Experimental Designs: Testing a Debugging Oracle Assistant

SERC-TR-120-P

Eugene H. Spafford ***Chonchanok Viravan***

Software Engineering Research Center
Department of Computer Sciences
Purdue University
West Lafayette, IN 47907–1398
(317) 494–7825

{spaf, viravan}@cs.purdue.edu

December 18, 1992

Abstract

This paper documents the design of an experiment to test a debugging oracle assistant. A debugging oracle is responsible for judging correctness of program parts or program states. A programmer usually acts as a debugging oracle. The goal of a debugging oracle assistant is to improve the programmer's speed and accuracy.

Factors that complicate our design process include: (1) programmer variability, (2) interaction between programmers and programs, (3) interaction between programs and faults, (4) possible confounding experimental factors, (5) any learning effect from the assistance, (6) any learning effect from the program, and (7) the lack of experienced programmers for our experimental studies.

This paper explains the rationale behind our design. It explains why the above factors can make other choices, such as a *Latin square design*, produce misleading results. It questions the validity of the so-called *within-subjects factorial design* when the experimental factors exclude programmers. It explains the factors related to programs, programmers, and faults that we need to control. It also explains why we prefer to use analysis of covariance to reduce experimental error caused by programmer variability instead of grouping programmers by expertise.

The paper also covers types of analysis to (1) test our hypotheses, (2) verify assumptions behind the analysis of variance, (3) verify assumptions behind the analysis of covariance, and (4) estimate adequate sample size. Lastly, we define the inference space to which we can generalize the experimental results.

Contents

1 Introduction

Debugging, a process of locating and fixing program faults, is considered one of the most serious bottlenecks in software development today [Agr91]. *Program faults*, or *bugs*, are physical evidence of *errors*; errors are inappropriate actions made during software development that may ultimately cause software to fail [IEE83].

Most debugging tools, or *debuggers*, assist the debugging process by providing as much program information as possible. Unfortunately, only a small subset of such information has been experimentally evaluated. Even so, existing experimental results already contradict several researchers' expectations. Shneiderman et al. [SMMH] and Gilmore and Smith [GS84] were surprised that detailed flow charts do not significantly improve debugging speed and accuracy. Weiser and Lyle [WL91] were surprised that static slices[1] do not help programmers improve debugging performance.

Experimental evaluations of debugging assistants can improve the quality of current debuggers. They can help check if a debugger provides information that significantly improves a programmer's performance. They can later evaluate the quality of tools or techniques that provide such information.

This paper documents the design of an experiment to test a particular type of debugging assistant, called the *debugging oracle assistant*. This assistant is explained in Section 2. Design and analysis aspects of the experiment are discussed in Sections 3 and 4 respectively.

2 Debugging Oracle Assistant

An *oracle* is responsible for determining correctness. A *testing oracle* is responsible for determining correctness of a program. A *debugging oracle* is responsible for judging correctness of the program parts or program states. A *program part* can vary from a collection of (not necessarily contiguous) program statements, to an expression within one statement, to an operation or a variable. A *program state* is composed of a control flow location and values of all variables visible at such a location [Agr91].

A programmer usually assumes the role of a debugging oracle. To check if faults lie in a suspicious program part, the programmers usually fix and rerun the program until they obtain correct output [Gou75, Ves85, ADS91]. To check variable values or flow of control in a program state, programmers usually rely on their intuitions and deductive abilities. Unlike a debugging assistant that may identify suspicious program parts (called *fault localization techniques*) [Agr91, Wei84, CC87, Sha83, Pan91], no automated debugging oracle assistant is currently available. Because an automated oracle is far-fetched (if not impossible) without using details of formal program specification, most fault localization techniques assume that programmers know enough to judge a program properly.

The presumption that a programmer is an accurate or reliable oracle lacks supporting evidence. When a programmer judges suspected locations, he can still waste time investigating the correct locations, or ignoring the faulty one [Vir91]. To judge a suspected location, randomly fixing and rerunning statements is not very efficient; when judging variables' values, programmers might not be able to

[1] A static slice of a variable x at location l is composed of all statements that may influence x at l.

distinguish the right from the wrong values.

The *goal* of a debugging oracle assistant (DOA) is to improve programmers' oracle abilities by improving their speed and their accuracy. Given a set of one or more suspicious program parts, or *hypothesized fault locations*, and/or program states as an input, the DOA should help the programmer decide on the correctness status of such locations: the DOA is not required to identify the actual type of fault. If the hypothesized locations are not faulty, the DOA should help programmers reject them as soon as possible.

To improve debugging speed, a DOA should help programmers to (1) select a *prime suspect* among the given *suspects* and (2) rule out *incorrect suspects* from the *suspect list*. A *suspect* is a program part suspected to be faulty or suspected to cause the program to enter a faulty state. A *suspect list* initially consists of all parts identified in the given hypothesized fault locations.[2] A *prime suspect* is the suspect that ranks highest in its likelihood to be faulty [Vir91]. An *incorrect suspect* is a suspect that is not faulty.

To improve debugging accuracy, a DOA should help a programmer verify a program part in question. Given a statement as a prime suspect, for example, a programmer usually restores and inspects one of its program states. He must verify correctness of either the values of variables at that program state or the statement itself. This task is not trivial when the specification of the variables/functions is absent. A DOA can do the next best thing by providing information that can enhance programmer understanding of the program. Viravan refers to this information collectively as *decision support evidence* [Vir91]. Brook's *beacon* [Bro83], the information that suggest the presence of a particular data structure or operations in the program, is also potential decision support evidence.

The experimental design presented here can test whether a DOA helps the programmer improve his speed or accuracy when he acts as a debugging oracle. The specific nature of a DOA under test may require slight modification to the design proposed here.

3 Experimental Design

Our hypothesis is that *the presence of an appropriate DOA will help programmers decide on the correctness status of hypothesized fault locations or program states significantly faster or more accurately.*

The formal model of our design to evaluate this hypothesis is shown in Figure 1. We refer to Y_{ijklm} as a dependent variable. We refer to $A_i, B_j, R_{(ij)k}, P_l$ as factors or independent variables. Other terms in the model represent the interaction, or the combined effects, between the above factors. An *interaction* between two factors means that a change in response between levels of one factor is different for all levels of the other factor [Hic73]. For example, if assistance and fault interact, we would not be able to tell if the presence of the assistant always helps improve verifying time or if certain fault type always required longer time to detect. We may be able to tell, however, that the assistance provides significant help when the program has that fault type.

This design guards against several complicating factors. In the following sections, we will explain how our experiment is structured to mitigate these factors.

[2]These definitions are broader than, but related to, the ones originally proposed by Viravan in [Vir91].

$$
\begin{aligned}
Y_{ijklm} &= \mu + A_i + B_j + AB_{ij} + R_{(ij)k} + P_l \\
& AP_{il} + BP_{jl} + ABP_{ijl} + RP_{(ij)kl} + \epsilon_{(ijkl)m}
\end{aligned}
$$

$$
\begin{aligned}
Y_{ijklm} &= \mathit{accuracy\ or\ time} \\
\mu &= \mathit{average\ of}\ Y \\
A_i &= \mathit{Assistance},\ i = 1,2 \\
B_j &= \mathit{Fault},\ j = 1,2 \\
R_{(ij)k} &= \mathit{Programmers},\ k = 1,2,3 \\
P_l &= \mathit{Program},\ l = 1,2 \\
\epsilon_{(ijkl)m} &= \mathit{Error},\ m = 1
\end{aligned}
$$

Figure 1: Mathmatical model of our experimental design

We assume that at least one error-revealing test case has been found and general requirements of the program are available. The design proposed here should work with a set of hypothesized fault locations, whatever its source. Either a programmer or a fault localization technique such as the ones described in [ADS93, PS93] can define this set.

3.1 Independent Variables

Four *independent variables* or *factors* for our experimental design are assistance, program, fault, and programmers. We also call assistance, program, and fault *treatments* or *main effects* and programmers *subjects* or *experimental units*.

3.1.1 Assistance

The two levels of the assistance factor correspond to the absence and the presence of DOA. The presence of the assistance may be offered off-line or on-line.

Off-line assistance is suitable when the DOA under investigation is not yet implemented. To test the effectiveness of information that is potential decision support evidence, for example, we can give additional information to programmers who debug programs manually. Off-line assistance also prevents the programmer from obtaining other helpful information that might confound the experiment.

On-line assistance is suitable when the DOA under investigation has already been implemented or when it offers dynamic information. Testing the ease-of-use of any tool or technique that generates helpful information, for example, should be done on-line. Testing the helpfulness of a program trace, for example, is better done on-line because the programmers can pick and probe at the specific program states he wishes to observe.

3.1.2 Program

The levels of program factor correspond to different programs. To keep the programming language from becoming a factor, we pick the programming language C (because our debugger prototype, Spyder [ADS93, Agr91] , works with C programs). To tap into an extensive collection of programs for the experiment, we use *archie*. Archie is a system that allows one to rapidly locate various public domain programs stored on hundreds of sites across the Internet.[3]

To keep the program domain from becoming a factor, we pick programs in the general domain, or at least, from the same domain. To keep the program size from becoming a factor, we pick comparable length programs. The programs should contain statements within the same hundreds of lines. If possible, we pick programs whose vocabulary sizes (Halstead's total number of unique operators and operands [Hal79]) are approximately the same.

To keep programming style from becoming a factor, we adjust both programs to make their style consistent:

- Adjust the indentation level to four spaces. According to the study by Miara et al. [MMNS83], 2-4 spaces should optimize the aid of indentation to program comprehension.
- Adjust the comment style, perhaps by leaving only header comments in each procedure.
- Adjust the programs to have approximately the same percentage of comments over non-blank lines and same percentage of blank lines.
- Adjust the program to have the same proportion of mnemonic terms.

To keep the program control structure from becoming a factor, we pick programs that contain no goto's. The study by Weissman [Wei74] shows a higher comprehension score with structured programs.

To keep the procedure interconnections from becoming a factor, we pick programs whose procedures have the similar number of parameters, if possible. The study by Woodfield et al. [WDS81] suggests that module interconnection may play a more important role in ease of comprehension than the level of modularization.

To keep program reading strategies from becoming a factor, we rearrange the procedures to follow (approximately) the execution order of the programs. The study by Jefferies [Jef82] shows that experts understand programs better because they read them in execution order whereas novices read programs in linear order.

We will vary only the complexity among programs. This will be done by varying the types of data structure and the number of nesting levels. To avoid the confounding problem, it is important to vary only one factor. Though the above list is not necessarily an exhaustive list of all factors that affect comprehension or debugging ability, it at least suggests the factors researchers must consider.

After adjustment to control program characteristics, we compile the programs to ensure that they contain no syntactic errors. We test the programs thoroughly before seeding a new fault. We find an error-revealing test case that reveals erroneous output caused by the seeded fault, then use a fault localization technique to generate the hypothesized fault locations.

[3] To access *archie*, telnet to "quiche.cs.mcgill.ca" and use the login name "archie". No password is required.

3.1.3 Faults

The levels of fault factor correspond to fault categories from which the fault types are randomly selected. To expand our inference space, we choose two frequently occurring fault categories: logic faults and data definition/handling faults. Most error studies [Lip84, Bow80, PAFB82, MB77] rank logic faults first and data definition/handling faults second in frequency of occurrence. A few studies, like [WO84], rank data definition/handling first.

To keep *fault presence* from becoming a factor, we select a fault type from a list of either fault of commission or fault of omission, not both. According to [Gla81], fault of omission (the failure to do something) is harder to find than a fault of commission.

To keep *fault location* from becoming a factor, we plant the fault in procedures in the same nesting level. The study by Atwood and Ramsey [AR78] reports that an error both lower in the propositional hierarchy[4] and lower in the program structure is more difficult to detect and correct than a similar error higher in the program structure.

To plant the fault, we use randomly selected statements (in the same nested procedure level) that are pertinent to the selected fault type. For example, when the fault type is an incorrect boolean operator, the statements in the list include if-then-else, while-do, case, etc. Simple syntactic manipulation in randomly selected statements for fault seeding, according to Knight and Ammann [KA85], can yield the diversity of mean-time-to-failure (MTF) similar to that of unintended faults.

3.1.4 Programmers

Programmers, our experimental subjects, will be graduate students or seniors in the department of Computer Sciences at Purdue University. All must have at least three years of programming experience and know the programming language C.

3.1.5 Notes

Note that we leave out hypothesized fault location as a factor. If the hypothesized fault location is a factor, a response variable to measure the accuracy will take on either a 0 or 1 value. Zero may represent a wrong judgment and one may represent a correct judgment. Variation in this type of data is difficult to detect with a small sample size.

We define and fix the characteristics of a set of hypothesized fault locations instead. Such characteristics may be expressed in terms of either the fault localization technique that generates them or by the restrictions under which they are selected. For example, the restrictions may state that five of the 10 non-overlapping locations have no effect on the erroneous output and the other five do. Thus, our response variables measure programmers' performance with respect to a set of hypothesized fault locations.

[4]Proposition hierarchy refers to the embedding or nesting of clauses in a sentence structure.

3.2 Dependent Variables

Two *dependent variables* or *response variables* we want to measure are time and accuracy.

3.2.1 Accuracy

The objective is to measure the accuracy of the programmers in judging correctness of the given set of hypothesized fault locations and program states. An *answer* is composed of judgements for all hypothesized fault locationss and program states presented. It is up to the experimenter to count either one statement as a location or one hypothesis with multiple statements as a location. A *judgement* for each location is I, C, or D. I stands for incorrect. C stands for correct. D stands for do not know yet. We add the D to avoid coincidentally-correct judgement.

We envision two types of accuracy measurements:

1. *Accuracy of an answer (AC)*

   AC is the percentage of correct judgements of an answer. For example, suppose the fault is in the second hypothesized fault location out of the given five locations. The answer CICCC has 100% accuracy. IICCC has 80% accuracy. CDDII has 20% accuracy. DDIDD has 0% accuracy.

2. *Average accuracy (AAC)*

   AAC is the sum of accuracy of all answers by a programmer divided by the number of answers. A programmer can get 100% AAC only when he gets his first answer correct. If the programmer fixes and reruns some statements three times before revising his answer from CDDII to CDDCI, we will count it as four answers: three CDDII and one CDDCI. Suppose he follows that by two CDDCC, one CDCCC, and one CICCC. His AAC is $(3*.20+.4+2*.6+.8+1)/8$ or 50%.

AAC should reflect the programmer's performance variability better than AC. If we only use AC, both programmers in the example above will get 100%. Lower AAC also suggests the amount of guessing involved.

3.2.2 Time

The time each programmer takes to judge the given locations correctly may either be an absolute measure or a relative measure. We envision three time measurements.

1. *Verification time*

   This is the absolute time measurement, measured in terms of minutes. This figure must exclude the *noise*, that is, the time that has nothing to do with the verification. Suppose the experiment is off-line and we designate some people to verify the correctness of the programmers' answer. If we do not designate enough people, some programmers may have to wait a while before they can check their answers. This waiting time must be subtracted from the total time.

2. *Number of times program parts are fixed and rerun*

This is a relative time measurement appropriate for when we monitor the programmers on-line.

3. *Number of program parts fixed and rerun*

   This is a relative time measurement appropriate for when we monitor the programmers off-line.

If a programmer fails to find the right answer at the end, then his time measurement is infinite. Data transformation is required to analyze data with infinite value. One possibility is to analyze the reciprocal of the time. The reciprocal value of infinity is zero. Vessey uses this approach to solve a similar problem in [Ves85].

3.3 Covariance

A *covariate* or a *concomitant variable* X is a variable that varies with the response variable Y [Hic73]. This supplementary measurement should, to some extent, predict the performance of the experimental units (e.g., programmers) or remove some biases that arise from uncontrolled variables in the experiment [Coc57]. *Analysis of covariance* can adjust the observed response variable Y for the effect of the covariate X [Mon91]. Without the adjustment, a covariate could inflate the experimental error term and make true differences in response caused by treatments harder to detect [Mon91]. We may find the differences among programmers greater than the effects of treatments [Ves85, Cur80, MS81]. We may need hundreds of programmers to see the statistical significance of our treatments.

We want to find a covariate X that can reduce the experimental error caused by programmer variability. To qualify as a covariate X, a measurement variable M must meet the following assumptions [Hic73].

- Y correlates linearly with M (e.g., regression model is linear).
- M can predict Y to a certain extent (e.g., regression coefficient is not zero).
- M is not affected by treatments given to the groups (e.g., the regression coefficients within each group are homogeneous).

Section 4.3 covers the test for these assumptions to ensure the validity of covariance analysis. If the treatment affects M, for example, the covariance analysis will remove some (or much) of the effect that the treatments had on the response variable and badly distort the analysis [NW74].

Candidates for X that might reduce programmer variability include:

1. Biographical data
2. Familiarity with the programming language C
3. Understanding of a program domain
4. Understanding of causal knowledge in a program
5. Accuracy and time to judge hypothesized fault locations with no assistance

Two promising biographical factors are experience (e.g., number of computer science classes) and aptitude (e.g., GPA). Moher and Schneider [MS81] found that both factors explain about 40% of the variations in program comprehension scores for student programmers (including novices).

Both programming language familiarity and the understanding of the domain are promising covariate candidates. The study by Pennington [Pen87] suggests that programmers need both forms of knowledge to achieve high program comprehension.

Causal knowledge is also promising. Causal knowledge is the understanding of causal connections in the program as the program executes. According to Littman et al. [LPLS86] programmers need it to modify programs correctly. Both program comprehension and modification are tasks related to debugging. Moher and Schneider found that a measurement of programmers performing one task correlates with a measurement of another task better than any biographical variables [MS81].

Accuracy and time to judge hypothesized fault locations with no assistance are promising covariate candidates because we will get the same unit of estimate as those from the experiment. Though the programmers carry out the same tasks, the measurements are not 100% guaranteed to work [PIS80]. The characteristics of the program and the fault we choose can still affect the programmer's performance.

Other possible covariates includes time measurement and software complexity metrics. Time measurement may reduce biases in the accuracy measurement. A software metric may adjust for variability in program complexity. According to the study by Curtis et al. [CSM79], both Halstead's E and MacCabe's v(G) are good predictors of time to find and fix bugs. A program's complexity covariate is not needed when each programmer sees all programs during the experiment. Such is the case in our proposed experimental design model.

Note that the formal model in Figure 1 does not include any covariate. If we find an appropriate covariate, we will add a term to the model to represent it.

3.4 Design Models

3.4.1 The proposed design model

Assistance	1						2					
Fault	1			2			1			2		
Programmers	1	2	3	4	5	6	7	8	9	10	11	12
Program 1												
Program 2												

Figure 2: Design Layout

The design layout of the mathmatical model in Figure 1 is shown in Figure 2. This is called a repeated measure design. A *repeated measure design* is a design that takes several observations from the same subject under different treatments [OM88]. Two kinds of comparisons in a repeated measure design are *between-subjects* and *within-subjects*. Between-subjects comparison is made when subjects are nested under the treatment levels. Thus, we compare assistance and fault between-subjects. Within-subject comparison is made when subjects are crossed (repeatedly measured) with treatment levels. Thus, we compare programs within-subjects.

Up to five things can be randomized in this design. First, three programmers will be randomly allocated for each fault and assistance treatment combination. Second, each programmer will see the two programs in random order. Third, the fault type will be randomly selected for each fault category. Fourth, the fault location will be randomly selected within a fixed nesting procedure level. Fifth, if hypothesized fault locations are not selected based on a particular fault localization technique, these locations can be randomly selected under restriction.

This design is presented in a minimal form. The increase in levels of any independent variable can extend the design. This is a conservative design, in the sense that we presume that several complicating factors can arise. These include (1) programmer variability, (2) interaction between programmer and program effects, (3) interaction between program and fault effects, (4) possible confounding experimental factors, (5) the learning effect from the assistance, (6) the learning effect from the programs, and (7) the difficulty in finding experienced programmers for the experiments. Section 3.4.2 explains why some of these matter. Section 3.4.3 shows some designs can produce misleading result because of them.

3.4.2 Rationale

We will address some obvious questions about our design choice.

1. Why use a repeated measure design?

A repeated measure design reduces the experimental error caused by programmer variability, making it possible to use fewer subjects to gather as much information as with a larger design [OM88]. The word *error* is not synonymous with "mistakes," but includes all types of extraneous variations. Such variations tend to mask the effect of the treatments [CC57].

A major problem in programmer-related experiments is that the effect of programmer variability is frequently greater than the effects of treatments [Ves85, Cur80, MS81]. The study by Sackman et al. [SEG68] points out a 28:1 performance difference among the professional programmers employed in the same position in the same firm. Dickey [Dic81] later points out that this figure is misleading because it encompasses all differences between (1) time sharing and batch systems, (2) JTS and machine language programmers, and (3) prior experience with time-sharing systems. After accounting for these differences, only a range of 5:1 can be attributed to programmer variability [Dic81].

The nested factorial design shown in Figure 3 can have a large error term that reflects programmer variability. Our repeated measure design should have a smaller error term because part of it now reflects variation within the same programmer.[5] Another problem is that experienced programmers willing to participate in an experiment are hard to find. This makes our design more attractive than the one in Figure 3, as it does not require as many programmers.

A repeated measure design does have its limitations, however. There should be

[5] The error of a repeated measure design is actually a combination of two terms, $R_{(ij)k}$ and $RP_{(ij)kl}$. This leaves $\epsilon_{(ijkl)m}$ equal to zero. $R_{(ij)k}$ is the between-subject error term. $RP_{(ij)kl}$ is the within-subject error term.

no carry-over effect from learning, training, ordering, or fatigue when a programmer is measured more than once [OM88]. We show the impact of these limitations later in this section.

Assistance	1								2							
Program	1				2				1				2			
Fault	1		2		1		2		1		2		1		2	
Programmers	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16

Figure 3: A Nested Factorial design

2. Why use more than one program?

More than one program is needed to define and expand our inference space. Brooks [Bro80] complains that "*the lack of knowledge about the description and specification of differences among subjects and programs has a damaging effect on the generalizability of the experimental finding. If there is no effective, replicable way to describe the characteristics of subjects or programs used in an experiment, then there is no way to determine whether the results of the experiment apply to real world situations.*"

With one program, we cannot readily generalize our findings. We do not know what aspects of that program influence our experimental results. With two or more programs, we can control the similarities and differences among them. We can define our inference space with respect to the style, the size, the domain, and the complexity of the programs that we control. Surprisingly, we have not yet found any similar experiments that specify programs in the level of detail we do in Section 3.1.

3. Why use more than one fault?

By selecting fault types randomly from two frequently occurring fault categories (logic and data definition/handling), we can generalize our findings to other fault types in both categories.

4. Why cross program and fault?

A design with multiple programs, each of which contains different faults, confounds the effects of program and fault. A *confounding problem* occurs when we cannot separate the effects of two (or more) factors [AM74]. When used wisely, confounding helps create special design arrangements that require fewer programmers or divide the experiment to be carried out in different time periods [Hic73]. Such design generally confounds two-way or higher interactions between factors. A design that confounds main effects, as in Figure 4, is a bad design.

To avoid this problem, we cross the program factor and fault factor. A cross produces multiple versions of each program, one for each fault type.

5. Why does each programmer not see all program versions?

A learning effect from seeing the same program twice is the problem. Vessey found that programmers cut their debugging time in half (despite the fault) when they see the same program again [Ves85].

By preventing advance study of the programs by experimental subjects, we create a more artificial environment. In real debugging scenarios, the users often are familiar with the software; in some cases, users are debugging software they may have maintained for decades. We can simulate this by letting the programmers fully learn about the programs before we begin our experiments, perhaps by letting them debug the programs a few times. Unfortunately, to test every possible DOA this way is too expensive. Once we find a promising DOA that deserve further investigation, then we can redesign the experiment to test its effects over a period of time.

6. Why include programmers as a factor?

A repeated measure design requires the subjects to become a factor. In our case, if the subject (programmer) factor is left out, the programmer effect and the interaction of programmers and other factors will confound with each other. The experimental results may become misleading.

This fact makes us question the validity of a popular design in empirical studies of programmers called *within-subjects factorial design*. This design allows each subject to see each level of each experimentally manipulated variable *once and only once*[6] [SCML79]. For example, Sheppard et al. generate 81 treatment combinations from their within-subject, 3^4 design. Each programmer sees three different treatment combinations. Twenty-seven programmers exhaust all 81 conditions. Nine other programmers repeat the tasks of nine previous participants [SCML79].

Though this design repeatedly measured programmers three times, it did not include programmers as a factor. As a result, Sheppard et al. admit that they cannot separate the variance attributed to individual programs from those attributed to programmers.

Despite this problem, a within-subject factorial design is still popularly used today. Many papers describe their models by words, as oppose to a mathematical model or a layout as shown in Section 2. Word description does not always make it clear whether the programmer factor is included. A within-subject factorial design also has additional problems when the programmer factor is included (see page 16).

6. Why randomize things in the design?

Randomization is a mean of ensuring that a treatment will not be continually favored or handicapped in successive replications by some extraneous source of variation, known or unknown [CC57]. Random allocation of programmers helps average out the effect of inhomogeneous experimental units (programmers). Random order of programs assigned to each programmer guards against systematic biases. In repeated measure design, the systematic biases may come from the learning effect and the fatigue effect. The former makes the second measure better than the first; the latter does the opposite [CC57]. Fault type selection and fault locations are also randomly selected to avoid introducing bias.

7. Why not group programmers by expertise?

Grouping programmers by expertise is another means to control programmer variability [Ves85]. We opt not to for two reasons. First, we are not interested in novices because their performance does not always scale up [Cur80, Jef82]. Second, we do not have a cost-effective, accurate, and reliable method to measure expertise. Vessey's *ex-post* classification [Ves85] is promising but costly because

[6] This meaning is not consistent with the meaning of within-subject comparison for a repeated measure design. See Section 3.4.1.

it requires analysis of verbal protocol during a debugging process. Biographical data, like years of experience, do not always predict programmer performances. Several studies [Ves85, LPLS86, SCML79] report that years of experience (beyond three years [SCML79]) do not correlate to the programmers' ability to understand, modify, or debug programs.

We instead post a restriction that our student programmers have three or more years of experience. Soloway and Ehrlich call them *advanced student programmers* [SE84]. The use of covariance analysis will provide the "handicap" for the programmers. No further grouping is required [Hic73].

8. Why use at least twelve programmers?

The answer lies in the *degree of freedom* of the estimate of error. A *degree of freedom (d.f.)* associated with any component is the number of independent parameters required to describe that component in the model [CC57]. When the number of degrees of freedom for error becomes smaller, the probability of obtaining a significant result decreases [CC57].

In our design, if we use eight programmers (for a sample size of two for each assistance and fault combination), the between-subjects error degree of freedom is four.[7] The test of assumptions in analysis of covariance can reduce this degree of freedom further. To remedy this problem, we increase the sample size to three. With twelve programmers, the error degree of freedom before adjusting for covariates is eight.

We are not claiming that twelve programmers are adequate. By adequate, we mean β, the chance of not finding significant difference caused by experimental treatment when it exists [MS81], is sufficiently low (e.g., like 5%). Twelve is what we need for an initial pilot study. Measurements from an initial pilot study can be used to estimate the actual number of programmers needed. Section 4 explains this in more detail.

3.4.3 Possible problematic designs

One obvious design choice we decided against is to repeatedly measure programmers under both levels of assistance. Such choice leads to possible problematic designs. To make the problems apparent, this section describes some of them.

The model in Figure 4 is the most intuitive, but incorrect. Every programmer is measured twice. The first time they evaluate Program 1 with Fault 1 with no assistance. The second time they evaluate the program 2 with Fault 2 with assistance. This design confounds all the main effects. The observed improvement may be caused by either the presence of the assistance, Program 2, Fault 2, or familiarity with the experimental procedure.

The model in Figure 5 avoids the confounding problem by crossing assistance with program factors and nested programmers under each program and fault combination. Each programmer will evaluate one buggy program. The assistant is provided for half of the hypothesized locations. *This design would have been a better design than the one in Figure 2 if no learning effect from the given assistance is guaranteed.* An example of such a learning effect is when the given assistance improves the understanding of one hypothesized location, it may indirectly improve the understanding of other locations as well.

[7] The computation of degree of freedom is explained in Montgomery [Mon91].

Programmers	1	2	3	4	5	6	7	8	9	10	11	12
Assistance 1	P1-B1											
Assistance 2	P2-B2											

Pi = Program i

Bj = Fault j

Figure 4: A design with all main effects confounded

Program	1						2					
Fault	1			2			1			2		
Programmers	1	2	3	4	5	6	7	8	9	10	11	12
Assistance 1												
Assistance 2												

Figure 5: A design that has learning effect

Program	1						2					
Programmers	1	2	3	4	5	6	7	8	9	10	11	12
Assistance 1	B1						B2					
Assistance 2	B2						B1					

Bj = Fault j

Figure 6: A 2x2 Latin square design with Assistance x Program

The model in Figure 6 avoids the learning effect caused by mixing the locations with and without assistance together in the same program. This is done by letting each programmer see both buggy versions of the same program, the first one without assistance, and the second one with assistance. Unfortunately, this design is not applicable because it has a potential learning effect from seeing the same program twice.

Fault	1	2
Programmers	1 2 3 4 5 6	7 8 9 10 11 12
Assistance 1	P1	P2
Assistance 2	P2	P1

Pi = Program i

Figure 7: A 2x2 Latin square design with Assistance x Fault

The model in Figure 7 avoids a learning effect from both the assistance and the program. Because a fault of the same type still takes different forms and locations in different programs, we believe the learning effect is negligible. The first group of programmers evaluates Program 1 containing Fault 1 with no assistance first, then evaluates Program 2 containing Fault 1 with assistance. The program order is reversed for the second group with Fault 2. No main effect is confounded. This is called a 2x2 *Latin Square* design.[8]

A Latin square design is a design in which each level of each factor is combined once and only once with each level of two other factors [Hic73]. According to Neter and Wasserman [NW74], it has three advantages. First, its use of two blocking variables (e.g., assistance and fault here) reduces experimental errors. Second, it minimizes the experimental units required. Third, it allows repeated measure design to take the order effect of treatments into account.

Besides the need for equal levels for all three effects (two levels in our case), another disadvantage of a Latin square is the assumption of *no* interaction between any of the three main effects [NW74, Mon91]. This assumption is frequently overlooked by researchers [OM88]. Several programmer-related studies (e.g., [GO86, Bae88]) use Latin square without mentioning that they verify this assumption.

Unfortunately, programs and faults do have a history of significant interaction. Studies by Sheppard et al. [SCML79] and Atwood and Ramsey [AR78] observe significant interaction between fault and program. If we risk using this design and find that interaction exists, *we cannot draw any conclusion* from the study.

The model in Figure 8 shows what happens to the layout of the so-called *within-subject* 2^3 *factorial design* for our problem when the programmer factor is included. We discussed within-subject factorial design earlier on page 13. The 2^3 represents Assistance x Program x Fault. When programmers become a factor here, fault order

[8] The design in Figure 6 is also a Latin square design.

Fault Order	1		2	
Groups	1	2	3	4
Programmers	1 2 3	4 5 6	7 8 9	10 11 12
Assistance 1	P1-B1	P2-B1	P1-B2	P2-B2
Assistance 2	P2-B2	P1-B2	P2-B1	P1-B1

Groups	1	2
Programmers	1 2 3	4 5 6
Assistance 1	P1	P2
Assistance 2	P2	P1

Fault Order	1	2
Assistance 1	B1	B2
Assistance 2	B2	B1

Groups	3	4
Programmers	7 8 9	10 11 12
Assistance 1	P1	P2
Assistance 2	P2	P1

Pi = Program i
Bj = Fault j

Figure 8: Latin squares layout of Within-subject factorial design

and groups of programmers become additional factors. Each group of programmers sees each fault and each program once. Every order of programs and faults is considered.

A word description of this design hides one important fact. A closer look at its layout reveals three Latin squares, one superimposed on the other two. On the top layer, we have a 2x2 Latin square with fault order, assistance, and faults as the main effects. On the next layer, we have two 2x2 Latin squares with program, group of programmers, and assistance as the three main effects. Thus, this design cannot tolerate interactions among fault order, assistance and faults and interactions among program, group of programmers, and assistance.

The first session will test the programmers' abilities to understand the semantics of C programs. The programmers will also fill out a questionaire that inquires about their academic background (e.g., number of computer science classes, number of programming languages known) and academic performance (GPA).

The second session will measure the time and accuracy for judging the given hypothesized locations. The programs used should contain faults from both logic and data definition/handling categories. After we collect each program, we will give the programmers a questionnaire. It will ask about the program functionality and the causal relations among program components.

3.4.4 The experiment

We set up the experimental conditions to represent the part of a debugging phase after the programmer already realizes the presence of faults (via code walkthrough or testing). We divide the experiment into two phases for each program with a break in between.

The actual procedure depends greatly on the nature of the DOA under test. However, one should allow for time for the programmers to get familiar with the program. Also, the programmers should not know about the number of faults in the program. Suppose we give them five hypothesized fault locations. If we tell them that the program contains only one bug, each programmer has 20% (= $1/5$) chance to get 100% AAC by guessing. If we do not, each has a 3% (= $1/2^5$) chance.

Note that beside accuracy and time, we can also ask for other "free" information. Information such as the confidence level of their answers and their comments may provide insightful hints that may help us interpret experimental results.

4 Analysis

We need to analyze the data to (1) test our hypothesis, (2) verify assumptions behind the analysis of variance, (3) verify assumptions behind the analysis of covariance, and (4) estimate adequate sample size. To generalize our findings, we must define the inference space also.

4.1 Test Hypotheses

Recall that our hypothesis is "*the presence of an appropriate DOA will help programmers decide on the correctness status of hypothesized fault locations or pro-*

gram states significantly faster or more accurately." To test this hypothesis, the best method is the analysis of variance (ANOVA).

ANOVA is a method of estimating how much of the total variation in a set of data can be attributed to certain assignable causes of variation (independent variables) [Har82]. It uses the F-test to test the following hypotheses:

H_0: Factor *f* causes no variation in the programmers' performance.

H_a: Factor *f* causes the variation.

ANOVA can simultaneously test the significance of terms in the model in Figure 1: (1) the assistant, (2) programs, (3) faults, (4) interactions among assistance, fault, and program, and (5) and interaction of program and programmers within assistance and fault.

4.2 Verify ANOVA assumptions

According to Hicks [Hic73], three assumptions for ANOVA should be checked:

1. The process is controlled, that is, it is repeatable.
2. The population distribution being sampled is normal.
3. The error variances are homogeneous.

Our design already meets assumption 1. The experiment is repeatable because we can let more programmers work on the same set of programs. To check for assumption (2), a normality plot and normality test can be used. To check for assumption (3), Bartlett's test of homogeneity of variances can be used. More discussion of these tests can be found in [Mon91, NW74]. If either one of the last two assumptions is not true originally, suitable transformations on the response variable Y may make it true [Hic73]. When the normality assumption is unjustified, alternatives include the Kruskal-Wallis test and the Chi-square test [Mon91].

4.3 Verify ANCOVA assumptions

Because ANCOVA is an extension of ANOVA, it inherits all ANOVA assumptions. Presuming that they are met, we can test if a variable M is a suitable covariate. To verify the assumption in Section 3.3, we can add into the model M and a term representing the interaction between M and treatments to the group. The significant effect of M on a response variable Y indicates that M correlated to a response variable Y linearly with non-zero slope. The nonsignificant effect of the interaction between M and the treatment assures that the treatment does not affect M.

If we find a good covariate, ANCOVA should make the treatment effects more apparent than ANOVA.

4.4 Estimate sample size

We still have to check if the sample size of twelve programmers is adequate. Based on (1) the expected difference in magnitude of the effects of experimental treatment,

(2) an estimate of variance, and (3) the size of risks we are willing to take[9] [MS81, Hic73], we can find out if β, the chance of *not* finding a significant difference caused by experimental treatment when it exists (or the chance of type II error) [MS81], is too high. If so, we need to use either a *power table*, a *power curve*, or *Operating Characteristics curve* [Hic73, Mon91] to estimate the required sample size. We need to replicate our experiment accordingly. We cannot readily estimate the sample size beforehand because we need the estimates of (1) and (2) from this initial pilot study.

4.5 Define inference space

From the result of our analysis, we can extend our findings to:

1. Programs that have the same size, domain, style, structure, and complexity as those that we use in the experiment.
2. Logic and data definition/handling faults.
3. Student programmers with at least three years of experience.
4. A set of hypothesized fault locations and the method that generates them.

5 Conclusions

The design of a seemingly simple experiment in this paper is complicated by several factors. These include (1) programmer variability, (2) interaction between programmer and program effects, (3) interaction between program and fault effects, (4) possible confounding experimental factors, (5) the learning effect from the assistance, (6) the learning effect from the programs, and (7) the difficulty in finding experienced programmers for the experiments.

To deal with factor (1), recent empirical studies of programmers found in [SI86, OSS87, KBMR91] still focus on grouping programmers by expertise rather than using the covariance alternative. Several studies overlook factors (2) and (3) when they use their within-subject design or a Latin square design. Combinations of factors (2), (3), (4) and (5) limit us to type of design applicable for our problem. Factor (7) leads many studies to focus on novice programmers abundant in the university. Many researchers do experiments with few programmers (like eight) without checking if the sample size is adequate.

Though our design is not fancy, it guards against all seven complicating factors. A literature review of experimental studies of programmers leads us to identify aspects of programs, faults, and programmers that need to be controlled. We point out the assumptions that must be verified before drawing any conclusion. Our inference space may be limited, but it is more realistic than infering the result to all programs.

Once our studies under this design suggest a DOA that deserves further investigation, a good follow-up study is to test its effectiveness over a period of time. Because this requires the programmers to see the same program repeatedly, we have to create a new design to handle the learning factor. This can be done by either taking the learning factor into account or by letting programmers become familiar with the programs prior to the experiments.

[9] For instance, α = the chance of falsely rejecting the null hypothesis = the chance of type I error = 5%.

According to Mitchell and Welty [MW88], computer science ranks almost the lowest among scientific disciplines in publishing experimental studies. This is not because this field does not need them. Mitchell and Welty suspect that many computer science researchers do not really know how to do experiments nor are they willing to spend time on it. We hope that this document of a design and analysis process can guide and encourage more experimental studies in computer science.

Acknowledgments

Our thanks are to Dr. Thomas Kuczek for his advice on experimental design and his review of an earlier draft of this paper; to Dan Flick and Sean Tang for answering several statistical questions; to Dr. H. E. Dunsmore for his recommendations of several pertinent references; to Viriya Upatising for introducing one of us to *archie*, and to McGill University for making archie available.

References

[ADS91] Hiralal Agrawal, Richard A. DeMillo, and Eugene H. Spafford. An Execution Backtracking Approach to Program Debugging. *IEEE Software*, May 1991.

[ADS93] Hiralal Agrawal, Richard A. DeMillo, and Eugene H. Spafford. Debugging with Dynamic Slicing and Backtracking. *Software Practice and Experience*, 1993. (To appear).

[Agr91] Hiralal Agrawal. *Towards Automatic Debugging of Computer Program*. PhD thesis, Purdue University, West Lafayette, IN, 1991.

[AM74] Virgil L. Anderson and Robert A. McLean. *Design of Experiments: A Realistic Approach*. Marcel Dekker, Inc., New York, 1974.

[AR78] M. E. Atwood and H. R. Ramsey. Cognitive Structures in the Comprehension and Memory of Computer Programs: An Investigation of Computer Program Debugging. Technical report, Army Research Institute for the Behavioral and Social Sciences, Alexandria, VA, 1978.

[Bae88] Ronald Baecker. Enhancing Program Readability and Comprehensibility with Tools for Program Visualization. In *Proceedings 10th International Conference on Software Engineering*, April 1988.

[Bow80] J. B. Bowen. Standard error classification to support software reliability assessment. In *AFIPS National Computer Conference Proceedings*, volume 49, pages 607 – 705, May 1980.

[Bro80] Ruven E. Brooks. Studying Programmer Behavior Experimentally: The Problems of Proper Methodology. *Communications of the ACM*, 23(4):207–213, April 1980.

[Bro83] R. Brooks. Towards a theory of the comprehension of computer programs. *International Journal of Man-Machines Studies*, 18:543 – 554, 1983.

[CC57] William G. Cochran and Gertrude M. Cox. *Experimental Designs*. John Wiley and Sons, Inc., New York, 1957.

[CC87] James S. Collofello and Larry Cousins. Toward automatic software fault localization through decision-to-decision path analysis. In *Proceedings of AFIP 1987 National Computer Conference*, pages 539–544, 1987.

[Coc57] William G. Cochran. Analysis of Covariance: Its Nature and Uses. *Biometrics*, 13(3):261 – 281, September 1957.

[CSM79] Bill Curtis, Sylvia B. Sheppard, and Phil Milliman. Third time charm: Stronger prediction of programmer performance by software complexity metrics. In *Proceedings 4th International Conference on Software Engineering*, pages 356–360, 1979.

[Cur80] Bill Curtis. Measurement and Experimentation in Software Engineering. In *Proceedings of the IEEE*, volume 68, pages 1144–1157, September 1980.

[Dic81] Thomas E. Dickey. Programmer Variability. *Proceedings of the IEEE*, 69(7):844, July 1981.

[Gla81] Robert L. Glass. Persistent Software Errors. *IEEE Transactions on Software Engineering*, SE-7(2):162–168, March 1981.

[GO86] Leo Gugerty and Gary M. Olson. *Comprehension Differences in Debugging by Skilled and Novice Programmers*, chapter 2, pages 13 – 27. Empirical Studies of Programmers. Ablex Publishing Corporation, Norwood, New Jersey, 1986.

[Gou75] J. D. Gould. Some psychological evidence on how people debug computer program. *International Journal of Man-Machines Studies*, 7:151 – 182, March 1975.

[GS84] D. J. Gilmore and H. T. Smith. An investigation of the utility of flowcharts during computer program debugging. *International Journal of Man-Machines Studies*, 20:357–372, 1984.

[Hal79] Maurice H. Halstead. *Elements of Software Science.* Operating and Programming Systems Series. Elsevier North Holland, Inc., New York, 1979.

[Har82] Donald L. Harnett. *Statistical Methods.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1982.

[Hic73] Charles Robert Hicks. *Fundamental Concepts in the Design of Experiments.* New York, Holt, Rinehart and Windston, 1973.

[IEE83] IEEE Standard Glossary of Software Engineering Terminology, 1983. IEEE Std. 729-1983.

[Jef82] R. A. Jefferies. Comparison of Debugging Behavior of Novice and Expert programmers. Technical report, Department of Psychology, Carnegie-Mellon University, Pittsburgh, PA, 1982.

[KA85] John C. Knight and Paul E. Ammann. An Experimental Evaluation of Simple Methods for Seeding Program Errors. In *Proceedings 8th International Conference on Software Engineering*, pages 337–342, August 1985.

[KBMR91] Jurgen Koenemann-Belliveau, Thomas G. Moher, and Scott P. Robertson, editors. *Empirical Studies of Programmer: Fourth Workshop*. Human Computer Interaction. Ablex Publishing Corporation, Norwood, New Jersey, 1991.

[Lip84] Myron Lipow. Prediction of Software Failure. *The Journal of Systems and Software*, 4(4):71 – 76, November 1984.

[LPLS86] David C. Littman, Jeannine Pinto, Stanley Letovsky, and Elliot Soloway. *Mental Model and Software Maintenance*, volume 1 of *Empirical Studies of Programmers*, chapter 6, pages 80 – 97. Ablex Publishing Corporation, Norwood, New Jersey, 1986.

[MB77] R. W. Motley and W. D. Brooks. Statistical Prediction of Programming Errors. Technical Report RADC-TR-77-175, 1977.

[MMNS83] Richard J. Miaara, Joyce A. Musselman, Juan A Navarro, and Ben Shneiderman. Program indentation and comprehensibility. *Communications of the ACM*, 26(11):861–867, November 1983.

[Mon91] Douglas C. Montgomery. *Design and Analysis of Experiments*. John Wiley and sons, Inc., New York, 1991.

[MS81] Tom Moher and G. Michael Schneider. Methods for improving controlled experimentation in software engineering. In *Proceedings of the Fifth International Conference on Software Engineering*, pages 224–233, 1981.

[MW88] Jeffrey Mitchell and Charles Welty. Experimentation in computer science: an empirical view. *International Journal of Man-Machines Studies*, 29:613–624, 1988.

[NW74] John Neter and William Wasserman. *Applied Linear Statistical Models*. Richard D. Irwin, Inc., Homewood, Illinois 60430, 1974.

[OM88] Bernard Ostle and Linda C. Malone. *Statistics in Research: Basic Concepts and Techniques for Research Workers*. Iowa State University Press, Ames, Iowa, 1988.

[OSS87] Gary M. Olson, Sylvia Sheppard, and Elliot Soloway, editors. *Empirical Studies of Programmer: Second Workshop*. Human Computer Interaction. Ablex Publishing Corporation, Norwood, New Jersey, 1987.

[PAFB82] D. Potier, J. L. Albin, R. Ferreol, and A. Bilodeau. Experiments with Computer Software Complexity and Reliability. In *Proceedings 6th International Conference on Software Engineering*, pages 94–103, 1982.

[Pan91] Hsin Pan. Debugging with Dynamic Instrumentation and Test-Based Knowledge. Technical Report SERC-TR-105-P, Software Engineering Research Center, Purdue University, West Lafayette, IN, 1991.

[Pen87] Nancy Pennington. *Comprehension Strategies in Programming*, pages 100–113. Empirical Studies of Programmer: Second Workshop. Ablex Publishing Corporation, Norwood, New Jersey, 1987.

[PIS80] Tom Di Persio, Dan Isbister, and Ben Schneiderman. An experiment using memorization/reconstruction as a measure of programmer ability. *International Journal of Man-Machines Studies*, pages 339–354, March 1980.

[PS93] Hsin Pan and E. H. Spafford. Fault Localization Methods for Software Debugging. *Journal of Computer and Software Engineering*, 1993. (to appear).

[SCML79] S. B. Sheppard, B. Curtis, P. Milliman, and T. Love. Modern coding practices and programmer performance. *Computer*, 12(12):41–49, December 1979.

[SE84] Elliot Soloway and Kate Ehrlich. Empirical Studies of Programming Knowledge. *IEEE Transactions on Software Engineering*, 10(5):595–609, September 1984.

[SEG68] H. Sackman, W. J. Erikson, and E. Grant. Exploratory experimental studies comparing online and offline programming performance. *Communications of the ACM*, 11(1):3–11, January 1968.

[Sha83] E. Y. Shapiro. *Algorithmic Program Debugging*. MIT Press, Cambridge, Mass., 1983.

[SI86] Elliot Soloway and Sitharama Iyengar, editors. *Empirical Studies of Programmer*. Human Computer Interaction. Ablex Publishing Corporation, Norwood, New Jersey, 1986.

[SMMH] B. Shneiderman, R. E. Mayer, D. McKay, and P. Heller. Experimental investigations of the utility of detailed flowcharts in programming. *Communications of the ACM*, 20:373–381.

[Ves85] I. Vessey. Expertise in Debugging Computer Programs: A Process Analysis. *International Journal of Man-Machines Studies*, 23:459 – 494, 1985.

[Vir91] Chonchanok Viravan. Fault Investigation and Trial. Technical Report SERC-TR-104-P, Software Engineering Research Center, Purdue University, West Lafayette, IN, 1991.

[WDS81] S. N. Woodfield, H. E. Dunsmore, and V. Y. Shen. The Effect of Modularization and Comments on Program Comprehension. In *Proceedings 5th International Conference on Software Engineering*, pages 215–223, 1981.

[Wei74] L. M. Weissman. A Method for Studying the Psychological Complexity of Computer Programs. Technical Report TR-CSRG-37, University of Toronto, Department of Computer Science, Toronto, Canada, 1974.

[Wei84] Mark Weiser. Program Slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, July 1984.

[WL91] Mark Weiser and Jim Lyle. *Experiments on Slicing-Based Debugging Aids*, pages 187–197. Empirical Studies of Programmer: Fourth Workshop. Ablex Publishing Corporation, Norwood, New Jersey, 1991.

[WO84] Elaine Weyuker and Thomas Ostrand. Collecting and Categorizing Software Error Data in an Industrial Environment. *The Journal of Systems and Software*, 4(4):289 – 300, November 1984.