"An Empirical Investigation Into Problem Decomposition Strategies Used In Program Design"

by

Jawed Iqbal Ahmed Siddiqi

A thesis submitted to the University of Aston in Birmingham
for the degree of
Doctor of Philosophy.

Department of Computer Science

October 1984

The University of Aston in Birmingham

An Empirical Investigation Into Problem Decomposition Strategies Used In Program Design

by

Jawed Iqbal Ahmed Siddiqi

Summary

In this thesis, findings are presented of a research investigation into general strategies for, and the effect of certain factors relating to, problem decomposition used in program design. The investigation involved two empirical studies, totalling six separate experiments, in which subjects trained in the broad principles of structured programming were asked to undertake various program design tasks associated with particular programming problems, solutions to which can be mapped through the use of 'process structure hierarchies' onto a small number of 'process decomposition paradigms'. Analysis of the results revealed that solutions based on primitive, as opposed to abstract, perceptions of problem structure were strongly preferred, initially easier to perceive though harder to complete and were more error-prone. A model of program designer behaviour together with generalised problem decomposition strategies are advanced, that view program design as a problem-solving activity. These proposals form an explanatory framework for interpreting the experimental results, which are shown to be consistent with the proposals. In particular, it is argued that aspects of problem presentation and subject familiarity with component parts of a problem, are major factors that influence problem decomposition, and were responsible for the observed strong bias towards simplistic solutions. Additionally, it is argued that such bias can also be caused by "perception difficulty" allied to inadequacies in abstraction skills attributable to previous training. The thesis concludes with a recommendation that more specific, "criteria-driven" forms of structured programming need to be taught and practiced.

Keywords: structured programming, empirical investigation,
problem decomposition strategies, program design

A thesis submitted to the University of Aston in Birmingham in October 1984 for the degree of Doctor of Philosophy.

Acknowledgements

I would like to acknowledge the encouragement and supervision given to me by Bryan Ratcliff throughout the entire project. I am particularly grateful for his critical reading of my manuscripts, because it assisted me in expressing many of the ideas in this work with a greater degree of clarity and precision than would have been otherwise possible.

I would also like to thank all the students who volunteered to be experimental subjects.

The first three years of this reseach was funded by the Science Research Council.

Contents

	PAGE
Chapter 1 - Introduction	1
Chapter 2 - Background Review	
2.1 Issues In Program Design	6
2.1.1 Introduction	6
2.1.2 Early Contributions	7
2.1.3 Programming Methodologies	12
2.1.4 Recent Developments	23
2.2 Empirical Considerations	25
2.2.1 Introduction	25
2.2.2 Methodological Issues	30
2.3 Conclusion	43
Chapter 3 - Report of Investigation	
3.1 Experimental Context	45
3.2 Hypothesis Testing	49
3.3 Methodological Specifics	52
3.3.1 Choice of Subjects	52
3.3.2 Choice of Experimental Material	54
3.3.3 Choice of Metrics	57
3.4 The Signal Study	61
3.4.1 Experiment 1	61
3.4.2 Experiment 2	65
3.4.3 Experiment 3	67
3.5 The Line-Edit Study	69
3.5.1 Experiment 1	70
3.5.2 Experiment 2	72

3.5.3 Experiment 3 74

Chapter 4 - Discussion

4.1 Problem Analysis and Design Evaluation 79

4.1.1 Introduction 79

4.1.2 The Signal Problem 81

4.1.3 The Line Edit Problem 87

4.1.4 Conclusion 99

4.2 Conceptual Model of Programmer Designer Behaviour 100

4.2.1 Formulation of the model 100

4.3 Problem Decomposition Behaviour 1o6

4.3.1 Strategies 106

4.3.2 Related Factors 111

4.3.3 Further Contributory Factors 114

4.4 Elaboration of Decomposition Paradigms 116

Chapter 5 - Conclusion 121

Appendices 126

Appendix 1.1 127

1.2 128

1.3 129

1.4 130

1.5 131

Appendix 2.1 132

2.2 133

2.3	134
Appendix 3.1	135
3.2	136
3.3	137
3.4	138
3.5	139
3.6	140
3.7	141
Appendix 4.1	142
4.2	143
4.3	144
4.4	145
4.5	146
Appendix 5.1	147
5.2	147
5.3	148
5.4	149
Appendix 6	149a
References	150

1. Introduction

In a society where considerable reliance is placed on computer software systems, it is imperative to constantly improve software construction methods and practitioner skills, so that ultimately we are able to justify, and have confidence in, this reliance. The motivation and perhaps the ultimate goal of this research is to attempt to have a direct bearing on this continuing need for improved software. However, the immediate aim to which this thesis addresses itself, is to contribute to the field of software engineering by improving our understanding of the program design process.

Whilst the overall direction of the research is closely allied to Dijkstra's compelling desire to change the current state of affairs, in which most of the programs written are totally "unfit for human appreciation" [1]; its particular line of attack is to undertake and attempt to counteract Weinberg's tempting challenge that "Perhaps programming is too complex a behaviour to be studied and must remain largely a mysterious process" [2]. Specifically, the research investigates the area of program design from a human-factors viewpoint. This is in line with Dijkstra's work, which considers programming as a purely human activity; as Weinberg himself states, "programming is a form - a complex form - of human behaviour" [ibid]. Therefore, this investigation, in common with many that involve the study of human behaviour, is empirical in nature. It employs the established principle of such research known as the 'scientific method', which consists of conducting experiments to gather, evaluate and interpret observational evidence.

A first major goal of the research was to analyse and synthesise the separate fields of program design and empirical evaluation into a coherent project. Chapter 2 of this dissertation provides a review of

appropriate background material and concludes with a summary in which four specific investigative research objectives are stated, these representing the basis of a study of problem decomposition strategies used in program design. Initially, the review surveys early contributions to program design, the structured progamming revolution and current programming methodologies. The survey ends with a long-term perspective of the issues central to program design. An overview of the rationale behind, and the mechanisms of, the scientific method is then presented, together with the methodological issues of experimental research into software engineering and the reasons why a suitable experimental methodology is necessary. The particular spectrum of investigations presented in this chapter is chosen because it illustrates important aspects concerning methodological issues central to programming experiments.

A further goal was to devise an experimental methodology, based on established principles of the scientific method but tailored for application to program design, that could be applied to each of the experiments to be carried out. After detailing the context within which experimentation was performed, chapter 3 describes this methodology, and in particular, the experimental methods specifically devised. These methods represent a contribution to techniques for analysing the nature and effects of problem decomposition strategies used in program design. They include the use of:

(i) hierarchical process structures for classifying programs;

(ii) algorithmic outlines as process structure cues in controlled experiments;

(iii) error frequencies as indicators of possible relationships between strategies and errors.

In total, six experiments are described. These experiments, which formed two sets of studies, were performed over a period of 18

months in academic environments, each set being associated with a different programming problem. The subjects used in the experiments were mainly computer science undergraduates, although in the very first experiment pre-university and post-graduate students also took part. Each experiment involved groups of subjects, all of whom had been previously taught to program in a "structured manner", undertaking various program design tasks. The problems chosen, namely, the 'signal problem' [3] and the 'line editing problem' [4] are such that their various solutions can be mapped respectively onto a small number of "process decomposition paradigms", corresponding to different algorith mic structures.

The investigation involved the collection and statistical analysis of data from both observational and comparative experiments, the details of which are also described in chapter 3. Both types of experiments employed non-parametric tests for two or more independent samples, as a decision mechanism for statistical significance. The observational experiments used measures of association as indicators for further investigation, whereas the controlled experiments tested causal relationships.

Chapter 4 details an informal step-wise refinement of the signal and line-edit problems and considers a design evaluation of their solutions, this being followed by the formulation of a model of program designer behaviour in which goal generation is hypothesised to be stimulus and knowledge activated. The application of the model to the two problems used in the experiments provides a description of presumed designer behaviour. Evidence consistent with the model obtained from the experimental studies is presented, the significant results of which are as follows.

Observations from an exploratory experiment in the set of investigatigative studies associated with the signal problem led to

proposals that subjects do not use an idealised "top-down" manner of design. Moreover, two generalised forms of problem-solving strategy were advanced, namely, 'process next item' and 'incremental design'. The former strategy is data-driven, - that is, its application in problem decomposition results in a program structure based upon some particular perception of the data stream and how the latter should be processed. In contrast, the latter strategy is requirements-driven - that is, it focusses on identifying, and then fullfilling, those processing requirements that are immediately attainable in developing a solution.

A further conclusion is that there appeared to be a strong tendency towards using strategies that produce solutions based on primitive perceptions of problem structure. Confirmatory observations were obtained from an experiment on the 'line-edit problem', in which it was predicted that applying the above-mentioned strategies would produce certain decompositions, the greater proportion of which would be based on primitive, as opposed to, higher-level, more abstract, perceptions. In addition, one implication of the result of the exploratory experiment on the signal problem was that such decompositions are easier to perceive; this was also confirmed via a subsequent controlled experiment.

Consideration of possible factors affecting such strategies, and why the 'primitive pathway' is more obvious, gave rise to two contributory factors being advanced: the nature of training received and of problem wording. Evidence was gathered, firstly by comparing results from two experiments investigating the effect of training. These results supported the view that subjects trained in the broad principles of structured programming possess less developed abstraction skills than those trained in a form of structured programming that incorporates decomposition criteria more specific

than those associated with simple "top-down refinement". In addition, an investigation into the effect of problem wording revealed that the presence of certain aspects of problem presentation, had a marked effect on decompositions produced.

Investigations concerning the effect of strategies on subjects' performance as measured in terms of correctness achieved and effort required, showed that solutions based on primitive perceptions, which were of poor quality in modularity terms, contained a greater proportion of errors. Moreover, it was observed that subjects' errors were strongly associated with the placement of those program components that contributed to the low degree of modularity.

The concluding chapter summarises the findings of the research and considers their implications. In particular, a recommendation is advanced that subjects should be trained in more "directed" forms of structured programming where abstraction skills receive greater emphasis. Some possible directions for future research are also considered and it is proposed that the model and hypothesised strategies constitute a framework for further experimentation into this broad "from-specification-to-decomposition" area.

2. Background Review

2.1 Issues In Program Design

2.1.1 Introduction

Expert practitioners have from time to time made numerous recommendations concerning how program design should be performed. However, little research has been carried out into how this process actually is performed and whether or not their recommendations are beneficial.

Two fundamental components appropriate to investigating program design are the design activity itself and the program it generates. Traditionally, a program is viewed as a series of instructions obeyed by a machine. This definition places emphasis on control flow and accords well with the obsolete method of using flowcharts for design but does not reflect the modern practice of emphasising structure. To do this, a more suitable definition of a program might be: "a structured representation of a task to be performed in order to solve a given problem, expressed (usually) in a procedural language". The activity of designing therefore involves deriving such a structure. Thus it is not suprising, that the use of hierarchical structuring to manage complexity (the basis of most, if not all, modern design methodologies) is in accordance with ideas forwarded in a number of early works [e.g.,5,6,7] that investigated the way people handled complexity.

However, before focussing attention on specific issues, consideration needs to be given briefly to the developmental stages of software production, of which design is an integral part. As this research is concerned with program, rather than system design, the

requirements analysis stage can be omitted. Hence, specification, design and implementation will be considered to be the stages required to engineer a program. Requirements specification produces in part a precise prescription of the program function. From this specification of "what" has to be achieved, design produces a specification of "how" it will be achieved, or, as many authors have stated: a process of transforming "the what into the how". Implementation involves producing, installing and maintaining the final product. The realisation of a program from a design specification involves coding (translating the design description into the required source language), testing (detecting errors in the program) and debugging (removing errors detected by testing). As with other engineering disciplines, software engineering is not a sequential, but rather a cyclic, process. This is because each representation of the problem, whether a specification, a design or a program, undergoes a process of validation which may reveal possible flaws that need to be corrected or resolved.

To fully appreciate the claimed benefits of ideas that have transformed program design from a mystery surrounded by folklore to a systematic discipline, it is instructive to start at its inception - that is, at the advent of computers.

2.1.2 Early Contributions

The history of computers has witnessed not only a rapid increase in the number of computer-based systems, but also an ever diversifying range of applications in which they are used. The constructional approach adopted for hardware was to build a "general-purpose" machine capable of executing a series of instructions, whilst software was to be "custom-designed" for each application. The obvious and most

significant benefit for software production that accrued from this separation was that the physical characteristics of the machine could be ignored. However, because hardware design involved building machines from physical components that had the specific task of executing computational processes, standardised techniques were developed and an "engineering" approach used. In contrast, software design involved constructing programs from abstract components that were required to carry out varied tasks; as a consequence, software design remained very much a mysterious process.

This state-of-the-art in software production, coupled with the dramatic growth in the size of software systems, led to observed, sometimes large differences between what was hoped for and what was actually achieved in the construction of those systems. There were widely held views as to the seriousness or otherwise of the problem; in retrospect, many have come to refer to it as the "software crisis". Boehm [8] provides a quantitative assessment of the cost of dealing with inadequate software, giving examples of software-hardware cost ratios ranging from 8:1 to 2:1 with an average of 7:3 for 1970. He pointed out that the cost of computer software compared to hardware had escalated and predicted a continued increase. For example, in 1955 the ratio was as low as 2:8; Boehm predicted this to be as high as 9:1 in 1985.

Despite the size of the problem and its impact, the primary condition for improvement is to accept its existence. Indeed, reports such as [9] contain documented experiences of development efforts which suggested that not only had the existence of the problem been acknowleged, but also that improved methods were being sought to develop software. In the late sixties, it became increasingly apparent that there was a need to provide a disciplined approach to software production. Consequently, much of the debate at that time

focussed on structuring the design process and on ordering design decisions.

One of the earliest and most significant contributors to the debate was E.W. Dijkstra [10], whose concern lay with "intrinsically large" programs. By this he meant "programs that are large due to complexity of the task, in contrast to programs that have exploded (by inadequacy of the equipment, unhappy decisions, poor understanding of the problem, etc.)" [ibid]. His method of attack was to carry out introspective "programming experiments"; these investigated "what techniques (mental, organisational or mechanical) could be applied in the process of program composition" [ibid] that would produce "an increase in our programming ability by an order of magnitude" [ibid] so as to overcome problems associated with large programs and enable the latter's correctness to be demonstrated. He overcame the anomaly of having to choose, for practical reasons, relatively small programs for his experiments "by treating problems of size explicitly and trying to find their consequences as much as possible by analysis, inspection and reflection" [ibid].

From Dijkstra's experiments [11] in the design and construction of multiprogramming systems, he suggested applying the principle of "divide and rule" for structuring the design process so that size-induced complexity could be controlled and the construction of such systems could be carried out in manageable steps. The application of this principle structures the system as a hierachical set of layers or "machines" where the relationship between consecutive layers is such that the machine at some non-primitive level is an abstraction or functional description of the machine at the next lower level, the latter providing the resources for the formulation of the former. Dijkstra [12] concluded from his development work on the "THE"-multiprogramming system that "the hierarchical structure proved

to be vital for the verification of the logical soundness of the design and the correctness of its implementation".

Whilst Dijkstra had laid down the guiding principle that program structure should be a layered hierarchy, there was - and still remains - some controversy as to whether the layers should be constructed from the "bottom-up" (i.e., starting with primitives) or from the "top-down" (i.e., starting with the target system). Gill [13] gives this practical and sound advice:

> "Clearly, the top-down approach is appropriate when the target system is already closely defined but the hardware or low-level language is initially in doubt. Conversely the bottom-up approach is appropriate when the hardware is given but the target system is only defined in a general way."

He points out that the weakness of both approaches is that early decisions will "be propagated through the layers and will finally cause trouble by proving undesirable and difficult to remove" [ibid] and that their success is dependent on the designer's ability to foresee the consequences of these problems. The need for clearly defined primitives and target systems is implicit in his recommendations, although he realised they rarely occurred in practice.

Randell's observations [14], based on work carried out on three major independent, yet related systems, revealed that their structure reflected the development process, and that each layer of the structure was a set of solutions to a set of problems considered to be closely related. On the ordering of decisions, he notes two features of the top-down approach. The first is that the designer "at each stage is attempting to define what a given component should do, before getting involved in decisions as to how the given component should provide this function" [ibid]. The other is suitability of the appproach "for the designer who has faith in in his ability to estimate the feasibility of constructing a component to match a set of

specifications". In contrast, the features of the bottom-up approach are that it "proceeds by a gradually increasing complexity of combinations of building blocks" [ibid] and is best suited "for the designer who prefers to estimate the utility of the component that he has decided he can construct" [ibid]. Like Gill, Randell warns against the dangers of strict adherence to the top-down or bottom-up approaches and points out that the then current emphasis on the former was an attempt to reduce the preponderance of the latter. Finally, in order to improve systems quality, Randell stressed the need for an effective methodological approach to design and for guidelines on the order in which decisions are made.

The need seriously to consider from an engineering perspective the desirability of a software components industry, analogous to its hardware counterpart, was noted by Perlis [15], and elaborated by McIlroy [16]. McIlroy's work stressed the desirability of building big systems from smaller standardised families of components. Analogies with hardware were noted at differing component levels. For instance, at the primitive level "software production in the large would be enormously helped by the availability of spectra of high quality routines, quite as mechanical design is abetted by the existence of families of structural shapes, screws or resistors" [ibid]. Moreover, at the conceptual level, the importance of exploiting the correspondence between interchangeability of hardware sub-assemblies and modularity of software was identified.

One of the earliest practised techniques that attempted to contain the complexity explosion of "monolithic" programs was modular programming. This was initally just the crude application of the divide-and-conquer principle in which the division of a program into modules was governed by implementation convenience rather than design needs. Not suprisingly, this approach proved to be an inadequate

rationale, although it allowed the exploitation of benefits that result in implementation from using common modules. Because the emphasis of modular programming was on the attainment of modular designs, its guidelines were to have a direct bearing on later methodologies. These considerations, in particular the notion of module independence and the idea of a central control module directing subordinate modules were to provide the impetus for the development of functional decomposition in program design [17].

In retrospect, it can be seen that such early and significant concepts as hierarchical structuring, the principle of divide-and-conquer and modularity, formed the basis of past and present programming methodologies, which are considered next.

2.1.3 Programming Methodologies

The Structured Programming Revolution

The start of the seventies saw a considerable amount of literature vigorously propounding a programming philosophy commonly referred to as "structured programming". Its origins lay in Dijkstra's earlier works and specifically in a paper entitled "structured programming" [10] in which he stated the following:

> "program testing can be used to show the presence of bugs
> but never their absence"

As a consequence of this self-evident, yet at the same time alarming maxim, his primary concern was (and still is) program correctness. From the outset, he perceived that the difficulties involved in proving the correctness of programs were such that, "unless measures were taken, the amount of labour involved might well (will) explode

with program size" [ibid]. To consider the measures required involved him addressing the question: "for what program structures can we give correctness proofs without undue labour, even if the programs get large?" [ibid]. Before turning attention to what those structures are, some consideration needs to be given to the advice, stated in Dijkstra's much quoted article "Go To Statement Considered Harmful" [18], that transferring control to labelled points should be avoided. This advice led to the misconception that a program without goto statements is necessarily structured. The rationale for restricting the use of GOTOs was (paraphrasing Dijkstra) to shorten the conceptual gap between the static program and the dynamic process so as to make the correspondence between the program text and the process taking place under its control as trivial as possible. Furthermore, Dijkstra warned against the practice of converting programs with goto statements into programs without goto statements, because it would lead to programs which are as opaque as their originals.

The structures proposed by Dijkstra to ease correctness proofs were such that they restricted sequencing of control to specific forms of concatenation, selection and repetition that possess modular characteristics (i.e., single entry and exit). The factors that contribute to the ease of use of these structures in correctness proofs are:

(i) they minimise the mental gap between the static program text and its dynamic process because their progress can be characterised by a combination of textual and/or dynamic indices (the former describes the place in the text for successive actions whilst the latter is associated with a 'repetition number' for repetition structures)

(ii) standard correctness proof propositions for each structure type can be formulated.

Structured programming, as a design technique in which program correctness is either self-evident or can be proved formally or rigorously was elaborated by Dijkstra in the classic monograph "Notes on Structured Programming" [19]. He suggests that program structure should be derived from hierarchical decomposition of the problem into sub-problems; hence the program produced is a hierarchy where intermediate levels consist of abstracted components which are defined in terms of "what they do" rather than "how they do it". A number of examples are used to illus trate this "step-wise" decomposition process and provide an "introspective exposition" of the methods Dijkstra had hitherto unconsciously applied.

The conceptual tools required to "understand" - in Dijkstra's sense, to prove the correctness of - a program are : Enumeration, Induction and Abstraction. The first tool, enumeration, is used to understand sequences of statements including selection statements. This means that, in practice, these statements can be understood/proved by giving consideration to each execution path; therefore it is an adequate tool provided the number of statements is small. In order to understand repetition constructs and recursive procedures, it is necessary to use mathematical induction. Its use in program correctness proving is similar to proving properties about integers or recurrence relations in number theory. Abstraction, the third tool, is probably the most powerful in program design; by using this the designer is able to concentrate on relevant properties of the problem and ignore irrelevant ones. For example, procedurisation allows us to synthesise the details of how a process works into an abstracted form specifying what it does; defining new data types allows one to manipulate the objects they described as abstract entities.

The application of structured programming produces levels of

conceptualisation in which each refinement represents some implicit design decision. Wirth [20] strongly stressed the need not only to make these decisions explicit so that "the programmer be aware of the underlying criteria" [ibid] used but also "to consider the the existence of alternative solutions" [ibid]. His guidelines in the step-wise refinement process are:

> "to decompose design decisions as much as possible, to untangle aspects which are only seemingly interdependent, and to defer decision which concern details of representations as long as possible" [ibid].

Moreover, he suggested that this refinement of program description into subtasks should be accompanied by a parallel refinement in the description of data that may be necessary for communication between sub tasks.

The qualified success of applying structured programming principles, reported by Aron [21] in an experiment described as "the superprogrammer project", provided the impetus for using these ideas in a production environment on a large-scale information retrieval project which involved over 80,000 lines of source code. The findings of this experiment revealed that the productivity of progammers increased five-fold and that the rate of detected errors produced by principal programmers was approximately one per man-year-effort [22,23]. Observations from introspective experiments [24], rather than quantative evidence from controlled experiments, such as "an experiment in structured programming" [25] supported the arguments expressed in favour of structured programming, which are, for example, that it facilitates the intellectual task of handling size-induced complexity and of proving design correctness.

Structured programming undoubtedly provides a framework for disciplined design, and there is some evidence supporting claimed benefits. However, it would be somewhat short-sighted to suggest

that its principles provide a completely adequate recipe for the production of correct and clear programs. Indeed, the detected error in Henderson's solution to the so called "telegram problem" [25] was in fact a direct result of not perceiving an appropriate level of abstraction. Moreover, the authors retrospectively observed that data concepts were obscured and recommended that these be elaborated in much the same way as algorithms. This could be seen as a step towards Jackson's emphasis on the role of data structuring in program design [26]. A similar criticism can be made of the undetected error in Naur's solution of the 'line edit problem' [4], in which the need for the data concept 'word' as a basis for structuring the design had not been identified. Both these examples illustrate the main weaknesses of structured programming, which are: the lack of specific decomposition criteria in formulating levels of abstraction and the absence of evaluation mechanisms to be used in the decision-making process for determining the best decomposition from a number of alternatives. In terms reflecting the incisive spirit of Occam's razor, we can say:

"whilst we have the knife, we do not know how to carve"

One of the primary objectives of most, if not all, disciplined programming methodologies is to achieve a modular design. However, the manner in which problem decomposition into modules is performed can often introduce a variety of complexities. The method of sequencing control as advocated by the structured programming school enables programs to be modular in terms of control flow - inter-connectivity of control between program components is reduced, thus eliminating the production of programs with a "spaghetti-like" structure. However, this restriction provides no guarantee of achieving modularity in terms of flow of information between modules. Current programming methodologies based on structured programming can

be viewed as attempts to augment its basic principles with additional design criteria. These criteria minimise information flow to preserve a "separation of concerns", i.e., decomposition is performed in such a way that "we don't lump concerns together that were perfectly separated to start with" [27].

Two schools of thought, the 'data structure' school represented by Jackson [26] and Warnier [28], and the 'data flow' school represented by Yourdon and Constantine [29], Myers [30] and Stevens [31] will now be considered. The former school considers it essential to base program design on the logical structure of data, whilst the latter school emphasises that program structure should be based on functional decomposition of the problem. For the purposes of the discussion, it is sufficient to focus attention on Jackson's work as representative of the data structure school and Constantine and Yourdon's work as representative of the other.

Structured Design

Structured Design, as defined in [29], "is the process of deciding which components interconnected in which way will solve some well-specified problem". The definition clearly recognises design as being a problem-solving exercise in partitioning and organising the components of a program. It aims to ease implementation, in particular testing, maintenance and modification, by structuring programs so that each program component corresponds to some "well-defined" piece of the problem, and the relationships between program components reflect existing relationships between parts of the problem. This strategy ensures "independence of modules", which can be seen as an alternative implementation of Dijkstra's notion of

"separation of concerns". Moreover, module independence originates from consideration of modularity [32] and criteria for good design [33]. These considerations culminated in Constantine's paper [17], from which the basis of the methodology is taken.

The methodology synthesises the program design concepts of modularity, hierarchical decomposition, levels of abstraction and design evaluation with systems theory notions of structural organisation and inter-connection of components. Indeed, Constantine [29], having acknowledged the influence of Dijkstra's works makes, the following statement of Emery's work [34] on systems theory:

> "From it I gleaned the essential concept of intercomponent coupling and firmed my commitment to a systems-theoretical view of the universe."

Therefore, not suprisingly, Structured Design views a program as an organised composition of aggregates and components, and uses the conceptual, linguistic and notational tools of systems theory in the statement of the methodology. More importantly, information flow is of primary consideration in Structured Design because it is not only used for orientating the design process, but is also at the heart of the measures used for design evaluation.

The steps applied in the design phase of the methodology are :

(i) Depict the problem as an information flow model by identifying the major data transformations; represent this model as a data flow graph, involving linear chains of processes, known as a "bubble chart";

(ii) Identify the afferent (importing) and efferent (exporting) data elements. This step leaves some transformations in the middle, which are termed as "central transforms";

(iii) Represent the information flow model as a hierarchy of modules with their imports and exports so that a

controlling module when activated will perform the entire task by calling upon Afferent, Central and Efferent subordinate modules;

(iv) Repeat steps (i) - (iii) for abstract subordinate modules.

There are no specific rules for structuring the data flow diagrams. However, central to this process is the perception of suitable levels of abstraction in data flow. The identification of the afferent section involves tracing the input stream from its primitive form to its highest level of abstraction. Similarly, finding the highest level of abstraction in the output stream determines the efferent data section.

Structured Design shares with structured programming the same, rather impractical, guideline for producing "good" decompositions in terms of modularity, namely, the perception of "appropriate" levels of abstraction. However, in contrast, it not only emphasises a specific a priori orientation for structuring - that of data flow - but also provides design evaluation mechanisms which are applied a posteriori. These evaluation mechanisms are directly related to the notion of module independence. A set of modules is said to exhibit a high degree of independence - in Structured Design terms, exhibit functional independence - if they satisfy two complementary characteristics, namely, minimal coupling and maximal cohesion. Coupling, or inter-module dependence, is a measure of the strength of association between a module and its external calling environments; cohesion or intra-module dependence is a measure of the degree of association within a module.

Structured Design's rationale of functional decomposition of the problem as series of procedural modules characterises it as a solution-oriented methodology [35]. Furthermore, it makes no major break with traditional modular programming but merely refines

pre-exisisting concepts of modularity. In contrast, the data structure school, as exemplified by the methodology to be consdidered next, is problem-oriented because it seeks to ascertain pertinent relationships in the problem and to transfer them to the data to be processed. Moreover, it infers that functional decomposition should not be carried out because function is implicit in data.

Jackson's Approach

The rationale behind Jackson's methodology can simply be summarised by the structuring principle that program structure should match problem structure. The method of achieving this is to base program structure on the logical structure of data. Consequently, program modularity reflects data structure rather than data flow. Furthermore, whilst the data flow school views a program as a hierarchy of functionally decomposed processes, Jackson relegates functional considerations to a later stage and promotes in its place the activity of modelling the real world. His fundamental design principle is firstly to produce an abstract model and then consider the functions required. The second step is to implement the abstract model. In common with most current methodologies therefore, Jackson preserves a separation between design and implementation, thereby absorbing those aspects of development that relate to producing an abstraction of the real world into a design and those that relate to realisation of the model into implementation.

Jackson argues that, for programs whose structure is based on problem structure, there will be no difficulty in associating the primitive operations required by the program with the components of that structure. This is because an abstract model of the problem environment (i.e., the program) perceives the real world through the

medium of its data structures (e.g., files). Jackson's approach therefore reflects the well-known principle that "data structures steer algorithms" [19,20]. The program therefore consists of operations concerned with manipulating data structures (i.e., reading and writing) and those operations directly concerned with the task to be performed (i.e., elementary actions required to perform the function). The claimed benefit of such an approach is that the program produced is easier to understand and modify because of the correspondence between problem and program structure.

The methodology employs the principles of structured programming in that levels of abstraction are expressed using the composite forms of sequence, selection and iteration. However, in relation to structured programming, Jackson says that it is insufficient merely to build programs from "structured" constructs - the crucial problem is to decide how these constructs should be fitted together and on what basis the structuring should be performed. The basic steps involved in Jackson's method are:

(i) Consider the problem environment and use this to describe the logical structure of each input and output data stream (using special structure diagrams);

(ii) Form a program structure based on the designed data structures;

(iii) Define the task to be performed in terms of elementary operations;

(iv) Allocate each operation to its appropriate component in the program structure;

(v) Determine the necessary conditions to control execution of selection and iteration components;

(vi) Translate the completed algorithm into Jackson's schematic logic (a special program design language) or the chosen target programming language.

Stage (i) is the creative phase because it involves the designer in abstracting that which is relevant to the problem requirements (i.e., perceiving the problem's logical structure) or, in structured programming terms, perceiving "appropriate levels of abstraction". The next stage involves finding components in the input data stream that, when processed, will produce components in the output stream. Combining these structures yields a single program structure. The methodology, however, provides no means of ensuring the necessary completeness of the operations list in stage (iii). The notion of "appropriateness" is at the heart of the design evaluation mechanism in Jackson methodology. It is applied at stage (iv). It can be stated as follows :

> if processing a data component X involves the primitive operation p, then p should be allocated to the program component corresponding to X.

In addition to his basic method, Jackson provides not only the means for the recognition of "structure clashes" (where the input and output data streams are not in correspondence and hence cannot be combined into one program structure) and "backtracking" (where the serial nature of the input stream prevents an a priori selection), but also standardised methods for their resolution. These allow the match between program structure and problem structure to be preserved. Structure clashes are modelled by communicating processes that can be implemented in a variety of ways, ranging from programs communicating via intermediate files using read and write operations, to concurrent processes that communicate via resume commands. Backtracking is modelled by assuming the validity of one of two possible outcomes of subsequent processing, with the proviso that it may prove necessary to reject the assumption, thereby admitting the validity of the other outcome, and hence having to "backtrack".

2.1.4 Recent Developments

It is instructive to compare the recent directions taken by the 'data-flow', 'data-structure' and 'structured programming' schools. The 'data-flow' school have adapted their work for systems development, systems analysis and specification [36,37] to provide a system design methodology. The methodology shows how to obtain a structured specification from the user requirements and how to use Structured Design tools to produce a system of programs. Furthermore, they have addressed themselves to administrative aspects of project control, planning and management [38,39]. These works depict methods that can enhance the productivity and effectiveness of a software engineering project; for example, Semprevivo [40] provides a set of practical guidelines for analysing, evaluating and improving team performance. The school aims to integrate the variety of structured tools and techniques already used for analysis, specification, design, coding, testing and maintenance with management guidelines and controls to yield a comprehensive methodology that covers every stage of the software engineering cycle.

Similarly, the data structure school have adapted their ideas to system design. For example, Orr [41] has married Warnier's ideas on program design with those of data base design to form a structuring tool for logical analysis, design and development of systems. Similarly, Jackson [42] has produced a development methodology which specifically addresses itself to system design and is not simply a "front-end" to his principles of program design; instead, it is a methodology in which the tools developed for program design are diffused throughout the systems development procedure.

Whilst the data-flow and data structure school have focussed attention on presenting guidelines for good designs and mechanisms to evaluate design quality, the structured programming school in

contrast has concentrated on correctness proofs. A major criticism of all three approaches is that there is an a posteriori application method for design assessment. However, advocates of structured programming have recognised this weakness of proving/evaluating the correctness of completed programs. Indeed, Jones [43] recommends a step-wise scheme in which, at each level of decomposition, proposed realizations are proved. Moreover, Dijkstra [1] not only recommends that development and proofs should proceed hand in hand, but that correctness conditions should steer program development. He has proposed a formal discipline in which, as Coleman [44] points out:

> "Dijkstra's guarded commands constitute a calculus for program development such that if the rules of the calculus are followed, the correctness of the resulting program is guaranteed".

The long term perspective for program design is difficult to visualise with any great assurance particularly with the increased interest in concurrency and new models for the semantics of computations, both of which require architectures that depart radically from the classical Von Neumann model underlying sequential procedural languages. There is an urgent need, and one which is likely to remain in the near future, to develop program design techniques for concurrent programs for the variety of multi-processor architectures [45] that support concurrent processing. The primary concern is developing methods to overcome problems of communication and synchronisation of component processors. Such stucturing methods as parallel composition of communicating sequential processes [46] and guarded commands [47] provide some of the necessary tools to increase our understanding of concurrent program design methods.

Attention has recently been focussed on the re-emergence of an alternative form of programming - that of functional programming [48,49] - in which the semantic model is applicative rather than procedural. Examples of applicative systems are Church's lambda

calculus [50] and McCarthy's pure Lisp [51]. In such systems, the notion of an algorithm is specified in functional terms (i.e as mapping from one set to another) and programs are built by combining functions using "functional forms" [48]. As Backus points out, a functional approach has many advantages over the conventional approach; these include a concise mathematical description of the underlying model and the fact that both programs and correctness proofs can be expressed in a languge that has the same associated algebra [ibid].

2.2 Empirical Considerations

2.2.1 Introduction

Many researchers in programming have been motivated by a belief that their recommendations will aid the programmer's task and therefore improve the quality of programs produced. Whilst the contributions made by "expert" programmers have been, to paraphrase Shiel [52], an unholy mixture of mathematics (e.g., Dijkstra 1968), literary criticism (e.g., Kern 74), and folklore (e.g., Brooks 75), their recommendations have been, in the majority of cases, couched in human factors terms. These recommendations have taken the form that a particular aspect of programming practice will make the programming task either easier, or faster, or less error-prone etc. Despite the authority and vigour with which these expert recommendations have been made and their common-sense appeal to our intuitive notions of programming, they do not constitute a scientific basis for acceptance but need to be empirically tested. Indeed, experimental evaluation can not only be a useful and powerful tool for assessing such proposals but can also provide evidence augmenting the contributions

of practitioners and experts in the field. Therefore, the temptation to accept experts' proposals without evaluation must be resisted.

Many researchers consider that Weinberg's classic work "The Psychology of Computer Programming" [2] was the catalyst for arousing a much-needed interest in human factors investigation generally. In particular, it was directly responsible for most of the investigations on the psychology of programmer team organisation [53,54,55]. The thrust of initial experiments in programming, and to a lesser extent current works, was in the vein of establishing whether a particular product or practice was in some sense better than others. For example, one of the earliest contributions was Sackman's experimental investigation [56] in response to the then current debate on the relative merits of time-shared and batch processing environments. The primary force responsible for the increased volume of work within the last decade has arisen from the debate caused by the structured programming movement with its radical ideas on programming practices and language constructs. This debate has provided experimental researchers the opportunity of empirically evaluating various claims made by proponents of the philosophy. Therefore, interestingly, there has been a parallel increase in both structured programming ideas and experimental work in programming. The impact of experimental results on language and software designers is difficult to assess. Indeed, views differ considerably. For instance, Sheil's article [52] is highly critical of the experimental techniques used and of the "shallow view of the nature of programming" held by experimental researchers; he argues that "the computing community has paid relatively little attention to the results". In contrast, Green [57] cites Embley's paper [58] (in which a proposed new construct is subjected to both empirical evaluation and formal analysis before agreeing upon a final version) as a hopeful sign of things to come.

As yet, it is too early to gauge the impact of experimental work on an area that is constantly undergoing radical change.

The two possible empirical evaluation paradigms available to researchers are observational and comparative experiments. Both types involve testing a relationship known as the 'null hypothesis'. This hypothesis asserts that there is no relationship between the independent variable, which is the variable under investigation and therefore the one the experimenter manipulates, and the dependent variable, which is the variable that is affected and therefore the one on which measurements are performed. A crucial aspect of designing an experiment is to ensure that the effect on the dependent variable is attributable to the independent variable under investigation. In order to achieve this, it is necessary to introduce experimental controls to constrain other independent variables that may affect the outcome. It is precisely because these controls are absent in observational experiments that there are a number of reservations about results obtained from them.

The simplest form of observational experiment is introspection and is probably the basis of many past recommendations, for example Dijkstra [10]. A variant of this rather subjective method, used with considerable success by Simon and Newell [59] in their pioneering work on problem solving, is protocol analysis. Traditionally, this technique involves recording individual subjects "talking aloud" about the task they are performing. The recorded speech transcription is divided into lines known as protocols. This technique has seen relatively little use in programming experiments, notable exceptions being Brooks [60] and Miller [61]. However, as Shneiderman [62] points out, whilst this technique can be "worthwhile when the subject is a capable sensitive programmer, since important insights may be obtained", there is no guarantee about similar behaviour of other

programmers.

Another method of gathering information using the observational paradigm is the case or field study. Examples include Knuth's empirical study of FORTRAN programs [63] and the earlier cited New York Times project [22]. The rationale behind this approach is that gathering large volumes of data can yield something significant. However, the lack of experimental controls provides no assurance as to the reliability or generality of the results obtained.

Before reviewing the scope of previous research concerned with human factors in software engineering, it is necessary to explain briefly the basis and the details of the scientific method as used by most, if not all, of the reviewed work. In the most general terms, the scientific method is to observe a system in order to gather knowledge. Therefore, in many cases, scientific investigations must initially start with observational experiments which provide the basis for subsequent hypothesis-testing comparative experiments. It must be made clear at the outset that the commonly held belief that knowledge obtained using the scientific method is unquestionably true because it is objectively proven or derived in some rigorous way, is a misconception since the method is based on the inductive principle. This principle can be stated as:

> If, for a wide variety of conditions, a hypothesis is confirmed by a large number of observations and, of all those observations, none refute the hypothesis, the latter is held to be universally applicable for those conditions.

The validity of making this inductive leap is a controversial issue of philosophy. One of the most simple and amusing illustrations of the dangers of this leap is Bertrand Russell's story [64] of the "inductivist turkey". The turkey observed that on arrival at the farm it was fed at 9 a.m.. However, as a good inductivist, it gathered a number of observations under a variety of conditions that confirmed

the initial observation and that led it to the obvious inductive inference about the pattern of being fed . Alas, the consequence of this inference proved to be disastrous on the morning of Christmas Eve. There have been a number of responses attempting to resolve this problem. One view as characterised by Feyerabend[65], who suggests abandoning the scientific method. Others believe that it is possible to provide probability measures associated with hypotheses and that each scientific theory is the best explanation available at that time, accepting that it may be necessary to revise the theories in the light of new observations.

The discussion so far constitutes only a partial account of the scientific method, because it is a process involving not only induction but also deduction. Once theories have been derived from observations of the system being studied, these theories can then be used to predict or explain the behaviour of the system using deductive reasoning. In summary, the fundamental cycle of the scientific method is:

(i) Record sufficient observations for varying sets of conditions of the model under investigation;

(ii) Formulate hypotheses to explain the observations;

(iii) Empirically evaluate the significance of these hypotheses;

(iv) Derive a theory or model from these hypotheses;

(v) Perform controlled experiments to evaluate model accuarcy;

(vi) Deduce behaviour hypotheses for the model;

(vii) Repeat from (i).

The above schema is a general one, and it is necessary to consider the issues involved in tailoring this methodology to suit the needs of human factor investigations in programming.

2.2.2 Methodological Issues

Having accepted at the outset that programming is a complex form of human problem-solving behaviour, it may seem tempting to consider what psychological theories of problem-solving behaviour have to offer. Unfortunately, as Green [57] points out, "Psychology does not have a general theory of thinking and is not likely to have one in any reasonable time to come". Sheil [52] observes that "although some psychological theory is very suggestive, it usually lacks the robustness and precision required to yield exact predictions for behaviour as complex as programming".

The need to establish a suitable experimental methodology was recognised by Weissman [66] and Shneiderman [67] nearly a decade ago. Since then, there has been little progress, with some notable exceptions [68,69,70]. Furthermore, as Moher and Schneider [71] the authors of one of the few recent papers on the problems of experimental research in software engineering observe, whilst "the literature contains numerous references to the use of experimental methods, there are few references on investigations into the methodology itself" [ibid]. At present, the enormity of the problems caused by the absence of an experimental methodology is such that "the study of experimental methodology is well beyond the scope of a single research problem" [ibid], and that, furthermore, experimental methodological considerations in programming constitute "an entirely new research area which will require the attention and energy of many researchers over a long period" [ibid]. Because of this absence, researchers investigating intuitively based claims of expert programmers have, in many cases, made methodological decisions that are, ironically, based on intuitive grounds. The review of experimental work that follows is not intended to be a comprehensive

survey of the literature (for such a treatment, see Shneiderman [62], Software Metrics [72]) but concerns itself specifically with the methodological issues central to programming experiments and the controls necessary for such experiments to be effective.

The aim of comparative behavioural experiments in programming is to create an environment in which subject behaviour can be observed and analysed effectively. Devising such environments obviously neccesitates the selection of suitable subjects, suitable materials that will yield the desired effect and the application of appropriate measures to analyse the effect produced. Therefore, the methodological issues at the heart of this type of experiment relate to a judicious choice of subjects, materials and measures; see [68,69].

Subjects

There are two primary concerns in the selection of subjects, according to Brooks [69]. First, the sample chosen should be representative, that is, the observed behaviour of the sample should be characteristic of the population under consideration. Second, the individuals in it should be relatively homogeneous as regards characteristics other than those under investigation, so as not to influence the results obtained. The reason for insisting that these requirements be satisfied is that, when an experimental sample is sub-divided into groups for differing treatments (i.e, the different procedures whose effects are to be measured), it is essential that any significant results obtained for any group are attributable to the treatments and not the characteristics of the subjects in that group. A priori, it is not always possible to know all the subject characteristics that will influence experimental results for any

programming-related task, although, in practice, for a given task it may be possible to determine which subject characteristics will introduce an experimental bias. For instance, in an experiment investigating the effect of particular programming practices, differences in such factors as intelligence, discipline studied, and level of education, could introduce an unwanted bias and therefore measures would need to be taken to control their effects.

One aspect of designing a "good" experiment is to minimise the effects of those subject characteristics that are responsible for experimental bias. There are various well-established techniques which reduce the effect of between-subject-variations; see [73,74,75]. These techniques include :

- random assignment of treatments;
- the use of "matched pairs", in which participants of an experiment are matched on some important characteristic; the consequence of this is that no group has a disproportionate number of biased subjects;
- a "within-subjects-design" where all the subjects undergo all experimental treatments.

In the case where the parent population exhibits a large degree of heterogeneity, the two desired goals of representativeness and uniformity become contradictory because, as Brooks [69] points out, whilst a sufficiently large sample size is required to ensure the former, the greater the sample size, the greater the variation among individuals.

There is some evidence of identifiable heterogeneity amongst subjects performing programming-related tasks. In Sackman's work [56] investigating the relative merits of time-shared and batch processing, variations in performance were observed as high as 25 to 1 across experienced programmers. Miller's observational study with novices

[76] yielded differences ranging from 4 to 1. Much more recent work investigating the behaviour of experts and novice problem-solvers, such as [77,78,79], reveals that there is a qualitative, rather than quantitative, difference between the two groups with regard to organisation of information and types of strategies applied. Contrary to our intuitive notions, the two earlier results taken together seem to suggest less variability amongst novice subjects. The more recent results, however, are in accordance with our intuitive notions, showing that experts are a more cohesive group in that they use organising principles and strategies that are domain specific (i.e., specific to the problem domain being investigated).

Methodologically the variability as found in Sackman's and Miller's works implies that careful consideration needs to be given to the sample composition of subjects. Ironically, the established practice in the vast majority of behavioural experimental investigations is to use restricted groups of subjects (usually undergraduate students). Indeed, as Weinberg [80] succinctly comments:

> "Whereas psychology may be the psychology of college freshmen, the psychology of programming could easily become the psychology of programmer trainees."

This apparent consensus over the proper sample composition of subjects (that is on the use of undergraduates) is based on convenience rather than any methodological criteria. Indeed,.where subjects have not been first-year students, the lack of agreement among researchers is well illustrated in Moher and Schneider's article [71]. They observe " that subjects have ranged from those with no prior computer experience to highly trained professional programmers". Futhermore, they cite Miller's work [76], in which subjects with no previous computing experience were asked to write sorting programs in a subset of BASIC, as an example of the use of naive subjects. Miller claims

the use of these subjects can lead to the detection of characteristics that have not been influenced by the effects of factors such as training and experience. In contrast, Moher and Schneider cite Young's findings [81] on programming errors, which revealed that the strategies used by experts and novices were radically different; for example, novices eliminated all the errors with the same degree of diligence whilst experts eliminated superficial errors with greater rapidity.

The implication of experimental investigations with novices suggests little justification for assuming that their findings are applicable to experienced programmers. However, comparative experiments involving both types of subjects need to be performed before such an implication is verified. Some researchers have attempted to design experiments so that the effect of variation in subject characteristics is brought under experimental control and have tried to conduct experimental investigations in such a way as to reveal the class of subjects to which their findings apply. An obvious and tempting way of controlling the effect of variability in subject characteristics is to use subjects that are undergoing similar training. However, as Brooks [69] argues, the use of intermediate programming classes can in some cases be problematic. He cites Shneiderman's work as an example which shows that significant differences can sometimes be attributed to relatively short differences in experience.

One possible way of ensuring that results obtained are representative of the parent population under consideration is to replicate experiments. This approach has been successfully adopted by researchers at the MRC SAPU unit [82]. They performed experiments with novice and expert programmers in such a way that findings could be compared for both groups of subjects. Their work is a long-term

investigation of the ease with which subjects can read, write and debug programs using different styles of conditional constructs [83,84]. In their exploratory experiments [85,86,87,88], naive subjects were chosen because interest was centered on occasional computer users rather than experienced programmers. It was considered that individuals in the latter group would be unlikely to have the same learning history, or that they might have preconceived prejudices about a particular style. For comparison purposes an experiment was conducted [82] using experienced programmers as subjects. The results regarding readability and debugging were found to concur with the exploratory experiments using novices.

The technique that is most effective in systematically controlling individual differences in performance between experimental treatments is the within-subjects-design, which has been used in a variety of studies [82,89,90,91]. The use of this technique is well illustrated by Love's experimental work [91], in which the primary objective was to show that controlled experiments can be designed to help to improve coding practices. The experimental aim was to investigate the effect of program structure on program understanding. The treatments in the investigation were complexity of control flow (at two levels: simple or complex) and paragraphing of source code (also at two levels: present or absent). Two groups of subjects differing in levels of experience were used. Experimental materials consisted of four Fortran programs written in four different forms corresponding to the two different levels of the two treatments. The experimental procedure consisted of randomly assigning each subject to one of the four groups, each of which received exactly the same set of programs to study and recall. Hence, each subject received both levels of the two treaments. The advantage of this design was that it enabled the investigation to measure the effect of two other

independent variables that could influence the results, namely, level of experience and sequence of programs. Its major disadvantages generally are that it involved the preparation of large amounts of material and, more importantly, that it could lead to subjects getting bored because of the number of experimental tasks they had to perform.

In summary it must be acknowledged that many researchers were, and still are, forced to use undergraduate students as subjects. In many cases, because of cost constraints, the use of professionals is impossible. However, the burden of proof still lies on the experimenter to show that the results obtained are representative of the population under consideration.

Materials

The second of the methodological concerns - the choice of experimental materials - is only one factor relating to a broader category, namely, that of "experimental environment" (i.e. that which encompasses all the available stimulus). As Moher and Schneider [71] point out, behavioural researchers have long realised that differences in results can often be attributed to a variety of factors in the experimental environment. Amongst the environmental factors that investigators need to consider, in their opinion, are:

- the choice of experimental materials;
- the physical setting in which programmers work, so that this can be reflected in the experimental setting;
- the different types of incentive (whether money, or the satisfaction of knowing the aims and subsequent achievements of the research, or being reassured that experimental results will not reflect course grades), so that these incentives can be used in a manner that ensures consistent performance of

subjects;

- various ways of presenting experimental instructions (i.e., whether in oral or written form, or whether presented informally or formally)

An illustration of the effect of the last factor is Weinberg and Schulman's investigation [92] of programmer performance; this revealed that small differences in statement of objectives can be responsible for large differences in results. Their work demonstrates that experimenters need to specify the goals of the experiment clearly, otherwise subjects will simply set their own goals that may not coincide with the experimenter's intention.

The main concern in controlling unwanted bias in the experimental stimuli lies with the choice of material used. There are two issues relating to this choice. Firstly, the material should allow the experimenter to elicit any existing differences in treatments; secondly, the effect of these differences should be attributable to these treatments. When considering the effects of subject variation, it was seen that these could be controlled by the use of a number of standard techniques. However, when choosing experimental material, the controls required for counteracting possible bias will vary from experiment to experiment.

Empirical investigations into programming language features provide examples of the types of material-choice problem encountered by researchers and their attempts to overcome the latter. These investigatons have used material that includes natural language [76], small sub-sets of a programming language [85], complete languages [93] and a special purpose query language [94,95]. The use of "micro-languages" (i.e., where a language comprises of only those operations and syntactic features that are under investigation) is advocated by Sime et.al. [85] in their work on different types of

conditional constructs. They consider that such languages allow researchers to focus on the specific issue being investigated, thereby avoiding any bias due to differences in subject training. Obviously, not all questions concerning language design are amenable to the use of micro-languages; its primary use is in comparing single linguistic features. Indeed, Gannon et.al. [93] point out that when it is necessary to investigate the interaction of language features, then the latter must be evaluated in the context in which they are used. Their work involved an experiment in which subjects wrote programs using two block-structured languages that differed with regard to nine specific features under investigation. These features included the use of the semi-colon as either a separator or as a terminator, and either automatic or requested inheritance of environments. Both Green [82], and Gannon [93] advanced a clear rationale in the choice of experimental material for detecting existing differences and made a reasonable case for their findings. However, both works have been the subject of criticism by Sheil [52]. He argues that the former work does not systematically control unwanted sources of variations in the experimental material, whilst his critique of the latter work questions the effectiveness of such an approach for yielding a clear interpretation of the results.

The measures taken by Sime et. al. [68] to control unwanted bias included devising a scenario that involved writing a series of cookery instructions for a mechanical hare. The hare responded to these instructions, fed to it in the form of edge-punched cards, by lighting lamps in its ears or sounding a buzzer. This simple scenario meant that subjects needed little training to adapt to the physical setting of the experiment which was an advantage over the conventional set-up of writing a computer program when (as in their case) using naive subjects. The presence of what Sime et. al [68]

term as "almost necessary effects" pertaining to the experimental material is a further factor that could produce an unwanted bias by increasing the already large between-subject-variance that is present in programming tasks. In Sime et. al's experiment [85] comparing IF-THEN-ELSE with the GOTO (i.e., conditionals involving an explicit transfer of control), an example of these effects is difference in program length, i.e., number of symbols needed and the amount of physical space occupied. Their solution was to provide subjects with a joystick pointing to a dictionary, so that the time spent by subjects in actually putting symbols into the programs was minimal. However, they considered that effects due to spatial differences were an important part of the comparison. A further problem in the same experiment was whether the provision of indentation would produce unwanted bias. Realising that, in general, it is not possible to indent a language with an explicit transfer of control so as to make its intended elaboration clear (as is the case with a nested language), they had to "decide whether indenting the nested language means providing an artifical prop for the subjects, or is merely taking full advantage of the structural features of the language in a realistic way".

Measures

The final methodological concern is the choice of measures. Human factor investigations in programming have used a variety of experimental metrics that seems to have resulted from a combination of necessity and a carte-blanche application of the principle "to measure is to know". Most experimental researchers would claim that their choice is based on necessity. However some concern has been expressed as to the relevance of some of the metrics in contributing to the

understanding of the program design process [57,69,96].

Software Science [97] is, to paraphrase Yeh [98], a unified and coherent field in which attributes of a computer program, such as implementation efforts, clarity, structure, error rates, language levels, etc. can be derived from metrics based on intrinsic characteristics of the program itself. Such metrics measure what Shneiderman [62] terms as the 'logical complexity' (i.e., the complexity due to control flow) of a program. These include: functions of frequencies of operators and operands in a program [97], the knot count [99] and McCabe's cyclomatic number based on graph theory [100]. Such metrics have the obvious advantage of facilitating automatic computation of measures from the program text, and the gathering of quantative evidence that readily lends itself to hypothesis-testing methods. Experimental studies reveal a high degree of association between attributes such as programming time [101], number of bugs [102],program clarity [103] and their proposed Halstead metrics. Investigations by Curtis et.al [104] using Halstead and McCabe metrics reveal that "these metrics appear to assess psychological complexity primarily where programming practices do not provide assistance" (i.e., they measure the difficulty in understanding programs which have been written in an "unstructured" manner). Such experiments exhibiting high correlations between factors and their proposed metrics therefore can offer useful quantative evidence. However, because these measures are based on intrinsic properties of the program, they take no account of the interaction between the program and the programmer.

Although there are a variety of metrics, the effect being measured in most cases has been the ease with which programs can be constructed and/or the ease with which existing programs can be understood. Experimentation involving program construction tasks

usually takes the form of comparing two groups of subjects: a control group and a group undergoing the treatment being investigated. The metric most commonly used for determining the effort required to develop a program is the time taken to write it. Some of the difficulties that can arise in using time as a metric have been noted by Brooks [69]. An obvious problem is identifying the time spent on aspects of the task that are not relavant to the investigation so that the former can be either eliminated or minimised. Brooks [ibid] suggests that time measures should be supplemented by evidence from other measures such as the number of debugging runs performed and the ratio of total number of recalled lines to program size. In addition to ascertaining the required times, problems can occur because time metric distributions are often skewed. This bias can be corrected using standard statistical transformations. For example, Sime et.al.'s data [85] resulted in a positively skewed distribution of times which they corrected through a logarithmic transformation. Other program-construction metrics relating to the ease with which a program can be constructed involve functions of errors made; for example, the investigations by both Sime. et.al [85] and Gannon et.al [93] used the frequency and persistency of errors as alternative metrics.

One of the earliest investigations on program "understandability", or what Weissman [66] termed "psychological complexity", proposed three measures of understanding. These related to subjects' effectiveness in: "hand-simulating" (i.e dry-running) programs, filling in blanks in a paragraph describing the program, and a subjective measure of how well subjects felt they understood the program. An obvious problem with Weissman's use of the first measure is that hand-simulation of a program can be performed on a statement-by-statment basis without knowledge of it s overall structure. Therefore, the decision to use a question/answer task to

obtain a second measure placed greater confidence in the investigation. Indeed, Weissman notes that "although hand-simulation as such is not a valid measure of understanding" [ibid], nevertheless "both reading and hand-simulation are important components of understanding a program" [ibid].

A commonly used technique for measuring program comprehensibilty is the use of 'memorisation-and-recall'. This technique, as used by Shneiderman [105] involves the subject reading a program and then re producing it as accurately as possible in every detail. The rationale for this work is based on Simon and Chase's work on chess [106], which suggests that experts have large amounts of organised knowledge and use high-level organisation principles. By analogy, Shneiderman hypothesised that for two forms of a program (executable and shuffled), experienced programmers would be able to re-construct the executable form with greater rapidity. Evidence from Shneiderman's experiments supports the use of memorisation-and-recall as a metric for measuring program quality and programmer comprehension. This technique has also been used by Love [91] to investigate the effect of complexity of control flow and indentation on program understanding. He bases the experimental rationale on Craik and Lockhart's theory of memory [107], which suggests that the probability of recalling information is dependent upon the depth of processing undertaken. Although there are a number of ways in which memorisation-and-recall can be applied, it s use as advocated by S hneiderman and Love is appropriate essentially for small programs. A suitable variant, as suggested by Brooks [69], for large programs would be to ask subjects to reconstruct a program that is close as possible to the original.

Finally, whilst it is desirable to conduct the "ideal experiment" (i.e., one in which unwanted bias due to between-subject-variation, non-uniform characteristics in experimental material and/or

inaccuracies in metrics, is negligible) so that the results obtained can be attributed solely to the treatment under investigation. In practice this is extremely difficult to achieve when investigating the complex tasks involved in programming. The options are to choose either:

- what Green [57] describes as the utopian solution, that is, "Once psychologists have taken the wrinkles out of a theory of thinking, programming can be treated as a special case and it will be obvious how to make it easier", or
- to conduct experiments as methodologically precise as is practically achievable so as to "chip away" at the problem under investigation.

2.3. Conclusion

The discussion presented in this chapter has considered practitioner recommendations on program design from a human-factors perspective and methodological issues appropriate to experimentation in program design. From this, objectives for suitable research can be identified, and these are presented in the following summary:

The proponents of structured programming view the design process as a complex problem-solving activity. Moreover, they believe that the use of cognitive tools such as stepwise refinement, hierarchical structuring, levels of abstraction etc., help to make the development task "easier". There are strong arguments in favour of this view from a problem-solving perspective because the overall approach in structured programming embodies the well-established problem-solving technique of problem reduction. However, whilst there is evidence for the benefits that are claimed for this method of design, it would be wrong to regard it as a panacea for designing programs. There is an obvious neccessity to investigate the effect on the program design

process of applying, on the one hand, structured programming principles, and on the other, practices incorporating those principles but involving more specific decomposition criteria.

Whilst this investigation acknowledges that the evidence obtained using the scientific method is not irrefutable, it does, however, take as axiomatic the view that using this method can provide a probability measure of the observation being representative of the system under investigation, so that the latter's significance can be assessed. Moreover, a model or theory based on the results from such observations then constitutes a proposed explanation of the behaviour of the system under investigation. The research was faced with the problem of applying the broad principles of the scientific method, rather than a suitably designed experimental methodology. However, the unwanted bias introduced because of this problem can be controlled by judiciously augmenting the scientific method with guidelines based on methodological decisions made in previous empirical investigations. Therefore, it was decided to make effective use of such guidelines so that an increased level of confidence could be placed in the results obtained.

In conclusion the specific research objectives were to investigate:

(i) the nature of problem decomposition strategies used in program design;

(ii) the factors related to these strategies;

(iii) the factors affecting these strategies;

(iv) the relationship between these strategies and errors made.

3. Report of Investigation

3.1 Experimental Context

The aim of this chapter is to describe the specifics of the investigation. Before these are detailed, consideration is given to two important aspects: first, the identification of the context within which experimentation was performed, and hence within which the research results are to be interpreted; second, the description of the experimental methodology employed - in particular, the assumptions made and the steps taken to provide a methodology tailored to the needs of the investigation.

Several factors contribute to the experimental context. The most significant include: the population under investigation, the physical setting and the size of problems to be investigated. Ideally, it was felt desirable to conduct the investigation so that the results:

- applied to a large cross-section of the programming community whose members' characteristics varied considerably with regard to ability, experience, training, etc.;
- were obtained from an experimental environment which closely resembled the physical setting within which programmers work;
- related to "realistic" programming problems;

In practice, however, the experimental context was considerably constrained because of the limits imposed by time, resources and availability of subjects.

At present, empirical research (whether conducted in an industrial or academic environment) on a complex problem-solving activity such as program design (an area in which there is a scarcity of empirical investigation), can have little hope of arriving at a satisfactorily complete solution. However, there is a difference

between investigations in industrial and academic environments. The former often involve large-scale experiments, whereas the latter are frequently constrained to small-scale experimentation. Therefore, academic studies are open to the often-voiced criticism that such studies deal with "toy", rather than "life-size", programs and use subjects from academic, rather than production environments performing tasks in artificial settings. The reason for this disparity between academic and industrial investigations is often attributable to availability differences in finance, resources and subjects. Some academic studies have attempted to counteract the effects of this disparity by such means as co-operating with commercial organisations (for example, the work by Hammond et. al. [108] used professional system designers), and developing courses in which subjects are encouraged/expected to participate in experimentation [55].

The circumstances surrounding this research were that no provisional arrangements had been agreed either for industrial co-operation (i.e. there were no commercial organisations who had agreed to supply volunteer subjects and/or make available resources) or for financing of programmers to act as volunteers. In addition, at the academic establishments where students were willing to be participants, there was no precedent for their being used as experimental subjects, which ruled out any serious possibility of organising experiments in students' free time. Moreover, because subjects' tutors were concerned about the possible disruption to their course of study, it was agreed that experimentation would be performed during one tutorial/practical session (i.e. a period of approximately fifty minutes) per term.

These above-mentioned circumstances dictated that:

(i) unpaid subjects be used;

(ii) since subject availabilty was restricted to infrequent, short

periods, the size-related complexity of the problems to be used as experimental material should be relatively small;

(iii) experimentation had to be performed in test-type conditions due to the necessity for adequate numbers of students to produce individual solutions to the same problem or sets of problems

Nevertheless, it was considered that despite these practical constraints, an experimental context in which computer-science undergraduates were asked to construct programs for "small" problems under experimental conditions, could constitute a meaningful research framework. This view could simply be justified on the principle that because of the scarcity of research in program design any contribution - even with severe constraints - could be a worthwhile one. However, a stronger case can be advanced:

- the chosen subjects represent a significant proportion of the programming community, as well as being potential future professional programmers;
- the specific objectives of the research meant that a number of important factors affecting strategies used in problem decomposition, other than problem size, could be investigated;
- the provision of a reassurance that subjects were participating in an experiment rather than a test, together with the "reward" of being allowed access to the outcome of the research, would help to motivate subjects, thereby overcoming possible adverse effects associated with the artifical setting of experimental conditions.

The overall direction that any programming research project using the scientific method follows, is an investigative path combining exploration and evaluation. In an approach where the former is emphasised, the intention is to "discover" from a human-factors

standpoint what features of a program makes its specification, construction, verification etc. more tractable. However, in an approach where emphasis is on the latter, the investigator posits, prior to experimentation, certain factors which are believed, or assumed, to be of interest; the aim then becomes to "measure" the effect of those factors. Investigations on programming style and language design by Sime et. al. [85] provide examples of the former approach, whilst the latter approach is exemplified by Weissman [66] and Gannon [93]. The present study chose essentially an exploratory path, albeit confined within an evaluative framework investigating the nature of, and the factors affecting, problem decomposition strategies. An approach with the alternative emphasis would have involved assuming that such factors as: problem size, programmer ability, design methodology, length of training etc. affect problem decomposition; and the validity of these assumptions would then be tested. This approach would make it easier to identify evaluative experimental hypotheses. However, because of the scarcity of empirical research on program design it was considered that initially the exploratory approach would prove more illuminating. One of the consequences of this decision was that an initial pilot experiment had to be performed so that the broad objectives of the research could be transformed into specific experimental aims and hypotheses.

The investigative methodology devised was based on the established principles of the scientific method. Its exposition, which follows, introduces concerns relating to experimentation in general before specific issues relating to the current research, are presented. The introduction on evaluation paradigms, statistical test procedures and choice of decision statistic provide the background necessary for assessing the type of conclusions that can be drawn from, and the confidence placed in, the results obtained.

Consideration of the choice of subjects, materials and measures completes the discussion.

3.2 Hypothesis Testing

Both observational and comparative evaluation paradigms involve testing an experimental hypothesis using statistical test procedures. However, as Leach [109] points out, they apply in two different situations, which differ in the degree of control applied and the type of conclusion that can be reached. The difference between the two paradigms depends on whether the variable under investigation is an attribute or a treatment. An experiment where the variable being investigated is an attribute (i.e., a property of the subjects participating in the experiment and therefore not under experimental control) is said to be an observational study. In contrast, if the variable being investigated is a treatment (i.e, is assigned to experimental subjects and therefore under experimental control), the study is said to be comparative. Therefore, observational studies need only involve one group of subjects in which the effect of the attribute under study is measured. However, a comparative experiment, in its simplest form, involves two or more groups of subjects in which each group is assigned one of the possible types of treatment.

The distinction between observational and comparative studies is crucial with regard to the conclusions that can be reached. With the former, one may establish only a correlative measure (i.e., the variables exhibit a measured degree of association) whilst, with the latter, one may also infer a causal relationship (i.e., the effect of the dependent variable is attributable to the treatment). Furthermore, the choice concerning the type of study has to be made at the start of the investigation because it effects both the designing

and performing of experiments. In some cases, for practical reasons, it may be difficult or impossible to carry out comparative studies. For example, studies investigating differences in intelligence due to gender or race must of neccessity be observational since the latter are attributes of the subjects. Therefore, differences in results obtained from such studies cannot infer a causal relationship which is directly attributable to gender or sex, because such differences may be due to other factors such as environment or culture.

A statistical test procedure is a decision mechanism, founded on the principles of mathematical probability theory, that transforms the experimental hypothesis and the set of collected observations by means of a decision statistic into an outcome that accepts or rejects that hypothesis. The similarity between the mechanics of a statistical procedure and the reasoning used in a court of law provides, as Leach [109] notes, a useful analogy to explain the force of argument used in the former. At the start of the experiment, we assume that there is no relationship between the variables in the experimental hypothesis (we assume the innocence of the accused). Therefore, the researcher (prosecuter) must aim to demonstrate on the basis of collected observations, the validity of the experimental hypothesis (must produce evidence that establishes the guilt of the accused) at some level of significance (beyond reasonable doubt).

The standard procedure for carrying out a statistical test is as follows:

(i) Posit the validity of the Null Hypothesis (i.e., assume that there is no relationship between the variables being investigated);

(ii) Choose the decision statistic to be used;

(iii) State the level of significance;

(iv) Compute, using the decision statistic chosen, the probabilty of

obtaining the observed sample, this probability being denoted by p;

(v) Reject the null hypothesis (and accept the experimental hypothesis) provided the computed probability exceeds the significance level.

The level of significance is the smallest probability value for the collected observations that would result in the null hypothesis being accepted. In theory, the value chosen is at the discretion of the experimenter and may vary from experiment to experiment depending on the degree of assurance required. However, in practice, the sole purpose of experiment is to verify the desired hypothesis and demonstrate the occurrence of an effect. Therefore, the smaller the significance level, the greater the confidence that an effect has occurred. The most frequently used value for the significance level in experimental psychology, so that the researcher can conclude that the observed effect is not the result of chance variation, is 0.05. However, many studies adopt the convention of using the value of the computed probability p, asserting that the result is signifcant at that level; for example, as Sheil [52] points out, effects have been reported as high as $p < 0.2$. There are obvious dangers in choosing "appropriate" significance levels after computing p. There is, however, an even greater danger, as Sheil warns, in choosing significance levels in such a manner, because the computed value for p is an estimate that an effect has occurred and not an estimate of the size of an effect.

The choice determining the decision statistic employed depends on the observed sample characteristics. These include the underlying nature of the population distribution from which the sample is collected and the type of data collected. The first feature determines whether the decision statistic is parametric or

non-parametric decision statistic, it is necessary to consider whether the type of data will be:

- categorical data, for example when subjects' solutions are classified into two mutually exclusive categories;
- ordinal data, for example when subject performance is measured via the number of correctly placed instructions in a program outline;
- continuous data, for example when a subject's "perception difficulty" is measured by the time taken to complete a task.

Two further features that determine the required decision statistic are the independence of data(i.e., whether measurements influenced each other) and the number of samples.

3.3 Methodological Specifics

The investigation can be viewed as two sets of studies, each one being associated with a particular programming problem and involving three separate experiments. Initially, for each of the two studies, it was prefered to perform an observational experiment where the overall aim was to discover something about the general nature of the strategies people use in program design. It would have been preferable to carry out the remaining experiments in both studies in a comparative manner; however, practical constraints (i.e., the absence of a control group of subjects) made this impossible for one of the experiments in the first study. The discussion that now follows details the methodological issues involved in choosing subjects, materials and measures.

3.3.1 Choice of Subjects

Two previously mentioned factors concerning training, and payment (i.e., subjects were familiar with step-wise refinement and were willing to be unpaid volunteers), restricted the population from which subjects could be chosen to that of computer science students trained in the broad principles of structured programming. In choosing subjects from this population, two differing criteria, dependent upon whether the experiment was observational or comparative, were adopted. For the observational experiments where the general aim was to discover those elements of the design process that are common to programmers, the criterion was to "cast the net fairly wide" so as to gather as much information as possible. In contrast, the comparative experiments had specific aims of establishing differences for a particular aspect of program design between two or more groups of subjects; this meant that the overall criterion was the need for homogeneity of subject characteristics.

The techniques considered in order to control the effects of between-subject-variance in relation to such factors as length of training, nature of training, intelligence etc. were : within-subjects-design, matched pairs and random assignment of treatments. Use of the first technique meant devising a number of problems (equal to the number of treatment levels) that were of equivalent complexity so that each subject could undergo all experimental treatments. The obvious difficulties in assessing complexity equivalence of programs ruled out this possibility. The second technique would have involved the pairing of subjects in relation to characteristics that might contribute to subject variance. In theory, this could be achieved by matching on length or course of study undertaken by subjects and course grades attained. However, this was only partly possible because, in practice, it was not known prior to experimentation which of the students would volunteer.

Therefore, the homogeneity assumption was based on choosing subjects from the same course (i.e., matching differences due to length and type of training) as well as randomly assigning treatments (i.e., assuming that effects of other factors such as skill levels would be randomly distributed across treatments).

3.3.2 Choice of Experimental Material

The most significant factor in choosing experimental material is deciding the type of task to be performed. Two possible choices are program construction and program comprehension tasks. The former type was considered more appropriate to the needs of the investigation. The material to be used for each experiment consisted of a problem specification where the task to be performed broadly involved designing a program for the problem so specified. It was considered essential that these problems each should possess more than one distinct solution in order that the experiments might yield evidence concerning the different design strategies that subjects employ.

Another important factor which influenced the choice of experimental material was the decision to restrict the scope of the investigation to problems whose general characteristics were similar to each other. The reason for choosing this approach was that it would have the advantage of reaching more detailed conclusions that - albeit derived from a limited problem arena - could with circumspection be extrapolated to a family of problems. Furthermore, it was felt desirable that the problems should be fairly "balanced" in their characteristics as this would avoid undue emphasis either on input data content or, alternatively, on processing requirement.

The problems chosen were considered to satisfy the above-mentioned requirements. For the first study, the problem used

was derived from Findlay and Watt's signal problem [3] (specified in appendix 1). Naur's line-edit problem [4] was used as a basis for the experimental material (specified in appendix 3) in two of the experiments in the second study. The four problem specifications (specified in appendix 5) for the remaining third experiment of this study were derived from both the signal and line-edit problems.

Program construction tasks provide an obvious means of investigating the nature of program design, although they have the serious disdvantage that it is difficult to devise comparative experiments involving their use (therefore, they are usually employed in observational studies). The source of the difficulty in the present context was that subjects' strategies were attributes and therefore not under experimental control. The obvious preference for comparative experimentation necessitates devising a scenario in which the variable under investigation is a treatment rather than an attribute. For two of the comparative experiments carried out, suitable experimental material was specially devised, this comprising of outline programs (hereafter known as a process structure cues) and lists of "elementary" instructions. The two process structure cues with their respective lists of actions used for the signal problem are provided in appendix 2, whilst the three cues and lists for the line edit problem are given in appendix 4.

The process structure cues used corresponded to different decompositions of the problem. Each was based on a particular (e.g primitive or abstract) perception of the problem structure. The cues were refined to a level such that a complete program could be obtained by allocating "elementary" actions (elementary in the sense that their functional description needed no further elaboration) to the former. Therefore, these cues in skeletal form (i.e., without the actions necessary to fulfill the processing requirements) consisted of

suitable key-words used to express sequence, selection, and iteration structures, with appropriate conditions for the latter two constructs.

The experimental procedure involved subjects having to construct programs by allocating actions from the action list to their given process structure cues. Hence, problem decomposition became a treatment in the experiment. Furthermore, in order to ensure that the effect of any significant differences could be attributed to the treatment rather than alternative sources of variation, the following measures were considered in developing the cues:

- The key-words used were from the subjects' main programming language (Algol68);
- The idea of labelling the action list so that subjects need write only numbers (say) in the spaces provided in the process structure cues was considered; however, the superficial convenience of reducing the task to "programming by numbers" was rejected on the grounds that it might have caused confusion with program readability and understanding as subjects assimilated the problem and developed the program;
- The positioning and size of blank areas in the process structure cues was such that no implied significance could be attached to them regarding the number, or placement, of instructions. The spacing of blank areas was such that, wherever subjects would have reasonably expected instructions to appear in relation to their knowledge of Algol68 syntax, a spatial area was left blank. In addition, these blank areas between key-words were made equal in size;
- The stylistic rules used regarding formatting and discriminability of key-words, choice of variable names etc. were in accordance with the conventions for program clarity as advocated on their programming courses.

The action list could have been formulated in a number of different ways depending on the elementary actions chosen. Though syntactical form would correspond to Algol68, there were a number of equivalent semantic forms. The list used could either be:

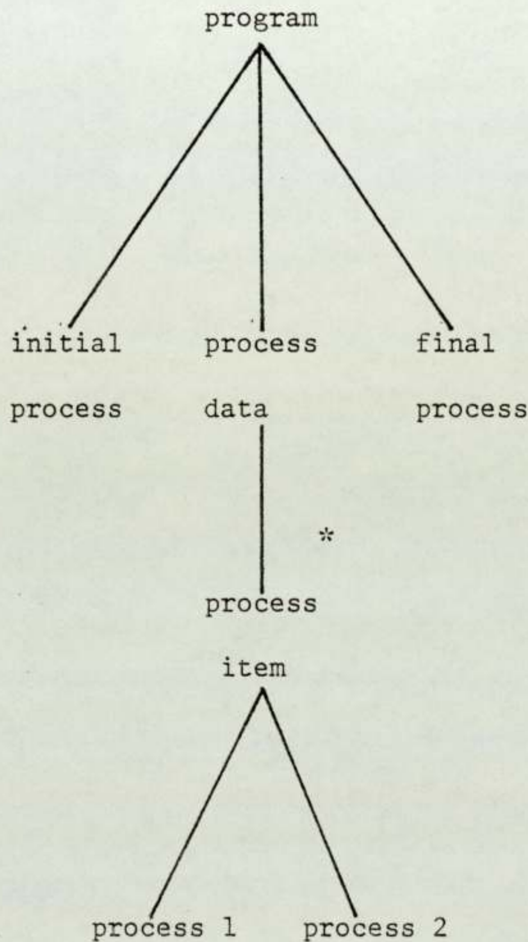
- a "complete set", where the required actions included all possible alternative forms;
- a "sufficient" set, where all the required actions include repetitions for those that would be required more than once;
- a "canonical" set, in which each of the required actions is given once only.

The latter alternative was chosen because it was considered that the first alternative would lead to a lengthy list involving a large number of actions that would not be used and would therefore be a source of confusion. Furthermore, in the second alternative, the actions that needed to be repeated were not dependent solely on the problem requirements but also on the syntactical rules of the language being used.

3.3.3 Choice of Metrics

Deciding upon suitable metrics depends largely on the variable being investigated and the type of task being performed . In the experiments where subjects developed a program from the specification alone, it was necessary to devise an investigative rationale/framework so that subjects' attempted solutions could be analysed. The problems were chosen on the basis that they possessed more than one distinct solution. Furthermore, the problems were such that their solutions could be mapped according to their constituent process abstractions onto one of a small number (e.g two or three) of decomposition paradigms. A first-level solution template for the family of problems

chosen can be represented as a hierarchy of abstract processes (where processes 1 and 2 are components of either a sequence or selection construct) as shown below.



The actual decomposition paradigms for a particular problem can be generated from this template by characterising the processes involved in the hierarchy - in particular, the process pair at the root of the hierarchy, hereafter referred to as the 'characteristic process pair'. By way of example, for the signal problem, two possible decomposition paradigms can be characterised as follows. On the one hand, specifying the characteristic process-pair as 'process vehicle signal' and 'process timing signal' leads to one paradigm. On the other hand, a characterisation which incorporates a process specified as 'process

waiting period', irrespective of the precise nature of the other process, leads to the other paradigm.

In the classification of attempted solutions, it was decided that the process description of the characteristic pair was to be the sole arbiter, so that factors such as syntax/notation used or positioning (correct or otherwise) of "elementary actions" (e.g input statements, assignments etc.) were not considered. In addition, the quantitative metric in analysing subjects' attempted solutions would be the frequency/proportion of solutions based on the different possible decomposition paradigms. It was considered that this metric would effectively quantify subjects' "preference" for a particular paradigm and would therefore be a useful contribution to the investigation.

In two of the experiments, there was a specific aim of investigating the relationship between different problem decomposition strategies and effort required. In these experiments, subjects were asked to produce solutions to specified problems using process structure cues and an action list, and it was necessary to devise metrics to analyse subjects attempted solutions. The effort required was measured in terms of the total time taken to develop a complete program. The rationale for using this measure is that the rate at which subjects are able to perform the task of accomodating actions into cues reflects the effort required to comprehend and elaborate the latter. Moreover, the justification for such a view is as follows. It can be assumed that, in decomposing a problem and designing a program, a subject produces some internal representation of the problem together with a subsequent model of its solution (this is consistent with Greeno's idea [110] of cognitive representation and Hoc's notion of a "systeme de representation et de traitment" [111] in the area of problem solving). It follows therefore, that the ease/difficulty with which subjects comprehend and elaborate cues

will be dependent upon the degree to which the latter "mirrors" their internal model. On this basis, it is reasonable to use this metric to evaluate different problem decompositions with respect to the effort required in their comprehension and elaboration.

A further metric used for measuring subjects' performance was defined in terms of the number of elementary actions correctly located in the structure cue. A better measure may have been achieved by weighting the components, as some were deemed more difficult to locate than others. However, as there were no objective criteria by which such weighting could be carried out, it was accepted that the choice of equal weight would prove satisfactory.

To obtain observational information about the relationship between decomposition strategies and subjects' errors, a metric was specially devised ♀ which measured the error frequencies in subjects' attempted solutions. The analysis required to compute these frequencies consisted of classifying and accumulating subjects' errors made in relation to certain features of the problem requirements, so that each frequency corresponded to the number of errors for a particular feature. More specifically, the chosen features were associated with refinements of the characteristic process-pair. The rationale for this choice was that such an analysis could yield an "in-depth" insight (i.e., one that is subsequent to the first level of refinement and focuses on those sub-processes which require the most elaboration) into possible relationships between strategies and errors.

In the next two sections, details of the plan, execution and analysis of the investigation, involving the two sets of experimental studies, are presented.

♀ (an example of an error frequency computation is shown in appendix 6)

3.4. The Signal Study

The problem specification used in the three experiments of this study possessed several different solutions, a feature considered pertinent to the objective of obtaining insight into subjects' design strategies.

3.4.1 Experiment 1

Aim

The aim of this exploratory experiment was not only to gather observational evidence on the kind of strategies used by subjects, but also to use this evidence to form evaluative hypotheses for further experimentation.

Subjects

The 129 subjects taking part were groups of computer science students attending a number of different educational establishments. Individuals therefore exhibited considerable variation with respect to the following characteristics:

- length of experience (from about 1/2 year to 3 years or more);
- level of training (from pre-university to postgraduate);
- the design notation they employed (this included flowcharts, Nassi-Shniederman diagrams and Algol-style language);
- primary programming language (from Basic to Algol 68).

Materials

Subjects were each supplied with a single-sheet computer print-out of the signal problem as given in appendix 1.

Procedure

The experiment was conducted during a 50-minute class period with different groups over a number of weeks. The subjects were given

instructions verbally by the same experimenter for each session. They were told that they were required to design a program in any design notation or high-level language with which they were familiar. It was emphasised that they were participating in an experiment and not a test, and that they should feel free to seek clarification of any aspect of the problem from the experimenter, though not from each other. They were encouraged to show any development or working carried out in obtaining the program.

Metrics

Various metrics were used to analyse the experimental data, these relating to: subjects' preference for a particular solution type, subject performance and frequencies of errors in solutions. Although several different hierarchical process structures can be identified for the signal problem (see 4.1.2), attempted solutions were categorised into only two decomposition paradigms , hereafter denoted by s1 and s2, this classification being considered adequate to the needs of the investigation. The s1 paradigm corresponds to a solution in which the characteristic process-pair is 'process vehicle signal' and 'process timing signal', whereas the s2 paradigm corresponds to a variety of solutions in which one process in the characteristic pair is 'process waiting period', the precise nature of the other process being immaterial. Representations of these two decompositions, accompanied by corresponding complete solutions, are given in appendix 1. The first metric above was used to evaluate the significance of subjects' preferences. The second and third metrics were used respectively to evaluate the significance of a trend between length of experience and performance, and error frequencies associated with fulfilling certain problem requirement goals .

Results

Data for 106 (90%) of the subjects was analysed, the remaining

subjects having made insufficient progress in producing programs that could be meaningfully analysed. The most illuminating statistic was the frequency of subjects' preferences for solutions based on s1 and s2 ; these were 97 (91.5%) and 9 (8.5%) respectively.

The performance scores (i.e., the number of elementary actions correctly located) ranged from a maximum possible of 9 to a minimum observed of 5. They were grouped into four sets corresponding to different lengths of experience. A table of each set with its mean performance score and frequency is given below:

length of experience	mean performance	group frequency
less than 1 year	6.81	31
between 1 & 2 years	7.02	50
between 2 & 3 years	7.21	14
greater than 3 years	7.55	11

The resulting data consisted of 4 independent samples of ordinal values, and there is a strong intuitive basis for assuming a correlation between experience and performance. Therefore, a Jonckheere trend test [112] was applied.

Null Hypothesis: There is no relationship between experience and performance.

Alternative Hypothesis: Performance improves with experience.

Decision Statistic: Normal approximation to the Jonckheere test with ties.

Significance level: 0.05

Computed probability: $p < 0.005$

Conclusion: Accept alternative hypothesis.

In addition to establishing the presence of this highly significant trend, it was decided to assess the degree of improved performance.

The use of an assymetrical measure of association is appropriate, since the focus of interest was on how length of experience improves performance, rather than the other way round. The value obtained using Somer's delta [113] was 0.23 which may be interpreted as saying that, if two subjects were selected from different sets, there would be a 23% chance that the subject from the more experienced group would perform better.

Before considering subjects' errors, a table is presented below showing frequencies with respect to the progress made as defined by specific "milestones" in the development of an sl-type solution:

Progress made	Freq
Satisfied the first two processing requirements only	18
Satisfied the first two requirements, and attempted the third,	45
Correct solution apart from "final check for longest wait"	30
Correct solution	6

The error analysis centered on problem-requirement features of the signal problem that are associated with the refinement of the characteristic process-pair, and in particular on the two processes involved in fulfilling the third processing requirement, as it is the attainment of the latter that is the main source of complexity with the signal problem. In practice, this involved inspecting solutions to determine whether certain actions had been omitted or misplaced. The relative frequencies (expressed as a percentage of the frequency of solutions in the second category above) of the two processes - namely, 'reset waiting period' and 'check for longest waiting period' - were 53% and 47% respectively. In addition, a significant error measurement for the 106 subjects was the absence of the 'final check for the longest waiting period' in 94% of the solutions.

3.4.2 Experiment 2

Aim

The aim of the second experiment was to obtain evidence as to why such a highly significant proportion of subjects favoured an s1-type solution. One obvious explanation for this preference, which became the investigative hypothesis for this experiment, was that an s1-type solution is in some sense easier to perceive. Since problem decomposition was merely an attribute of the participants in experiment 1, it was now made a treatment. To achieve this, subjects were split into two groups, one being guided or "cued" to s1, the other to s2.

Subjects

The subjects were 20 second-year computer science undergraduates trained in step-wise refinement and Algol68. They were divided randomly into two groups of equal size.

Materials

Supplied to each subject were:

(a) A specification of the problem as for experiment 1;

(b) A skeletal process structure cue corresponding to either s1 or s2 (see appendix 2);

(c) A list of actions necessary to develop a complete program (see appendix 2).

Procedure

The procedure was essentially the same as for the first experiment. Subjects were instructed that, from the materials they received, they had to produce a solution to the given problem by allocating actions from the action list to appropriate positions in the skeletal cue. They were informed that certain actions might have

to be inserted more than once in order to obtain a complete solution, and that no implied significance should be attached to the size or positioning of blank areas in the outline program structure. The start time of the experiment and the finishing times for subjects were recorded.

Metrics

The ease with which subjects comprehended their cues was measured in terms of the total time taken to accomplish the task of developing a program.

Results

A table of times (to the nearest minute) taken by individuals of the two groups in producing completed solutions is given below.

Group s1	8	13	14	14	14	15	15	17	20	21
Group s2	21	22	23	23	23	24	29	29	30	

Note, that of the 20 subjects who took part, 1 in group s1 and 2 in group s2 produced solutions that had to be rejected due to insufficient development. The times taken formed two independent samples of ordinal data and therefore were analysed using a Mann-Whitney test [114] (although informal inspection of the table suggests a significant difference between the two groups).

Null Hypothesis: The two groups do not differ in the time taken to develop a program from the given cue and action list.

Alternative Hypothesis: The group cued to decomposition s2 will take longer.

Decision Statistic: Mann-Whitney

Significance Level: 0.05

Computed Probability: $p < 0.005$

Conclusion: Accept alternative hypothesis

Furthermore, the size of the difference was measured using a Hodges-Lehman estimate [115], which indicated that a subject cued to s2 would take some 9 minutes longer to complete the task.

3.4.3 Experiment 3

Aim

If the signal problem's processing requirements are imposed upon the sequence of input signals, the outcome will be a (logical) data structure corresponding to s2. It would therefore be expected that a population of subjects who had received training in Jackson's principles would produce a significantly greater proportion of s2-type solutions than the participants of experiment 1. The aim of the third experiment was to test this prediction. The obvious course of action would have been to conduct a comparative experiment involving a population divided into two groups, so that one group received training in step-wise refinement and the other group was trained to use a data-emphasis structuring principle. However, practical circumstances precluded comparative experimentation (no subjects who had received both types of training were available), and thus another observational investigation similar to experiment 1 was conducted.

Subjects

The 34 subjects were second-year undergraduates from various disciplines who had chosen programming as a "complementary studies" option. As part of their training, they were required to attend a weekly series of first-year computer science undergraduate lectures on step-wise refinement and Pascal. In addition, they received separate parallel instruction that stressed the need to consider the logical structure of data as a means of obtaining an outline algorithim.

Materials, Procedure and Metrics

UNIVERSITY LIBRARY

As for experiment 1.

Results

Five subjects failed to produce solutions which could be meaningfully analysed. The frequencies (with their percentages) of s1-type and s2-type solutions produced by the 29 subjects were 16 (55%) and 13 (45%) respectively. Comparing this result with that of experiment 1 yielded two independent samples of categorical data.

Null hypothesis: There is no difference in the division of s1 and s2 frequencies between subjects in experiments 1 and 3.

Alternative Hypothesis: Subjects trained in Jackson's principles will produce a greater proportion of s2 solutions.

Decision Statistic : A normal approximation to the Fisher exact test [115].

Significance level: 0.05

Computed probabilty: $p < 0.000005$

Conclusion: Accept alternative hypothesis.

An estimate of the size of the difference using a Somer delta [113] revealed that it was 36% more likely that subjects from experiment 3 would produce a s2-type solution than those from experiment 1. The frequencies of performance values ranging from the minimum observed to the maximum possible for the two groups is given below.

	scores					
	4	5	6	7	8	9
Group s1	2	8	2	2	2	
Group s2			1	3	8	1

The results form two independent samples of ordinal data. Although informal inspection reveals a marked difference between groups, data was nevertheless analysed for significance using a Mann-Whitney test

[116]. Moreover, as an s2-type solution was considered superior to the other because it possesses a greater degree of modularity, a test was carried out for increased performance in the s2-group.

Null Hypothesis: There is no difference in performance scores between the two groups.

Alternative hypothesis: The performance of group s2 is significantly greater.

Decision Statistic: Normal approximation to the Mann-Whitney test with extensive ties.

Level of Significance: 0.05

Computed probability: $p < 0.0001$

Conclusion: Accept alternative hypothesis

In addition, an estimate of the size of the effect was made using Somer's delta [113], which revealed that there was a 78% chance that a subject from the s2 group would perform better than from the other.

One possible contributory factor that was advanced to explain the apparent ease with which s1-type solutions were perceived, was the presence of certain key-words or phrases in the problem wording and the particular combination of processing requirements presented therein. This formed the basis for the second investigative study.

3.5 The Line-Edit Study

It is almost axiomatic that problem wording will influence problem solving and hence the solutions produced. The aim of the first two experiments in this study was to characterise subjects strategies and gather observational evidence regarding the conjecture (from the first study) that certain features of the problem wording would act as cues for decomposition. Furthermore, the problem chosen was considered appropriate to the needs of the investigation because

it contained explicit references to both primitive and abstract problem specification features. In the third experiment, the previously mentioned conjecture was systematically investigated by devising four problems, constructed from the specifications of both the signal and line-edit problems.

3.5.1 Experiment 1

Subjects

These consisted of 36 third-year computer science undergraduates trained in step-wise refinement and Algol68.

Materials and Procedure

These were as for experiment 1 of the signal study except that each subject was supplied with a specification of the line-edit problem as given in appendix 3.

Metrics

As in the observational experiments of the signal study, subjects' attempted solutions were mapped onto decomposition paradigms. The three paradigms (respectively refered to hereafter as L1, L2 and L3) that were judged adequate for classification, in terms of the characteristic process-pair were:

(i) 'build a line of m chars', 'adjust line and then output';

(ii) 'process space', 'process character';

(iii) 'build a word', 'output a word'.

Representations of the corresponding hierarchical process structures with their completed solutions are given in appendix 3.

The error analysis carried out was similar to that of the signal study, the frequencies of errors associated with fulfilling the following problem requirement features being accumulated:

(a) removing successive spaces;

(b) inserting a single space between words;

(c) preventing a space being output before the first word;

(d) preventing a space being output before the end of a line.

These were chosen because their attainment, or lack of, was the source of several mistakes, as noted by Goodenough and Gerhart [117], in Naur's [4] original, and other subsequent published solutions.

Results

Of the 36 solutions, 5 were not classified as they had either not made sufficent progress or did not match one of the three decomposition paradigms. The division of frequencies for the remaining subjects, respectively corresponding to decompositions L1, L2 and L3 were 15 (48%), 6 (20%) and 11 (32%), did not reflect a strong preference for a particular decomposition type. On the basis of interpreting the preference for s1-type solutions in the signal study as an indication that subjects favour solutions based on primitive perceptions, comparisons of frequencies based on primitive (i.e., L1 & L2) and abstract (i.e., L3) solutions were performed.

Null Hypothesis : There is no significant difference between frequencies of solutions based on primitive, as opposed to, abstract perceptions.

Alternative Hypothesis: There is a preference for solutions based on primitive perceptions.

Decision Statistic: Normal approximation to a Binomial test [118]

Significance Level: 0.05

Computed Probability : $p < 0.024$

In addition to the division of frequencies for progress made (i.e. frequencies corresponding to solutions satisfying either all three or the first two problem requirements), the error analysis also involved calculating for each solution type:

- the error frequency for each of the four above-mentioned

features;

- the error percentage, which is the total number of subject errors expressed as a percentage of the maximum possible number of errors that subjects could make the latter simply being four times the group frequency). The results are as follows:

Solution Type	Progress all 3	Progress first two	Errors a	Errors b	Errors c	Errors d	Error %
L1	4	11	3	14	15	15	78
L2	6	0	3	1	2	5	45
L3	11	0	1	1	1	10	30

Since decomposition strategy was an attribute in this experiment, a second controlled experiment was performed using process structure cues to make the variable under investigation a treatment.

3.5.2 Experiment 2

The first experiment in the line-edit study revealed that all subjects' attempts, irrespective of solution type, were based on erroneous decompositions. This was because they had not realized that certain additional predicates were necessar y to ensure that solutions did not contain the errors previously mentioned in (3.5.1). It was considered that the inclusion of these predicates in the process structure cues would unecessarily increase the complexity of the cues and might also be a source of confusion. Hence, the cues used in this experiment were erroneous, in that they were in a form that non-cued subjects might be expected to produce when attempting to solve the problem (as in the first experiment). Furthermore, since the completed solutions presented in appendix 3 are based on these

decomposition cues, they are also incorrect.

Aim

In the signal study, the preference for primitive solutions was attributed to the ease with which primitive decompositions were perceived. In general terms, the effort required to produce a solution is not simply the effort involved in perceiving a decomposition. More accurately, it can be described as the sum of the effort required to perceive a decomposition and elaborate this resulting decomposition into a completed solution. It might therefore be reasonable to assume that in the case of a problem for which there is no strong preference for a particular decomposition type (e.g., the line edit problem), possible differences in times taken to produce a solution may be indicative of elaborative effort required to complete that solution. Moreover, on the basis that the effort required in elaborating decomposition types will vary from one decomposition to another, the following experiment to compare differences in elaborative efforts between groups was performed.

Subjects

The 24 subjects were third-year computer science undergraduates trained in step-wise refinement and Algol68. They were divided randomly into three groups.

Materials

Each subject was provided with : a specification of the line-edit problem, a process structure cue corresponding to either L1, L2 or L3, and a list of actions to develop a complete program from the cue (see appendix 4).

Procedure

As for experiment 2 of the signal study.

Metrics

The effort required to produce a program was measured in terms of the

time taken to develop a complete solution.

Results

Of the 24 subjects that participated, one in each group produced solutions that had to be rejected due to insufficient progress. The data for individual times taken (to the nearest minute) by the three groups is shown in the table below.

Group L1	21 22 23 24 28 33 36
Group L2	15 15 19 19 20 20 22 24
Group L3	14 15 20 20 21 21

The data was analysed for differences between groups using a Kruskal Wallis test [119]. This revealed a significant difference between groups ($p < 0.005$). Moreover, pair-wise comparisons testing for differences between groups using a Mann-whitney test [114] with a significance level appropriate to the comparisons rather than the experiment (i.e., 1/3 of 0.05), yielded the following:

- times of group L1 > times of group L2 ($p < 0.005$)
- times of group L1 > times of group L3 ($p < 0.005$)
- times of groups L2 and L3 do not differ significantly.

3.5.3 Experiment 3

Aim

The aim of this experiment was to investigate the effect of problem specification on problem decomposition by varying the processing requirements and key-words in the input data description. More specifically, the experimental hypothesis was that explicit references to primitive or abstract problem specification features are responsible for corresponding (i.e., primitive and abstract)

decomposition types.

Subjects

53 second-year computer science undergraduates trained in step-wise refinement and Algol68 were used. They were randomly divided into four groups, each group being assigned one of four treatments.

Materials

The four treatments were devised by systematically manipulating two factors, namely, removing the need to fulfill primitive processing requirements, and/or introducing the presence of abstract data items. The four problem specifications (specified in appendix 5), which represent the four experimental treatments, are characterised as follows:

(i) Problem I is the signal problem unaltered;

(ii) Problem II is formed from the data description of the line-edit problem containing references to an abstract data item, namely, 'word' and a set of processing requirements that are equivalent to those of the signal problem;

(iii) Problem III is the signal problem with only the third requirement. The specification removes the explicit presence of primitive processing requirements (i.e., those that correspond to single elementary actions), thereby emphasise the presence of the remaining abstract one;

(iv) Problem IV is formed by taking the data description of the line-edit problem and adding a processing requirement which is equivalent to the third requirement of the signal problem.

The hypothesis to be tested was that the proportion of abstract to primitive decompositions should increase from group solving problem I,

having the smallest ratio, to the group solving problem IV, having the largest.

Procedure and Metrics

As for experiment 1.

Results

The data for 46 (87%) of the solutions was analysed, as the remainder had made insufficient progress for analysis. The frequencies for primitive and abstract solution types for each group were:

Group	Frequencies	
	primitive	abstract
I	11	0
II	10	3
III	7	5
IV	4	6

The data reveals that the preference for primitive solutions can not only be counteracted, but actually reversed, with appropriate cues in the specification. Furthermore, a Jonckheere test [112] was applied for a trend in group order I (II, III) and IV, where the middle two groups were combined as there was no obvious a priori rationale for distinguishing an order difference between them. The test indicates the presence of a highly significant trend ($p < 0.005$).

Summary

In theory, significant results obtained from comparative experiments imply that the treatment under investigation is responsible for the observed effect, provided that other factors were

experimentally controlled. In practice however, this high degree of control was not achievable for two reasons. First, it was difficult to determine prior to experimentation exactly what factors might need controlling. Second, control of between-subject-variance via the within-subjects-design technique would have implied producing programming problems of equivalent complexity. Therefore, a more reasonable, and at the same time more cautious interpretation is that the results from comparative experiments represent a higher degree of association between the treatment and the observed effect than those obtained from observational experiments. Bearing this in mind, the results of the investigation can be summarised as follows:

- There is a marked preference for solutions based on primitive perceptions of problem structure as observed in the first experiments of the signal and line-edit studies;
- There is a difference in effort required to produce solutions based on primitive and abstract perceptions. The second experiment on the signal problem suggests that greater effort is required for solutions based on abstract perceptions, whereas the corresponding experiment on the line-edit problem indicates the reverse;
- The effect of prior training of subjects to look for logical data abstractions produces, as seen in experiment 3 of the signal study, an increase in solutions based on abstract perceptions of the problem structure;
- The results of the last experiment in the line-edit study strongly suggest that the presence of certain key-words and processing requirements in the problem specification can influence the decompostion strategy employed by subjects;
- Observations from the initial experiments in each study indicate that there is a relationship between subject error

frequency and the decomposition strategy employed.

4. Discussion

4.1. Problem Analysis and Design evaluation

4.1.1 Introduction

The first part of this chapter details an analysis of the signal and line-edit problems and their respective possible solutions, an informal "top-down" exposition of which is presented. Initially, each solution is characterised in terms of an "item-type-to-be-processed", hereafter refered to as ITEM. Essentially, an Item is that perception obtained from consideration of input data and/or processing requirements (including output) which becomes pivotal to the subsequent decomposition of the problem. The various alternative solutions to each problem based on different ITEMs are mapped onto characteristic process structure pairs corresponding to the decomposition paradigms presented in appendices 1 and 3; the various characteristic pairs are then refined to obtain process structure hierarchies. With regard to the line-edit problem, modifications needed to produce correct versions of the erroneous solutions given in appendix 3 are also discussed. In addition, all solutions are subjected to a design evaluation based on the notion of "modularity", as characterised by certain stated criteria.

Since one aspect of this discussion concerns design evaluation, it is relevant to consider what constitutes a "good design". At present, there is no universally accepted method of quantitative design assessment, although one frequently stressed qualitative property that is considered necessary for a good design is a high degree of modularity. This notion is, to paraphrase Dijkstra [27], the partitioning of the original amorphous knot of obligations,

constraints and goals (i.e., the problem specification) into a set of "separate concerns" (i.e., levels of abstractions). To arrive at an effective separation of concerns, the levels of abstraction should be "internally coherent" and "externally isolated", or in Structured Design terms "tightly cohesive" and "loosely coupled". More specifically, the implications of these requirements, as pointed out by Liskov [23], are that:

- the combined activity of functions of an abstraction level supports that process abstraction (i.e, the task implied by the process specification);
- each level of abstraction has resources (e.g., data) which it owns exclusively and which other levels are not permitted to access;
- the flow of information between levels should be in the form of data passed as explicit arguments via functions;
- the direction of control flow between levels should proceed from the top to the bottom; i.e., higher level functions may call lower level functions, but the latter are not aware of the existence of the former.

The qualitative assessment of each solution type for the two problems is carried out by considering three design evaluation parameters associated with the corresponding characteristic process pair. These parameters, which readily suggest themselves from the above requirements, are: the process specification/name, the functions performed by the process and its resources. In practice, the approach adopted is to use the above requirements as "benchmarks" to categorise a solution as being either of "high" quality if it satisfies all of the above four requirements, or "low" quality if it violates one or more of those requirements. Additionally, those specific features of the solution that contribute to its categorisation are highlighted.

4.1.2 The Signal Problem

Problem Analysis

For the signal problem, there are two possiblities for ITEM namely: a "primitive" signal or a "chunked" (waiting) period. Both these perceptions inevitably influence the manner in which the input stream is viewed, the processing of which then dominates the elaboration of the decomposition. There are in fact several distinct, but correct, perceptions of the input stream, each with its characteristic process structure pair, the details of which are presented below:

ITEM	Input Stream	Characteristic pair
t*	(v* & t*)* or (t* & v*)*	process vehicle period process waiting period
t*	(v* ! t*)*	process vehicle period process waiting period
t*	(v ! t*)*	process vehicle signal process waiting period
v ! t	(v* ! t)*	process vehicle period process timing signal
v ! t	(v ! t)*	process vehicle signal process timing signal

(v = "vehicle signal", t = "timing signal" and !, & and *

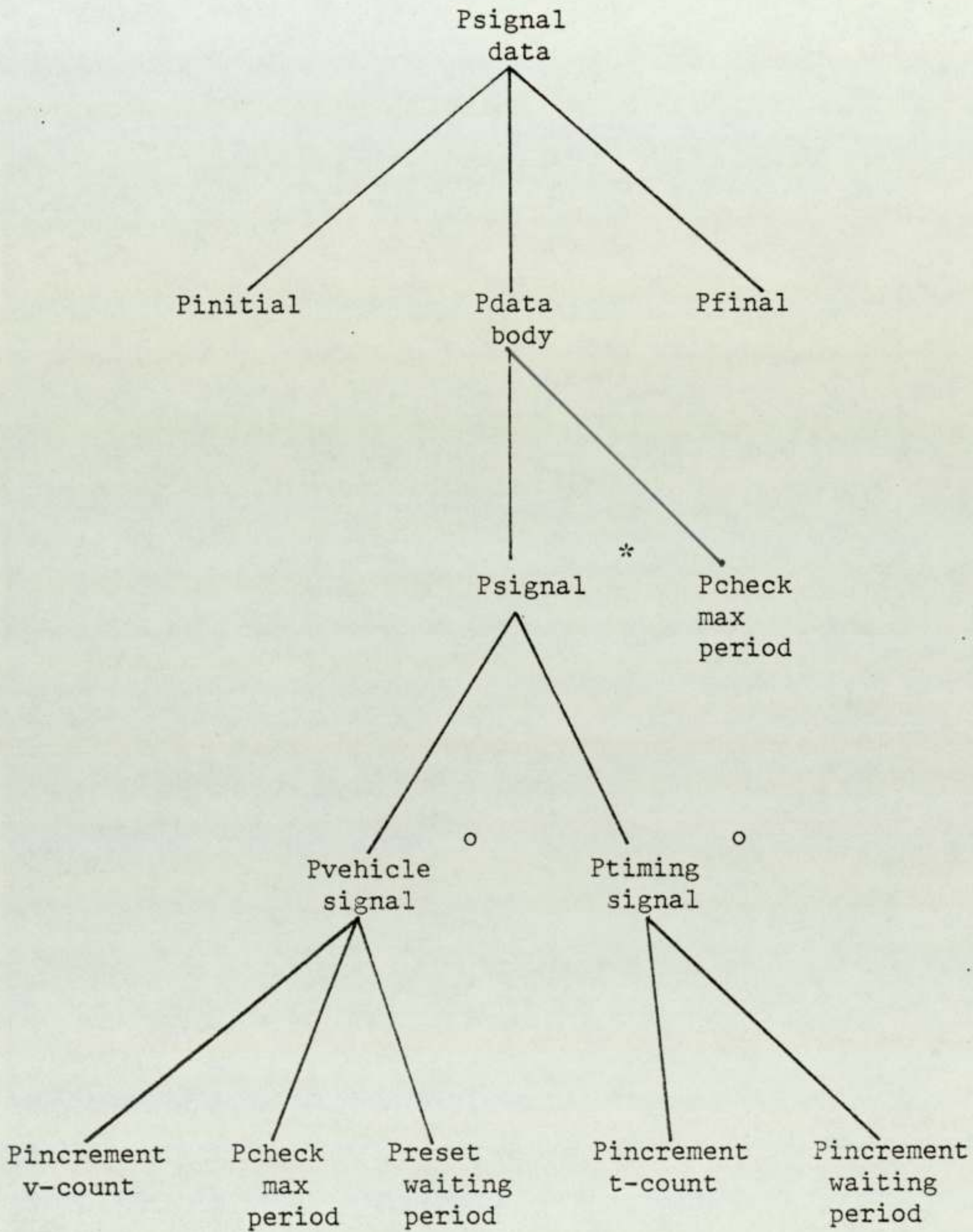
respectively denote unordered alternation, concatenation and unbounded iteration).

The first three alternatives map onto an s2-type decomposition, because of the presence of a t* process which deals with a waiting period, whereas the latter two alternatives map onto an sl-type decomposition. Although the structure t* & v is incorrect, it is worth noting because it appeared in some of the solutions of experiments 1 and 3 of the signal study. Whilst the fourth alternative is interpreted as an sl-type decomposition, it is regarded as a perverse solution of the problem since a v* component bears little relevance to the processing requirements; not suprisingly, it never occured among subjects' solutions

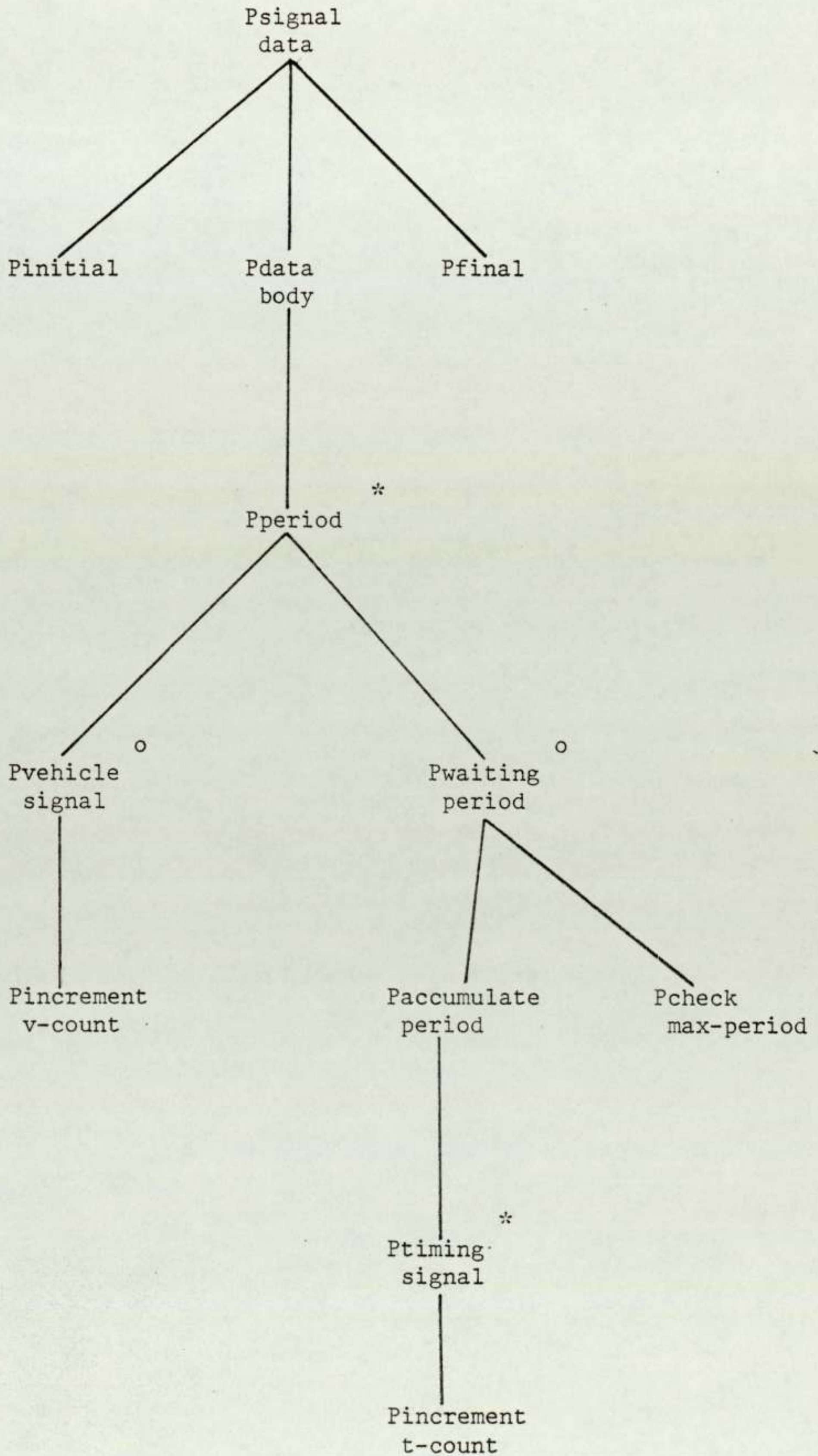
Since the third and fifth alternatives respectively correspond to the standard s2 and sl paradigms, consideration is now given to the refinement of their characteristic process structure pairs. In relation to the fifth alternative, the elaboration of 'process timing signal' is simply two elementary actions, that of incrementing waiting, and total survey times. The elaboration of 'process vehicle signal' is, however, more complex because it involves a sequence of three processes. The first and third are elementary actions that respectively correspond to 'increment a vehicle count' and 'reset waiting period', whilst the second process is composite and involves the 'check for a possible longest waiting period'. This latter component is also needed as part of 'final process', to ensure that the waiting period between the last vehicle and the end of the survey is also compared against 'the longest waiting period'.

The refinements of the process pair associated with the s2 paradigm are such that 'process vehicle signal' is simply the elementary action of incrementing a vehicle count, whilst 'process waiting period' consists of the two composite processes 'process

accumulate period' and 'check for longest waiting period'. This completes the refinements of the two paradigms and representations of their process structure hierarchies are presented below:



sl-type Process structure hierarchy



s2-type Process structure hierarchy

Design Evalauation of s1 and s2 paradigms

The characteristic process pair for an s1 paradigm in terms of its design evaluation parameters are:

(i) specification - process vehicle signal

functions - increment vehicle signal, check max period and reset waiting period

resources - num of vehicles, longest waiting period, waiting period

(ii) specification - process timing signal

functions - increment length of survey and waiting period

resources - length of survey and waiting period

The s1-type is not a high quality design because it violates two of the four previously mentioned requirements for modularity (see 4.1.1). The latter two functions of "process vehicle signal" do not support its abstraction, which violates the first requirement; and the resources of both processes are not exclusively owned, which violates the second requirement. Re-arrangement of the program code so as to remove these violations produces:

```
IF signal = 1
THEN
    (* process vehicle *)
    increment vehicle count;
    set v-arrived
```

```
ELSE
      (* process waiting period *)
      IF v-arrived
      THEN
         check for max period;
         reset waiting period;
         reset v-arrived
      FI;
      increment waiting period;
      increment survey  length

FI
```

This re-arranged version has a resource item v-arrived that is used as a flag to communicate between the two processes, violating the last two requirements. In addition, it highlights the placement of actions that contribute to its poor quality, namely, 'reset waiting period' and 'check for max period'. Similarly, it can also be shown that the 'final check for max period' is a further contributory factor to low quality. In Structured design terms, this solution type is of low quality because it is control coupled (one process commands the other process on what action it should take via the flag variable). From a Jackson perspective, this decomposition type would be deemed inappropriate because it does not possess a 'process waiting period' component to which actions associated with a waiting period such as 'reset waiting period' should be allocated.

The s2-paradigm is of a high design quality because all four requirements are satisfied. The component 'process vehicle' performs a single function that supports its abstraction and exclusively owns a data resource which it communicates as an explicit argument. The

other component 'process longest waiting period' performs two functions (i.e., 'process waiting period' and 'process check max-period'), both of which support its abstraction; its data resources are exclusively owned and communicated as explicit arguments. The feature that contributes to this design being high quality is the presence of the component 'process waiting period', which produces a design encapsulating an abstraction level that preserves an effective "separation of concerns", or from a Jackson viewpoint, reflects the logical structure of the data.

4.1.3 The Line-edit Problem.

Problem Analysis

In order to analyse the line-edit problem in the same manner as the the signal problem, the following observations concerning the former should be noted: that it includes non-trivial input and output streams and that ITEM can be perceived as either a line, word or character. Each of these perceptions defines a different solution; these are presented as three separate cases (s = space character, c = non-space character, n = newline):

Line-driven

The implications for the input and output streams of perceiving ITEM as a line are that there are two possible perceptions for the former, namely: (s ! c)* or (s* & c*)*, whilst the latter's structure is ((s & c*)* & n))*. The characteristic process-pair for the first input perception is 'build a line of m characters' and 'adjust line and then output', whereas for the second it is 'build a line' and 'output a line'. In the first perception, a line-item is visualised

as a simple repetition of characters, which implies that there is the possibility of 'build a line of m characters' encountering a line break in the middle of a word; the second perception, however, visualises a line-item as a repetition of words and therefore 'build a line' produces a repetition of complete words. Interestingly, all subjects' solutions based on a line ITEM corresponded to the first alternative, which can be refined into the L1-type solution presented in appendix 3.

Character-driven

There are three different solutions that are based on ITEM as a character. One possible way of arriving at a program that is essentially Naur's original solution [4] is to visualise the input stream as (s ! c)*, the characteristic process-pair as being 'process space' and 'process non-space-character'. The other two possible solutions have input streams that correspond to (s* ! c)* and (s ! c*)*. For all three solutions, the output stream structure is ((s & c*) ! (n & c*))*, though it is the input stream perception that dominates the initial decomposition. Of the three, only the first alternative can be elaborated to an L2-type solution as given in appendix 3. The latter two are "hybrid" in the sense that some "chunking" (i.e., character grouping) is present. Of the six subjects who produced solutions that were categorised as L2-type decompositions, there was one of each hybrid type.

Word-driven

There are two alternative solutions where ITEM is a word. One solution is based on a perception of the input stream as (s* & c*)*,

with a characteristic process pair of 'build a word' and 'output a word'; the other is based on (s* ! c*)* with a characteristic process pair of 'process spaces' and 'process non-space characters'. The output stream structure for both solutions is the same as in the previous case. Note thus that, from a Jackson perspective, making ITEM a word achieves a degree of perceptual correspondence between the input and output stream structures that is absent in the other two cases. Note further that, the former alternative can be refined to an L3-type solution, whilst the latter never appeared amongst subjects' solutions.

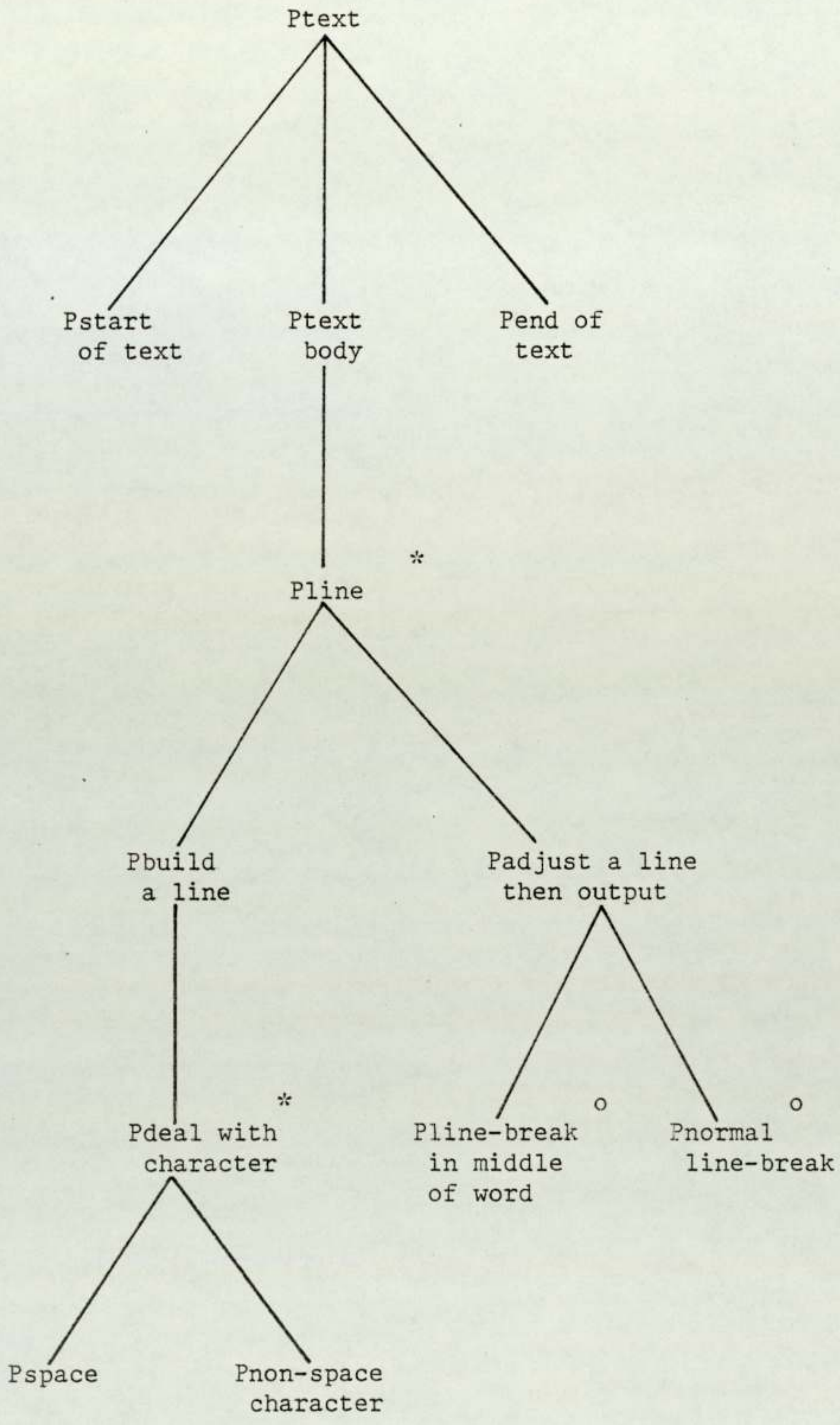
Refinement of characteristic pairs

Having specified the characteristic pairs for the three decomposition types, consideration is now given to their refinements so as to produce detailed process structure hierarchies. Both characteristic processes of the line-based perception, namely, 'build a line' and 'adjust a line and then output', involve composite sub-processes that need considerable refinement. The first process repetitively adds a non-redundant character to the current line, whilst the second process adjusts the line if there was a line break in the middle of a word, and then outputs the line. This latter task is elaborated in terms of two composite processes that need further refinement (as can be seen from the completed solution). However, the process structure hierarchy for an L1-type solution shown in Figure 1 is adequate for the needs of this discussion.

The elaboration of 'process non-space character', a characteristic process of an L2-type decomposition, consists of two elementary actions adding a character to a word and incrementing the size of the word. The refinement of 'process space',the other

characteristic process, is more complex. This is due to the need to distinguish between the cases when a space is either redundant or acts as control character for output, in which latter case a space denotes either the end of a word or a line. The process hierarchy for an L2-type solution is shown in Figure 2.

'Process output word', one of the characteristic processes for a word-based decomposition, distinguishes between the cases in which a word is output on either the current line or a new line, so that the word can be preceded by an appropriate separator. The function of 'process build a word' is to get the next word; this involves two repetitive processes, one skipping over spaces, the other concatenating non-space characters to form the next word. The process hierarchy for an L3-type solution is shown in Figure 3.



Figurel

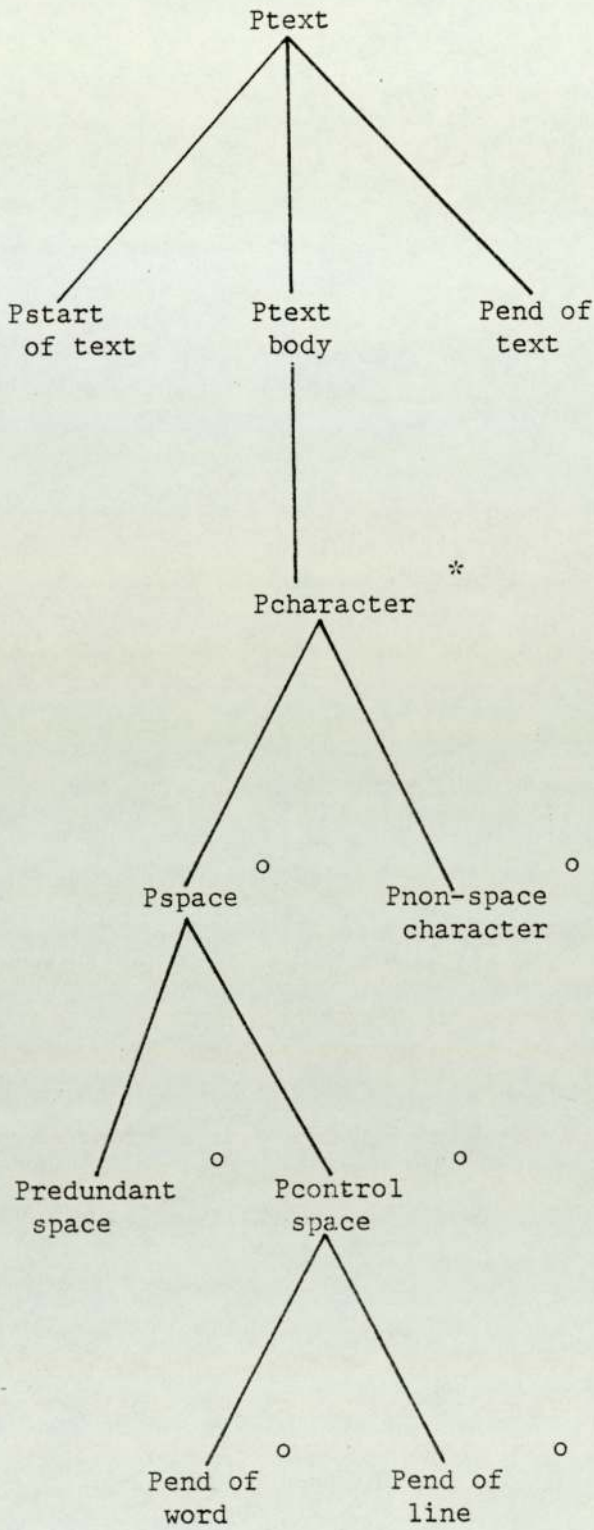


Figure2

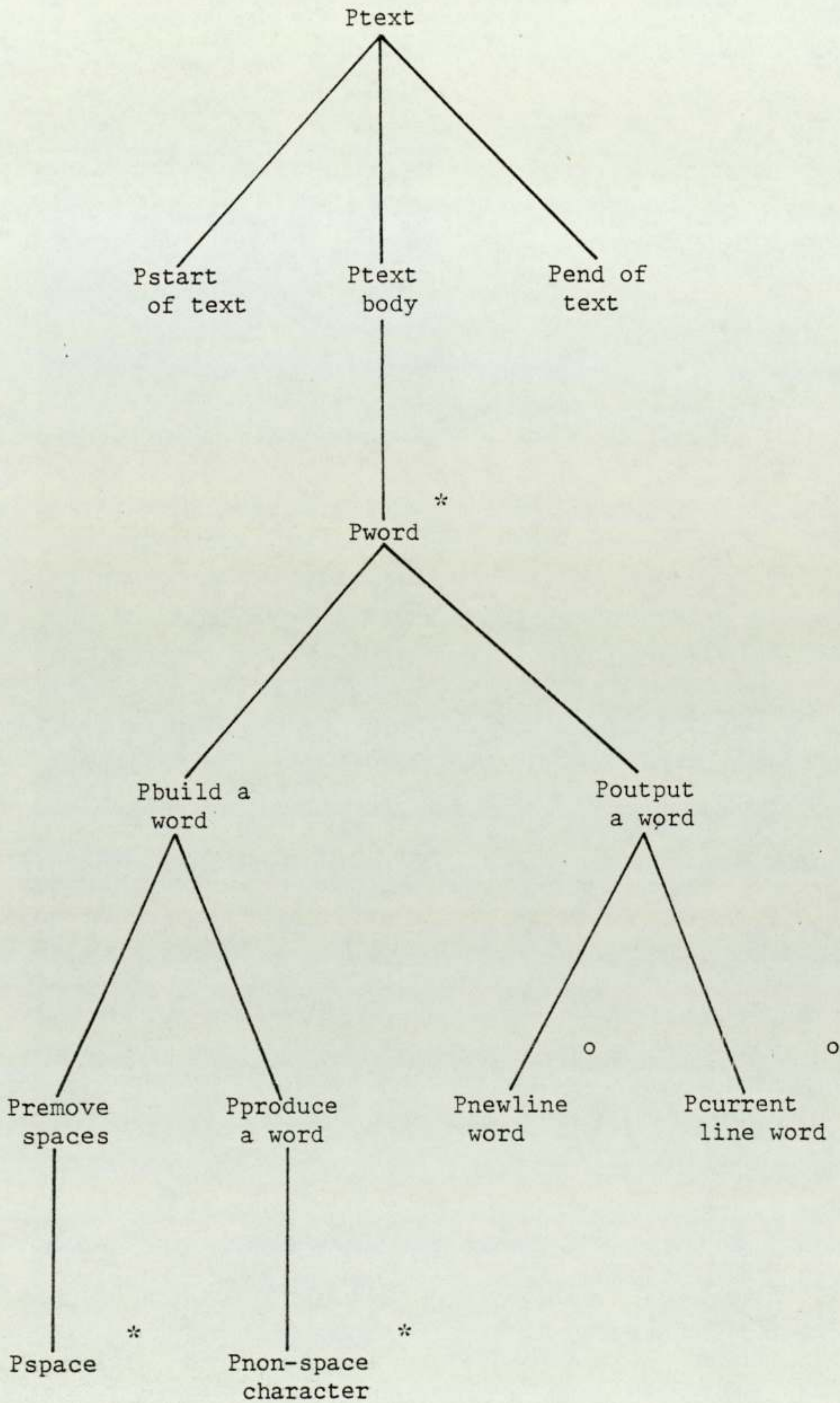


Figure3

Design Evaluation

It is appropriate for the purposes of this discussion to evaluate design quality of decomposition paradigms based on subjects' actual - albeit incorrect - attempts rather than on the correct versions. For the L3-type decomposition, the characteristic pair described in terms of its design evaluation parameters is as follows:

- one process specifies its abstraction as "build a word"; it "imports" character items and "exports" word items via a function that performs only the task specified in its process abstraction;
- the other process, specified as "output a word", imports and exports word items via a single-task function.

The design satisfies all four previously mentioned requirements: first, the function of each process support the its abstraction; second, the resources form an effective "separation of concerns" because 'build a word' conceals the details of processing characters to form words from 'output a word'; third, both processes communicate data as explicit arguments; fourth, control flow is "top-down" and therefore, the design quality is assessed as "good". The presence of the two single-task functions, each of which owns its resources exclusively, is responsible for this evaluation.

The functions performed by 'process space', a characteristic process of the L2-type design, are to remove redundant spaces and to output words. Because the process abstraction (i.e., the task implied in its specification) is not supported by these functions and in addition, there is no effective "separation of concerns" in resources (i.e., words and characters), neither of which is exclusively owned by either process, the design quality is assessed as poor. The major

factors that contribute to this classification lie with 'process space', or more specifically, with the functions of this process.

The characteristic processes for the line-based decomposition, 'build a line' and 'adjust line and then output', violate the second requirement because both processes own character and line (i.e., there is no effective separation of data). Furthermore, the information being passed from the former process to the latter appears to be a single explicit data item. However, closer examination reveals that two parameters are actually being passed, namely, the line and the next input character. The latter item acts as a flag which transmits control from the first to the second process.

Interestingly, from a Jackson perspective, the L3-type decomposition is based on the logical structure of the input data, whilst the other two designs would be rejected as their structure is inappropriate to the problem requirements. From a Structured Design viewpoint, the L3-type decomposition is functionally cohesive, whereas the other two solutions are such that their characteristic processes are control coupled.

Program Modifications

The errors in the solutions presented in appendix 3 are related to the output of a space at the start, and/or end, of output lines. The modifications needed for each of the three solutions are discussed as three separate cases:

(i) L1-type solution

The component controlling the output is:

IF char ≠ space OR prev char ≠ space

THEN

```
        line := line + char;
        line size := line size + 1
    FI;
```

The function of the above predicate can be mistakenly interpreted as detecting the presence of a non-redundant character (which corresponds to the situation of the input character being one of two possibilities: either a non-redundant space or a non-space character). More accurately, its function is simply the removal of successive spaces. To correct this mistake involves two major changes: first, the need to separate the two components of the conditional expression, and second, to introduce the conditions 'line size $\neq$ 0' and 'line size + 1 $<$ m' to ensure that there is no space respectively at the start and finish of lines. With these modifications, the corrected version of the component is:

```
    IF char = space
    THEN
        IF prev char ≠ space
        THEN
            IF line size ≠ 0 and line size + 1 < m
            THEN
                    add space to line;
                    increment line size
            FI
        FI
    ELSE
        add char to line;
        increment line size
    FI;
```

(ii) L2-type solution

The component 'process space' controlling the output is:

```
IF word size ≠ 0 (*non-redundant space *)
THEN
     output word;
     set word size to zero;
     set word to empty string
FI;
```

Since 'output a word' prints a word preceded by a separator, the first word will be preceded by a space. However, this will only be the case for the first line, since, for every subsequent line, the first word will be preceded by a newline character. The remedy chosen by Goodenough and Gerhart [117] is to specify a blank line at the beginning of the output text, which will result in the first word of every output line being preceded by a newline separator. The effect of this change can be achieved, as they point out, either by:

- conjoining the predicate 'line size ≠ 0' with the existing condition that checks whether the current word should be output on the current line or on a newline;
- or by setting the length of the line to "m" (i.e., maximum number of characters on a line) at the start of the program.

A simpler modification that does not change the specification is to nest the condition 'line size ≠ 0' within the condition for a non-redundant space.

(iii) L3-type solution

The nature of the error and the reason for its occurrence are the same as in the previous case, and therefore "guarding" the component 'process output a word' with the condition 'line size ≠ 0' in the same way as above ensures that the first line does not begin with a space.

The approach taken in the above discussion is to consider what

modifications need to be introduced to the code of certain incorrect programs to produce corrected version. An alternative approach is to consider what aspect of the abstract model on which the program design is based is incorrect; having identified and corrected this error in the design, the code is then modified accordingly. The former strategy is in marked contrast to the latter, because it does not make use of the refinement levels produced in the derivation process to trace the source of the error, but simply "patches" the program code. Applying the alternative approach to the line-edit problem and viewing the ITEM as a "word-item", the argument is as follows. The program design model for the L3-type solution processes each word in the same manner. However, from the above discussion it can be deduced that the first word in the text should be processed differently. Therefore, the required perception of ITEM is (s* & c*) & (s* & c*)*, resulting in the program:

```
build word;
print(word);
WHILE word ≠ "*"
DO
    build a word;
    output a word
OD
```

In relation to design assessment of corrected versions, it can be seen that "inserted patches of code" would reduce modularization, because their presence would not support the process abstraction, thereby violating, or adding to the existing violation, of the first requirement. However, the program design based on the modified model for the L3-type does not violate this requirement and would therefore be assessed as "good". This consequence is a strong argument for advocating an error removal strategy that is based on re-designing the

program rather than "patching" it.

4.1.4 Conclusion

It can be seen that the decision regarding the choice of an ITEM is a significant determinant in the formulation of subsequent, refinement levels of an algorithm. Moreover, the input and output streams are two obvious factors that influence this choice. Indeed, the L1-type solution is an 'output-driven' design, whereas the s1 and L2 paradigms are 'input-driven' designs. In certain cases, the problem requirements may suggest the possibility of a further alternative for an ITEM, for example in the case of the signal problem 'a waiting period' and in the case of the line-edit problem 'a word'.

In relation to design quality, the discussion reveals that solutions based on abstract perceptions are superior to those based on primitive perceptions. Two examples of program structure illustrating this comparative difference in quality are s1 with s2 and L2 with L3. The "poor" quality of an L1-type program structure shows that abstract perceptions do not necessarily correspond to "simply chunked", but rather "appropriately chunked", perceptions. With regard to modifying programs, it is noted that a good design (e.g., an L3-type) is easier to correct than a poor one (e.g., an L1-type). Furthermore, a program modification strategy based on re-design rather than "patch" is preferable.

A conceptual model will now be presented. This model is based on : the literature review covering issues in program design; further reading in relævent cognitive psychology; and the experimental results reported above.

4.2 Conceptual Model Of Program Designer Behaviour

Software practitioners and human factor researchers, whose common goal is that of easing the programmer's task, also share an approach to representing the results of their investigations as a synthesised system or theory, which is frequently expressed in terms of principles and notions from other disciplines. For example, Dijkstra has expressed his ideas [1] on how a program should be designed as a calculus using mathematical principles, whilst Constantine [17] and Jackson [26] incorporate into their methodologies concepts from sytems theory and information modelling respectively. Human factor researchers in programming have used notions from cognitive psychology and problem-solving to produce conceptual models of programmer behaviour for various programming-related tasks. For instance, Allen [120] cites several examples of such models including: Sime et. al's investigations [86] into nested conditionals where results are explained in terms of a theory of "taxon" and "sequence" information; Shneiderman and Mayer's proposal [121] of a syntactic/semantic model of programmer behaviour and Atwood and Ramsay's work [122] which applies to program comprehension the notion of Kintsch "hierarchical schema" [123] on text comprehension. This investigation follows the same tradition by proposing a conceptual problem-solving model of programmer behaviour for the program design process. The model explains the behaviour of an aggregrate of programmers trained in structured programming principles.

4.2.1 Formulation of the model

The program design task is hypothesised to involve the problem solver, at any given time, carrying out one of three distinct types of

behaviour:

- problem understanding;
- solution planning (i.e., the generation of a set of goals);
- solution representing (i.e., recording the solution sequence).

Furthermore, since the overall design strategy in structured programming is essentially step-wise refinement, it can be said to be reductionistic (i.e., the problem to be solved is reduced into several sub-problems, with the reduction process being repeated on each sub-problem). Therefore, the three stages mentioned above need to be performed repeatedly.

Problem solving, as viewed in its most general form, is an activity which transforms an initial state, by applying a given set of operators, to produce a solution sequence that leads to a final state [59]. Therefore, the correspondence between program design (in structured programming terms) and problem-solving can be specified as:

- initial state : problem specification;
- final state : the program in a formal notation;
- operators : decomposition, abstraction, concatenation, selection and repetition;
- solution sequence : levels of refinement.

The model components - the problem solver, the problem and the solution sequence - will now be characterised from a problem solver perspective.

The problem solver is viewed as an information processor, whose structure is hypothesised to consist of: a set of knowledge structures relevant to program design, memory for storing and processing information and a facility for planning. The former, as Shneiderman [121] has pointed out, is a complex multi-levelled body of concepts and techniques that he refers to as "semantic knowledge". In general terms, this knowledge includes general methods for constructing

programs, strategies for producing specific programs or program "segments" and the effects of various program statements. For example, with respect to all the subjects that participated in the experiments, it could be said that their knowledge included high-level notions such as the structured programming operators previously specified, and strategies for producing program segments that ranged from simple segments such as accumulating a count, to intermediate segments such as finding the largest element of a list.

The memory structure adopted is based on Greeno's work [110] on problem solving and Shneiderman's model of programmer behaviour [121]. The structure consists of three components: short-term memory, long-term memory and working memory. The former stores information from the outside world to which the problem-solver pays attention but has a relatively limited capacity (Miller [124] suggests seven plus or minus two "chunks"), although information from it is easily retrieved. The knowledge acquired through experience by the problem solver is permanent and resides in semantic form in the long-term memory, whose capacity is essentially unlimited and retrieval from which is systematic. The stored information is assumed to be hierarchically structured, as hypothesised by Lindsay and Norman [125], in terms of categories of concepts; these are organised in the form of a semantic network (i.e., a tree structure), in which each node represents a generic concept that is related to its sub-nodes by an "ISA" (i.e., is an instance of) relationship. The component termed as "working memory" (due to Feigenbaum [126]) is not a permanent store but has a greater capacity and longer retention time than short-term memory. Information from short-term and long-term memory can be integrated in this component to produce solutions during problem solving.

Solution planning is viewed as goal generation, where a goal

structure defines the current state, the desired state and a set of possible strategies to transform the former into the latter. The mechanism for generating goals needs to be considered because goal elaboration in program design terms is equivalent to problem decomposition. Structured programming has no specific well-defined decomposition criteria, and therefore no systematic mechanism for goal generation can be defined. However, two distinct approaches can be hypothesised, which occur when the designer's primary focus of attention is on one of the two main ingredients of the descriptions of most programming problems, namely: either the data (it should be noted here that the term "data" in this context is intended to include both input and output) specification or the processing requirements. The two approaches can be respectively termed as "data-driven" and "requirements-driven". In addition, it is hypothesised that goal generation can be characterised by cognitive processes that are a function of an "availability heuristic". The rationale for this characterisation is influenced by Pollard's application [127] of Tversky and Kahneman's theory of nonrational intuitive judgement to logical reasoning tasks [128]. This theory proposes that a subject's judgements are mediated by an availability heuristic. Pollard suggests that this heuristic is responsible for two different types of availability effects, one being the availability derived from the subject's experience and the other from the salient characteristics of the stimulus. Both types have an essential common feature: they directly "cue" the subject's response. Thus the response is a function of this cueing and is not based on a rational reasoning process.

The implications of interpreting availability theory in context of solution planning is that goal generation is not necessarily based on rational reasoning but is a function of two possible sources that

are responsible for cue availability. Hence, two different types of activations are hypothesised, termed as "stimulus" and "knowledge" activation, occurring when planning is steered by specific characteristics of the stimulus and the knowledge structures of the problem solver respectively. This characterisation of designer behaviour implies that human-centered factors in program design (e.g., the level of difficulty) are not simply attributes of the task alone but are also related to problem solver knowledge. Furthermore, such a characterisation attempts to view program design as an accquired skill, and in so doing, takes note of Sheil's critique [52] that "programming is clearly a learned skill, and, therefore, what is easy or difficult is much more a function of what skills an individual has learned than any inherent quality of the task".

The application of two different types of problem decomposition strategy are proposed, being based on what Greeno [110] refers to as "reproductive thinking" and "productive thinking". The former is essentially a retrieval process, occuring when the subject understands the problem being solved, remembers the strategy for solving it and then transfers it in an integrated form from long-term memory to working memory. In contrast, the latter is a reconstruction process that takes place when the problem-solver does not have an existing strategy for solving the problem. In such a case, the task becomes one of constructing a solution plan by transforming existing strategies. An illustrative example from Greeno's discussion is Wertheimer's area-of-a-parallelogram problem, where knowing the formula for the area involves a simple retrieval of a strategy, and therefore is reproductive, whereas realising that a parallelogram can be transformed to a rectangle is productive.

The input to the model is the problem specification (the stimulus), which is hypothesised to be a function of the cues

(primitive or abstract) in the problem wording. In particular, a distinction is made between cues that stem from the data content and processing requirements of the specification. Both are assumed to be possible sources of cues, given that subjects trained in structured programming should have been taught to pay attention to data specification as well as processing requirements as a basis for solution structuring.

The output from the model (the solution sequence) is hypothesised to be the stages of program development performed by subjects. For a particular model, as Card et. al [129] point out, the "grain of analysis" (i.e., the level of detail) is defined by the operators used. The operators in this model are simply the strategies that transform one state to another state. However, in order to provide a more detailed description of designer behaviour, the approach adopted in documenting the output when applying the model to a given problem is that strategies are described informally and the states of the model, which correspond to program development stages, are characterised in algorithmic form. Thus, an operational overview of the model is:

(i) The process of problem understanding yields a description of the problem-to-be-solved which enters the short-term memory;

(ii) The available cues in this description are the primary sources for activating cognitive processes that generate goals in the working memory; these goals are then elaborated;

(iii) Each problem refinement is recorded.

The implications of advancing a model involving stimulus and/or knowledge activated goal generation processes based on availabilty theory are that problem decomposition strategies need to be hypothesised. Furthermore, hypotheses regarding preference for, and effort associated with, these strategies and contributory factors

affecting goal generation can then be formulated. The next section considers these hypotheses and provides empirical evidence that is consistent with the proposed model.

4.3 Problem Decomposition Behaviour

4.3.1 Strategies

Top-down exposition of a design cannot be regarded as proof of program development in a step-wise manner. Indeed, Wirth [130], in relation to his step-wise refinement method, is quite explicit: " I should like to stress that we should not be led to infer that program development proceeds in such a well organised, straightforward, top-down manner". The implied premise, which will act as a starting-point for postulating various strategies, is that programs are not necessarily developed using this idealised way of thinking. This view is supported by arguments resulting from the application of the proposed model to the signal and line-edit problems. These arguments detail the goals, knowledge and/or stimulus activated processes or other mechanisms for goal generation, and problem decomposition strategies associated with such goals. A partial record is thus provided of the "chain of thought" that a typical subject might undertake in the initial stages of problem decomposition.

First, let us suppose for the signal problem a data-driven approach where the processing requirements initially are a secondary consideration. In this case, the goal generation process is both stimulus and knowledge activated. This is because availability is derived from the presence of the prevailing signal stream emphasis in the data specification as well as from the subjects' familiarity with

the strategy associated with a hypothesised goal, broadly described as: 'process a repetition of two different types of signal'. The representations of this goal structure and the relevant portion of the problem description are transferred into the working memory. These representations are then processed by the application of a general hypothesised data-driven strategy that is assumed to be reproductive (i.e., part of the knowledge set of the problem solver) and is characterised as: 'Process Next Item' (PNI). The transferred problem description, views an "item" in its most trivial form i.e., as either (1 or 2) signal. The application of the strategy results in a problem refinement that corresponds to an s1-type solution and is of the form:

```
read (signal);
WHILE signal ≠ 0
DO
    IF signal = 1
    THEN
        process a 1-signal
    ELSE
        process a 2-signal
    FI;
    read (signal)
OD
```

Alternatively, consider a requirements-driven approach. A hypothesised goal of: 'accumulate counts for vehicle and timing signals', derives its availability from the first two processing requirements. Furthermore, if it is assumed that the goal is one with which the subject is familiar, then, the process generating it is both stimulus, and knowledge activated. Transference of the appropriate part of the problem description and the goal into the working memory is followed by the application of a hypothesised general

requirements-driven strategy termed as 'incremental design' (ID); its somewhat bottom-up character can be encapsulated informally by the phrase: "do what you can and make the rest fit around it". The resulting component is :

```
IF signal = 1
THEN
    increment vehicle count
ELSE
    increment survey length
FI
```

Preservation of this component in the subsequent steps produces an s1-type solution.

Naturally, a deeper analysis of the signal problem (i.e., one that is not based simply on the readily available cues) is needed to generate goals that would elaborate to an s2-type solution. This involves either :

- perceiving appropriately chunked items which will result in the generation of the abstract goal: 'process a repetition of two types of item', where one of these is itself a subsequence;
- or recognising subgoals which reside at a higher level in the problem structure and therefore require a certain amount of refinement themselves, which in the signal problem, context relates to determining the length of a waiting period.

Thus, in developing an s2-type decomposition, the problem-solver realises the advantage gained from including of a "subsequence of 2 signals" component, as these chunks relate directly to satisfying the requirement of determining 'the length of the longest waiting period'. The resulting component is:

```
WHILE signal = 2
DO
```

```
    process signal;
    read (signal)
  OD;
```

For the line-edit problem, the three possibile cues in a data-driven approach are character, word and line. However, since neither of the three cues is strongly emphasised in the problem wording, it is considered unlikely that these cues will be the major source of the availability effects. Similarly, in a requirements-driven approach, none of the requirements appear to contain explicit features that could be a major source of availability effects. Hence, for both approaches it is conjectured that goal-generating processes are both stimulus, and knowledge, activated.

An L2-type solution results from a data-driven approach in which a hypothesised goal of the form: 'process a repetition of two different types of characters' is both stimulus and knowlegde activated; this is because availability is derived partially from the presence of a character emphasis in the problem wording and also partially from problem-solver familiarity with the hypothesised goal. The general strategy associated with such a goal is the reproductive PNI strategy. The transference of this strategy with a description of the appropriate portion of the problem into the working memory is processed to yield a decomposition corresponding to an L2-type solution of the form:

```
  read (char);
  WHILE char ≠ "*"
  DO
    IF  char = space
    THEN
        process space
    ELSE
```

```
        process a non-space char
    FI
    read (char);
OD
```

To arrive at an L1-type solution from a data-driven approach involves hypothesising a goal of the form: 'process lines of characters'. The generation of such a goal is considered unlikely for two reasons. First, availabilty effects would have to originate from a not particulary pronounced line-item emphasis in the problem wording. Second, for goal generation to be knowledge-activated would involve assuming that subjects were familiar with a strategy for processing a repetition of items (i.e lines) where the item is itself a subsequence. However, since the results of the first observational experiment in the signal study indicate a paucity of solutions based on subsequences, the validity of the latter assumption is questionable. In a requirements-driven approach, the second requirement, namely, 'no line will contain more than m characters and each line will be filled as far as possible', is stimulus activated. This requirement is also knowledge-activated because the hypothesised goal of: 'fill a line of m characters', which satisfies the requirement, is one for which it is reasonable to assume that the problem solver will be familiar with a program component of the form:

```
WHILE not m characters
DO
    process a character;
    read (char)
OD
```

An L3-decomposition type requires a goal of the form: 'repeatedly, build a word and then process it'. The presence of 'word' in the data description is not sufficiently emphasised to act as an

an available cue and neither of the three processing requirements are possible sources of availability effects; therefore, it is considered unlikely that the goal is generated from these effects alone. One possible explanation is that the problem solver may have generated goals of the form: 'repeatedly build a word' and 'repeatedly process a word' which derive their availabilty from a combination of the stimulus and the subject being familiar with a "process next word" (i.e., PNI-type) strategy; then, in a piece-meal manner reminiscent of ID, the subject combines the goals to form the required problem decomposition. An alternative explanation is that the subject, through a logical reasoning process, perceives the chunking of words, recognises the necessity for appropriate abstract goals, i.e., 'build a word' and 'process a word', and uses a productive strategy to combine them to produce an L3-type decomposition.

The two problem decomposition strategies hypothesised are assumed to be of general applicability. The PNI strategy, which is strongly associated with a data-driven approach, bears some similarity to Hoc's [131] findings that strategies for program construction are influenced by the role of "mental execution of the program": it is reasonable to assume that the most likey self-elaboration of the task is that subjects would visualise, having contemplated the data, would be to deal with the list an item at a time. Also, Hoc's ideas of "strategies of progressive generalisation of a sequential procedure" and "mechanisms of adapting known procedures to computer operation" [ibid] provides an alternative perspective to the role ID plays in a requirements-driven approach.

4.3.2 Related Factors

From the strategies discussion on the signal problem, it was

observed that there are strong availability effects in both data, and requirements driven approaches; moreover, these effects both reinforce the generation of goals whose elaboration leads to an sl-type decomposition. Therefore, for the signal problem, a strong preference for sl-type solutions can be predicted, as was confirmed by the observed bias for sl-type solutions, in experiment 1 of the signal study. However, for the line-edit problem the strength of availability effects are not mutually aligned towards one particular goal, but are in fact responsible, in differing strengths, for generating distinct goals that elaborate to the three different decomposition types. The model does not predict the relative frequencies of the three decomposition types, because of the difficulty in quantitatively assessing availability effects. It does however suggest that all three types will be present with relatively significant frequencies. The frequencies corresponding to L1, L2 and L3 decomposition types obtained in the first experiment of the line-edit study are in accordance with this prediction.

An alternative view of problem solving behaviour presented by postulating availability effects is, that subjects are not conciously inclined to seek other possible decompositions but instead adopt a route of "least initial resistance". The word "initial" is important here since the ease with which a first-level decomposition is accomplished is likely not to be related to the ease of its subsequent elaboration. Such an interpretation means that decompositions produced by goals that result from availability effects rather than a logical reasoning process are easier to comprehend, although not necessarily easier to elaborate. In case of the signal problem, since all the availability effects are in mutual alignment for an sl-type decomposition, this means that the latter will be significantly easier to perceive. The result of the controlled experiment in the signal

study strongly supports this prediction.

The general implication of the above is that goals generated on the basis of availability will be based primarily upon simplistic, rather than abstract perceptions and that the former will occur with greater frequency, as was observed in two of the experiments. In the first experiment of the signal study, the bias towards an s1-type solution can be re-interpreted as a strong preference for solutions for a decomposition based on a primitive perception. Similarly, in the first experiment of the line-edit study, comparison of frequencies of solutions based on abstract and primitive perceptions revealed a significant preference for the latter. Conversely, solutions based on abstract perceptions are inherently harder to perceive because they are more likely to be the product of a logical reasoning process rather than being triggered by availability effects.

A relevant consideration at this point is the distinction between novice, expert and experienced subjects. The ability to handle abstractions has been identified as one major attribute of experts that distinguishes them from novices [132]. Therefore it may be argued that the results obtained simply reflect that subjects were novices at structured programming who had had insufficient time to develop the abstraction capabilities that characterise an expert. Whether the second and third year undergraduates who took part in the experiments were still novices is a matter of debate. Similarly, whether any of the participants were experts is something that is difficult to establish. It is, however, considered that gaining experience involves applying acquired techniques over a prolonged period, but this does not necessarily develop skills of any particular kind. Indeed, the results of the first experiment of the signal study, where the range of experience of subject groups was the most diverse, revealed that neither the failures nor the few s2-type

solutions produced were monopolised by any one particular group in the population. Therefore, it can be argued that the degree to which abstractions become revealed to subjects during problem decomposition is largely due to other contributory factors to which consideration is now given.

4.3.3 Further Contributory Factors

One of the factors responsible for goal generation is subject experience. In fact, as predicted by the model and as the experimental evidence indicates, subjects whose background could be characterised as experienced in only a broadly "structured approach" are inclined towards simplistic data, or requirements, driven reasoning. The generalised converse of this is that abstraction skills are likely to be more developed in subjects taught structured programming which incorporates more specific decomposition criteria (of whatever kind) where perception of abstractions receives greater emphasis. The third experiment in the signal study, in which the aim was to investigate the effect of training on problem decomposition, attempted to make abstract perceptions act as response cues, this being achieved by training subjects to look for logical abstractions in data and therefore enabling cues to derive their availability from subjects' training. The experimental results indicate an increased proportion of solutions based on abstract perceptions and therefore lends support to the view that training in the application of more specific criteria for decomposition can lead to an improvement in abstraction skills.

As already noted another factor which may influence decomposition strategy is the problem specification itself. The presence of certain key-words and phrases, the ordering of constituent parts, or other

textual features, may cause attention to be focused on a particular problem component, thereby triggering off some decomposition pathway. For example, primitive features in the description of the signal problem, namely, the signal stream and the first two processing requirements, could both act as available cues and therefore promote simplistic reasoning. In the line-edit problem, however, the presence of "words" in the data specification is considered to be in some part responsible for abstract goal generation. Observational evidence supporting the view (and its converse) that primitive features are responsible for primitive decompositions can be obtained by comparison of the proportions of simplistic-to-abstract-based decompositions in the first experiments of both studies.

The model's prediction that changes in problem wording will imply a change in availibilty effects is supported from the results of the third experiment of the line-edit study where certain primitive and abstract problem specification features were manipulated to produce an increased number of abstract decomposition types. The four problem specifications corresponding to the four experimental treatments were:

- problem I contained two cues responsibile for availability effects that produce primitive decomposition;
- problems II and III contained cues responsible for availability effects that yield both primitive and abstract decompositions;
- problem IV contained two cues responsible for availability effects that result in abstract decomposition.

The experimental results revealed the predicted increase in abstract decompositions. In addition, a more specific hypothesis testing the implied trend in the above treatments was also verified. Furthermore, the results also revealed a greater proportion of abstract decompositions for solutions based on the third treatment than for those based on the second. Since the former treatment corresponds to

emphasising abstract features pertaining to processing requirements rather than data, it would appear that the role of emphasing such requirements has a greater influence than that of the data specification. A possible explanation for this result is that the training subjects received, places greater emphasis on functional decomposition rather than inspection of data structure.

The precise effects of previous training and problem description in any given circumstance will depend on the individual characteristics of subjects and the specification with which they are confronted. As both effects were aligned in the signal problem - the latter effect magnifying elementary problem components, the former providing no positive compensation - the result was a bias towards a simplistic solution, whereas in the line-edit problem the data description provided only a partial positive compensation. Therefore, it can be concluded that these factors can in general mitigate against a wholly top-down approach being employed. Indeed, for problem decomposition to be performed in a top-down manner requires the designer having a set of decomposition rules rather than merely being cued in a possibily non-rational manner to some "least-resistance" decomposition pathway. This lends cre dence to the original assertion that problem decomposition is often carried out in a somewhat disorganised, piece-meal, bottom-up manner.

4. 4 Elaboration of Decomposition Paradigms

To yield further understanding of program designer behaviour, attention is now focused on the frequencies and nature of errors made in the elaboration of a decomposition to a completed solution. An explanatory framework based on the notion of generic concepts, as described in the model, is presented to provide reasons for the

occurrence of errors. Moreover, a relationship between decomposition quality and and error frequen cy is noted.

The elaboration of either decomposition of the signal problem is essentially the fulfillment of a goal that satisfies the third requirement, namely: "find the largest accumulated waiting period". The strategy associated with this goal, hypothesised to involve a productive reasoning process, can be visualised as categorising components into appropriate "clusters" and allocating these clusters to the existing decomposition structure.

For the s2-type, clustering of components and their allocation to process structures is hypothesised to be relatively error free. The rationale for this view is that the three actions 'increment waiting period', 'reset wait' and 'check for longest waiting period', and the process component to which these actions are allocated, namely 'process timing signal', all belong to the same generic category, namely 'time'. Therefore, clustering of these components is simply performed through generic grouping (the basis upon which storage and retrieval of information takes place within the model). The observational evidence from the first experiment in the signal study supports the above view because only a small percentage of s2-type solutions contained errors associated with the placement of the actions required to satisfy the third requirement. Note that, from a design evaluation perspective, the abstraction level corresponding to the component 'process waiting period' is precisely the cluster of actions associated with the generic category 'time'.

In the case of an sl-type decomposition, for the problem solver to arrive at a correct solution, the hypothesised productive reasoning process involves recategorising the cluster of three components differently from that based on generic categories. The process is predictably, therefore, relatively error prone. Supporting evidence

for this hypothesis is based on the first experiment of the signal study; in nearly all subjects' solutions, the component 'final check for the longest waiting period' was absent, and of those that attempted satisfying the third requirement, approximately half the solutions contained errors associated with the components 'reset waiting period' and/or 'check for longest waiting period'. The explanation for the first mistake is that a subject's focus of attention is on the refinement of the characteristic process-pair (i.e., retrieving the necessary knowledge structures needed to satisfy the third requirement, transforming them into program components and deciding upon their placement) and therefore becomes, as Rumelhart [133] terms, "sensitive to the local context". In so doing, the designer "loses sight of" (i.e., no longer retains in the working memory) the overall design structure, which is necessary to arrive at a correct placement of the action in question because it forms part of the initial level of refinement. Similar programmer behaviour during which subjects "lose sight of the overall view of the procedure" has been reported by Hoc [111]. In relation to the last two mistakes, it was noticable that components associated with the generic category 'time', that form part of 'process vehicle signal', were placed within 'process timing signal'. Although it is difficult to explain the exact reason for choosing this placement, the influence of wanting to retain things with the generic category to which they belong cannot be ignored. It is also worth noting that the solution features with which errors were associated correspond to actions whose placement, in a correct solution, contribute to the solution's poor modularity.

Further evidence which substantiates the proposition that the elaboration process for primitive perceptions is more error-prone than that for abstract perceptions is the error frequencies for L1, L2 and L3-type solutions from experiment 1 of the line-edit study; these

frequencies were respectively 75%, 45% and 30%. Moreover, the highest error frequencies were associated with those solution features which contributed to a solution's poor modularity measure.

From the above discussion, particularly the description of the elaboration of s1-type and s2-type decompositions, it would appear that the elaboration of poor quality decompositions is not only relatively more error prone but also requires greater effort. The reason for the latter is that in elaborating a poor quality decomposition correctly, additional effort is required to either recategorise actions in an unobvious manner or introduce conditions whose need was not apparent in the initial decomposition. Therefore, on this basis, it can be hypothesised that high-quality decompositions are easier to elaborate than low quality ones.

The results of the second experiment in the line-edit study, which revealed that different decompositions require differing amounts of effort, supports the hypothesis which relates effort to decomposition quality. Furthermore, comparisons of effort required to elaborate poor quality decompositions (L1 and L2 types) with one of good quality (the L3-type) also provide partial support for the hypothesis because they indicate significant differences between L1 and L3 but not between L2 and L3. The factors responsible for these differences cannot be explained by availabilty effects, although the previously mentioned relationship between effort and the quality of the decompositions provides one possible source of explanation. The exact reason for there not being a significant difference, as would be predicted, in the effort required between L2 and L3 types is difficult to establish, but a possible cause is the inaccuracy in measuring "effort". Two possible sources of this inaccuracy are: first, the difficulty in relating two factors, where one factor is assessed qualitatively (i.e., categories) and the other is assessed

quantitatively (i.e.,time); second, the validity of assuming that the effort required in perceiving different decomposition types is approximately the same.

5. Conclusion

Initially, the broad aim of the research was to investigate whether structured programming is a completely effective design technique. Therefore, the original motivation for conducting experiments was simply to gather empirical evidence that would validate or refute hypotheses concerning this design technique. One conclusion from the pilot study was that both theory and application of structured programming are still problematic areas, because analysis of subjects' attempts at solving a reasonably simple programming problem, in what was judged to be an adequate time to complete the task, yielded relatively high percentages of incorrect and incomplete solutions. This conclusion supports the doubts raised in the background review as to whether structured programming really is entirely sufficient for the production of high quality, correct programs. Moreover, such doubts are shared by many others; Green for example questions "the wisdom of propounding it [the principle of divide and rule] as the single vital principle that allows a program to be produced mechanically and errorlessly" [57]. The pilot study's confirmatory evidence concerning the sufficiency of structured programming led to the formulation of the more specific objective of developing a better understanding of how program design actually is performed, so that ultimately advice might then be given as to how it should be taught and practiced.

The specific line of attack chosen was to investigate the nature of problem decomposition strategies and certain factors related and contributory to those strategies. The results of the two experimental studies conducted revealed that decompositions based on simplistic, as opposed to abstract, perceptions of problem structure were: significantly more frequent in subjects attempts, required

considerably less effort to perceive but relatively more effort to complete, and produced solutions that contained a greater proportion of errors. More importantly, the experimental work provided significant insight into the various decomposition strategies that are employed by subjects who have been taught, and in principle, practice top-down structured programming. A model of program designer behaviour was then devised in the light of this insight gained, which would provide an explanatory framework for interpreting the experimental results.

The model views program design as a problem-solving task where solution planning is regarded as a goal generation activity. The fulfillment of a goal yields some particular (partial) decomposition of the problem, possibly accompanied by "tying loose ends together," i.e., fitting collections of program segments piece by piece into a (partially) developed program structure. The view that program design is actually performed in an idealised top-down manner is rejected in favour of the alternative view that such aspects as problem specification, subject familiarity with component parts and the level of abstraction skills developed in previous training, are major contributory factors responsible for the strategies by which problem decomposition is effected. The model (in conjunction with assumptions regarding generalised knowledge structures and problem decomposition strategies possessed by subjects), when applied to the signal and line-edit problems, yields a description plausibly corresponding to a subject's chain of thought. The experimental results are then interpreted within the behavioural framework provided by this description.

However, the degree to which the model adequately reflects the behaviour of a typical participant of the experiments is to some extent a matter of debate. Whilst, for example, the findings suggest that the two contributory factors advanced certainly play an

influencing role in problem decomposition, it would be somewhat short-sighted to propose that they are solely responsible for "shaping" this complex task. Nevertheless, the model provides a richer description, and perhaps captures more of the flavour, of how program design might proceed than the traditional top-down exposition.

Although the research findings have certainly yielded answers to some of the questions posed, it is a characteristic tendency of an empirical investigation that further issues are then raised; these issues generate more hypotheses that hopefully prove easier to test than in preceding stages of an investigation. In particular, attention needs to be given to whether subjects' approaches are broadly data and/or requirements driven and what role is played by the strategies of PNI and ID respectively in these approaches. In order to provide a more detailed description of program designer behaviour, a further possible direction for future research is the use of video and/or verbal "protocols" to obtain more detailed behavioural evidence which can then be represented using production systems that model human cognition [134].

In relation to methodological issues, the following points are noted. First, the extent to which these findings are applicable to experts is difficult to establish. Although attempts made to enlist groups of presumed experts for this research were unsuccessful, a comparison of these findings with those involving experts would be a useful augmentation to this work. Second, whilst the materials used, (e.g., programming tasks, process structure cues), the method of analysing solutions used (e.g., process structure hierarchies) and the measures employed (e.g., completion time as a measure of effort) have been reasonably successful in eliciting experimental results, nevertheless, replication of their use in further program design experiments would provide valuable evidence as to their effectiveness

in this area. In particular, the problems encountered in trying to determine effort required to perceive and elaborate decompositions, using process structure cues and completion times, merit further investigation.

As already noted, the two generalised rudimentary forms of data-driven and requirements-driven strategies that have been proposed, when applied to the problems used in the experiments, lead to inferior solutions. A natural induction is that this might be the case for the majority of problems. This gives rise to a certain amount of concern since the experimental evidence indicates a preference for such strategies (apparently irrespective of background, training, experience etc.). If that is the case, it is important to minimise the effect of any factor that promotes usage of these strategies and their possibile entrenchment in a person's general approach to program construction. One possible way of achieving this amelioration is by providing of training in more "directed" forms of structured programming that concentrates on the development of abstraction skills by providing "concrete" models on which decompositions can be based. This training would, hopefully, advance subjects' general design know-how and therefore possibly also help to counteract any tendency to adopt - without further analysis - an inferior strategy implicitly suggested by aspects of problem presentation, for example.

Finally, in relation to the initial aim of assessing the effectiveness of step-wise refinement as a problem decomposition strategy, the conclusion is that in structured programmming, the importance of the rule of Descartes:

> "Divide each problem that you examine into as many
> parts as you need, to solve them more easily."
>
> Rene Descartes, Oeuveres, vol. VI,
> Discours de la Methode, Part II

has been appreciated, but the warning of Leibnitz:

> "This rule of Descartes is of little use, as long as the art of dividing ... remains unexplained. By dividing his problem into unsuitable parts, the unexperienced problem solver may increase his difficulty."
>
> Gottfried von Leibnitz
> Philosophiche Schriften, vol. VI

remains unheeded.

APPENDICES

Appendix 1.1 The Signal Problem

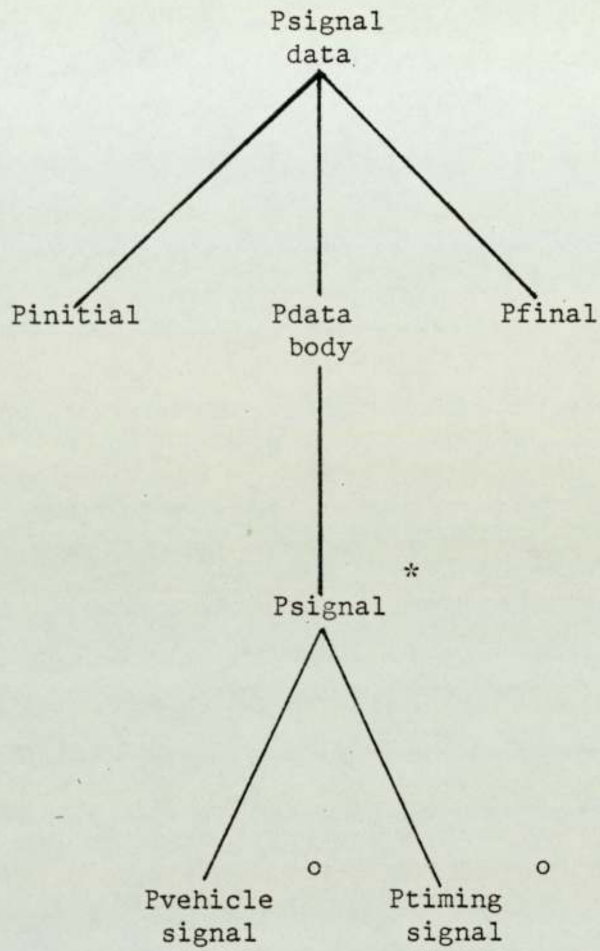
A traffic survey is conducted automatically by placing a detector at the road side connected by data-links to a computer. Whenever a vehicle passes the detector, it transmits a signal consisting of the the number 1. A clock in the detector is started at the beginning of the survey, and at one second intervals thereafter it transmits a signal consisting of the number 2. At the end of the survey the detector transmits a 0. Each signal is received by the computer as a single number (i.e. it is impossible for two signals to arrive at the same time). Design a program which reads such a set of signals and outputs the following:

(a) the length of the survey period;

(b) the number of vehicles recorded;

(c) the length of the longest waiting period without a vehicle.

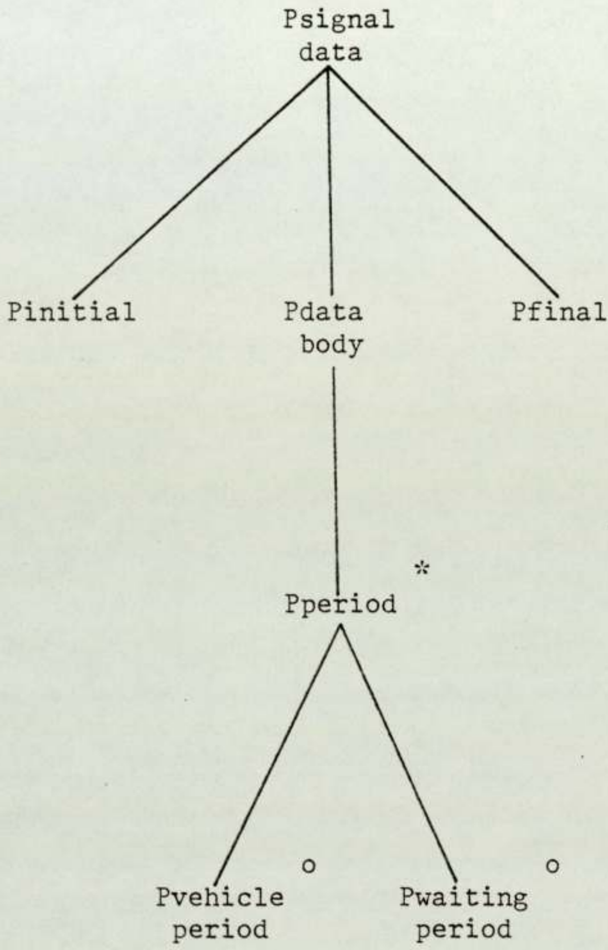
Appendix 1.2 sl-type Decomposition



Appendix 1.3 Complete Solution (sl-type)

```
num of vehicles := 0; length of survey := 0; waiting period := 0;
longest waiting period := 0;
read(signal);
WHILE signal ≠ 0
DO
  IF signal = 1
     THEN
          (*process a vehicle signal*)
          num of vehicles := num of vehicles + 1;
          IF waiting period > longest waiting period
          THEN longest waiting period :=  waiting period
          FI;
          waiting period := 0
     ELSE
          (*process a timing signal*)
          waiting period := waiting period + 1;
          length of survey := length of survey + 1
     FI;
     read(signal)
OD;
IF waiting period > longest waiting period
THEN longest waiting period := waiting period
FI;
print(length of survey, num of vehicles, longest waiting period)
```

Appendix 1.4 s2-type Decomposition



Appendix 1.5 Complete Solution (s2-type)

```
num of vehicles := 0; length of survey := 0;
longest waiting period := 0;
read(signal);
WHILE signal ≠ 0
DO
    IF signal = 1
    THEN
        (*process a vehicle*)
        num of vehicles := num of vehicles + 1;
        read(signal)

    ELSE
        (*process a waiting period*)
        waiting period := 0;
        WHILE signal = 2
        DO
            length of survey := length of survey + 1;
            waiting period := waiting period + 1;
            read(signal)
        OD;
        IF waiting period > longest waiting period
        THEN longest waiting period := waiting period
        FI
    FI
OD;
print(length of survey, num of vehicles, longest waiting period)
```

Appendix 2.1 Skeletal program structure cue for sl-type solution

```
.
.
.
.
.
.

WHILE . . . . . . . signal ≠ 0
DO
  .
  .
  .
  .
  .
  .

  IF signal = 1
  THEN
      .
      .
      .
      .
      .
      .

  ELSE
      .
      .
      .
      .
      .
      .

  FI
  .
  .
  .
  .
  .
  .
OD
.
.
.
.
.
.
```

Appendix 2.2 Skeletal program structure cue for s2-type solution

```
.
.
.
.
.

WHILE . . . . . signal ≠ 0
DO
  .
  .
  .
  .
  .

  IF signal = 1
  THEN
      .
      .
      .
      .
      .

  ELSE
      .
      .
      .
      .
      .

       WHILE . . . . . signal = 2
       DO
          .
          .
          .
          .
          .
       OD
      .
      .
      .
      .
      .
      .

  FI
  .
  .
  .
  .
  .
OD
.
.
.
.
.
```

Appendix 2.3 List of elementary actions

```
read(signal)

num of vehicles := 0

length of survey := 0

waiting period := 0

longest waiting period := 0

num of vehicles := num of vehicles + 1

length of survey := length of survey + 1

waiting period := waiting period + 1

IF waiting period > longest waiting period
THEN longest waiting period := waiting period                    (*)
FI

print(length of survey, num of vehicles, longest waiting period)
```

(*) Strictly, this is not an "elementary" action; however, determining its location in the skeletal structure was considered to be an integral part of the design task.

Appendix 3.1 The Line-edit problem

A piece of text consisting of words separated by one or more space characters is terminated by an *.

It is required to convert it to line by line form in accordance with the following rules:

(a) Redundant spaces between words are to be removed;

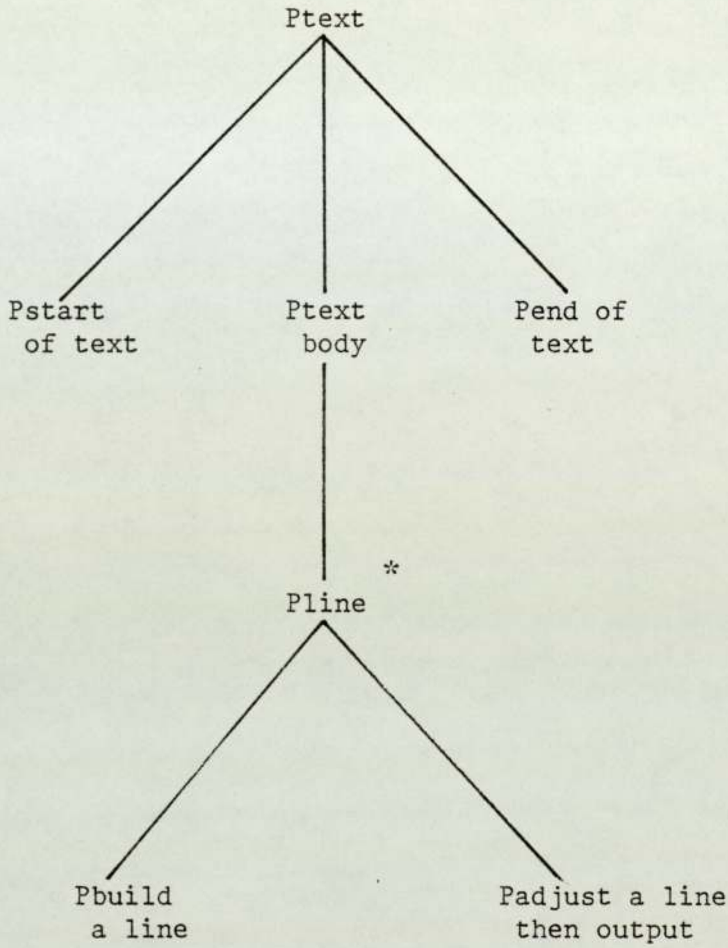
(b) No line will contain more than m characters and each line is filled as far as possible;

(c) Line-breaks must not occur in the middle of a word.

(You may ignore the presence of line-feed characters and the possibility of a word being greater than m characters).

Design a program to read the text and output it in accordance with the above rules.

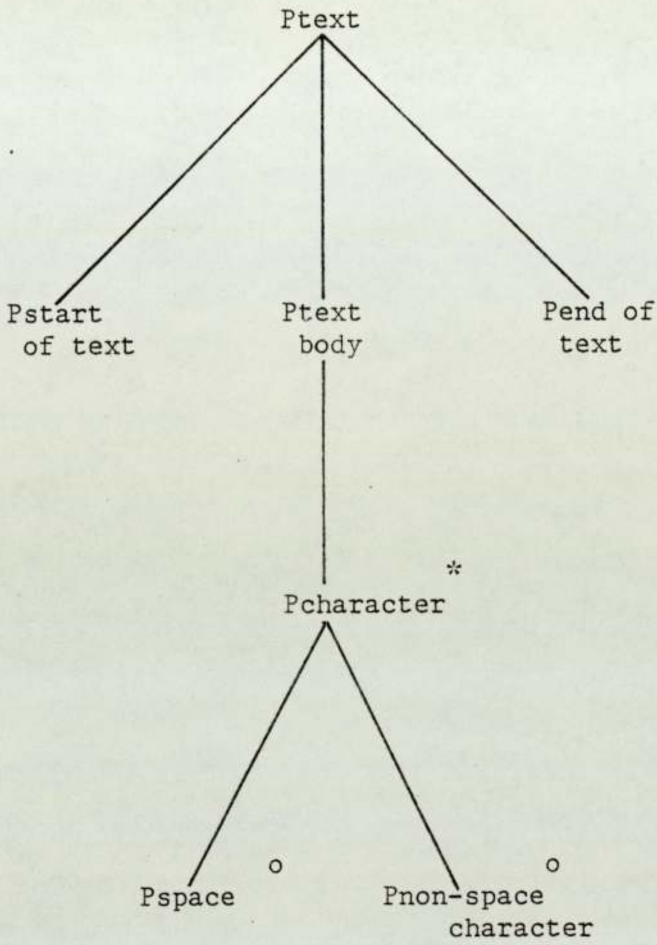
Appendix 3.2 L1-type Decomposition



Appendix 3.3   L1-type Solution

```
line := empty;
line size := 0;
prev char = space;
read (char);
WHILE . . . . char ≠ "*"
DO
  .
  .
  .
 WHILE . . . . (line size < m) AND (char ≠ "*")
 DO                        (*loop to build a line of m chars
   .                        with redundant spaces removed*)
   .
   .
   IF (char ≠ space) OR (prev char ≠ space)
   THEN
       line := line + char;
       .                   (*add non-redundant space or character*)
       line size := line size + 1

   FI;
   .
   prev char := char;
   read (char)

 OD;
  .
  .
  .
 IF (char = space) OR (char = "*")
 THEN
     print (line);
     print (newline);
     line size := 0; line := empty

 ELSE                                (*process line with a possible
     remove partial word (part word, word size);      break in the
middle of a word*)
     print (line); print (newline);
     line := part word; line size := word size

 FI;
  .
  .
  .
OD;
```

Appendix 3.4 L2-type Decomposition



Appendix 3.5   L2-type Solution

```
line size := 0;
word size := 0;
word := empty;
read (char);

WHILE . . . . . char ≠ "*"
DO
  .
  .
  .
  IF char = space
  THEN
      .
      .
      .
      IF word size ≠ 0    (*not a redundant space*)
      THEN
          .
          .
          .
          (*output a word on, current or new, line*)
          IF line size + word size < m
          THEN
              .
              print (space);
              line size := line size + 1

          ELSE
              print (newline);
              line size := 0
              .

          FI;
          .
          print (word); line size := line size + word;
          word := empty; word size := 0

      FI
      .
      .
      .
  ELSE (*build a word*)
      word := word + char;
      .
      word size := word size + 1

  FI;
  .
  read (char)
  .

OD
```

Appendix 3.6   L3-type Decomposition

Ptext

Pstart of text

Ptext body

Pend of text

Pword *

Pbuild a word

Poutput a word

Appendix 3.7 L3-type Solution

```
.
line size := 0;
read (char);

WHILE . . . . . char ≠ "*"
DO
  .
  .
  .
  WHILE . . . . . char = space DO   read (char)    OD;
  .
  .
  word size := 0; word := empty;

  WHILE . . . . . char ≠ space AND char ≠ "*"
  DO
    word := word + char;
    word size := word size + 1;                (*build a word*)
    read (char)

  OD;
  .
  .
  .
  (*output a word on current, or new, line*)
  IF line size + wordsize < m
  THEN
      print (space);
      line size := line size + 1
      .
  ELSE
      print (newline);
      line size := 0;
      .
  FI;
  .
  print (word);
  line size := line size + word size

OD
```

Appendix 4.1 Process Structure Cue for L1-type Decomposition

```
.
.
.
WHILE . . . . . char ≠ "*"
DO
  .
  .
  .
  WHILE . . . . . (line size < m) AND (char ≠ "*")
  DO                      (*loop to build a line of m chars
    .                      with redundant spaces removed*)
    .
    .
    IF (char ≠ space) OR (prev char ≠ space)
    THEN
        .
        .                 (*add non-redundant space or character*)
        .
    FI
    .
    .
    .
  OD
  .
  .
  .
  IF (char = space) OR (char = "*")
  THEN
      .
      .
      .
  ELSE                    (*process line with possible break in the
      .               middle of a word*)
      .
      .
  FI
  .
  .
  .
OD
.
.
.
```

Appendix 4.2   Action List for L1-type Cue

```
line size := 0

line size := line size + 1

line := line + char

print (line)

print (newline)

read (char)

prev char := char

line := empty      (*empty is the null string*)

remove partial word (part word, word size)          ****

line := part word

line size := word size
```

**** NOTE : "remove partial word" removes part of the word "part word" of size "word size" from the end of a line i.e., when there is a line break in the middle of a word.

Appendix 4.3 Process Structure Cue for L2-type decomposition

```
.
.
.
WHILE . . . . char ≠ "*"
DO
 .
 .
 .
 IF char = space
 THEN
     .
     .
     .
     IF word size ≠ 0    (*not a redundant space*)
     THEN
         .
         .
         .
         (*print a word on, current or new, line*)
         IF line size + word size < m
         THEN
             .
             .
             .
         ELSE
             .
             .
             .
         FI
         .
         .
         .
     FI
     .
     .
     .
 ELSE (*build a word*)
     .
     .
     .
 FI
 .
 .
 .
OD
.
.
.
```

Appendix 4.4 Action list for 12 and 13 type solutions

```
line size := 0

word size := 0

line size := line size + 1

word size := word size + 1

line size := line size + word size

word := word + char

print (space)

print (newline)

print (word)

read (char)

word := empty      (*empty is the null string*)
```

Appendix 4.5 Process Structure Cue for L3-type Decomposition

```
.
.
.
WHILE . . . . char ≠ "*"
DO
  .
  .
  .
  WHILE . . . . . char = space DO . . . . . OD    (*remove
spaces*)
  .
  .
  .
  WHILE . . . . . char ≠ space AND char ≠ "*"
  DO
    .
    .               (*build a word*)
    .
  OD
  .
  .
  .
  (*output a word on current, or new, line*)
  IF line size + wordsize < m
  THEN
      .
      .
      .
  ELSE
      .
      .
      .
  FI
  .
  .
  .
OD
.
.
.
```

Appendix 5.1

(see appendix 1.1)

Appendix 5.2

A line of text consisting of words separated by one or more spaces is terminated by an *.

Design a program to input the text and output the following:

(a) the number of non-space characters;

(b) the number of spaces;

(c) the length of the longest word.

Appendix 5.3

A traffic survey is conducted automatically by placing a detector at the road side connected by data-links to a computer. Whenever a vehicle passes the detector, it transmits a signal consisting of the the number 1. A clock in the detector is started at the beginning of the survey, and at one second intervals thereafter it transmits a signal consisting of the number 2. At the end of the survey the detector transmits a 0. Each signal is received by the computer as a single number (i.e. it is impossible for two signals to arrive at the same time). Design a program which reads such a set of signals and outputs the length of the longest waiting period without a vehicle

Appendix 5.4

A line of text consisting of words separated by one or more spaces is terminated by an *.

Design a program to input the text and output the length of the longest word.

Appendix 6 : A typical solution containing errors for the signal problem.

```
num of vehicles := 0;       length of survey := 0;
waiting period  := 0; longest waiting period := 0;
read(signal);
WHILE signal ≠ 0
DO
    IF signal = 1
    THEN
         (* process vehicle signal *)
         num of vehicles := num of vehicles + 1;
         waiting period := 0
    ELSE
         (* process timing signal *)
           waiting period := waiting period + 1;
           IF waiting period > longest waiting period
           THEN longest waiting period := waiting period
           FI;
           length of survey := length of survey + 1
     FI;
     read(signal)
OD;
```

The error frequency for above solution would be 2, since :

(i) 'check for longest waiting period' has been placed within 'process timing signal' rather than 'process vehicle signal'

(ii) 'final check for longest waiting period' is absent

REFERENCES

1. E.W. Dijkstra 1976

   A Discipline of Programming, preface xiii

   Prentice Hall

2. G.M. Weinberg 1971

   The Psychology of Computer Programming, pp. 27

   Van Nostrand Reinhold Company

3. W. Findlay and D.A Watt 1979

   PASCAL An Introduction to Methodical Programming pp 60-66

   Pitman

4. P. Naur 1969

   Programming by Action Clusters

   BIT Vol. 9, No. 3, pp 250-258

5. A. Koestler 1964

   The Act of Creation

   Hutchinson

6. C. Alexander 1966

   Notes on the Synthesis of Form

   Harvard University Press

7. H.A. Simon 1969

   The Sciences of the Artificial

   M.I.T Press

8. B.W. Boehm 1973

Software and Its Impact: A Quantitative Assessment

Datamation, Vol. 19, No. 5, pp. 48-59

9. P. Naur, B. Randell and J.N. Buxton (Editors) 1976

Software Engineering Concepts and Techniques

Petrocelli Charter

10. E.W. Dijkstra 1968

Structured Programming

in P. Naur, B. Randell and J. N. Buxton, pp. 222-226

11. E.W. Dijkstra 1968

Complexity controlled by Hierarchical Ordering of Function and Variability

in P. Naur, B. Randell and J. N Buxton, pp. 114-116.

12. E.W. Dijkstra 1968

The Structure of the T.H.E Multiprogramming System

Communications of the ACM, Vol. 11, No. 5, pp. 341-346

13. S. Gill 1968

Thoughts on the Sequence of Writing Software

in P. Naur, B. Randell and J.N. Buxton, pp. 116-118

14. B. Randell 1968

Towards a Methodology of Computing System Design

in P. Naur, B. Randell and J.N. Buxton, pp. 127-129

15. A.J. Perlis 1968

Keynote Speech

in P. Naur, B. Randell and J. N. Buxton, pp. 87-88

16. M.D. McIlroy 1968

Mass Produced Software Components

in P. Naur, B. Randell and J. N. Buxton, pp. 88-95

17. W. Stevens, G. Myers and L. Constantine 1974

Structured Design

IBM Systems Journal, Vol. 13, No. 2,pp. 115-139

18. E.W. Dijkstra 1968

Goto statement considered harmful (Letter to the Editor)

Communications of the ACM, Vol. 11, No. 3, pp. 147-148

19. E.W. Dijkstra 1972

Notes on Structured Programming

in O-J. Dahl, E.W. Dijkstra and C.A.R. Hoare

Structured Programming

Academic Press, London

20. N. Wirth 1971

Program development by Stepwise Refinement

Communications of the ACM, Vol. 14, No. 4, pp. 221-226

21. J.D. Aron 1968

The Superprogrammer Project

in P. Naur, B. Randell and J.N Buxton, pp. 188-190

22. F.T. Baker 1972

Systems Quality through Structured Programming

Proceedings of the Fall Joint Computer Conference

AFIPS Press, Vol. 41, Part 1, pp. 339-343

23. F.T. Baker 1975

Structured programming in a production programming environment

Proceeding of the International Conference on Reliable Software

ACM SIGPLAN Notices, Vol. 6, No. 10, pp. 172-185

24. R.A. Snowdon 1974

Interactive use of a computer in the preparation of structured programs.

University of Newcastle upon Tyne

Ph.D thesis

25. P. Henderson and R. Snowdon 1972

An Experiment in Structured Programming

BIT, Vol. 12, No. 1, pp. 38-53

26. M.A. Jackson 1975

Principles of Program Design

Academic Press

27. E.W. Dijkstra 1976

A Discipline of Programming, pp. 211-212

Academic Press

28. J.D. Warnier 1974

Logical Construction of Programs

Van Nostrand Reinhold

29. E. Yourdon and L.L Constantine 1979

Structured Design

Prentice Hall

30. G.J. Myers 1975

Reliable Software through Composite Design

Van Nostrand and Reinhold

31. W.P. Stevens

Using Structured Design

J. Wiley & sons

32. D.L. Parnas 1972

On the Criteria to be Used in Decomposing Systems into Modules

Communications of the ACM, Vol 12, No. 15, pp. 1053-1058

33. B.H. Liskov 1972

A Design Methodology for Reliable Software Systems

Fall Joint Computer Conference

AFIPS Press, pp. 191-199

34. J. Emery 1964

The Planning Process and its Formalisation in Computer Models.

Proceedings of the 2nd Congress of Information System Science.

35. S.N. Griffiths 1979

Design Methodologies - A Comparison

in G. Bergland and R. Gordon (editors)

Tutorial: Software design strategies

IEEE Computer Society Press, pp. 189-213

36. T. DeMarco 1978

Structured Analysis and System Specification

Yourdon Inc.

37. C. Gane and T. Sarson 1979

Structured Systems Analysis: Tools and Techniques

Prentice Hall

38. E. Yourdon 1979

Managing the Structured Techniques

Yourdon Inc.

39. T. DeMarco 1979

Controlling Software Projects

Yourdon Inc.

40 P.C Semprevivo 1979

Teams in Information Systems Development

Yourdon Inc.

41. K.T. Orr 1977

Structured Systems Development

Yourdon Inc.

42. M.A. Jackson 1983

System Development

Prentice Hall

43. C.B. Jones 1980

Software Development: A Rigorous Approach

Prentice Hall

44. D. Coleman 1978

A Structured Programming Approach to Data, pp. 209

Macmillan Press Ltd.

45. P.C. Treleaven 1978

Exploiting program concurrency in computing systems

Technical report University of Newcastle

46. C.A. Hoare 1978

Communicating Sequential Processes

Communications of the ACM Vol. 21, No. 8, pp. 666-677

47. E.W. Dijkstra 1975

Gaurded commands, nondeterminancy and formal derivation of programs

Communications of the ACM, Vol. 18, No. 8, pp. 453-457

48. J. Backus 1978

Can Programminmg Be Liberated from the von Neumann style ?

A Functional Style and Its Algebra of Programs

Communications of the ACM, Vol. 21, No. 8, pp. 613-641

49. P. Henderson 1980

Functional Programming: Applications and Implementation

Prentice Hall

50. A. Church 1941

The Calculi Of Lambda-Conversion

Princeton Univ. Press

51. J. McCarthy 1960

Recursive Functions of Symbolic Expressions and their computations by machines

Communications of the ACM, Vol. 3, No. 4, 184-195

52. B.A. Sheil 1981

The Psychological Study Of Programming

Computing Surveys, Vol. 13, No. 1, pp. 101-120

53. F.T. Baker and H.D. Mills 1973

Chief Programmer Teams

Datamation Vol 10, pp. 58-61

54. F.T. Baker 1972

Chief programmer team management of production programming

I.B.M Systems Journal, Vol 11., No.1, pp. 56-73

55. V.R. Basli and R.W. Reiter 1981

A Controlled Experiment Quantatively Comparing Software Development Approaches

IEEE Transactions on Software Engineering, Vol. 7, No. 3 pp. 299-320

56. H. Sackman, W. Erikson and E. Grant 1968

Exploratory Experimental Studies Comparing Online and Off-line Programming Performance

Communications of the ACM Vol. 11, No. 1, pp. 3-11

57. T.R.Green 1980

Programming as a Cognitive Activity

in, H.T. Smith and T.R.G. Green (editor)

Human Interaction With Computers, pp. 271-320

58. D.W. Embley 1978

Empirical and formal language design applied to a unified control construct for interactive computing

International Journal of Man-Machine Studies, 10, pp. 197-216

59. A. Newell and H.A. Simon 1972

Human Problem Solving

Prentice Hall

60. R. Brooks 1977

Towards a theory of the cognitive processes in computer programming

International Journal of Man-Machine Studies, 9, pp. 735-751

61. M.L. Miller 1978

A structured planning and debugging environment for elementary programming

International Journal of Man-Machine Studies, 11, pp. 79-95

62. B. Shneiderman 1980

Software Psychology

Winthrop Publishers

63. D.E Knuth 1971

An Empirical study of FORTRAN programs

Software Practice and Experience, Vol. 1, No. 2, pp. 105-133

64. A.F. Chalmers 1982

What is thing called Science ?

The Open University Press, pp. 14

65. P.K. Feyerabend 1970

Philosophy of Science:A subject with a great past

in R.H. Stuewer (Editor)

Historical and Philosophical perspectives of Science

Minnesota Studies in Philosophy of Science, Vol. 5

University of Minnesota Press

66. L. Weissman 1974

Psychological Complexity of Computer Programs:An Experimental

Methodology

ACM SIGPLAN NOTices, 9, pp. 25-36

67. B. Shneiderman 1975

Experimental testing in programming languages, stylistic

considerations and design techniques

AFIPS Conference Proceedings, National Computer Conference

pp. 447-452

68. T.R.G. Green, M.E. Sime and M. Fitter 1975

Behavioural Experiments on Programming Languages: Some

Methodological Considerations

MRC Social and Applied Psychology Unit, Memo No. 66

69. R. Brooks 1980

Studying Programmer Behaviour Experimentally: The problems of

a Proper Methodology

Communications of the ACM, Vol. 23, No. 4, pp. 207-213

70. B. Shneiderman and D. McKay 1976

Experimental investigations of computer program Debugging and Modification

Proceedings of the 6th International Ergonomics Association

Santa Monica 1976

71. T. Moher and G.M. Schneider

Methodology and Experimental Research in Software Engineering

International Journal of Man-Machine Studies, 16, pp. 65-87

72. A. Perlis, F. Sayward and M. Shaw 1981 (Editors)

Software Metrics

MIT Press

73. D.R. Cox 1958

Planning of Experiments

Wiley Publications

74. R.A. Fisher 1966

Design of Experiments

Oliver and Boyd, (8th Edition)

75. W.G. Cochran and G.M. Cox 1950

Experimental Designs

Wiley Publications

76. L. Miller 1974

Programming by Non-Programmers

International Journal of Man-Machine Studies, 6, pp. 237-260

77. E. Soloway, K. Ehrlich, J. Bonar and J. Greenspan

What Do Novices Know About Programming?

in A. Badre and B. Shneiderman (Editors)

Directions in Human/Computer interactions, pp. 27-54

Ablex Publishing Corporation

78. K.B. McKeithen, J.S. Reitman, H.H. Rueter and S.C. Hirtle 1981

Knowledge Organisation and Skill Differences in Computer

Programmers

Cognitive Psychology, 13, 307-32

79. B. Adelson 1981

Problem solving and the development cof abstract categories in

programming languages

Memory and Cognition No. 4, Vol 9, 422-433

80. G. Weinberg 1971

The Psychology of Computer Programming, pp.33

Van Nostrand Reinhold Company

81. E.A. Youngs 1974

Human errors in programming

International Journal Of Man-Machine Studies, 6, pp. 361-376

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

83. A.T. Arblaster, M.E. Sime, and T.R.G Green 1979
Jumping to some purpose
The Computer Journal, Vol. 22, No. 2 pp. 105-109

84. T.R.G. Green 1980
IFs and THENs: Is Nesting just for the Birds ?
Software-Practice and Experience, 10, pp. 373-381

85. M.E. Sime, T.R.G. Green and D.J. Guest 1973
Psychological Evaluation Of Two Conditional Constructions Used In Computer Languages
International Journal of Man-Machine Studies, 5, pp. 105-113

86. M.E. Sime, T.R.G. Green and D.J. Guest 1977
Scope marking in computer conditionals a psychological evaluation
International Journal of Man-Machine Studies, 9, pp. 107-118

87. M.E. Sime, A.T.Arblaster and T.R.G. Green 1977
Reducing programming errors in nested conditionals by prescribing a writing procedure
International Journal of Man-Machine Studies, 9, pp. 119-126

88. M.E. Sime, A.T. Arblaster and T.R.G Green 1977

Structuring the programmer's task

Journal of Occupational Psychology, 50, 205-216

89. J.D. Gould 1975

Some Psychological Evidence on How People Debug Computer Programs

International Journal of Man-Machine Studies, 7, 151-182

90. S.B. Sheppard, M.A.Borst, P. Millman, and T.love 1979

Modern coding practices and programmer performance

Computer, Vol. 12, No. 3, pp. 41-49

91. T. Love 1977

An experimental investigation of the effect of program structure on program understanding

ACM SIGPLAN NOTICES, 12, 3, pp. 105-113.

92. G.M. Weinberg and E.L. Schulman 1974

Goals and Performance in Computer Programming

Human Factors, 16, 1, pp. 70-77

93. J.D. Gannon and J.J. Horning 1975

The Impact of Language Design on the Production of Reliable Software

IEEE Transation on Software-Engineering, 1, pp. 179-191

94. P. Reisner, R.F. Boyce and D.D Chamberlin 1975

Human factors evaluation of two data base query languages: SQUARE and SEQUEL

Proceedings of the national computer conference, AFIPS Press

95. P. Reisner 1977

Use of psychological experimentation as an aid to development of a query language

IEEE Transaction on Software Engineering, 3, pp. 218-229

96. B. Curtis, S.B. Sheppard and P. millman 1979

Third time charm: Stronger prediction of programmer performance by software complexity measures

Proceedings of the 4th International conference on software engineering

IEEE, New York, pp. 356-360

97. M.H. Halstead 1977

Elements Of Software Science

Elsevier

98. R.T. Yeh 1979

In Memory of Maurice H. Halstead

IEEE Transaction on Software Engineering, Vol. 5, No. 2,

99. M.R. Woodward, M.A. Hennel and D. Hedley 1979

A Measure of Control Flow Complexity in Program Text

IEEE Transaction on Software Engineering, Vol. 5, No. 1, pp. 45-50

100. T. McCabe 1976

A Complexity Measure

IEEE Transaction on Software Engineering, 2, 6, pp. 308-320

101. R.D. Gordon and M.H. Halstead 1976

An Experiment comparing FORTRAN programing times with software physics hypothesis
Purdue University, Tech. Rep. CSD-TR-167

102. L.H. Cornell and M.H. Halstead 1976
Predicting the number of bugs in a program module.
Purdue University, Tech. Rep. CSD-TR-205

103. R.D. Gordon 1979
Measuring Improvements in Program Clarity
IEEE Transaction on Software Engineering, Vol. 5, No. 2, pp. 79-90

104. B. Curtis, S.B. Sheppard, P. Millman, M.A. Borst and T. Love 1979
Measuring the Psychological Complexity of Software Maintainance Tasks with the Halstead and McCabe metrics
IEEE Transaction on Software Engineering, 5, 2, pp. 96-104

105. B. Shneiderman 1977
Measuring computer program quality and comprehension
International Journal of Man-Machine Studies, 9, pp. 465-478

106. W.G. Chase and H.A. Simon 1973
Perception in chess
Cognitive Psychology, 4, pp. 55-81

107. F.I.M. Clark and R.S. Lockhart 1972
Levels of Processing: A Framework for Memory Research
Journal of Verbal Learning and Verbal Behaviour, 11, pp. 671-684

108. N. Hammond, P. Barnard, A.H. Jorgensen and I. Clark. 1982
Naming Commands: An Analysis of Designers' Naming Behaviour.
Proceedings of a Conference on the Psychology of Problem
Solving with Computers: Cognitive Engineering
Vjrie University, Amsterdam

109. C. Leach 1979
Introduction to Statistics
A Nonparametric Approach for SocialScientist
John Wiley & Sons

110. J.G. Greeno 1973
The structure of memory and the process of solving problems.
in, Contemporary issues in cognitive psychology:
the Loyola Symposium (ed. R.L. Solso) pp 103-131
V.H.Winston & Sons

111. J.M. Hoc 1977
Role of mental representation in learning a program language.
International Journal Of Man-Machine Studies, 9, 87-1o5

112. C. Leach 1982
Op. Cit., pp. 169-180
John Wiley & Sons

113. C. Leach
Op. Cit., pp. 80-85.

114. J.Greene & M. d'Oliveira 1978

Cognitive Psychology

Methodology Handbook, part 1, pp 56-58

Open University Press

115. S. Siegel 1956

Nonparametric Statistics for the Behavioral Sciences

McGraw-Hill

116. C. Leach 1982

Op. Cit., pp 65-66

117. J.B. Goodenough and S.L. Gerhart 1975

Towards a theory of test data selection

IEEE, Transactions on Software Engineering, SE-1,2, 156-173

118. C. Leach 1982

Op. Cit., pp 134-135

119. J.Greene & M. d'Oliveira 1978

Op. Cit., pp. 59-61

120. R.B. Allen 1981

in, A.Badre & B.Shneiderman (Editors)

Cognitive factors in human interaction with computers

Directions in Human/Computer Interaction, pp 1-26

Ablex Publishing Corporation

121. B. Shneiderman & R. Mayer 1979

Syntactic/Semantic interactions in programmer behaviour:

A Model and experimental results
International Journal of Computer and Information Sciences,
Vol 8, No 3, pp. 219-235

122. M.E. Attwood & H.R. Ramsey 1978
Cognitive structure in the comprehension and memory of computer
programs:An investigation of computer program debugging.
Army Research Institute

123. W. Kintsch 1978
Comprehension and memory text
in W.K. Estes (Editor)
Handbook of learning and cognitive
processes Vol 6
Lawerence Erlbaum Associates

124. G.A. Miller 1956
The magical number seven, plus or minus two:
Some limits on our capacity for processing information
Psychological Review, 63, pp. 81-97

125. P.H. Linsday & D.A. Norman 1977
Human Information Processing, pp 381-417
Academic Press

126. E.A. Feigenbaum 1970
Information processing and memory
in D.A. Norman, (Editor)
Models of memory, pp. 451-469
Academic Press

127. P. Pollard 1982

Human reasoning:Some possibile effects of availabilty

Cognition 12, pp. 65-96

128. A. Tversky & D. Kahneman 1973

Availability:a heuristic for judging frequency and probability

Cognitive Psychology 5, pp. 207-232

129. S.K. Card, T.P. Moran & A. Newell 1980

Computer text editing:An information processing analysis of

a routine cognitive skill

Cognitive Psychology 12, pp. 32-74

130. N. Wirth 1974

On the composition of well-structured programs

Computing Survey 6, pp. 247-259

131. J.M. Hoc 1982

Analysis of beginners problem solving strategies in programmming

Proceedings of a conference on the psychology of problem solving

with computers: Cognitive Engineering

Vrijie University Amsterdam

132. H. Kahney 1983

The behaviour of novice and expert problem solvers.

Artificial Intelligence and Simulation of Behaviour Quarterly

Issue 48, pp. 20-24

133. D.E. Rumelhart 1977

Introduction to Human Information Processing

John Wiley & Sons

134. R. Young 1979

in, D. Michie (editor)

Expert Systems in the Micro-electronic age

Production systems for modelling human cognition pp. 35-45

Edinburgh University Press