Experimental Design and Analysis in Software Engineering

Part 3: Types of Experimental Design

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

In previous tutorials, we have looked at why you might want to performance experiment, and how you go about planning one. Next, we address the different types of experimental designs.

TYPES OF EXPERIMENTAL DESIGN

It is useful to know and understand the several types of designs that you are likely to use in software engineering research, since the type of design constrains the type of analysis that can be performed and therefore the types of conclusions that can be drawn. (A description of these analysis techniques can be found in [Kitchenham 1992].) For example, there are several ways to calculate the F-statistic for an analysis of variance; the choice of calculation depends on the experimental design, including the number of variables and the way in which the subjects are grouped and balanced. Similarly, the measurement scale of the variables constrains the analysis. Nominal scales simply divide data into categories and can be analyzed by using statistical tests such as the Sign test (which looks at the direction of a score or measurement); on the other hand, ordinal scales permit rank ordering and can be investigated with more powerful tests such as Wilcoxon (looking at the size of the measurement differences). Parametric tests such as analysis of variance can be used only on data that is at least of an interval scale.

The sampling also enforces the design and constrains the analysis. For example, the amount of random variance should be equally distributed among the different experimental conditions if parametric tests are to be applied to the resulting data. Not only does the degree of randomization make a difference to the analysis, but also the distribution of the resulting data. If the experimental data is normally or near-normally distributed, then you can use parametric tests. However, if the data is not normally distributed, or if you do not know what the distribution is, nonparametric methods are preferable.

Many investigations involve more than one independent variable. In addition, the experiment invokes changes in the dependent variable as one or more of the independent variables changes. An independent variable is called a **factor** in the experimental design. For example, a study to determine the effects of experience and language on the productivity of programmers might have two factors: experience and language. The dependent variable is productivity. Various values or classifications for each factor are called the **levels** of the factor. Levels can be continuous or discrete, quantitative or qualitative. If experience is measured in years of experience as a programmer, then each integer number of years can be considered a level. If the most experienced programmer in the study has eight years of experience, and if there are five languages in the study, then the first factor has eight levels and the second factor five.

There are several types of factors, reflecting such things as treatments, replications, blocking and grouping. This article does not tell you what factors should be included in your design. Neither does it prescribe the number of factors nor the number of levels. Instead, it explains how the factors can be related to each other, and how the levels of one factor are combined with the levels of another factor to form the treatment combinations. The remainder of this section explains how to derive a design from the number of factors and levels you want to consider in your investigation.

Most designs in software engineering research are based on two simple relations between factors: crossing and nesting; each is discussed separately.

Crossing

The design of an experiment can be expressed in a notation that reflects the number of factors and how they relate to the different treatments. Expressing the design in terms of factors, called the **factorial design,** tells you how many different treatment combinations are required. Two factors, A and B, in a design are said to be **crossed** if each level of each factor appears with each level of the other factor. This relationship is denoted A X B. The design itself is illustrated in Figure 1, where ai represents the levels of factor A and bj the levels of factor B. The figure's first row indicates that you must have a treatment for level 1 of A occurring with level 1 of B, and of level 1 of A occurring with level 2 of B. The first column shows that you must have a treatment for level 1 of B occurring with each of the two levels of A. In the previous example, the effects of language and experience on productivity can be written as an 8 X 5 crossed design, requiring 40 different treatment combinations. This design means that your experiment must include treatments for each possible combination of language and experience. For three factors, A, B and C, the design A X B X C means that all combinations of all the levels occur.

Crossed		Factor B	
		Level 1	Level 2
A	1	a1 b1	a1 b2
	2	a2 b1	a2 b2

Figure 1. Example of a Crossed Design

Nesting

Factor B is **nested** within factor A if each meaningful level of B occurs in conjunction with only one level of factor A. The relationship is depicted as B(A), where B is the nested factor and A is the nest factor. For example, consider again the effects of language and experience on productivity. However, let factor A be the language, and B be the years of experience with a particular language. Now B is dependent on A, and each level of B occurs with only one level of A. That is, B is nested within A. A two-factor nested design in depicted in Figure 2.

Nested

Factor A			
Level 1		Level 2	
Factor B		Factor B	
Level 1	Level 2	Level 1	Level 2
a1 b1	a1 b2	a2 b1	a2 b2

Figure 2. Example of a Nested Design

Nesting can involve more than two factors. For example, three factors can be nested as C(B(A)). In addition, more complex designs can be created as nesting and crossing are combined.

There are several advantages to expressing a design in terms of factorials. First, factorials ensure that resources are used most efficiently. Second, information obtained in the experiment is complete and reflects the various possible interactions among variables. Consequently, the experimental results and the conclusions drawn from them are applicable over a wider range of conditions than they might otherwise be. Finally, the factorial design involves an implicit replication, yielding the related benefits in terms of reduced experimental error.

On the other hand, the preparation, administration and analysis of a complete factorial design is more complex and time-consuming than a simple comparison. With a large number of treatment combinations, the selection of homogeneous experimental units is difficult and can be costly. And some of the combinations may be impossible or of little interest to you, wasting valuable resources. For these reasons, we look at how to choose an appropriate experimental design for your situation.

SELECTING AN EXPERIMENTAL DESIGN

There are many choices for how to design your experiment, but the ultimate choice depends on two things: the goals of your investigation and the availability of resources. Here are guidelines on how to decide which design is right for your situation.

Choosing the number of factors

Many experiments involve only one variable or factor. These experiments may be quite complex, in that there may be many levels of the variable that are compared (e.g. the effects of many types of languages, or of several different tools). One-variable experiments are relatively simple to analyze, since the effects of the single factor are isolated from other variables that may affect the outcome. However, it is not always possible to eliminate the effects of other variables. Instead, we strive to minimize the effects, or at least distribute the effects equally across all the possible conditions we are examining. For example, techniques such as randomization aim to prevent variability among people from biasing the results.

But sometimes the absence of a second variable affects performance in the first variable. That is, people act differently in different circumstances, and you may be interested in the variable interactions as well as in individual variables. For instance, suppose you are considering the effects of a new design tool on productivity. The design tool may be used differently by designers who are well-versed in object-oriented design from those who are new to object-oriented design. If you were to design a one-factor experiment by eliminating the effects of experience with object-oriented design, you would get an incomplete (and probably incorrect) view of the effects of the tool. It is better to design a two-factor experiment that incorporates both use of the tool and designer experience. That is, by looking at the effects of several independent variables, you can assess not only the individual effects of each variable (known as the main effect of each variable) but also any possible interactions among the variables.

To see what we mean by interaction, consider the reuse of existing code. Suppose your organization has a repository of code modules that is made available to some of the programmers but not all. You design an experiment to measure the time it takes to code a module, distinguishing small modules from large. When the experiment is complete, you plot the results, separating the times for those who reused code from the times of those who did not. This experiment has two factors: module size and reuse. Each factor has two levels; module size is either small or large, and reuse is either present or absent. If the results of your experiment resemble Figure 3, then you can claim that there is no interaction between the two factors.

Notice that the lines in Figure 3 are parallel; the parallelism is an indication that the two variables have no effects on each other. Thus, an individual line shows the main effect of the variable it represents. In this case, each line shows that the time to code a module increases with the size of the module. In comparing the two lines, we see that reuse reduces the time to code a module, but the parallel lines indicate that size and degree of reuse do not change the overall trend. However, if the results resemble Figure 4, then there is indeed interaction between the variables, since the lines are not parallel. Such a graph may result if there is considerable time spent searching through the repository. For a small module, it may actually take more time to scan the repository than to code the module from scratch. For large modules, reuse is better than writing the entire module, but there is still significant time eaten up

in working with the repository. Thus, there is interaction between size of module and reuse of code.

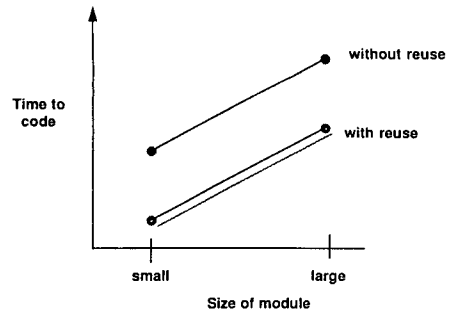

Figure 3. No interaction between factors

Thus, there is far more information available from the two-factor experiment than there would have been from two one-factor experiments. The latter would have confirmed that the time to code increases with the size of the module, both with and without reuse. But the two-factor experiment shows the relationship between the factors as well as the single-factor results. In particular, it shows that, for small modules, reuse may not be warranted. In other words, multiple-factor experiments offer multiple views of the data and enlighten us in ways that are not evident from a collection of single-factor experiments.

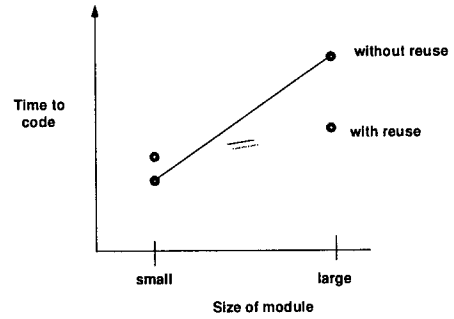

Figure 4. Interaction between factors

Another way of thinking about whether to use one factor or more is to decide what kind of comparison you want to make. If you are examining a set of competing treatments, you can use a single-factor experiment. For example, you may want to investigate three design methods and their effects on quality. That is, you apply design methods A, B and C to see which yields the highest quality design or code. Here, you do not have other variables that may interact with the design method.

On the other hand, you may want to investigate treatment combinations, rather than competing treatments. For example, instead of looking just at design methods, you want to analyze the effects of design methods in conjunction with tool assistance. Thus, you are comparing design method A with tool assistance to design method A without tool assistance, as well as design method B with tool assistance and without tool assistance. You have a two-factor experiment: one factor is the design method, and the other is the absence or presence of tool assistance. Your experiment tests n1 + n2 different treatments, where n1 is the number of levels in factor 1, and n2 is the number of levels in factor 2.

In the next issue, we will look at other design issues, including deciding between factors and blocks, how to choose between nested and crossed designs, and how to match subjects.

REFERENCE

Kitchenham, Barbara (1992), DESMET Handbook of Data Collection and Metrication Book 3: Analysing Software Data, National Computer Centre, Manchester, England.

Revolutionary DCF System to Replace CMM

Matt Sejnowski
9016 Yucca Mt, Rd
Austin, TX 78759
MattSejnowski@Wayne.Com

API Austin – First there were software metrics. With these, software developers and their management could finally measure something for the output of the software creation process. In the 80's these techniques flourished. Funny names for these measurements emerged, like "McCabe complexity" and "software volume".

Soon it was realized that there needed to be a way not only to measure the quality of the software output, but also to measure the quality of the engineering organization itself. The Capability Maturity Model, CMM, was developed in the early 90's. Organizations are audited by professionals and rated on a scale of 1 to 5. Low scores mean the software production process is chaotic, while 5 means that all aspects of software development are fully understood and carefully applied, all but assuring a quality product every time. Sadly, most software organizations today weigh in at a meager 1, and there's a surprising number of 0's out there.

Now, a revolutionary new measurement technique has been developed by a small startup consulting firm in Austin, Texas. The new system is simply known as DCF. The simplicity and elegance of the new measuring system belies its power in accurately judging the soundness of a software organization.

The inventor of DCF and founder of the DiCoFact Foundation, Matt Sejnowski, says the new measurement system is "simple and fool-proof, but modifications are being made to make it management-proof as well".

One Sunday morning Matt was performing his normal ritual of reading the most important parts of the newspaper first, when he came across his favorite comic strip, "Dilbert" by Scott Adams. Matt and his work colleagues loved this comic strip and were amazed by how many of the silly storylines reminded them of actual incidences at their company. They even suspected that Scott Adams was working there in dis-