Empirical Software Engineering, 1, 241–277 (1996)

Repeatable Software Engineering Experiments for Comparing Defect-Detection Techniques

CHRISTOPHER M. LOTT[1] lott@bellcore.com
Bell Communications Research, 445 South Street, MCC 1H-331B, Morristown NJ 07960, USA

H. DIETER ROMBACH rombach@iese.fhg.de
Fraunhofer Institute for Experimental Software Engineering, Sauerwiesen 6, 67661 Kaiserslautern, Germany and Department of Computer Science, University of Kaiserslautern, 67653 Kaiserslautern, Germany.

Abstract. Techniques for detecting defects in source code are fundamental to the success of any software development approach. A software development organization therefore needs to understand the utility of techniques such as reading or testing in its own environment. Controlled experiments have proven to be an effective means for evaluating software engineering techniques and gaining the necessary understanding about their utility. This paper presents a characterization scheme for controlled experiments that evaluate defect-detection techniques. The characterization scheme permits the comparison of results from similar experiments and establishes a context for cross-experiment analysis of those results. The characterization scheme is used to structure a detailed survey of four experiments that compared reading and testing techniques for detecting defects in source code. We encourage educators, researchers, and practitioners to use the characterization scheme in order to develop and conduct further instances of this class of experiments. By repeating this experiment we expect the software engineering community will gain quantitative insights about the utility of defect-detection techniques in different environments.

Keywords: code reading by stepwise abstraction, functional testing, structural testing, controlled experiments, empirical software engineering.

1. Introduction

Systematic software development approaches distinguish between verification and validation techniques. These techniques may be used to detect defects in requirements, design, code, test, or other artifacts. According to the IEEE definitions, verification techniques "evaluate a system or component to determine whether the products of a given development phase satisfy the conditions imposed at the start of that phase," and validation techniques "evaluate a system or component during or at the end of the development process to determine whether it satisfies specified requirements" (IEEE, 1983). Verification techniques are primarily applied by personnel in an off-line manner. Examples of these techniques include formal proofs based on Hoare or Mills semantics (Hoare, 1969; Gannon et al., 1987), semi-formal techniques such as reading by stepwise abstraction for design and code documents (Linger et al., 1979), active design reviews (Parnas and Weiss, 1985), and scenario-based reading for requirements documents (Porter et al., 1995). Validation techniques are applied on-line by running a software system with a set of test cases as inputs. Examples of these techniques include functional testing, in which the specification is primarily used to develop

test cases (Howden, 1980; Myers, 1978), and structural testing, in which the source code is primarily used to develop test cases (Howden, 1978; Marick, 1994).

The primary argument for using both off-line verification techniques and on-line validation techniques is the need to capture defects in requirements, design, and code artifacts as early as possible. Unless code is reused extensively, or formal notations for requirements and design documents are used, testing can first be applied only late in the development process. Therefore requirements and design defects that are not detected until testing begins will be extremely expensive to repair. To address this problem, developers must use both verification and validation techniques.

Experience has shown that project pressure will cause developers to abandon off-line, human-based techniques such as reading by stepwise abstraction unless they are convinced of the utility of those techniques. Therefore, an effective approach for convincing professional developers and university students of the utility of off-line techniques is needed. We recommend using the same principle underlying Watts Humphrey's Personal Software Process (Humphrey, 1995), namely asking developers to apply and measure the techniques themselves. This type of experience may be gained by conducting low-risk projects (the approach advocated by the Personal Software Process) or by taking part in repeatable software engineering experiments (the approach advocated in this paper).

An experimenter may repeat portions of a previously defined software engineering experiment, or may faithfully replicate all aspects of that experiment. In doing so, the experimenter may (or may not) be able to reproduce previous results. We see strong parallels to physics, where students repeat classic experiments such as measuring the charge on the electron, although they do not necessarily reproduce the expected results. By repeating experiments, physics students learn about working in the laboratory and improve their understanding of fundamental concepts. The learning benefits that accrue to the participants in a software engineering experiment are similar to those in physics. However, there is only one well-accepted value for the charge on the electron, so success for replicated physics experiments means reproducing previous results. We believe that success for a software engineering experiment should not be defined solely in terms of reproducing earlier results, but rather in terms of measuring and understanding the variation factors that affect how human beings perform in different environments. Therefore, we believe that the software engineering community can benefit greatly from a set of repeatable experiments that can be used in software engineering education as well as technology transfer programs. The work presented here is a first step in this direction.

This paper discusses experiments that compare the utility of verification (reading) and validation (testing) techniques for detecting defects in source code. The focus is on understanding variation factors and educating the participants, not on reproducing the results obtained by others. A characterization scheme for the many dimensions to be considered in these experiments is presented in Section 2. Section 3 surveys four similar experiments that compared defect-detection techniques and gives their results. These example experiments offer concrete, reusable goals, plans, and procedures. Special emphasis is placed on an experiment that we conducted at the University of Kaiserslautern. Finally, Section 4 draws some conclusions and identifies future work.

2. A Characterization Scheme for Experiments

Many have called for an infrastructure for experimental software engineering (Curtis, 1980; Basili et al., 1986; Fenton et al., 1994). The characterization scheme of this section is a contribution towards achieving that goal for experiments that evaluate source-code verification and validation techniques. The characterization scheme draws on work that appeared in Preece and Rombach (1994) and is summarized in Table 1. The characterization scheme permits the comparison of results from similar experiments and establishes a context for cross-experiment analyses. The scheme is divided into four major parts, namely the goals and hypotheses that motivate an experiment, the plan for conducting the experiment, the procedures used during the experiment, and finally the results.

2.1. Goals, Hypotheses, and Theories

A statement of goals determines what an experiment should accomplish, and thereby assists in designing and conducting the experiment (Basili et al., 1986). This section discusses the definition of experimental goals, addresses the derivation of testable hypotheses from goals, introduces a goal template for developing concrete goals suited to this class of experiments, and discusses the development of a theory that will explain the results.

2.1.1. Defining Experimental Goals

Many techniques are candidates for performing a specific software engineering task. Choosing one requires the practitioner to answer the question "which technique is best suited for achieving a specific task in my own environment?" The question may be alternately stated as "is technique 'A' better suited to the project's needs than technique 'B?' " If the subjective terms "best" and "better" can be defined quantitatively and objectively for a specific context, we can define concrete goals for experiments that will yield the information needed to answer these questions. An objective definition of a term such as "better" may be "detects 30% more defects" (i.e., is more effective) or "costs 10% less as measured in staff-hours" (i.e., is more efficient).

A statement of goals is vital for planning and conducting an experiment, although only part of the planning phase. For example, defect-detection techniques may be compared based on the time required to develop test cases or on the number of failures revealed. The choice of an aspect to be evaluated (e.g., effectiveness or efficiency of a defect-detection technique) will further dictate what data must be collected, and determines in part how that data can be analyzed.

We suggest using the GQM Paradigm to support the processes of stating goals, refining goals in an operational way into metrics, and interpreting the resulting data. The GQM Paradigm supports the definition of goals and their refinement into concrete metrics (Basili and Rombach, 1988; Basili et al., 1994; Differding et al., 1996). The idea behind the GQM Paradigm is that measurement should be based on goals. By stating goals explicitly, all data collection and interpretation activities are based on a clearly documented rationale.

Table 1. Characterization scheme for experiments that compare defect-detection techniques.

I. Goals, Hypotheses, and Theories
   A. Aspects of a goal
      1. Object of study (e.g., code reading, functional testing, ...)
      2. Purpose of study (e.g., compare, analyze, ...)
      3. Quality focus of study (e.g., effectiveness, efficiency, ...)
      4. Point of view (e.g., practitioner, experimenter, ...)
      5. Context (e.g., subjects, objects, environment, ...)
   B. Hypotheses
      1. Type (e.g., direct observations, context factors, ...)
      2. Expected result (i.e., null and alternative hypotheses)
   C. Theories:
      1. Mechanisms that predict and/or explain results
      2. Derived from beliefs or related work

II. Experiment Plan
   A. Experimental design
      1. Independent variables (e.g., techniques, objects, order, ...)
      2. Dependent variables (e.g., defects found, time required, ...)
      3. Randomization (e.g., match of subject, object, and technique)
      4. Repeated measures (e.g., within-subject designs)
      5. Manipulation of independent variables (e.g., full-factorial, partial-factorial, ...)
      6. Null hypotheses (e.g., technique A has no effect on ...)
   B. Defect-detection techniques for source code
      1. Type (e.g., reading, functional testing, structural testing, ...)
      2. Other aspects (e.g., test-case development, termination criteria, ...)
   C. Objects
      1. Source-code modules (e.g., length, complexity, ...)
      2. Faults (e.g., number, types, interactions, ...)
   D. Subjects
      1. Selection criteria (e.g., participants in a course)
      2. Experience, training, and background (e.g., students, professionals, ...)
      3. Ethical issues (e.g., right to withdraw, anonymity, ...)
      4. How many are required (assess power of analysis procedure)
   E. Data collection and validation procedures
      1. On-line and off-line collection procedures (e.g., forms, videotape, counts of runs, ...)
      2. Validation approaches (e.g., independent sources, interviews, ...)
   F. Data analysis procedures
      1. Significance level for inferential statistics (e.g., $p < 0.05$)
      2. Parametric techniques
      3. Non-parametric techniques

Table 1. Continued.

III. Experiment Procedures
   A. Training activities (e.g., independent work, controlled setting, ...)
   B. Conducting the experiment (e.g., time periods, data validity, ...)
   C. Giving feedback to subjects (e.g., comparing expectations with results, ...)

IV. Results
   A. Data (i.e., the raw data collected during the study)
   B. Interpretations (i.e., statements about the hypotheses)

Goal template. We can derive a series of goals for this class of experiments using the following goal template borrowed from the GQM Paradigm:

Analyze (*n*) **techniques for detecting software defects**
for the purpose of **(understanding, comparison, ...)**
with respect to their **(effectiveness, cost, efficiency, ...)**
from the point of view of the **(researcher, practitioner, ...)**
in the context of **(a specific context)**.

The first facet ("object of study") states that some number of defect-detection techniques will be analyzed or compared. The second facet ("purpose") states whether the experiment will characterize, compare, etc. the techniques under study. The third facet ("quality focus of study") is the primary effect that is the focus of the experiment. The fourth facet ("point of view") states the perspective from which the experiment will be viewed (i.e., for whom the results should be useful).

The fifth facet ("context") briefly states what personnel (i.e., subjects) and code artifacts (i.e., objects) will be used, and any other relevant information. A specific project may have a highly specific context that dictates how the experiment should be planned and conducted. Example context factors may be "experience of subjects," "application domain," or "implementation language." Therefore, the derivation of testable hypotheses must account for the context, possibly by stating expectations about the effect of various context factors on the results. For example, poor results might be expected if subjects are confronted with a technique that differs sharply from their experience.

2.1.2. Deriving Testable Hypotheses

Based on the goals, hypotheses must be derived that will be tested by the experiment. Statements about the expected results that can be tested using the experiment are called testable hypotheses. For example, the statement "technique A is good for task T" is not testable, whereas the statement "technique A requires less time than technique B to accomplish

task T" is testable. To support testing such statements using inferential statistical methods, these statements are eventually formulated as *null hypotheses*, and the original statement is called the *alternative hypothesis* (Judd et al., 1991). A null hypothesis simply states that no difference exists between some set of data. In the previous example, the corresponding null hypothesis would be "technique A requires the same amount of time as technique B for accomplishing task T." These null hypotheses will be tested by applying an inferential statistical analysis procedure to the data collected during an experiment (Box et al., 1978). That analysis will yield a probability value for the possibility that the results are due to chance (i.e., whether the null hypothesis must be rejected or accepted). The discussion of null hypotheses is continued in Section 2.2.2.

2.1.3. Theories to Explain the Results

Empirical work must lead to a deep understanding of phenomena, not stop at a demonstration of statistically significant differences. When constructing the hypotheses to be tested empirically, the researcher ordinarily relies on a theory that he or she believes will predict and explain the results. This theory (sometimes called a mechanism) must be testable (falsifiable). It further guides the selection of what to measure in the experiment. For example, if a researcher believes that the experience of subjects dwarfs the results of using different defect-detection techniques, then the researcher must find some way to measure subject experience. If validated, these theories represent the sorely needed understanding about software engineering techniques.

If the researcher has no theory, and no comparable work exists that might suggest a theory, then it may be premature to conduct an experiment that focuses on a quantitative evaluation. Instead, the researcher might want to consider conducting qualitative research that would assist with developing a theory as well as with identifying the constructs and variables pertaining to that theory.

2.2. Experiment Plan

This section discusses planning an experiment that will evaluate defect-detection techniques. The foundations of controlled experiments are discussed first. Then the issues of experimental design, defect-detection techniques, objects, subjects, data collection procedures, and data analysis procedures are presented.

2.2.1. Foundations

The foundations of controlled experiments include establishing credibility of the work; distinguishing between correlation and causality; addressing issues of construct validity, internal validity, external validity, and validity tradeoffs; and deciding upon the scope of the experiment. All are discussed next.

Credibility. Credibility is the cornerstone of empirical studies in any field. Genuine credibility stems from an experimenter's painstaking efforts in building up a collection of supporting evidence for his or her claims. This evidence should include a clear statement of goals and hypotheses, a falsifiable theory to explain the results, a convincing demonstration of the results, and a thorough discussion of the reasoning used to obtain the results.

Correlation versus causality. A correlation between two events merely states that the two events have been known to occur together. However, event A is said to cause event B only if a certain set of conditions holds (Votta and Porter, 1995). First, there must be some nonspurious association between the two events. Second, event A must happen before event B (temporal precedence). Third and finally, the experimenter must propose a theory (i.e., a mechanism) that explains why event A causes event B, and that further can be tested in the experiment. Both correlation and causality can be tested, but the value of a controlled experiment is its power in establishing causality.

Construct validity. Construct validity refers to the degree to which a given measure accurately characterizes some construct under study (Judd et al., 1991). For example, a count of computer science courses may be a poor measure of a subject's experience with a language (has poor construct validity), while an estimate of the number of lines of code that subject has written in that language is a better measure of experience (has acceptable construct validity). The experimentalist must never forget that all measures are imperfect, and must apply sound reasoning when choosing measures.

Internal validity. Internal validity refers to the extent to which causality is established as a credible explanation for the relationship between the presumed causes and measured responses (Judd et al., 1991). Campbell and Stanley identify eight threats to internal validity (Campbell and Stanley, 1966, pp. 5–6). One of the most important threats in software engineering experiments is called a selection effect. An example selection effect is a subject who has a natural ability for the experimental task due to many years of experience. Selection effects can be measured in part by using within-subject designs (i.e., measuring the same subject on multiple tasks). Selection effects can be addressed highly effectively if the subjects are known extremely well, an approach sometimes used in research on human-computer interface design. Another aspect of a selection effect can be mitigated by allowing subjects to withdraw from an experiment at any time. This freedom should ensure that the internal validity of the experiment is not threatened by stresses on the subjects that are invisible to the researchers.

External validity. External validity refers to the confidence with which the results can be generalized beyond the experiment (Judd et al., 1991). External validity is affected not only by the design but also by choices made for the objects chosen and the subjects who participate. For example, external validity of an experiment that evaluates defect-detection techniques can be increased by using experienced software engineers who are experts with the techniques to be evaluated. External validity in software engineering experiments may also be achieved, at high cost, by repeating an experiment many times, varying the subjects and objects each time.

Validity tradeoffs. The issues of construct, internal, and external validity are related.

Table 2. Scope of empirical studies (Basili et al., 1986).

Number of teams	Number of projects: One	Number of projects: More than one
One	Single project	Multi-project variation
More than one	Replicated project	Blocked subject-project

Choices that result in high internal validity may damage external validity, and vice versa. For example, a tightly controlled experiment may use very small pieces of code and undergraduate students to attain high internal validity. However, using toy code modules and inexperienced subjects sacrifices external validity, because the results will not necessarily hold in an environment where professionals test large software systems (Brooks, 1980). These tradeoffs demonstrate the value and importance of developing an explicit statement of experiment goals.

Scope of an experiment. Choosing the scope of an experiment depends on the desired external validity of the results (i.e., the context for which the results should be useful). For example, specialized case studies (narrow scope) may offer results that, although limited to a specific context, are extremely useful for that context (Lee, 1989; Glass, 1995). Basili et al. (1986) classified the scope of empirical studies using repeatability and organizational context as criteria, as reprinted in Table 2. The upper-left box in the table ("single project") might also be called a case study; one team (or even a single subject) participates in a single project (exercise). For example, one person may apply a single defect-detection technique to a single program, or a team may apply design inspections in the course of a development project. For maximum internal and external validity, a controlled experiment (as characterized by the lower right-hand corner of Table 2) will be required.[2]

2.2.2. *Experimental Designs*

We use the term "experiment" to mean a controlled investigation in which some random assignment of subjects to levels of independent variables has been done (Spector, 1981). The experimental design explains how this assignment is done, and thereby controls factors that will permit causality to be inferred (Campbell and Stanley, 1966). The design is fundamental to internal validity, and determines in large part the external validity of the results as well. This section discusses independent variables, dependent variables, randomization, repeated measurements, manipulation strategies, and null hypotheses.

Independent variables. An independent variable is a factor believed to influence the results of the experiment (i.e., a causal factor). The experimenter manipulates the values

assumed by an independent variable to study the effects of those different values on the results. The independent variables that primarily determine the external validity of this class of experiments are the defect-detection technique, the types and sizes of the programs (objects) to which the techniques are applied, and the types and number of faults in the programs. Given a within-subjects design, the independent variables that primarily determine the internal validity of this class of experiment are the order in which subjects apply the techniques, and the order in which subjects see the programs. Independent variables are said to be confounded if their values do not vary independently of each other. For example, if all subjects were to test the same program with the same technique, the variables 'technique' and 'program' would be said to be confounded.

Uncontrolled independent variables (also called subject variables) are those factors which may influence the results but which the experimenter cannot manipulate, or chooses not to manipulate. Examples of these variables include a subject's experience and motivation. However, the experimenter may attempt to measure these factors to test whether any correlation is seen with the results.

Dependent variables. The dependent variables measure the effects of manipulating the independent variables. For example, dependent variables may be defined to measure the responses of the subjects when they use a defect-detection technique. In this class of experiments, possible dependent variables include elapsed time, counts of failures revealed, counts of failures recorded, counts of faults isolated, etc.

Randomization. Randomization refers to the random assignment of subjects to different levels of the independent variables. In experiments that evaluate defect-detection techniques, subjects are randomly assigned the combination of a defect-detection technique and a code object. Randomization is also an important prerequisite for the use of certain statistical analysis procedures (see Section 2.2.7).

Repeated measurements. An important issue in designs that observe each subject multiple times (within-subject designs) is repeated measurements. Repeated measurements permit subjects to be compared with themselves, helping quantify selection effects. The experimenter also gains badly needed data points without recruiting and training a new set of subjects. However, having a subject perform several related tasks within a short time interval may cause nonnegligible learning effects.

Manipulation strategy. A design's manipulation strategy explains what combinations of causes (i.e., independent variables) are examined. An independent variable is commonly called a *factor* in the terminology of experimental design, and the values that each factor can assume are called *levels*. Factors (actually the levels) are commonly manipulated in an experiment by crossing them (Pfleeger, 1995a). A complete crossing of some number of factors is called a *full factorial design*. A full-factorial design tests all possible combinations of factors. This seemingly ideal situation permits the experimenter to measure interaction effects between the various independent variables. For example, the factors T (technique) and P (program) are crossed if all levels of factor T are matched with all levels of factor P, and would be expressed as "$T \times P$." The number of trials required for a full-factorial design is the product of the number of levels of each factor. Assuming three techniques

(three levels for factor T) and three programs (three levels for factor P), a minimum of $3 \times 3 = 9$ trials would be required.

Due to the extreme variability in skill among human subjects, software engineering experimentalists would ideally like to use repeated measurements as well as full-factorial designs in their experiments. Thus ideally an experimenter would like to have the same subject test the same program with many different defect-detection techniques. However, learning effects make it ridiculous for subject to apply a second, different technique to a program in which she or he already detected defects once. A compromise is needed.

Software engineering experiments commonly use a partial-factorial design to permit repeated measurements. A *partial-factorial design* omits some combinations of factors for various reasons. Because learning effects prevent an experimenter from testing all combinations of subject, technique, and object, realistic designs for this class of experiments must omit the combinations in which a subject sees the same object (program) more than once. There are further tradeoffs between a full-factorial and an efficient experimental design. For further information, see for example (Pfleeger, 1995b).

Null hypotheses. The design determines a minimum set of null hypotheses regarding external and internal validity of the experiment. The hypotheses concerning external validity correspond directly to the testable hypotheses derived from the goals; the rest check for threats to internal validity. All of the null hypotheses derived from the design are commonly tested using inferential statistical analysis procedures. These tests are discussed in Section 2.2.7.

In the class of experiments discussed here, the primary null hypothesis for external validity states that the different techniques (i.e., the various levels for factor "technique") have no effect on the values of the dependent variables. Additionally, null hypotheses concerning internal validity issues help the experimenter quantify threats such as selection or learning effects. For example, a null hypothesis may state that the different objects or subjects used in the experiment have no effect on the values of the dependent variables.

Two errors are possible when an experimenter considers whether to accept or reject a null hypothesis (Judd et al., 1991, pp. 396 ff.) The first error (called a "Type I error") consists of rejecting the null hypothesis although it was in fact true. For example, an experimenter may erroneously state that a technique helps subjects detect defects more rapidly, although it in fact did not. The second error (called a "Type II error") consists of failing to reject the null hypothesis although it was in fact false. For example, an experimenter may erroneously state that two techniques helped subjects detect the same percentage of defects, although in fact they differed.

2.2.3. *Defect-Detection Techniques*

The defect-detection techniques are the primary target of study in this class of experiments. We prefer to avoid the term *treatments* as used by some researchers (see e.g., Votta and Porter, 1995) because in psychological and medical experiments, the experimenters apply some *treatment* such as medication to subjects and measure the subject's responses (see e.g., Aronson et al., 1985). In contrast, the class of experiments discussed here requires subjects

to apply some techniques to some objects, and then to measure the responses themselves. Thus the subject's ability to apply the technique may obscure the technique's power. For example, a subject may succeed in revealing a failure, but then fail to notice the revealed failure. In that case, the technique was successful, but the subject measured the response poorly.

Because the subject applies the technique, the process to be followed must be well-defined and documented. Issues of process conformance, motivation, and bias can then be addressed. The issue of process conformance asks whether the subject's application of the technique deviates from the prescribed process (Sørumgård, 1996). Motivation is an issue in all experiments, and especially so because the subject's creativity and insight is critical when developing test cases. A less than motivated subject may exert little effort in applying the technique and consequently detect few defects, a poor result that cannot entirely be blamed on the technique. Finally, if the subject is biased for (or against) the language or defect-detection technique, the subject may perform better (or worse) as a result. It is difficult to find a measure of motivation or bias that exhibits high construct validity. Still, even using an imperfect measure it may be possible to mitigate the effects of biased and motivated subjects by spreading subjects of different motivation and bias levels equally over all combinations. This can be done by defining groups (blocks) based on some assessment of bias and motivation, and then randomly assigning members of each block to the combinations of technique and object.

The classes of defect-detection techniques considered in this class of experiments are inspections and other reading (verification) techniques, functional testing, and structural testing. Each is discussed next.

Reading techniques. These verification techniques are used to detect defects in code without using the computer. Software practitioners use many different off-line reading techniques. Notable differences among the techniques include the use of individuals versus teams, the availability of tool support, and the use of meetings (Porter et al., 1995a). Example verification approaches include walk-throughs (subjects "walk through" the code, "executing" it on paper), structured inspections (subjects detect defects on their own and hold meetings to collect the defects) (Fagan, 1976; Porter et al., 1995), and code reading by stepwise abstraction (subjects write their own specification of the code and compare it with the official specification) (Linger et al., 1979). The code reading approach used in several experiments surveyed in this paper may be considerably more rigorous than either a walk-through or code inspection.

Functional testing. Also known as "black box" testing, this validation technique is characterized by using primarily the specification to develop test cases. Approaches such as equivalence-class partitioning and boundary-value analysis may be used to choose test data; these approaches serve as termination criteria. In general, subjects use the specification to develop test cases, run the test cases to reveal failures, and use the specification to identify failures.

Structural testing. Also known as "clear box" or "white box" testing, this validation technique is characterized by using primarily the source code to develop test cases. The termination criteria for structural testing are defined in terms of *coverage criteria*; i.e., some

statement is made about what percentage of the code or percentage of paths through the code must be exercised by the set of test cases. Many different coverage criteria may be selected; one example is 100% statement coverage, meaning that each statement is executed at least once (Myers, 1979). In general, subjects use the source code to develop test cases, run the test cases to obtain the desired coverage and reveal failures, and use the specification to identify the failures.

2.2.4. *Objects*

The objects (sometimes called instruments or artifacts) are the pieces of faulty code to which the subjects apply the chosen defect-detection techniques during training and experimental exercises.

Code modules. The use of objects of a nontrivial size is important for external validity. However, the maximum amount of effort that subjects can expend may be sharply limited, forcing the use of small pieces of code. The minimal requirements for the code modules used in this class of experiments are that the implementation language is known to the subjects and that the size is appropriate for completing in the time allotted. Optimal requirements for the objects will satisfy the needs of the researcher and the context (i.e., consider the viewpoint and context of the experiment as stated in the goal). For example, from the viewpoint of a researcher who is primarily interested in internal validity, optimal objects are generally small, stand-alone programs that can be analyzed relatively quickly. In contrast, from the viewpoint of a practitioner or development manager who is interested in external validity, optimal objects would be those taken from a current project.

Faults. The experimenter needs code modules that exhibit some number and type of faults that manifest themselves as failures at run-time. Ideally, code modules with naturally occurring faults can be used; otherwise, the experimenter must seed faults into some code modules. A balance must be found between an extremely low number of faults in the source code (subjects may not find any) and an unrealistically high number of faults (subjects may become disgusted with the task). Further, the experimenter will want to balance the choice of faults of various types. We classify faults using two orthogonal facets as discussed in Basili and Selby (1987). Facet one (omission, commission) describes whether the fault was due to the absence of necessary code or the presence of incorrect code. Facet two (initialization, computation, control, interface, data, cosmetic) describes the nature of the fault in more detail; i.e., what code construct was written incorrectly. One caveat is that the classification of faults is a subjective process. Finally, the experimenter must make a tradeoff between internal and external validity with respect to faults and their interactions. For example, internal validity is improved at the expense of external validity if all faults cause observable failures and interact as little as possible.

2.2.5. *Subjects*

This section discusses the selection of subjects, their experience, ethical issues, and the number of subjects needed.

Selection. Ideally, the subjects would be a random sample of the population of computer science professionals. However, attaining a random sample is difficult. Instead, subject groups are commonly an accidental sample of this population (Judd et al., 1991), meaning that they are selected based on their participation in university or industrial courses. However, it is important *not* to ask for volunteers to avoid the problems of self-selection. Instead, all course participants should be used.

Experience. Both university students and professional developers have been used as subjects in this class of experiments. In the software engineering domain, these two types of experimental subjects have been classified as performing experiments *in vitro* (i.e., with students) versus *in vivo* (i.e., with professionals) (Votta and Porter, 1995). The argument is that experiments should be piloted in universities to detect problems in the design and procedures at a relatively low cost. A more costly study involving professionals can then be performed with the knowledge that all experimental materials have been rigorously tested. These two groups have different levels of experience, which determines how seriously the threat of learning effects must be taken. A subject's knowledge of the application domain, the test environment, the operating system, and the support tools all influence learning effects. If subjects lack this experience, learning effects are likely to dominate the results. An optimal subject would have a detailed knowledge of the implementation language as well as all of the defect-detection techniques that will be analyzed.

Ethical issues. Many ethical issues that arise in psychological experiments using human beings are happily absent from this class of experiments. These include involving people without their consent, deceiving subjects, invading the privacy of subjects, or withholding benefits from control groups (Judd et al., 1991, Chapter 20). However, student subjects are commonly required to participate in experiments as a condition of passing a course, a practice that is considered acceptable provided that no *undue* coercion is present, *and* that the procedures provide an educational benefit to the subject (see Judd et al., 1991, pp. 491–492) for a detailed discussion). Further, it is important that quantitative results are not used when determining course grades, just the fact that the student did or did not participate. In the context of an experiment run in industry, job pressures to participate in an experiment are also undesirable.

Another part of experimental ethics, and a way of reassuring the subjects that their performance will not be evaluated and used against them, involves preserving their anonymity. Preserving anonymity is especially difficult when an educator wants to ensure that all members of a course have participated in an experiment. Still, some semblance of anonymity can be preserved by assigning subjects an identifier, and using strictly that identifier on all data collection forms. The correspondence between identifiers and true identities need not ever be recorded.

How many? The question "how many subjects should be used" can be answered by examining first the design and second the power of the statistical analysis technique that will be used. The number of independent variables and the number of levels that each variable can assume sets an absolute minimum, because the experimenter wants at least one observation of each combination. Based on an experimenter's choice of statistical

analysis technique and significance level, the power of the technique must be examined. The power of an analysis technique refers to its sensitivity and is defined as the probability of making an error of Type II; i.e., incorrectly accepting the null hypothesis. The power of a technique is commonly expressed in a power table. For a given analysis technique, a power table relates the four factors sensitivity (power level), effect size, significance level, and number of observations (i.e., subjects). These tables can be found in references such as Cohen (1988). Larger samples lead to greater powers for a given analysis technique. An especially difficult part of power analysis is estimating the effect size that is expected in the experiment. See also Miller et al. (1995) for a discussion of power analysis in software engineering experiments.

2.2.6. *Data Collection and Validation Procedures*

The subjects will collect most of the values for the dependent variables during the experiment using data-collection forms. Answering the following questions about the dependent variables (metrics) will help plan the content and number of forms that are required for the study.

- What data is collected once?
- What data is collected periodically?
- What collection can be automated?

We recommend testing the forms before conducting the experiment; a training phase may serve this purpose if there is sufficient time between the end of the training phase and the start of the experiment to repair any problems found.

After the subjects have completed the exercises, we recommend performing detailed interviews of subjects to validate their data (see also (Basili and Weiss, 1984) for a discussion of the dangers of not conducting interviews). These interviews let the experimenter inquire whether the subjects applied the techniques as prescribed (process conformance), determine whether the subjects understood how to supply the data that were demanded of them (data validity), and check other issues that might cause misleading results.

2.2.7. *Data Analysis Procedures*

An inferential statistical analysis procedure is commonly used to test the null hypotheses that the experimenter derived from the design. These procedures offer a judgement as to the probability that the results were due to chance. Selecting an appropriate analysis procedure to evaluate the results of an experiment can be a challenging task. In the case of software engineering experiments, small data sets and significant restrictions on those already limited data are common (Zweben et al., 1995). The data analysis procedure should ideally be selected when the experiment is planned, because the choice of data analysis

procedures is dependent on the experimental design as well as the collected data. We discuss briefly significance level, power analysis, parametric analysis procedures, nonparametric analysis procedures, and other pattern-recognition approaches. See Briand et al. (1996) for a comprehensive discussion of data analysis procedures in the software engineering domain.

Significance level. An important issue is the choice of significance level. Common practice dictates rejecting the null hypothesis when the significance level is ≤ 0.05 (Box et al., 1978, p. 109). If multiple hypotheses will be tested simultaneously, a lower value must be used. On the other hand, research in the social sciences will sometimes use a value of 0.10, depending on the design. Regardless of the value, it should be chosen before the experiment is conducted.

Power of analysis technique. An important selection criteria is the power of an analysis technique, as discussed in Section 2.2.5, meaning the variation that a technique can detect given the available observations and the desired significance level.

Parametric analyses. Parametric analysis procedures infer whether the differences among sets of data are significant by testing whether the variance among sets of data is larger than the variance within those sets. These procedures assume that the distribution for the data can be adequately described with a number of parameters. A common assumption is that the data exhibit a normal distribution. A simple test for normality in a data set involves exploratory data analyses such as graphing the data to check for outliers and deviations from the expected bell curve of data. For example, parametric techniques such as ANOVA are commonly used when randomization has been performed and assumptions of normality can be justified. Some good news for experimenters is that parametric techniques are robust to nonnormality under certain conditions, most importantly randomization (Box et al., 1978, pp. 46ff. and p. 104). Even if the assumptions are violated, analysis procedures such as the t test become more conservative; i.e., they will not make a type I error, but may lead the experimenter to make a type II error (Briand et al., 1996).

Nonparametric analyses. If the data lie on an ordinal scale, randomization was not part of the experiment, or a normal distribution for the population is badly violated due to many outliers, then a nonparametric analysis procedure is an appropriate choice. These procedures assume nothing about the distribution of the data. They function by ranking the data points within a data set and then analyzing only the rank information. Although these analysis methods offer a low risk of making a Type I error, they may lead the experimenter to make a Type II error because of their reduced power when compared to the parametric analysis procedures.

Other approaches. A number of pattern-recognition approaches have been applied to software engineering data sets. These include classification trees (Selby and Porter, 1988) and optimized set reduction (Briand et al., 1993). These approaches make no demands or assumptions about the distribution of the data sets, but generally require a large number of data points before they are helpful. Because of those requirements, they are not usually suitable for analysis of the relatively small data sets obtained from case studies or controlled experiments.

2.3. *Experiment Procedures*

This section discusses the procedures involved in conducting an experiment to compare defect-detection techniques. Experimental procedures describe precisely how the experiment will be performed, including how subjects will apply the chosen techniques to the chosen objects, as well as what aspects of monitoring, computer lab space, etc. must be considered. Procedures are essentially the implementation of the design, plus some extra details that are not independent variables but can nonetheless be the downfall of an experiment.

2.3.1. *Phase 1: Training*

If the techniques to be applied are unknown to the subjects, or if the subjects are known to have different levels of expertise with the techniques, then a training phase is highly recommended. The purpose of a training phase is to reduce the differences in the primary uncontrolled independent variables, namely the experience and capabilities of the subjects. Training reduces, but cannot eliminate, learning effects, and thereby can improve the internal validity of an experiment. A training session also familiarizes subjects with the techniques and experimental procedures such as data collection forms. Further, a training session can serve as a dry run of the experiment; i.e., it offers the experimenter a chance to debug the experimental procedures and data collection forms.

Ideally the training phase and the experiment will be performed with as little time separating them as possible. We suggest performing the training activities within a week's time, and running the experiment the following week.

2.3.2. *Phase 2: Experiment*

In the experiment, the subjects apply the techniques to the experimental objects. We recommend that the experimenters monitor the rooms in which subjects work. The experimenters can answer questions from subjects, assist subjects in filling out the forms used to capture data, and prevent any undue influences that might result from subjects helping each other. Depending on the nature of the experiment and the granularity of data that the experimenters require, other data collection methods may also be used. For example, experiments that investigate human-computer interaction issues such as the usability of a test-support tool may require the subjects to be videotaped during their exercises to permit additional analyses (Preece and Rombach, 1994).

2.3.3. *Phase 3: Analysis, Interpretation, and Feedback*

To improve the value of the experiment as a educational exercise, an interpretation session should be scheduled for reporting the results to the subjects as soon as possible following the experiment. The results can be discussed as soon as the researchers have performed

preliminary statistical analyses. This provides the subjects with feedback that may confirm or refute the subject's expectations. Feedback also adds objective, empirical data to the subject's beliefs about the utility of the techniques used in the experiment.

2.4. Results

The final aspects of any experiment are the raw data, the results of any inferential statistical analyses performed on the raw data, and interpretations of the statistical analyses. The power of the statistical analysis technique must be considered and reported with the results. If application of the statistical analysis technique yields results that indicate no significant differences, this should be interpreted to mean that the differences were less than the variation that can be detected by the analysis technique (i.e., its power). For example, if a power table reports that the combination of technique, significance value, and number of observations yields a power of 90%, then the technique will not detect significant differences that are less than $1 - 0.9 = 10\%$. Section 3 surveys results from four comparable experiments.

Failures to reproduce earlier results. The results of a replicated experiment sometimes do not agree with the original results. To help understand failures to reproduce other's results, researchers must supply as much information as possible to permit an analysis of possible differences between the experiments. The characterization scheme of Table 1 is a guide to the information that should be reported. Special attention should be paid to the design, the subject profiles (education, experience, training), measurement procedures (bias, reliability, questionnaires), and materials (code, documents, instructions) used in the experiment. This information may assist others in identifying confounding variables or cultural differences that the researcher did not or could not control.

3. Survey of Comparable Experiments

This section uses the schema from Table 1 to survey four comparable experiments that compared reading and testing techniques. Special emphasis is placed on an experiment that we conducted at the University of Kaiserslautern. As in previous work (see, e.g., Basili and Weiss, 1985), this section demonstrates the difficulty of comparing experiments in software engineering and drawing meaningful conclusions from the results.

3.1. Hetzel (1976)

Hetzel performed a controlled experiment that compared three defect-detection techniques (Hetzel, 1976).

Table 3. Hetzel's experimental design (Hetzel, 1976).

	Group A			Group B			Group C		
Subject	s. 1	s. 2	s. 3	s. 1	s. 2	s. 3	s. 1	s. 2	s. 3
1	DR1	MT3	ST2	DR2	MT1	ST3	ST1	MT2	DR3
2	DR1	MT3	ST2	DR2	MT1	ST3	MT2	DR3	ST1
...									
12	DR1	ST2	MT3	DR2	ST3	MT1	ST1	DR3	MT2
13	ST2	DR1	MT3	MT1	ST3	DR2	DR3	ST1	MT2

Goal. The template presented in Section 2.1.1 was used to construct a goal for this experiment.

Analyze **three defect-detection techniques**
for the purpose of **comparison**
with respect to their **effectiveness at revealing failures**
from the point of view of the **researcher**
in the context of a **controlled experiment**.

Experimental design. Table 3 summarizes Hetzel's experimental design. Subjects were randomly divided into groups A, B, and C to match the technique with the program. Then the order in which they saw the programs and applied the defect-detection techniques was randomized. For example, subject 2 from group A applied disciplined reading (DR) to program 1 during session 1, applied mixed testing (MT) to program 3 during session 2, and applied specification testing (ST) to program 2 during session 3.

Defect-Detection Techniques. The following three techniques were compared.

DR (disciplined code reading): A specification was developed by characterizing the lowest-level code structures ("paragraphs") first, then computing the effects of the total program by combining the effects of the paragraphs. This specification was compared with the official specification to detect defects (inconsistencies).

ST (specification testing): In this functional test technique, only the specification was used to derive test cases. No other criteria for developing test cases were specified. Defects were detected by evaluating the output with the specification.

MT (mixed testing): In this mix of functional and structural testing (also called selective testing in the original source), subjects could use the specification to develop test cases, but were also given a goal of achieving 100% statement coverage. Defects were detected by evaluating the output with the specification.

Objects. Three highly structured PL/I programs of 64, 164, and 170 statements were used. The programs contained faults that caused 9, 15, and 25 different types of failures, respectively.

Subjects. The 39 subjects were selected based on their experience with PL/I. The subjects averaged over 3 years of programming experience and were a mix of graduate students and professionals. Selection and motivation were driven by a significant monetary reward that varied with performance (minimum US $75, maximum US $200 in 1976 dollars).

Data collection procedures. Data collection forms were used to collect data about the subject's background, the subject's attitudes towards the techniques, and the faults and failures detected while applying the techniques.

Data analysis procedures. Parametric statistical procedures (ANOVA) were used to analyze the data.

Experimental procedure. The training session and experimental sessions were performed during a span of 6 days (Monday through Saturday). The training session consisted of an introduction to the techniques, a presentation of the specifications, and an overview of the structure of the three programs that would be used during the experiment. Thus in contrast to some experiments, the subjects were partially acquainted with the object before beginning the exercises, but they had not practiced the testing tasks.

Batch processing of jobs was used; turnaround time averaged 15 minutes. Monitors submitted jobs and fetched output. Six sessions were held to reduce the load on the machine (odd-numbered subjects participated in three of the sessions, and even-numbered subjects participated in the other three).

A time limit was enforced (3.5 and 4.5 hours maximum, depending on the program). Any breaks or pauses that the subjects took were not charged against the time limit. Only limited data were collected about the actual time required.

Results. No significant difference in effectiveness was detected among the two testing techniques. However, the data showed that the subjects who applied the reading technique performed less effectively than those who applied the testing techniques (significance level < 0.01). The differences attributable to the programs were even larger than the differences attributable to the techniques, suggesting a strong selection effect. Measures of the individual's effectiveness correlated well (some significance levels < 0.01) with the number of computer science courses they had taken. Hetzel also noted that the subjects observed only about 50% of the revealed failures, and that the separation of best and worst performers was a factor between 2 and 3.

3.2. Myers (1978)

Myers performed a study that compared three defect-detection techniques (Myers, 1978).

Goal. The template presented in Section 2.1.1 was used to construct a goal for this experiment.

Table 4. Myers' experimental design (Myers, 1978).

Group A: Functional test	Group B: Structural test	Group C: Walk./inspec.
PL/I rating: 1.5	PL/I rating: 2.1	PL/I rating: 2.4
W/I rating: 0.2	W/I rating: 0.3	W/I rating: 0.6

Analyze **three defect-detection techniques**
for the purpose of **comparison**
with respect to their **effectiveness and efficiency at revealing failures**
from the point of view of the **researcher**
in the context of an **industrial course**.

Experimental design. Table 4 summarizes Myers' experimental design. Each subject was observed once; no randomization was performed. To reduce bias, subjects were divided into three groups based on their experience with PL/I and walkthrough/inspection techniques, as shown in the table. Each subject's PL/I rating was computed using an ordinal scale (1 = no understanding, 2 = rudimentary understanding, and 3 = advanced understanding of the language), and the walk-through/inspection rating was computed using a different ordinal scale (0 = no experience, 1 = some experience with these techniques). The values in Table 4 are group means.[3] However, an insufficient number of subjects were experienced with PL/I and walk-throughs/inspections to permit balancing the groups based on these experience ratings. The result was that subjects applied the technique with which they had the most experience (average over the group, not true for all individuals). Groups A and B had 16 subjects each; group C had 9 teams of 3 people each.

Defect-Detection Techniques. The following three techniques were applied by the subjects.

FT (functional testing): Subjects in Group A only had access to the program's specification and an executable version. No method for deriving test cases was specified.

ST (structural testing): Subjects in Group B had access to the program's specification and its source code. No method for deriving test cases was specified, nor were any coverage criteria required.

W/I (walk-through/inspection method): Subjects in Group C were divided into teams of 3 and then asked to "test" the program using an *ad hoc*, manual walkthrough/inspection method. Individuals prepared separately, then met to collect defects.

Object. A single PL/I program of 63 statements that formats input text on a line-by-line basis was used. This program contained 15 faults.

Subjects. The subjects were 59 software developers who participated in a course held for IBM employees.

Data collection procedures. Data was collected via data-collection forms.

Data analysis procedures. Myers used nonparametric methods (specifically Kruskal-Wallis) and correlation tests to analyze the data.

Experimental procedure. Subjects were pretested to assess their experience with PL/I and with the walkthrough/inspection technique. The results of the pretests were used to form groups of similar abilities (see discussion of design).[4] No training phase is described, although lectures were held on program testing during the course.

During the experiment, the subjects who performed testing were instructed to test until they believed they had found all of the failures (no time limit). The amount of time that they required was recorded. The subjects who performed the walkthrough/inspection method took the materials with them, prepared on their own, and reported their time. The subsequent collection session was limited to 90 minutes.

Results. The data showed no significant differences in effectiveness among the three techniques. However, differences in time per fault were judged to be highly significant (no level reported), the walkthrough/inspection method required the most time, functional testing somewhat less, and structural testing the least amount of time. Based on the large variability observed among individuals, correlations of the results with uncontrolled independent variables such as testing experience and performance on the pretest were checked, but no large, significant correlations were found. Even ignoring significance issues, Myers reported that the results were somewhat disappointing: these experienced subjects largely ignored unconventional input cases in favor of conventional ones, overlooked many revealed failures, and isolated on average only one-third of the known faults.

3.3. Basili & Selby (1987)

Basili & Selby developed a controlled experiment that compared three defect-detection techniques (Selby, 1985; Basili and Selby, 1987). They conducted their experiment three times. The first two repetitions used a total of 42 advanced students, but those results are not presented here. The discussion below focuses on the last of the three repetitions, when professionals were used.

Goal. The template presented in Section 2.1.1 was used to construct a goal for this experiment.

Analyze **three defect-detection techniques**
for the purpose of **comparison**
with respect to their **effectiveness and efficiency at revealing failures**
from the point of view of the **researcher**
in the context of an **industrial development group**.

Experimental design. Table 5 summarizes Basili & Selby's experimental design. The subject's experience was considered an independent variable and was characterized in terms of years of professional experience and academic background. A random match of subjects,

Table 5. Basili & Selby's experimental design (Basili and Selby, 1987).

Subjects		Code reading	Functional testing	Structural testing
Advanced	S1	P4	P3	P1
	S2	P3	P1	P4
	...			
	S8	P1	P4	P3
Intermed.	S9	P3	P1	P4
	S10	P4	P3	P1
	...			
	S19	P1	P4	P3
Junior	S20	P3	P1	P4
	S21	P1	P4	P3
	...			
	S32	P4	P3	P1

techniques, programs, and experience levels was made. Subjects worked with the programs in a fixed order; i.e., all subjects first worked with program P1 on day 1, then with P3 on day 2, and finally with P4 on day 3. (There was a program P2, but it was not used in the third repetition.) Given this ordering, the table shows the randomization of the order in which the techniques were applied. For example, advanced subject S2 applied functional testing to P1 on day 1, applied code reading to P3 on day 2, and applied structural testing to P4 on day 3.

Defect-Detection Techniques. The following three defect-detection techniques were compared.

CR (code reading): Individuals use the source code and the technique of "code reading by stepwise abstraction" to write a specification for the program. Using this technique, individuals identify prime subprograms (basic blocks) in the source, write a specification for that subprogram, and repeatedly combine specifications into larger ones until they have captured the behavior of the program. The subjects compare their specifications with the official specification to detect inconsistencies.

FT (functional testing): Subjects analyze the specification to identify equivalence classes in the input data. They then choose test cases based on that analysis by focusing on equivalence-class boundaries, run the test cases, and compare the actual output with the expected output to detect failures.

ST (structural testing): Subjects use a source-code listing to construct test cases that will lead to 100% statement coverage. After running the tests, individuals compare the actual output with the expected output to detect failures.

Objects. Three FORTRAN programs of 169, 147, and 365 source lines were used. These programs had 9, 6, and 12 faults, respectively. One of the programs (P4) was rewritten in FORTRAN and reused from the Hetzel study, and one program (P1) was rewritten in FORTRAN and reused from the Myers study. The objects are available (see appendix). Faults were classified into different types and classes to enable checking whether the techniques differed in the detection of failures caused by different kinds of faults.

Subjects. The 32 subjects who participated in the third repetition were professional programmers. These subjects had anywhere from 1.5 to 20 years of professional experience.

Data collection procedures. Data-collection forms were used, and some assistance was gained from the computer.

Data analysis procedures. A parametric analysis procedure was used, specifically ANOVA.

Experimental procedure. The training session consisted of a 4-hour tutorial on the techniques to be applied; the subjects had previously used only functional testing. Subjects applied the techniques in three separate sessions. Following the three sessions, a follow-up "debriefing" session was held to discuss the experiment.

Results. Analysis of the data showed that the code readers detected the largest percentage of inconsistencies (comparable with failures in the testing techniques, significance level < 0.01), and that the functional testers detected more failures than the structural testers (significance < 0.01). However, the subjects spent a similar amount of time applying all three techniques (no significant differences were found), resulting in code reading being the most efficient at revealing failures (significance level < 0.01).

As in the Hetzel study, a disappointing result was that on average only 50% of possible failures were revealed and recorded. Basili & Selby noted that code readers have revealed inconsistencies (comparable with failures) and done much work towards isolating the faults, whereas the testers have only revealed failures. This was the motivation for the extension done in the Kamsties & Lott experiment.

3.4. Kamsties & Lott (1995)

Kamsties & Lott extended the design and techniques originally used by Basili & Selby (1987) and conducted the resulting experiment twice (Kamsties and Lott, 1995). The extension consisted of a step of fault isolation following failure detection. Next we present an overview of the Kamsties & Lott experiment according to the characterization scheme of Table 1. First the goals from that experiment are presented. Then the experiment plan is presented, including the design, techniques, objects, subjects, data-collection procedures, and data-analysis procedures. Finally the procedures are stated, and results are given. Guidelines for reusing all of these aspects are also presented.

3.4.1. Goals and Testable Hypotheses

We present a detailed discussion of the goals and hypotheses from the Kamsties & Lott experiment so that the interpretation and explanation of the results can be substantiated with

this qualitative information. The following four concrete goals also demonstrate the use of the goal template from Section 2.1.1.

Goal 1: Effectiveness at revealing failures. The goal is stated as follows:

Analyze **code reading, functional testing, and structural testing**
for the purpose of **comparison**
with respect to their **effectiveness at revealing failures/inconsistencies**
from the point of view of the **researcher**
in the context of a **university lab course** using **small C programs**.

The effectiveness of revealing failures[5] is defined as the percentage of the total possible failure classes that were revealed. Provided that faults do not interact, one failure class is defined for each fault. Only failures that were both revealed by the subject's detection efforts *and* were recorded by the subject are counted to compute this percentage. Therefore meeting goal 1 requires answering the following questions:

Q1.1: What percentage of total possible failures (i.e., total possible failure classes) did each subject reveal and record?

Q1.2: What effect did the subject's experience with the language or motivation for the experiment have on the percentage of total possible failures revealed and recorded?

These questions may be answered by collecting data for the following metrics:

M1.1: The number of different, possible failure classes.
A count of total possible failure classes is derived from the count of total faults. If faults do not interact, then each fault produces only one failure class.

M1.2: The subject's experience with the language, estimated on a scale from 0–5.

M1.3: The subject's experience with the language, measured in years of working with it.

M1.4: The subject's motivation for the experiment, estimated on a scale from 0–5.

M1.5: The subject's mastery of the technique, estimated on a scale from 0–5.

M1.6: The number of times a test case caused the program's behavior to deviate from the specified behavior (i.e., revealed some failure class).

M1.7: The number of revealed deviations (i.e., unique failure classes) that the subject recorded.

Testable hypotheses are derived from the statement of goal 1, the questions, and the metrics as follows:

H1.1: Subjects using the three defect-detection techniques reveal and record a different percentage of total possible failures. Rewritten as a null hypothesis: Subjects record the same percentage of total possible failures.

H1.2: A subject's experience and motivation affect the percentage of total possible failures s/he reveals and records. Rewritten as a null hypothesis: Measures of experience and motivation have no correlation with the data for percentage of failures.

Goal 2: Efficiency at revealing failures. Goal 2 differs from goal 1 only in that we replace the word "effectiveness" with "efficiency." It is stated as follows:

Analyze **code reading, functional testing, and structural testing**
for the purpose of **comparison**
with respect to their **efficiency at revealing failures**
from the point of view of the **researcher**
in the context of a **university lab course** using **small C programs.**

We define efficiency at revealing failures as the percentage of total possible failures divided by the time required. Again failures must have been revealed by a subject's test case as well as recorded by the subject to be counted. Therefore meeting goal 2 will require us to answer the following questions:

Q2.1: How many unique failure classes did the subject reveal and record per hour?

Q2.2: What effect did the subject's experience with the language or motivation for the experiment have on the number of unique failure classes revealed and recorded per hour?

The question may be answered by reusing much of the data that was collected for the metrics of goal 1, plus one more:

M2.1: The amount of time the subject required to reveal and record the failures.

Testable hypotheses are derived from the statement of goal 2, the questions, and the metrics as follows:

H2.1: Subjects using the three defect-detection techniques reveal and record a different percentage of total possible failures per hour. Rewritten as a null hypothesis: Subjects record the same percentage of failures per hour.

H2.2: A subject's experience and motivation affect the number of failures s/he reveals and records per hour. Rewritten as a null hypothesis: Measures of experience and motivation have no correlation with the data for failure-detection rate.

Goal 3: Effectiveness at isolating faults. Previous goals addressed failures (inconsistencies for the code readers), and in a group dedicated to testing, work stops there. However, a development group may also be interested in the next step, namely isolating faults (i.e., the aspect of the source code that may cause a failure at runtime). In other words, after applying a given defect-detection technique, how difficult is fault isolation? This goal addresses that follow-on activity (the extension to the Basili & Selby experiment) and is stated as follows:

Analyze **code reading, functional testing, and structural testing**
for the purpose of **comparison**
with respect to their **effectiveness at isolating faults**
from the point of view of the **researcher**
in the context of a **university lab course** using **small C programs.**

Of course the defect-detection technique does not isolate a fault, but rather a person isolates a fault. We define fault-isolation effectiveness as the percentage of known faults that were isolated by the subject after having applied a technique. An important caveat for counting isolated faults was requiring that a failure corresponding to the isolated fault had been revealed. Without this requirement, the subject could have isolated the fault purely by chance, not based on the use of a defect-detection technique. Therefore meeting goal 3 will require us to answer the following questions:

Q3.1: What percentage of total faults (that manifested themselves in failures) did each subject isolate?

Q3.2: What effect did the subject's experience with the language or motivation for the experiment have on the percentage of total faults isolated?

These questions may be answered by collecting data for the following metrics:

M3.1: The number of faults present in the program (known in advance because faults are seeded).

M3.2: The number of faults that manifested themselves in failures.

M3.3: For all faults that manifested themselves in failures, the number of those faults that were isolated.

Testable hypotheses are derived from the statement of goal 3, the questions, and the metrics as follows:

H3.1: Using the information resulting from applying one of the three defect-detection techniques, subjects then isolate a different percentage of total faults. Rewritten as a null hypothesis: subjects isolate the same percentage of faults.

H3.2: A subject's experience and motivation affect the percentage of total faults s/he isolates. Rewritten as a null hypothesis: Measures of experience and motivation have no correlation with the data for percentage of faults.

Goal 4: Efficiency at isolating faults. Parallel to goals 2 and 1, Goal 4 differs from goal 3 only in that we replace the word "effectiveness" with "efficiency." It is stated as follows:

Analyze **code reading, functional testing, and structural testing**
for the purpose of **comparison**
with respect to their **efficiency at isolating faults**

from the point of view of the **researcher**
in the context of a **university lab course** using **small C programs**.

We define efficiency at isolating faults as the number of faults isolated, divided by the time required to do so. All caveats from goal 3 also apply here. Therefore, meeting goal 4 will require us to answer the following questions:

Q4.1: How many faults did the subject isolate per hour?

Q4.2: What effect did the subject's experience with the language or motivation for the experiment have on the number of faults isolated per hour?

These questions may be answered by reusing the data that was collected for the metrics of goal 3, plus one more:

M4.1: The amount of time the subject required to isolate faults.

Testable hypotheses are derived from the statement of goal 4, the questions, and the metrics as follows:

H4.1: Using the information resulting from applying one of the three defect-detection techniques, subjects isolate a different number of faults per hour. Rewritten as a null hypothesis: subjects isolate the same number of faults per hour.

H4.2: A subject's experience and motivation affect the number of faults s/he isolates per hour. Rewritten as a null hypothesis: measures of experience and motivation have no correlation with the data for fault-isolation rate.

Reuse guidelines. The goals presented in the previous section may be reused verbatim or be tailored to the specific needs of an experiment. To assist with the tailoring activity, Table 6 captures the traceability between the goals and the questions presented in the previous section, and Table 7 presents the traceability between the questions and the metrics. If the previously defined goals are reused verbatim, the supporting questions and metrics can also be reused verbatim. If the goals cannot be reused verbatim as stated, they may be tailored in many different ways. For example, if a different quality focus (one other than effectiveness or efficiency) is introduced, then all questions and supporting metrics pertaining to the quality focus must be revised. If the previously developed models (definitions of effectiveness or efficiency) are not suitable, then the existing goals must be tailored for the revised models.

3.4.2. Plan

This section discusses the design, techniques, objects, subjects, data-collection procedures, and data-analysis procedures from the Kamsties & Lott experiment.

Experimental design. We used a within-subjects design in both replications. The design as used in the second repetition[6] appears in Table 8. The four factors (i.e., independent

Table 6. Traceability among example goals and questions

Questions	Goal 1: Effec. fail.	Goal 2: Effic. fail.	Goal 3: Effec. faults	Goal 4: Effic. faults
Q1.1, percentage of failures	√			
Q1.2, exp./mot. and failures	√			
Q2.1, failures per hour		√		
Q2.2, exp./mot. and fail./hr.		√		
Q3.1, percentage of faults			√	
Q3.2, exp./mot. and faults			√	
Q4.1, faults per hour				√
Q4.2, exp./mot. and faults/hr.				√

Table 7. Traceability among example questions and metrics

Metrics	Q1.1	Q1.2	Q2.1	Q2.2	Q3.1	Q3.2	Q4.1	Q4.2
M1.1, total failure classes	√	√						
M1.2, experience (rel.)		√		√		√		√
M1.3, experience (abs.)		√		√		√		√
M1.4, motivation		√		√		√		√
M1.5, mastery of technique		√		√		√		√
M1.6, failures revealed	√	√	√	√				
M1.7, failures recorded	√	√	√	√				
M2.1, time to reveal failures			√	√				
M3.1, total faults					√	√		
M3.2, faults that caused failures					√	√	√	√
M3.3, faults isolated					√	√	√	√
M4.1, time to isolate faults							√	√

variables) are the techniques, programs, subjects, and order of applying the techniques. Random matches of techniques, programs, subjects, and order of applying the techniques were performed. Subjects were divided into six groups to manipulate the order in which each subject applied the techniques. However, the order in which subjects saw the three programs used in the experiment was fixed, not randomized. This prevented the subjects

Table 8. Kamsties & Lott's experimental design (Kamsties and Lott, 1995).

Program and day	Code Reading (CR)	Functional testing (FT)	Structural testing (ST)
Pgm. 1 (day 1)	groups 1, 2	groups 3, 4	groups 5, 6
Pgm. 2 (day 2)	groups 3, 5	groups 1, 6	groups 2, 4
Pgm. 3 (day 3)	groups 4, 6	groups 2, 5	groups 1, 3

from discussing the programs among themselves outside the experiment and thereby influencing the results. In the jargon of controlled experiments, the factors "program" and "day" are confounded with each other. For example, subjects in group 1 applied code reading to program 1 on day 1, applied functional testing to program 2 on day 2, and applied structural testing to program 3 on day 3.

This design permits the derivation of five null hypotheses that incorporate the hypotheses from Section 3.4.1. The following null hypotheses can be tested by applying an appropriate statistical analysis procedure:

D.1: The technique has no effect on the results (i.e., the techniques do not differ in their effectiveness and efficiency); this incorporates hypotheses H1.1, H2.1, H3.1, and H4.1.

D.2: The program and day have no effect on the results; i.e., no selection effects.

D.3: The order in which subjects apply the techniques has no effect on the results; i.e., no learning effects.

D.4: The subjects have no effect on the results; i.e., all subjects perform similarly (no selection effects).

D.5: Measures of the subject's experience and motivation have no correlation with their performance (incorporates hypotheses H1.2, H2.2, H3.2, and H4.2).

Reuse guidelines. This design strives for high internal validity and a high educational value for the subjects. Both of these issues lead toward a repeated-measures design, a design that demands considerable time from the participants. Also, if the experimenter can be confident that the subjects will not discuss the objects outside the experiment, the design can easily be reworked to separate the confounded factors 'program' and 'day.'

Defect-detection techniques. The following defect-detection techniques were compared.

Functional testing (FT). In step 1, subjects receive the specification but do not see the source code. They identify equivalence classes in the input data and construct test cases using the equivalence classes, paying special attention to boundary values. In step 2, the subjects execute their test cases on the computer. They are instructed *not* to generate additional test cases during step 2, but we can neither prevent nor measure this. Step 2 concludes when the subjects print out their results and log off the computer. In step 3, the

subjects use the specification to observe failures that were revealed in their output. After recording the failures, the subjects hand in a copy of their printed output, receive the printed source code in exchange, and begin step 4. In step 4, the extension to the Basili & Selby experiment, the subjects use the source code to isolate the faults that caused the observed failures. No special technique is specified for the fault-isolation activity. Step 4 concludes when the subjects hand in a list of observed failures and isolated faults.

Structural testing (ST). In step 1, subjects receive printed source code but do not see the specification. They try to construct test cases that will achieve 100% coverage of all branches, multiple conditions, loops, and relational operators as measured by the Generic Coverage Tool (Marick, 1994). For example, 100% coverage of a multiple condition using a single "logical and" operator means that all four combinations of true and false must be tested, and 100% coverage of a loop means that it must be executed zero, one, and many time(s). In step 2, the subjects use an instrumented version of the program to execute their test cases and view reports of attained coverage values. The subjects develop additional test cases until they reach 100% coverage, or believe that they cannot achieve better coverage due to various pathological cases. After executing the test cases, the subjects log off the computer, hand in a copy of their printed output, receive a copy of the specification in exchange, and begin step 3. In step 3, the subjects use the specification to observe failures in their output. The extension to the Basili & Selby experiment is step 4, in which the subjects isolate the faults that caused the observed failures. No special technique is specified for the fault-isolation activity. Step 4 concludes when the subjects hand in a list of observed failures and isolated faults.

Code reading (CR). In step 1, subjects receive printed source code but do not see the specification. They read the source code and write their own specification of the code based on the technique of reading by stepwise abstraction (Linger et al., 1979). Subjects identify prime subprograms, write a specification for the subprogram as formally as possible, group subprograms and their specifications together, and repeat the process until they have abstracted all of the source code. After writing their own specifications in this manner, the subjects allow their specifications to be photocopied, receive the official specification in exchange, and begin step 3. (To simplify comparisons with the other defect-detection techniques, code reading has no step 2.) In step 3, subjects compare the official specification with their own to observe inconsistencies between specified and expected program behavior (analog to failures in the other defect-detection techniques). The extension to the Basili & Selby experiment is step 4, when the subjects isolate the faults that led to the inconsistencies. No special technique is specified for the fault-isolation activity. Step 4 concludes when the subjects hand in a list of identified inconsistencies and isolated faults.

Reuse guidelines. The techniques as compared in this experiment are strict; e.g., the structural testers do not see the specification until after they have finished their test runs. This strictness may be unrealistic for an industrial software organization. In an industrial setting, the defect-detection techniques should be tailored to meet the specific needs of the organization or replaced with techniques that are practiced locally. Because dramatic changes in the defect-detection techniques may invalidate some of the assumptions and models behind the data collection procedures, the data collection forms must be rechecked if the techniques are changed.

Objects. Three C programs of 260, 279, and 282 source lines were used. Differences among the objects in terms of length and fault density were minimized. The programs had a total of 11, 14, and 11 faults, respectively, in the second repetition.

Reuse guidelines. Objects from the Kamsties & Lott experiment are coded in C and are available (see appendix). If the implementation language is not a match with the subject's experience, then code reuse may be neglected in favor of reusing only the specification.

However, the reuse of objects from any experiment depends highly on the goals behind the new experiment. Objects that are reused unchanged will result in results that can be easily compared with other repetitions of the original experiment. However, to attain maximum external validity with respect to other projects within a given organizational context, the experimenter should use code selected from systems developed within that context. The experimenter would also be well advised to consider the nature of the faults encountered in that context, and to select faults that are most similar to the organization's characteristic fault profile. See Laitenberger (1995) for an experimental design that addresses these issues.

Subjects. Some 27 university students participated in the first run of the experiment, and 23 students participated in the second. The two runs are examples of "in vitro" experiments.

Data collection and validation procedures. Subjects used data collection forms to record information about the time they required. The researchers analyzed the documents turned in by the subjects to collect additional data about visible failures. The computer counted the number of test runs.

Reuse guidelines. All data collection forms used in the Kamsties & Lott experiment are available (see appendix). These forms are highly specific to the goals, techniques, quality focuses, and the implementation language. Tailoring the goals and metrics to be collected will therefore always require tailoring the forms. For example, if assessing efficiency is not a goal, then no collection of time data is required.

Data analysis procedures. Based on the randomized approach for matching subjects, objects, and techniques, primarily parametric statistics (ANOVA) were used to test the null hypotheses. The ANOVA procedure was chosen because the design included randomization (thus satisfying a fundamental assumption), and all analyses involved more than two groups of data. A nonparametric correlation test was used to evaluate the correlations of subject factors with the results.

Reuse guidelines. If the design is reused unchanged, the analyses can proceed identically to those documented in Kamsties and Lott (1995). Dramatic changes to the design, especially if the randomization step cannot be performed, may prevent the use of ANOVA; nonparametric statistics such as the Kruskal-Wallis test can then be used.

3.4.3. Procedures

The Kamsties & Lott experiment consisted of three phases, namely training, experiment, and feedback.

Phase 1: Training. The training phase consisted of a lecture portion and a practical portion. Subjects first listened to several lectures explaining the techniques in the context of a software engineering lecture course. A training phase was then performed within a single week using three short C programs (44, 89, and 127 source lines). Each subject gained experience with all three defect-detection techniques by applying one technique to one of the short C programs. Subjects worked independently and chose their own times to perform the training exercises. The training period also allowed the subjects to become familiar with the working environment, the data-collection forms, etc.

Phase 2: Experiment. The experiment was run at fixed times on three separate days over the course of a week. Subjects who applied code reading worked in a conference room. Subjects who applied the two testing techniques worked in computer labs, separated as much as possible. All rooms were monitored.

A token prize was offered to the subjects who were the most effective and efficient with a given technique, thus a total of three prizes were available. No other incentives were offered.

Phase 3: Feedback. A feedback session was held about two weeks following the last experiment session. At that time, the prizes were awarded, the results of the preliminary analyses were presented, and the subjects had a chance to ask questions and give the experimenters their own feedback.

Reuse guidelines. The training activities as described above may need to be extended depending on the subjects. For maximum benefit, we recommend that the training exercises be structured similarly to the experiment so as to force all subjects to gain relevant experience. Also, if the work is not handed in or checked, students are likely to ignore it. If the design is not changed, procedures for the experiment should be reusable verbatim. The amount of time spent on preliminary analyses and on running feedback sessions will depend on the context. Professionals may be especially interested in the results of the experiment and so will demand more information from the experimenter than a student subject.

The experiment also makes nontrivial demands on a subject's time, which for students was possible. The total time commitment is about 2–3 hours per training exercise and 3–4 hours per experiment exercise, for a total of 15–21 hours. The time commitment could be reduced for experienced subjects by reducing or omitting the training phase. If the experimenter is only interested in how subjects reveal and record failures, the required time can be shortened by omitting the final fault-isolation step for each technique.

3.4.4. Results

This section gives an overview of the results from the second repetition of the Kamsties & Lott experiment. All analyses and interpretations are presented in detail in Kamsties and Lott (1995b).

No significant differences were observed among the techniques with respect to the percentage of failures observed (i.e., hypothesis D.1 could not be rejected with respect to effectiveness of observing failures). The code readers and the functional testers isolated ap-

proximately the same percentage of faults, with the structural testers performing less well. Stated differently, hypothesis D.1 was rejected with respect to effectiveness of isolating faults at significance level 0.02. The functional testers required the least amount of total time to reveal failures and isolate faults, with no significant differences among the code readers and structural testers. In other words, hypothesis D.1 was rejected with respect to efficiency of the functional testers at significance level 0.01. Combining these results, the functional testers observed failures most efficiently (significance level 0.00) and isolated faults most efficiently (significance level 0.01).

No significant differences among the programs were observed (hypothesis D.2 must be accepted). Some significant differences among the subjects and order were seen, suggesting the presence of both selection and learning effects (i.e., hypotheses D.3 and D.4 had to be rejected for some aspects). Finally, no significant correlations between measures of experience and the results were found (i.e., hypothesis D.5 must be accepted).

To summarize, the techniques did not differ significantly with respect to effectiveness in the hands of our subjects. However, the functional testers observed failures most rapidly, required much additional time to isolate faults, and yet were still overall most efficient at isolating faults. Code readers required much time to identify inconsistencies, but were then able to isolate faults in extremely little additional time. Similar to previous studies, large individual differences were observed, and only 50% of known defects were detected on average.

4. Conclusions and Future Work

This paper presented a characterization scheme for experiments that evaluate techniques for detecting all types of defects in source code. This scheme is intended to reduce the effort required to develop and conduct further instances of this class of experiment as well as ease the comparison of results from similar experiments. To ease the task of conducting an experiment further, a laboratory package has been made available with all materials necessary for repeating the experiment that was conducted at the University of Kaiserslautern (see appendix). We note that experiments for other technologies have been similarly packaged; see for example, the comparisons of different inspection techniques (Vander Wiel and Votta, 1993; Laitenberger, 1995; Porter, Votta, and Basili, 1995). Future directions include performing more repetitions in order to analyze different variation factors and to strengthen the credibility of the existing results.

Conducting experiments in software engineering has benefits for students, professionals, and the community. Students can experience first-hand the relative strengths and weaknesses of the techniques that are introduced in their courses. Professionals can gain confidence in new techniques before they apply the techniques in a revenue-producing project. The software engineering community can accumulate a body of knowledge regarding the utility of various techniques under varying project characteristics. We therefore recommend that repeatable experiments be adopted as a standard part of both software engineering education and technology transfer programs.

We strongly believe that researchers and practitioners need to view software engineering as an experimental discipline (Rombach et al., 1992). Techniques such as defect-detection

techniques that are fundamental to software development need to be understood thoroughly, and such an understanding can only be gained via experimentation. Other fundamental techniques that need to be thoroughly understood include various design methodologies and different styles of documentation. Many of the experiments necessary to build a body of knowledge about these techniques are too large for any single organization; they must be repeated in different contexts. The International Software Engineering Research Network (ISERN) has facilitated the repetition of experiments in different contexts. Researchers and practitioners from the following organizations are members of ISERN and take part in repeating experiments: University of Maryland at College Park, University of Kaiserslautern, VTT Electronics, University of New South Wales, Nara Institute of Science and Technology, University of Rome at Tor Vergata, University of Bari, Macquarie University, AT&T Bell Laboratories, Daimler-Benz Research Center, and Computer Research Institute of Montreal. Organizations interested in joining ISERN may contact any ISERN member or send electronic mail to isern@informatik.uni-kl.de.

Appendix: Laboratory Package

The materials needed to repeat the experiment as performed at the University of Kaiserslautern are available on the World-Wide Web. Please access these URLs:
`ftp://ftp.wkap.com/pub/emse/`
`http://www.cs.umd.edu/users/cml/`

5. Acknowledgments

We thank our colleagues in the Software Engineering Research Group at the University of Kaiserslautern and in the Fraunhofer Institute for their contributions, especially Alfred Bröckers for his guidance on data analysis procedures and Erik Kamsties for his help in conducting the experiment. The assistance provided by ISERN members in reviewing the Kamsties & Lott experiment was much appreciated. Finally, we deeply appreciate the comments on this paper from Lionel Briand, Shari Lawrence Pfleeger, Larry Votta, and the anonymous reviewers. Their criticisms and suggestions greatly improved the paper.

Notes

1. This work was conducted while the author was with the Department of Computer Science, University of Kaiserslautern, 67653 Kaiserslautern, Germany.
2. The term "blocked" has a highly specific meaning in experimental design, and differs from the usage in Table 2.
3. Strictly speaking, a mean cannot be computed using data from an ordinal scale. The values reported by Myers should only be understood to give a rough characterization of the group's experience.
4. See Campbell and Stanley (1966, p. 15) for a discussion of artifacts due to regression to the mean and other problems that arise when trying to correct for differences in ability by grouping subjects based on extreme values on pretests.
5. In the code reading technique, failures are not revealed; instead, inconsistencies between the user's specification and the official specification are identified. These inconsistencies are analogous to failures in the validation techniques. We use "failure" to keep the discussion brief.

6. This design differs from that used in the first repetition in that it controls for all 6 orders of applying the techniques. We did not consider order to be important when we first planned the experiment, so the first repetition only used groups 1, 4, and 5 as shown in Table 8.

References

Aronson, E., Brewer, M., and Carlsmith, J. M. 1985. Experimentation in social psychology. *Handbook of Social Psychology* (Lindzey, G., and Aronson, E., eds.) Vol. 1, 3rd ed. New York: Random House.

Basili, V. R., Caldiera, G., and Rombach, H. D. 1994. Goal question metric paradigm. *Encyclopedia of Software Engineering*, (Marciniak, J. J., ed.) vol. 1. John Wiley & Sons, pp. 528–532.

Basili, V. R., and Rombach, H. D. 1988. The TAME project: Towards improvement–oriented software environments. *IEEE Transactions on Software Engineering* SE-14(6): 758–773.

Basili, V. R., and Selby, R. W. 1987. Comparing the effectiveness of software testing techniques. *IEEE Transactions on Software Engineering* 13(12): 1278–1296.

Basili, V. R., Selby, R. W., and Hutchens, D. H. 1986. Experimentation in software engineering. *IEEE Transactions on Software Engineering* SE-12(7): 733–743.

Basili, V. R., and Weiss, D. M. 1984. A methodology for collecting valid software engineering data. *IEEE Transactions on Software Engineering* SE-10(6): 728–738.

Basili, V. R., and Weiss, D. M. 1985. Evaluating software development by analysis of changes: Some data from the software engineering laboratory. *IEEE Transactions on Software Engineering* 11(2): 157–168.

Box, G. E. P., Hunter, W. G., and Hunter, J. S. 1978. *Statistics for Experimenters*. New York: John Wiley & Sons.

Briand, L., El Emam, K., and Morasca, S. 1996. On the application of measurement theory in software engineering. *Journal of Empirical Software Engineering* 1(1): 61–88.

Briand, L., Basili, V. R., and Hetmanski, C. J. 1993. Developing interpretable models with optimized set reduction for identifying high-risk software components. *IEEE Transactions on Software Engineering* 19(11): 1028–1044.

Brooks, R. E. 1980. Studying programmer behavior experimentally: The problems of proper methodology. *Communications of the ACM* 23(4): 207–213.

Campbell, D. T., and Stanley, J. C. 1966. *Experimental and Quasi-Experimental Designs for Research*. Boston: Houghton Mifflin. ISBN 0-395-30787-2.

Cohen, J. 1988. *Statistical Power Analysis for the Behavioral Sciences*. Lawrence Erlbaum Associates.

Curtis, B. 1980. Measurement and experimentation in software engineering. *Proceedings of the IEEE* 68(9): 1144–1157.

Differding, C., Hoisl, B., and Lott, C. M. 1996. Technology package for the goal question metric paradigm. Technical Report 281-96, Department of Computer Science, University of Kaiserslautern, 67653 Kaiserslautern, Germany, April.

Fagan, M. E. 1976. Design and code inspections to reduce errors in program development. *IBM Systems Journal* 15(3): 219–248.

Fenton, N., Pfleeger, S. L., and Glass, R. L. 1994. Science and substance: A challenge to software engineering. *IEEE Software* 11(4): 86–95.

Gannon, J. D., Hamlet, R. B., and Mills, H. D. 1987. Theory of modules. *IEEE Transactions on Software Engineering* 13(7): 820–829.

Glass, R. L. 1995. Pilot studies: What, why and how. *The Software Practitioner*, January, 4–11.

Hetzel, W. C. 1976. *An Experimental Analysis of Program Verification Methods*. PhD thesis, University of North Carolina at Chapel Hill.

Hoare, C. A. R. 1969. An axiomatic basis for computer programming. *Communications of the ACM* 12(10): 576–580, 583.

Howden, W. E. 1978. An evaluation of the effectiveness of symbolic testing. *Software–Practice and Experience* 8(4): 381–398.

Howden, W. E. 1980. Functional program testing. *IEEE Transactions on Software Engineering* SE-6: 162–169.

Humphrey, W. H. 1995. *A Discipline for Software Engineering*. Addison-Wesley.

Institute of Electrical and Electronics Engineers. 1983. *Standard Glossary of Software Engineering Terminology*.

Judd, C. M., Smith, E. R., and Kidder, L. H. 1991. *Research Methods in Social Relations*. Holt, Rinehart and Winston, sixth edition.

Kamstics, E., and Lott, C. M. 1995. An empirical evaluation of three defect-detection techniques. *Proceedings*

of the Fifth European Software Engineering Conference (Schäfer, W., and Botella, P., eds.), Lecture Notes in Computer Science Nr. 989, Springer–Verlag, 362–383, September.

Kamsties, E., and Lott, C. M. 1995. An empirical evaluation of three defect-detection techniques. Technical Report ISERN 95–02, Department of Computer Science, University of Kaiserslautern, 67653 Kaiserslautern, Germany, May.

Laitenberger, O. 1995. Perspective-based reading technique, validation and research in future. Student project (Projektarbeit), Department of Computer Science, University of Kaiserslautern, 67653 Kaiserslautern, Germany.

Lee, A. S. 1989. A scientific methodology for MIS case studies. *MIS Quarterly*, 33–50, March.

Linger, R. C., Mills, H. D., and Witt, B. I. 1979. *Structured Programming: Theory and Practice*. Addison-Wesley Publishing Company.

Marick, B. 1994. *The Craft of Software Testing*. Prentice Hall.

Miller, J., Daly, J., Wood, M., Roper, M., and Brooks, A. 1995. Statistical power and its subcomponents—missing and misunderstood concepts in software engineering empirical research. Technical Report RR/95/192, Department of Computer Science, University of Strathclyde, Livingstone Tower, Richmond Street, Glasgow G1 1XH, Scotland.
http://www.cs.strath.ac.uk/CS/Research/EFOCS/Research-Reports/EFoCS-15-95.ps.Z.

Myers, G. J. 1978. A controlled experiment in program testing and code walkthroughs/inspections. *Communications of the ACM* 21(9): 760–768.

Myers, G. J. 1979. *The Art of Software Testing*. New York: John Wiley & Sons.

Parnas, D. L., and Weiss, D. M. 1985. Active design reviews: principles and practices. *Proceedings of the Eighth International Conference on Software Engineering*, IEEE Computer Society Press, 132–136.

Pfleeger, S. L. 1995a. Experimental design and analysis in software engineering, part 3: Types of experimental design. *ACM SIGSOFT Software Engineering Notes* 20(2): 14–16.

Pfleeger, S. L. 1995b. Experimental design and analysis in software engineering, part 4: Choosing an experimental design. *ACM SIGSOFT Software Engineering Notes* 20(3): 13–15.

Porter, A. A., Siy, H., and Votta, L. G. 1995. A survey of software inspections. Technical Report CS-TR-3552, UMIACS-TR-95-104, Department of Computer Science, University of Maryland, College Park, Maryland 20742, October.

Porter, A. A., Votta, L. G., and Basili, V. R. 1995. Comparing detection methods for software requirements inspections: A replicated experiment. *IEEE Transactions on Software Engineering*, 21(6): 563–575.

Preece, J., and Rombach, H. D. 1994. A taxonomy for combining software engineering and human-computer interaction measurement approaches: Towards a common framework. *International Journal of Human–Computer Studies* 41: 553–583.

Rombach, H. D., Basili, V. R., and Selby, R. W. (eds.) 1992. *Experimental Software Engineering Issues: A Critical Assessment and Future Directions*. Lecture Notes in Computer Science Nr. 706, Springer–Verlag.

Selby, R. W. 1985. Evaluations of Software Technologies: Testing, CLEANROOM, and Metrics. PhD thesis, Department of Computer Science, University of Maryland, College Park, MD 20742, May.

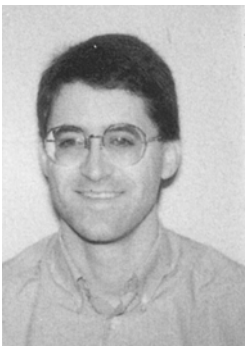
Selby, R. W., and Porter, A. A. 1988. Learning from examples: generation and evaluation of decision trees for software resource analysis. *IEEE Transactions on Software Engineering* 14(12): 1743–1757.

Spector, P. E. 1981. *Research Designs*. Sage University Paper series on Quantitative Applications in the Social Sciences, series no. 07-023. Beverly Hills: Sage Publications.

Vander Wiel, S. A., and Votta, L. G. 1993. Assessing software designs using capture-recapture methods. *IEEE Transactions on Software Engineering* 19(11): 1045–1054.

Votta, L. G., and Porter, A. 1995. Experimental software engineering: A report on the state of the art. *Proceedings of the Seventeenth International Conference on Software Engineering*, IEEE Computer Society Press, 277–279.

Zweben, S. H., Edwards, S. H., Weide, B. W., and Hollingsworth, J. E. 1995. The effects of layering and encapsulation on software development cost and quality. *IEEE Transactions on Software Engineering* 21(3): 200–208.



Christopher M. Lott received the B.S. degree in computer science from The Ohio State University, and the M.S. and Ph.D. degrees in computer science from the University of Maryland. Lott is a research scientist in the Applied Research Area of Bell Communications Research in Morristown, New Jersey, USA. Previously, he worked as a faculty research associate with the Department of Computer Science, University of Kaiserslautern, Germany, where he helped establish the Fraunhofer Institute for Experimental Software Engineering in Kaiserslautern. He is a member of the IEEE Computer Society.

Dr. H. Dieter Rombach is a Full Professor in the Fachbereich Informatik (i.e., Department of Computer Science) at the Universitaet Kaiserslautern, Germany. He holds a chair in software engineering research institute (SFB), and is director of the Fraunhofer Institute for Experimental Software Engineering (IESE) which aims at shortening the time needed for transfering research technologies into industrial practice. His research interests are in software methodologies, modeling and measurement of the software process and resulting products, software reuse, and distributed systems. Results are documented in more than 70 publications in international journals and conferences.