Chapter 1.5

Methodological Issues in the Study of Programming

David J. Gilmore

Psychology Department, University of Nottingham, Nottingham, NG7 2RD, UK

Abstract

The primary purpose of this chapter is to provide readers with a sufficient understanding of methodological issues to enable them to consider critically the numerous experimental results presented elsewhere in the volume, particularly given the potential conflicts between controlled laboratory studies and real-world observations. The chapter commences with a review of the many reasons why data might be collected in the study of programming, before looking at the choice of programming tasks for investigation. This is followed by an introduction to the key concepts of experimental design – statistical significance, effect size, sample size and power – since understanding the distinctions between these is critical in applied research, even when behaviour is being observed in real-world contexts. Methods for observing programming behaviour *in situ* are presented next, followed by a brief discussion of some special issues which arise when we try to generalize from applied research. The chapter concludes with some example case studies which are intended to emphasize the complementary nature of controlled methods versus observation and artificiality versus the real world.

Psychology of Programming
ISBN 0-12-350772-3

1 Types of data collection

When planning or assessing research it is important to consider the reasons why the research is to be, or was, conducted. This is a topic which I shall return to later, but it is necessary here to outline four of the main reasons why data is collected. It would be a mistake to believe the distinctions are as clear as I present them here, particularly since research may be conducted for more than one reason.

1.1 Hypothesis testing

This is the main tradition in cognitive psychology and it is used when contrasting theoretical positions make different predictions about the effect of some manipulation on behaviour. In these circumstances it is desirable to set up experimental situations with everything constant, except for the essential manipulation. In this way any observed difference in performance can be attributed to that one difference.

For example, one could have a theory of program comprehension that claims comprehension is attained through an initial analysis of syntactic structure and, therefore, that the use of indentation to indicate syntactic structure will lead to improvements in all aspects of program comprehension. An alternative theory might simply be that much program comprehension is possible without an analysis of syntax. This theory predicts benefits for indentation only when the programming task requires understanding of the program's syntactic structure. In this, rather simple, scenario an ideal hypothesis-testing experiment can be constructed. Subjects can be supplied with indented, or non-indented programs and then be required to perform tasks which may or may not require comprehension of the syntactic structure.

Unfortunately life is not always as simple as in this example. Frequently it is not possible to make one manipulation without 'knock-on' effects resulting in experimental conditions that differ in more ways than the hypothesis requires. For instance, in the above example, indentation might mean that long lines must be wrapped onto the next line. In this case, the experimental design is impure, since any effect observed may be due either to indentation or to line wrapping, or the two effects may cancel each other out. In these circumstances the researcher must introduce new conditions that may not seem realistic, but which allow the hypothesized effect to be properly observed. In the indentation example one would have to compare indentation with and without line wrap, and line wrap with and without indentation.

Assessing the outcome of such experiments relies on statistical tests to determine the probability that the observed differences are due to chance variations in performance (assuming there is no difference in reality), rather than systematic variations. This probability is described as the statistical significance of the effect (see below).

A vital component in any hypothesis testing research is the theoretical framework, since this both enables the researcher to make decisions about what tasks to provide and what data to measure and it provides the context within which the results can be explained. Hypothesis-testing research is not possible without a theoretical framework and such research is better described as a comparison.

1.2 Comparisons

Similar to hypothesis testing, but less systematic, are comparisons. In these the researcher attempts to discover which of two (or more) alternatives is easier to use,

or whether some change to the programming process has an effect on performance. Comparisons are different from hypothesis testing in that they are intended to observe, but not explain any effect. Whereas hypothesis testing is used mainly in theoretical developments, comparisons are commonly used to recommend courses of action. Although comparisons cannot provide explanations, since the cases to be compared often differ on many criteria simultaneously, they are excellent at stimulating hypotheses and theoretical frameworks.

To return to our indentation example, our research question may simply be whether indentation provides a more usable representation of a computer program. For this comparison, although the results themselves may differ according to the presence or absence of line wrap in the actual experimental materials, line wrap is not important in interpreting the results, since it is a necessary feature of the indentation of real programs.

The results of such comparisons are conventionally tested for statistical significance, even though the more useful measure would be one of the size of the difference between the two conditions. This is known as the effect size (for more details, see below). The effect size is critical when results lead to recommendations for action, since there may be hidden costs in the action that were not assessed in the comparison (e.g. time spent producing flowcharts, printing facilities for flowcharts, etc.), but which need to be taken into account before action is taken.

1.3 Evaluations

The distinction between evaluations and comparisons is a very fine one and there is considerable overlap. However, an evaluation occurs when we are asking a question like 'Can people use flowcharts?', rather than 'Are flowcharts better than structure diagrams?'. Evaluations may also be performed simply with the intention of improving a system. Thus, it is unnecessary to compare it with something; rather, one looks for the weaknesses in the system. Whereas with comparisons we may design our experiment with only one or two measures of performance, in evaluations it is common to measure as much as possible. Subjective, non-performance measures (i.e. questionnaires) are usually used as well, since preferences may be as important as performance. Decisions as to which of the many measures are most informative will often be made *post hoc*. For example, in evaluating a novice programming environment we may be interested in showing that 50% of programs run first time or that 50% of bugs are corrected within 5 minutes. These criteria may have been derived from observations of other systems, but note that no direct comparison occurs.

1.4 Exploration

Exploratory data collection occurs when the research question begins 'What happens if ...?', or 'How do people do ...?'. Such questions are particularly common when the main research theme is novel and we lack enough information to engage in the other types of data collection (as when studying new paradigms, such as object-oriented, parallel or logic programming). For example, if we are interested in teaching aids for some programming language we may decide to collect examples of programs written by novices, in order to look at the sorts of errors they make, how they correct them, etc. Alternatively we may wish to look at problems that arise in system design

generally, rather than just in the coding stage, which may involve observations of a number of examples of system design.

The prime difficulties with exploratory data collection are that it can be very difficult to collect and record such data and it is rare for the resultant data to be well defined. Thus, we can try to count novice programming bugs, but it will be difficult to decide what counts as a single bug, and the number produced will not help us to describe the errors. Likewise there are no obvious numbers to be collected from observations of the system design process. Although numbers are not essential to all research, the description and interpretation of non-numeric data can be very difficult.

1.5 Summary

The above methods of data collection are each appropriate in different circumstances. The results of each need to be interpreted differently, and the results from each are often necessary for the design of further research using other approaches.

There is no simple relationship between the ease of data collection and result interpretation. Thus, hypothesis-testing research often requires much work in the construction of experimental conditions and materials. However, if designed correctly the results should be directly interpretable in terms of the theoretical justification of the experiment. Conversely, although exploratory data is often easier to collect, its analysis can be extremely time consuming.

2 The experimental task

As already noted, there are many different kinds of research goals, and as well as requiring different methods, they are also best examined through different tasks. Exploratory data collection will require the use of realistic programming tasks, since the main goal is to observe real programming activity. In contrast, the hypothesis-testing approach is not bound by the need for realism and can construct experimental tasks better suited to testing the hypothesis under consideration.

Many different tasks have been given to programming subjects. The more obvious include asking subjects to write a program and 'think aloud' while they do it, or giving them a program and setting them a comprehension quiz about it. Another very natural technique is to scrutinize code, often from students, either to classify types of errors (Spohrer and Soloway, 1986) or to compare frequencies of different solution types (Ratcliff and Siddiqi, 1985). But there are also less obvious, more artificial tasks, such as timed question answering with miniature programming languages (Sime *et al.*, 1977), or recalling progams that have only been viewed briefly (Kahney, 1983) Each task has its place. In this section I shall illustrate some of the reasons for choosing different types of tasks.

There are two important considerations – the object to be studied (the person or the system), and the programming task of primary interest (see Chapter 1.3). I shall examine these in turn, but the same conclusion can be drawn from each: 'You can't study people using one system and assume your results generalize; you can't study a single task and assume you have found out all the important facts'.

The prime interest in studying the person has been in the comparison of experts and novices, with particular interest in how experts represent programming knowl-

edge. The most popular approach has been exploratory techniques, such as verbal protocols and recall techniques (revealing what is recalled first, what is forgotten, etc.). In both, the results depend upon the investigator's judgement to extract meaningful patterns from the data. Subsequent testing to discover whether the patterns are real or spurious requires more rigorous approaches based on comparisons and hypothesis testing, such as the priming study by Pennington (1987) outlined below, or 'fill-in-the-blank' (Cloze) tests, in which we predict the ease with which different programmers will be able to complete different types of program statements.

Studying the system, unlike studying the person, does not seem to invite exploratory study. Typical approaches have been the comparison of different notational structures (Sime *et al.*, 1977) and studies of different browser designs (Monk and Walsh, 1988). These are experiments where performance was compared in two or more different experimental conditions, using very orthodox laboratory designs. The tasks used in such studies usually cover a range of programming activities. For example, Sime *et al.* used carefully timed question answering and coding, measuring time, number of syntactic errors, correction time, etc. The use of miniature programming languages, novice subjects and a game-like scenario enabled accurate conclusions to be drawn about notational differences, though at the expense of reality.

Studies of different programming tasks (i.e. writing, reading, debugging, modifying, etc.) offer another perspective on the choice of experimental tasks. The majority of studies of program writing have used observational techniques, either video (with or without verbal protocols) or *post hoc* analysis of code. Exceptions include the work of Hoc (1981) comparing different environments for solving data-driven and procedure-driven problems, and the work of Sime *et al.* on writing programs in different notations. Studies of program comprehension, in contrast, have used a great variety of different techniques. A common technique is a comprehension quiz, but this is not as simple as it might seem. Some studies have failed to detect effects of their manipulations (e.g. indentation) because when all the test items are averaged together there is no effect, but there may be large differences on a few of the items. The challenge to the researcher is then to explain why those few items are different. Other comprehension techniques include hand-execution, 'fill-in-the-blank' tests (Soloway and Ehrlich, 1984; Thomas and Zweben, 1986) and the recognition of program structures (Cunniff and Taylor, 1987). Studies of debugging have used both observational techniques and also comparisons of performance with and without various tools (Weiser, 1982, 1986; see below).

Here again there is an important warning: whilst single tasks, single subject populations (e.g. students) and single languages may be used to disprove a theoretical hypothesis, they are of little use when they confirm hypotheses. Early studies of programming languages often used a single task, such as hand-execution, to compare different notations. However, Green (1977) showed that different procedural notations indistinguishable on one task (reasoning about control-flow), could produce very different performance on a second task (reasoning about conditional structures). Later, Gilmore and Green (1984) were successful in showing that the effect partially reversed with declarative languages. Reasoning about control-flow revealed notational differences, where reasoning about conditional structures did not (more details are given in Chapter 2.2).

In summary, therefore, for truly generalizable conclusions, a variety of tasks, a variety of types of programmers and a variety of languages should be studied. When

all these variations are not present, then care must be taken not to overgeneralize the results.

3 Important concepts in experimental design

3.1 Statistical significance

This concept is probably familiar to readers, since it is the most frequently reported in published papers. It provides a measure of the likelihood of the observed difference in scores being a result of chance variations in the performance of four subject groups, assuming that they were drawn from the same population and that the experimental manipulations had no influence on their performance. As researchers we are interested in drawing the conclusion that our subject groups, although from the same population originally, have been differentially affected by the manipulation and can now be regarded as being from different populations. Thus, the statistical significance of a result assesses the probability that there is not a difference in the scores achieved by the different groups. Only when this probability is small can we draw the inference that our experimental manipulation had an effect on performance.

By convention a probability of 5% is taken as this small level beyond which we conclude that we have observed a real effect. Thus an observed difference with a 6% significance is commonly not reported, where a 4% probability might lead to publication. Because statistical significance is simply a measure of probability, we should take care to interpret it as such. A probability of 5% represents a one in twenty chance, which means that if we conduct twenty statistical tests, then there is a high probability that one will reveal a significance of around 5%, without any implication that our samples are from different populations. This can be a substantial problem when gathering large quantities of data, since this may require numerous statistical tests. The best solution is to use hypotheses to restrict the number of tests performed and to use significance levels smaller than 5%.

3.2 Effect size

A concept which is frequently confused with statistical significance and which is of the utmost importance when interpreting applied research is that of effect size. This reflects the influence on a score of the experimental manipulations, though not just by magnitude, but also by variability. Thus effects which are easiest to detect are either small in magnitude and totally reliable across subjects, or more variable, but large in magnitude. Much harder effects to detect are those which have both small magnitude and large variability.

Unfortunately effect size is not usually a calculable quantity, though estimates can sometimes be made by statistical analysis (see Hoc and Leplat, 1983, for further details). Nevertheless, the larger the effect the easier it will be to observe in an experiment. In such circumstances fewer subjects may be needed, or noisy measures of performance[1] may be used.

A statistically significant effect is not equivalent to an ecologically relevant effect, since the effect may be small, but the experimental precision sufficiently high that

[1] A noisy measure is one in which the task context or the mechanism of measurement introduces extra variability into the data (e.g. a task context in which many different strategies are available to subjects).

it is detected statistically. Correspondingly, a non-significant effect is not equivalent to an ecologically negligable effect, since the experimental precision may have been so poor that the effect could not be detected statistically. This second point is of great importance, since it is quite common for substantive conclusions to be based on the flimsy fact that the results were not statistically significant. Landauer (1987) provides further discussion on this topic in relation to HCI generally, along with numerous examples.

3.3 Sample size

The problem of sample size is one of the most difficult to deal with, since it seems abundantly clear that small samples must be a bad thing. This intuition is usually based on the problem of the representativeness of the sample, which is determined by the method used to select the sample.

However, if the sampling method is unbiased, then small samples are not inherently bad. Indeed, the same levels of statistical significance from small and large samples would be suggestive of larger- and smaller-effect sizes, respectively. If the same experiment were conducted with either twenty or forty subjects and the same average scores and significance levels were obtained, then contrary to intuition the experiment with twenty subjects represents a larger and probably more ecologically important effect.

The assessment of statistical significance is based on the sample size, and it is incorrect, therefore, to talk of a result being 'more significant' because of the sample size (whether small or large).

The main problem with small samples arises when the results are not significant (i.e. $p > 5\%$), since the use of a larger sample, all else being equal, might reduce p to below 5%. Thus again, one must be cautious when interpreting non-significant results, especially when the statistical significance is in the 5% to 20% region[2].

It is important to note also that the sample size is not the total number of subjects used, but the number used in each experimental condition. The choice of sample size is frequently constrained (by availability, expense, etc.), preventing the use of an adequate number for the anticipated effect size. When this situation arises, it is necessary to increase the power of the experiment through more careful design.

3.4 Power

The power of an experiment is a tricky concept. It is a measure of the quality of an experimental design. It is affected by sample size and by the quality of the measures of performance, the accuracy of the classification of subjects into groups and by the quality of the materials used.

Imagine we are investigating the benefits of a new debugging tool. The effect size is constant but, according to the experimental design, our ability is reflected by the accuracy of these estimates. Power will be affected by the quality of our performance measures, not just in terms of millisecond accuracy, but also in terms of how well defined and reliable across subjects our measures are. We could also use either a

[2]In general, it seems to be the case that the sensitivity of an experiment is proportional to the square root of the number of subjects. Thus to double the sensitivity of an experiment around four times as many subjects will be required. Adding a few extra subjects to the sample is unlikely to make much difference.

within-subjects design (where each subject performs in each experimental condition) or a between-subjects design (where each subject performs in only one experimental condition). In general a within-subjects design is more powerful, producing the same level of significance with fewer subjects. However, the repetition in a within-subjects design (which may lead to learning and order effects) may lessen the quality of the performance measures, thereby decreasing the power.

The main skill of experimental design is constructing an experiment of sufficient power to detect an effect, without overkill. When contradictory results are discovered (one with no significant differences and one with significant differences) it is important to remember that they may be due to differences in the power of the experimental designs.

3.5 Summary

(1) Statistical significance measures probability and is not a direct measure of effect size.

(2) Significance levels reflect both effect size and the power of the experimental design.

(3) Large effects require less-powerful experimental designs.

(4) Small sample sizes produce less-powerful designs.

(5) Noisy measures of performance produce less-powerful designs.

(6) Weak designs may fail to reveal genuine, but small, effects.

4 Observational techniques

Within the life sciences generally, use is made of both observational and experimental techniques. The former serve in the process of discovery when existing theoretical frameworks are unable to precisely define hypotheses, whilst the latter can validate hypotheses. The former can be particularly valuable for uncovering new and interesting phenomena, but they are unable to fully explain them. The latter will produce well-defined information, but are unlikely to discover something new. Observational techniques are a form of exploratory data collection, but they are becoming increasingly sophisticated and, thus, deserve a separate section of this chapter.

In the study of programming it may seem surprising that initial research focused on experiments, when observational methods would seem more appropriate to a new discipline. However, the first investigators either performed simple comparisons or developed their hypotheses from existing theoretical frameworks in psychology. The shift to observational techniques has grown from doubts about both the value of generalizing from context-bound comparisons and the further applicability of the theoretical frameworks.

Green (1980) provided a good review of some of this early research, covering three theoretical frameworks: 'Languages as Notations', 'Programming and Natural Languages' and 'Programming as Problem Solving'. The integration of these frameworks is the challenge to future research, requiring observational studies to stimulate the development of new unifying ideas.

Many of these observational techniques involve verbalization by programmers about either their current activity or, retrospectively, about some earlier activity. This is known as a protocol, and the analysis of such data is known as protocol analysis. Due to problems associated with the reliability of people's insight into their own thought processes, there are many complex issues in this area. Ericsson and Simon (1984) provide a clear and comprehensive coverage of these issues and is essential reading for any researcher contemplating the use of protocol analysis.

There are a number of different areas where such protocols might be collected:

(1) Interviews with programmers: Prior to the use of more formal techniques a researcher may conduct an interview with a programmer about the nature and organization of the work. The information acquired should not be assumed correct since there may be a discrepancy between actual activity and the planned, intended activity as perceived by the programmer. The more formal observations will provide an opportunity to verify these initial findings.

(2) Video plus verbalization: Video provides a relatively easy means of observing the programmer's 'natural' programming activities. These may be accompanied by simultaneous verbalizations, or the programmer may review their performance later on. The former may interfere with the subject's normal programming style (through increased mental workload), whilst the latter may lead to a rationalized account of unstructured behaviour. The advantage of such methods is that, unlike simple error counts, etc., they are able to capture the full dynamic nature of programming activity.

(3) Constructive interaction: A more natural form of observation may be to have two users co-operating on the solution to a problem. In this way their conversation provides natural verbalization. The main problem with this method is that of making an individualistic interpretation of the results. This method has not been used in relation to programming problems, though examples of its use can be found in O'Malley *et al.* (1984).

(4) Longitudinal studies of learning: Over the length of a programming course attitudes and perceptions can be assessed and submitted programs collected. The sheer volume of data that this produces is a problem in its own right, with any analysis being extremely time consuming and frequently subjective. These methods are further complicated by the uncontrollability of the situation. Prior experience, books read, experiments performed, effort expended are all uncontrolled variables which may influence the learning process.[3]

(5) Social and organizational processes: The larger context of programming (in a commercial/industrial world) is affected by many social and organizational issues. These cannot really be studied using traditional hypothesis-testing techniques and require observational methods. Chapter 4.1 provides an example of how such research may be conducted.

[3] In fact, these variables affect all studies of programming, though hypothesis-testing methods can control for them, in part, through careful experimental design. The only controls available for observational techniques are *post hoc* statistical ones (i.e. multiple linear regression). However, the increasing availability of easy-to-use statistical packages means that such controls may become more powerful than experimental ones.

In fact, it is misleading to pretend that there is a clear distinction between hypothesis-testing and observational techniques, since there are increasing number of studies that use aspects of both. For example, Widowski (1987) uses a simulation of a programming environment to capture elements of natural behaviour, but within a controlled environment. Leventhal and Instone (1989) describe a pilot longitudinal study in which formal experimental techniques are used at regular intervals throughout a programming course. This mixed research is likely to prove the most fruitful over the next few years.

5 Applying research

5.1 What was the question?

In the hypothesis-testing approach the question addressed by a piece of research is usually straightforward and clearly presented. Thus, for example, we might ask the question 'can we demonstrate circumstances in which novice behaviour is superior to that of experts?' (Adelson, 1984). But it is rare that we have such precise questions to answer. This is usually more of a problem for an assessor of the research than for the researcher. It is not unusual for research to be criticized for failing to consider certain factors or issues, when they were not relevant to the researcher's own question.

For example, the research conducted at the University of Sheffield in the 1970s (Sime *et al.*, 1977) into the design and use of different conditional structures (e.g. If ... GOTO versus IF ... THEN ... ELSE ...) was intended to further our understanding of how program representation interacts with the programming task to determine performance. Although it was not intended that the results be taken as strict recommendations for future language design, Sheil (1981) has dismissed these research efforts with

> 'how compelling does one find a psychological study of novice programming which found novices' fears of dealing with a teletype so overwhelmed their effects that they studied 'programming' in the context of a bizarre game which involved feeding a mechanical rabbit!'

The statistical argument behind Sheil's criticisms is that through using experimental design with high power, Sime *et al.* have detected effects of small size, which are of no practical value, an argument that is valid within the comparison approach to research, but not within hypothesis-testing, for which the design of Sime *et al.* was well suited.

Thus, hypothesis-testing research may not be directly useful for the production of human factors guidelines, though the resultant clarification of theory should be.

5.2 Interpreting causes

Conversely, in the human factors paradigm it is often sufficient just to show, for example, a 30% improvement in performance with system X as opposed to system Y. In these situations one must be wary of trying to draw theoretical conclusions from the design difference between X and Y since such inferences are often not valid. For example, we may compare experts in different programming languages, but any differences cannot then be attributed simply to language differences since the programmers may also differ in their programming education, motivation and programming problems

experienced. Gilmore and Green (1988) found marked differences in the nature of debugging by BASIC and Pascal programmers.

Their language-specific conclusions were based on the fact that the subjects in these different languages had apparently had similar computing experiences and that the subjects performed comparably overall. But Davies (1988) has shown that BASIC programmers can perform like Pascal programmers when they have received tuition about program design skills. Given the difficulties of finding Pascal programmers who have not received such training, there has been no investigation of whether Pascal programmers without design skills perform like BASIC programmers. For now, we can only conclude that there are education effects and probably language differences too.

Alternatively, a study may compare novices and experts in the same language, gaining considerable knowledge about their differences, but without understanding the causes of the differences. This is especially important when we use our knowledge of novice-expert differences to recommend particular teaching practices. The critical factor may be amount of programming experience, or range of problems studied or the numbers of programming languages known.

This problem of understanding the cause of an observed effect is closely related to the problem of experimental validity, which is of great importance in assessing the value of any piece of research.

6 Experimental validity

The problem of validity arises when we try to generalize from our results to other samples, to other languages, or to theoretical positions. There are many different types of validity and it is important not to confuse them.

Two important types of validity are internal and external validity. Internal validity describes our ability to be sure that our explanation of the observed differences is the only likely explanation, and it is closely related to the statistical issues discussed earlier. External validity describes our ability to make correct generalizations about the implications of the results. In most cases external validity is only an issue when we have high internal validity. Hypothesis-testing research must place most emphasis on internal validity, while comparisons and evaluations need to emphasize external validity.

For example, the Sime *et al.* studies (and the later Gilmore and Green studies mentioned above) have high internal validity.

The differences observed cannot be due to anything other than the differences in the notations presented to subjects. These studies demonstrated that the subjects needed different types of information for different tasks and that an important property of the notations used was the match between the information easily obtained and that needed for the task.

These studies have moderate external validity, since the result cannot be unquestionably generalized to all programming notations and programming tasks (a result that would be of great interest to anyone developing new notations). The results should generalize, however, to novice programmers learning a first language and probably to novices learning any language. Since the subjects in the experiments did not become experts in the notations, it is hard to be sure of the external validity in relation to expert programmers.

Although not lacking in external validity, these studies do lack face validity, a concept that is easily confused with external validity. Whereas external validity reflects our actual ability to generalize, face validity reflects our apparent ability to generalize.

Face validity can be illustrated through one of Sheil's criticisms of these experiments. Sheil appeals to his readers' intuitions about the compelling nature of research involving a 'mechanical rabbit' without discussing whether the important conclusions depend upon the method used. Given that the main implications of the Sime *et al.* studies derive from the theoretical position, the method of study is of little relevance.

Unfortunately, research can have high face validity without either internal and external validity. For example, Gannon (1977) assessed the value of data typing in programming languages. To do this, he developed two complete languages that were identical in all respects except data typing and, unavoidably, any other features affected by data typing. Although his results show an advantage for statically typed languages, it is not clear whether this is solely due to data typing, or whether the advantage derives from the 'better primitives with which to solve string-processing problems', or the fact that 'most students learn a statically typed language as their first language'. Any explanation of these results cannot be shown to be the only valid explanation (an absence of internal validity). Without internal validity there is unlikely to be a high level of external validity.

Face validity is an important property of research, especially when there is a need to convince others of the value of the research. However, it should be considered *after* internal and external validity, not before. Programming behaviour is extremely complex and in order to acquire statistically useful measures of performance simple tasks and unreal measures of performance are often essential. This is particularly so when we wish to make inferences about the causes of observed differences in performance.

7 Methodological case studies

7.1 Current concerns rigorously applied

There is a considerable literature on the nature of expertise in programming (see Chapter 3.1), but very little of it has used the hypothesis-testing approach to validate the many observational results. Pennington (1987) carried out a complex study that took a standard psychological methodology and applied it to programming. Although the study has little or no face validity, it has high internal validity, through a tradition in cognitive psychology (McKoon and Ratcliff, 1980). Pennington's experiment addressed the expert programmer's mental representation of a computer program. The subjects' task was to comprehend and learn a program.

Pennington tested the programmer's memory for particular lines of the program through a recognition test. However, before presenting the statements to be recognized, she presented subjects with a momentary glimpse of another statement from the program. If the mental representation of this 'priming' statement is close to the 'to be recognized' statement then the speed with which recognition occurs will be quicker. If there is no mental association between prime and test statements then the recognition time will be slower.

By using primes that were either procedurally or functionally connected to the test item it is possible to infer the mental distance between statements in the program.

Pennington's results revealed that

> procedural rather than functional units form the basis of expert programmers' mental representations. (Pennington, 1987, p. 295)

This result, which is different from others in the area, is important because of the high internal validity. It serves not to further applications of existing results, but to qualify them and point out possible errors of interpretation in earlier experiments. This reveals the value in using the hypothesis-testing approach for theories that have been derived from comparative and evaluative data.

7.2 The real world raises problems of representation

At the opposite end of the spectrum we can consider a valuable observational study with high face validity, but which does not lend itself to generalizable conclusions. Letovsky *et al.* (1987) looked at an IBM code inspection process. They analysed one sixty-five minute video tape in detail and found that three goals were being achieved during the inspection (clarity, correctness and consistency) through the use of three behaviours (reconstruction, simulation and cross-checking). A feature of their analysis was that although they were observing a high-level process, the problems encountered during the code inspection varied from syntactic concerns to problems of specification. Of particular interest are the syntactic difficulties which arose with the use of 'case-statement' in which there was conflict between the 'natural' task structure and the most efficient code structure. The design team had to make decisions about balancing comprehensibility and efficiency.

Although it is not clear how an alternative syntax could have solved this problem, it illustrates that the mapping from syntax to semantics can be a critical part of the programming process. Thus, unlike a hypothesis-testing experimental study, which tends to look for one clear-cut result, observational studies can provide interesting results along a variety of dimensions.

7.3 The challenge to external validity

Weiser (1982) investigated aspects of expert debugging strategies and found that experts use 'slices' when debugging. A 'slice' is that part of the program which can alter the value of a particular variable (the buggy variable). An important property of a slice is that the statements within it need not be textually contiguous, but may be scattered throughout the program.

In his experiment, Weiser's subjects had to debug Algol-W programs of seventy-five to 150 lines, which contained just one error in the subroutine which performed the bulk of the calculations. To test the programmers' use of slices, they were given a recognition memory test after debugging all three programs. The test examined their memory for slices versus contiguous code and for relevant (to the bug) and irrelevant statements. Since the interest was in semantic understanding, details such as variable names were altered before the lines were presented for recognition.

The results of the experiment revealed that relevant slices were recognized as accurately as relevant contiguous statements, but significantly better than the irrelevant slices. From this Weiser concludes that although programmers do not use only slices when debugging; this is clear evidence that slices are used as part of the debugging strategy.

Given that slices can be formally defined, the natural progression of this research was to build slice-based debugging tools. Weiser and Lyle (1986) evaluated the utility of such tools for debugging. Despite the apparent validity of the early experiments, these evaluations revealed that although 'pencil and paper' simulations of slicing aids were beneficial in speeding up debugging, the on-line tool showed no such benefits. Weiser and Lyle conclude that

> perhaps, slicing is like watching a beautiful sunset – a computer can do it, but it just isn't the same. (p. 189)

Weiser and Lyle went on to look at dicing tools, which reduce the size of a slice still further (slicing the slice), and found that these tools did provide the programmers with an advantage when debugging, even though there were no results to show that expert programmers use dice when debugging.

This example reveals the need to be cautious when assessing the external validity of some research. The research results might seem quite conclusive, but the step from them to application may be considerable.

8 Summary

This chapter has presented various important concepts in experimental research and demonstrated that the study of computer programming skills requires a wide range of approaches to the collection of data.

An understanding of hypothesis-testing techniques is essential, since almost all of the early research on programming has this form and, at least until there is an agreed theoretical framework, it will continue to be of great importance in studying programming. But recently, observational techniques have become popular due to their ability to capture the more complex aspects of programming. Such techniques are now an invaluable part of the empirical repertoire, even though there are still some analytical problems.

The case studies demonstrate how a variety of approaches, tasks and methods should be viewed not as a problem or a weakness, but as an essential property of a rapidly changing area which has lots of pressing, but very different questions (e.g. theoretical, practical, speculative). There are many more examples to be found and hopefully this chapter has helped to prepare the reader to appreciate those that are presented in the rest of this book.

References

Adelson, B. (1984). When novices surpass experts: the difficulty of a task may increase with expertise. *Journal of Experimental Psychology; Learning, Memory and Cognition,* **10**, 483-495.

Cunniff, N. and Taylor, R.P. (1987). Graphical versus textual representation: an empirical study of novices' program comprehension. *In* G. Olson, S. Sheppard and E. Soloway (Eds), *Empirical Studies of Programmers: Second Workshop.* Norwood, NJ: Ablex.

Davies, S. (1988). The nature and development of programming plans. Paper presented at *International Conference on Thinking,* Aberdeen, 1988.

Ericsson, K.A. and Simon, H. A. (1984). *Protocol Analysis: Verbal Reports as Data.* Cambridge, MA: MIT Press.

Gannon, J.D. (1977). An experimental evaluation of data-type conventions. *Communications of the ACM,* **20,** 584-595.

Gilmore, D.J. and Green, T.R.G. (1984). Comprehension and recall of miniature programming languages. *International Journal of Man-Machine Studies,* **21,** 31-48.

Gilmore, D.J. and Green, T.R.G. (1988). Programming plans and programming expertise. *Quarterly Journal of Experimental Psychology,* **40a,** 423-442.

Green, T.R.G. (1977). Conditional program statements and their comprehensibility to professional programmers. *Journal of Occupational Psychology,* **50,** 93 - 109.

Green, T.R.G. (1980). Programming as a cognitive activity. *In* H.T. Smith and T.R.G. Green (Eds), *Human Interaction With Computers.* London: Academic Press.

Hoc, J-M. (1981). Planning and direction of problem solving in structured programming: an empirical comparison between two methods. *International Journal of Man-Machine Studies,* **15,** 363-383.

Hoc, J-M. and Leplat, J. (1983). *International Journal of Man-Machine Studies,* **18,** 283-306.

Kahney, H. (1983). Problem solving by novice programmers. *In* T.R.G. Green, S.J. Payne, and G. van der Veer (Eds), *The Psychology of Computer Use.* London: Academic Press.

Landauer, T. (1987). Cognitive psychology and computer system design. *In* J. Carroll (Ed.), *Interfacing Thought.* Cambridge, MA: MIT Press.

Leventhal, L. and Instone, K. (1989). Becoming an expert: the process of acquiring expertise among highly novice computer scientists. Paper presented at *Psychology of Programming Interest Group: First Workshop.* Warwick University, January, 1989.

Letovsky, S., Pinto, J., Lambert, R. and Soloway, E. (1987). A cognitive analysis of a code inspection. *In* G. Olson, S. Sheppard and E. Soloway (Eds), *Empirical Studies of Programmers: Second Workshop.* Norwood, NJ: Ablex.

McKoon and Ratcliff (1980). Priming in item recognition: the organisation of propositions in memory for text. *Journal of Verbal Learning and Verbal Behaviour,* **19,** 369-386.

Monk, A.F. and Walsh, P. (1988). Browsers for literate programs: avoiding the cognitive overheads. *Proceedings of ECCE4, Fourth European Conference on Cognitive Ergonomics.* Cambridge, UK, September, 1988.

O'Malley, C. E., Draper, S. W. and Riley, M. S. (1984). Constructive interaction: a method for studying human-computer-human interaction. *In* B. Shackel (Ed.), *Proceedings of Interact 84.* Amsterdam: Elsevier North-Holland.

Pennington, N. (1987). Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive Psychology,* **19,** 295-341.

Ratcliff, B. and Siddiqi, J.I.A. (1985). An empirical investigation into problem decomposition strategies used in program design. *International Journal of Man-Machine Studies,* **22,** 77-90.

Sheil, B. (1981). Coping with complexity. Paper presented at Houston Symposium III: Information Technology in the 1980s.

Sime, M.E., Arblaster, A.T. and Green, T.R.G. (1977). Structuring the programmer's task. *Journal of Occupational Psychology,* **50,** 205-216.

Soloway, E. and Ehrlich, K. (1984). Empirical studies of programming knowledge. *IEEE Transactions on Software Engineering,* **5,** 595-609.

Spohrer, J.C. and Soloway, E. (1986). Analysing the high frequency bugs in novice programs. *In* E. Soloway and S. Iyengar (Eds), *Empirical Studies of Programmers.* Norwood, NJ: Ablex.

Thomas, M. and Zweben, S. (1986). The effect of program dependent and program independent deletions on software cloze tests. *In* E. Soloway and S. Iyengar (Eds), *Empirical Studies of Programmers.* Norwood, NJ: Ablex.

Weiser, M. (1982). Programmers use slices when debugging. *Communications of the ACM,* **25,** 446-452.

Weiser, M. and Lyle, J. (1986). Experiments on slicing-based debugging aids. *In* E. Soloway and S. Iyengar (Eds), *Empirical Studies of Programmers.* Norwood, NJ: Ablex.

Widowski, D. (1987). Reading, comprehending and recalling computer programs as a function of expertise. *Proceedings of CERCLE Workshop on Complex Learning.* Grange-over-Sands.