The psychology of programming for non-programmers

This item was submitted to Loughborough University’s Institutional Repository by the/an author.

Additional Information:


Metadata Record: https://dspace.lboro.ac.uk/2134/26882

Publisher: © Nick P. Rousseau

Rights: This work is made available according to the conditions of the Creative Commons Attribution-NonCommercial-NoDerivatives 2.5 Generic (CC BY-NC-ND 2.5) licence. Full details of this licence are available at: http://creativecommons.org/licenses/by-nc-nd/2.5/

Please cite the published version.
This item was submitted to Loughborough University as a PhD thesis by the author and is made available in the Institutional Repository (https://dspace.lboro.ac.uk/) under the following Creative Commons Licence conditions.

Attributes-NonCommercial-NoDerivs 2.5

You are free:

- to copy, distribute, display, and perform the work

Under the following conditions:

**Attribution.** You must attribute the work in the manner specified by the author or licensor.

**Noncommercial.** You may not use this work for commercial purposes.

**No Derivative Works.** You may not alter, transform, or build upon this work.

- For any reuse or distribution, you must make clear to others the license terms of this work.
- Any of these conditions can be waived if you get permission from the copyright holder.

Your other rights are in no way affected by the above.

This is a human-readable summary of the Legal Code (the full license).

For the full text of this licence, please go to:
http://creativecommons.org/licenses/by-nc-nd/2.5/
The Psychology of Programming for Non-Programmers

by

Nick P. Rousseau

A Doctoral Thesis
Submitted in partial fulfillment of the requirements for the award of Doctor of Philosophy of the Loughborough University of Technology

21 August 1990

© by Nick P. Rousseau 1990
The Psychology of Programming for Non-Programmers

Intermittent computer users, generally unable to program, often need more flexibility than current applications can offer them. A first step to providing such flexibility is to consider the psychological issues underlying the users' needs and the communication of these needs. This thesis does this by exploring the possibility of "Automatic Programming" where users communicate their requirements and the computer generates programs to meet them. A series of studies of the communication of needs locate the main issues as being in the processes of:

* deciding what one wants (requirements generation),
* describing it (requirements communication),
* deciding whether the program offered is adequate (evaluation).

In deciding on their needs, subjects generally generate a concrete description of a program that meets them. The extent of the knowledge of the constraints involved will determine how appropriate the program specification is. As the programming flexibility increases, the constraints involved become more complex. Offering the user greater flexibility will still necessitate, therefore, an increased body of knowledge.

The means of describing the program required should, ideally, be both familiar to the user and appropriate to what is to be communicated. While "Natural Language" is familiar, it is argued that it is not appropriate. Appropriate languages, implying the use of structured notations, will only be acquired through practice, however.

Finally, the user must decide whether the program offered is to be accepted. Effective evaluation must involve employing appropriate information gathering techniques and a means for determining how satisfactory the program is. Difficulties that exist in this area for untrained users and limitations to the extent to which the computer can support these processes are considered.

The thesis discusses the sources of problems with requirements specification as a means to Automatic Programming for users with minimal expertise. The conclusions of the thesis are as follows. For the experienced users it is debatable whether they would want a tool that must necessarily take some control out of their hands. For the intermittent user, without considerable advances in Artificial Intelligence, the constraints that may have to be imposed on what can be made available could result in the generation of applications no better than those currently available. It seems likely that the most satisfactory solution would involve both a prototyping tool and a human expert assistant.
Do you not know?
Have you not heard?
The Lord is the everlasting God, the Creator of the ends of the earth.
He will not grow tired or weary, and his understanding no-one can fathom.
He gives strength to the weary and increases the power of the weak.
Even youths grow tired and weary, and young men stumble and fall;
but those who hope in the Lord will renew their strength.
They will soar on wings like eagles;
they will run and not grow weary,
they will walk and not be faint.

ACKNOWLEDGEMENTS

My supervisor, Mark Lansdale, for support, advice and encouragement throughout.

The staff, technicians and students of the Human Sciences Dept. for their cooperation and help, particularly Andrew Shepherd and the other members of the Cognitive Ergonomics Research Group.

ITT (Europe) and Alcatel N.V. for making this work possible.

Psion plc for the provision of equipment.

Eleanor, for struggling with me.

Colin Garner and my many other friends whose prayerful support has been invaluable.

My family for providing places to escape to.

Trademarks:

HyperCard, HyperTalk, Macintosh and MacroMaker are trademarks of Apple Computer, Inc.
Visual Interactive Programming (VIP) is a trademark of Mainstay.
Prototyper is a trademark of SmethersBarnes & SmethersBarnes Publishing Division.
LabVIEW is a registered trademark of National Instruments Corporation.
Matrix Layout is a trademark of Matrix Software Technology Ltd.
World Builder is a trademark of Silicon Beach Software, Inc.
Wingz is a trademark of Informix Software, Inc.
Prograph is a trademark of the Gunakara Sun Systems Ltd.
Psion is a registered trademark and Organiser II is a trademark of Psion plc.
# TABLE OF CONTENTS

Chapter 1: Introduction ............................................. 1  
1.1. Introduction ................................................... 2  
1.2. The Dilemma Faced by Company Managers ................... 5  
1.3. Application User Versus Application Developer .............. 7  
1.4. The Characteristics of the Non-Programmer ................ 9  
1.5. A Statement of the Goal of the Thesis ...................... 10  
1.6. Research Methodology in Human-Computer Interaction ...... 12  
1.7. The Approach Adopted in this Thesis ....................... 20  
1.8. Summary ...................................................... 25  

Section 1: The General Nature of the Solution ................. 26  

Chapter 2: The Nature of Programming Expertise ............... 27  
2.1. Introduction .................................................. 27  
2.2. The Nature of Programming .................................. 27  
2.3. The Nature of Programming Expertise ...................... 31  
2.4. The Possibilities for Improving the Acquisition of  
Programming Expertise .......................................... 43  
2.5. Conclusion ................................................... 44  

Chapter 3: What Can be Learnt From Existing Programming  
Tools? .................................................................. 46  
3.1. Introduction ..................................................... 46  
3.2. The Approach Taken .......................................... 46  
3.3. Aids to Program Design ...................................... 49  
3.4. Support for Writing Code .................................. 60  
3.5. Support for Reading Code ................................... 70  
3.6. Disguising the Programming Process ....................... 84  
3.7. Discussion ...................................................... 85  
3.8. Conclusions .................................................... 90  

Chapter 4: What Can We Learn From Human-Human  
Dialogue? .............................................................. 92  
4.1. Introduction ...................................................... 92  
4.2. Current Understanding of Requirements Specification  
Dialogues Between Humans ....................................... 93  
4.3. Programmer-Client Dialogue Study ........................ 99  
4.4. The Method ...................................................... 100  
4.5. The Observations and Discussion ........................... 103  
4.6. The Search for a Way Forward .............................. 123  
Dialogues .......................................................... 126  
4.9. Conclusions ..................................................... 133
Section 2: The Elements of the Interaction .......................... 135

Chapter 5: Generating a Specification ...................... 136
  5.1. Introduction ............................................. 136
  5.2. Complete Task Analysis - JEEVES in Control ........ 137
  5.3. The User Controls the Process ....................... 139
  5.4. Observations from the Programmer-Client Study ...... 140
  5.5. What Information is to be Communicated and How .... 141
  5.6. Mental Representations Arising From Information Presented .................. 145
  5.7. Communicating Information During the Dialogue .... 147
  5.8. The Use of a Program Skeleton as a Basis for Specification Generation .... 148
  5.9. Method .................................................. 149
  5.10. Observations .......................................... 151
  5.11. Discussion ........................................... 154
  5.12. The Use of Descriptions of Generic Program Facilities and Examples .......... 156
  5.13. Method .................................................. 156
  5.14. Observations and Discussion ........................ 161
  5.15. Discussion ........................................... 169
  5.16. Conclusion ............................................ 178

Chapter 6: The Communication of the Specification ....... 180
  6.1. Introduction to the Chapter ......................... 180
  6.2. What Users are Wanting to Communicate ............. 181
  6.3. Is "Natural Language" The Best Means of Communicating With JEEVES? .............. 182
  6.4. What Language Would be Appropriate for Communication With JEEVES? ............ 188
  6.5. A Framework for the Virtual Program Specification Dialogue ..................... 188
  6.6. An Environment for Specifying the Virtual Program .................. 189
  6.7. The Language of Specification ....................... 192
  6.8. Empirical Study of the Use of Structured Notations by Novices .................. 194
  6.9. Method .................................................. 196
  6.10. Results and Discussion ................................ 203
  6.11. Conclusions ............................................ 214

Chapter 7: End-User Evaluation: A Consideration of the Processes and Requirements ...... 216
  7.1. Introduction ............................................. 216
  7.2. The Importance of Evaluation ....................... 217
  7.3. The Basic Elements of Evaluation .................... 218
  7.4. The Study ............................................... 219
  7.5. Method .................................................. 219
  7.6. Comments Noted ........................................ 222
  7.7. The Presentation of the Proposed Program ........... 226
  7.8. The Process of End-User Evaluation ................ 227
  7.9. Testing Programs ....................................... 228
  7.10. A Study of End-User Evaluation .................... 229
  7.11. Method .................................................. 230
  7.12. Results .................................................. 234
  7.13. The Process of Evaluation ........................... 251
  7.14. The Relevance to JEEVES ............................ 256
  7.15. Conclusions ............................................ 260
Chapter 8: Discussion and Conclusions ................. 261
8.1. Introduction .................................................. 261
8.2. Review of Thesis ............................................ 261
8.3. New Research Techniques Developed ................. 265
8.4. A Discussion of the Results of This Work ............... 267
8.5. Relevance to the Design of JEEVES ..................... 268
8.6. Relevance to the Future of "Intelligent" Interfaces in
     General ..................................................... 273
8.7. Relevance to Cognitive/Social Science ................. 276
8.8. Conclusions .................................................. 278

References ...................................................... 282

Appendices

Appendix I Material from the Programming-Client Study:
   A) The Task Descriptions Given
   B) Detailed Description of the Taxonomy Used
   C) Two Examples of SNOBOL4 Programs Used
   D) Graphs of the Dialogues

Appendix II Material from the Bibliographies Study:
   A) Summarised Descriptions of Subjects' Past Experience of
      Reference Systems
   B) Summarised Descriptions of Subjects' Ideal Reference System
   C) The Dialogues between the Subjects & the Experimenter
      Represented in Terms of the Different Activities & the Duration of
      Each
   D) The Relationships between the Needs & Specifications across the
      Different Stages, for the Different Subjects

Appendix III Material from the Study Comparing Notations
   A) The Description of Flowcharts & JSDs Used
   B) The Flowchart & JSD of the Recipe for Making Lasagne Used in
      Tasks One & Two
   C) The Diagram Segments Used in Task Three
   D) The Cake Recipe Used in Task Four

Appendix IV Material from the Study of End-User Evaluation
   A) Demonstration of the Diary of the Psion Organiser II Model LZ
   B) List of Basic Diary Tasks that Psion Could not Support.
CHAPTER 1

INTRODUCTION

Abstract to Chapter 1

In this chapter the overall goal addressed in this thesis is introduced and some important issues in relation to how it is achieved are discussed. It starts by describing the situation of the managing executive whose computer tools do not provide the facilities required without being programmed. This is seen as too costly an activity for someone in such a position. The characteristics of users in this position are described. They have considerable knowledge of their own domain but little detailed knowledge of computers or the sorts of abstractions they employ.

The goal of this thesis is to consider the psychological issues of providing programming power to such users. This extends to a consideration of the questions that must be answered in the design of any tool that would aim to do that combined with an assessment of the feasibility of such a project. It is anticipated that the results of this would have wider implications for "intelligent" interfaces in general.

The methods appropriate to this work are then discussed. Problems with the traditional, formal approach to empirical work are identified and alternatives that could be seen to offer valuable, if less rigorous, results described. In particular, it is argued that it is not appropriate to try to construct a solution to the problem at this stage of the work.

Figure 1.1a. The options available to John. The available packages do not allow John to manipulate or present the data in the desired manner.
1.1. Introduction

Picture the scene. John Smith, product manager of Zippo, the latest washing powder that removes dirt and odours while enhancing the original colours and impregnating the laundry with a pleasant smell, has to make some crucial decisions about the next publicity campaign. He has some ideas but to justify these to the rest of his team he needs facts and figures about sales and the state of the market. He has a terminal on his desk that can access all the information he needs but this is in the form of raw data and it needs to be interpreted and presented in a convincing light. What can he do? Figure 1.1, a, b & c show the options available to John with current technology.

![Diagram](image_url)

*Figure 1.1b. The problem with using programmers can lie both in communicating what is wanted and in ensuring that it will be done in time.*

The database package that is available can generate some pretty smart graphics but none of the graphs that are on offer really show the information in quite the right light. Also, there would need to be some massaging of the figures if they are to support the arguments that John wants to present. It seems likely that generic software will not be sufficient (Figure 1.1a). The next option is to send the problem down to the programming team and get them to put together a data presentation package that will provide all the tools and facilities required. This will require a full specification from John of what is required and, knowing their back-log of work, it will not be ready in time for the meeting on Monday (Figure 1.1b). It looks like John has no choice but to do the coding himself. It is quite a while since he last had a look at the programming manual and he will probably have to remind himself of just about every function needed. And then he will have to debug it. The meeting is, however, in only three
days time. It looks like he will have to resort to paper and pencil again for the presentations and have to cope without the necessary statistics being readily available (Figure 1.1c). John starts to dream...

John imagines a computer that would be as good as the best butler/secretary. He would call it JEEVES. He would be able to communicate his needs to it in an easy way & it would generate the appropriate code (Figure 1.2).

JEEVES would not be constrained by a set of available graphical options. John could tell it to construct novel ones from simple building blocks, following his instructions.
He would not mind having to explain what he wanted, just so long as he did not have to spend ages learning some strange language to do it in. He would even be happy with having to teach JEEVES some basic tasks at the outset that could then be called upon when needed. After all that is what one has to do with a new secretary.

The scenario presented above occurs, with variations, every day for vast numbers of computer users. This thesis explores some of the psychological issues that are raised by such an ideal computer as John is imagining. As such, it performs a similar role to the science fiction novel, albeit in a more rigorous and hopefully convincing manner. Science fiction, by accepting reality but with the extension of some fantastic idea can explore the implications of that concept (Philmus, 70 and Griffiths, 80). In a similar way, this thesis assumes that current qualitative constraints hold on both computing facilities and our understanding of human cognition. The conclusions drawn will not depend on solving all the problems of interface design & psychology before becoming relevant. It explores how programming could be made available to Non-Programmers and the sorts of issues that must necessarily arise if tools that provide this are to be developed. This, like science fiction, forces one to consider what may well not be a current problem and, by having considered it, puts one in a better position to deal with the problem as it starts to manifest itself (Griffiths, 80).

This thesis makes no attempt to suggest how such a computer could be built or whether it would be technically feasible. Rather it looks at the sorts of dilemmas and problems that would need to be faced if it were ever to become a reality. The questions addressed initially in the thesis relate to current approaches to making programming accessible to non-programmers. This leads to a consideration of how much of programming as we know it must be changed for this goal to be realisable. It is argued that there is a need for a fundamental re-shaping of the process by means of the use of tools supporting "automatic programming". The central psychological issues in the design of such tools are identified and the problems inherent in them considered. This work is found to have value for the wider areas of "intelligent" interface design and cognitive psychology in general. Finally, the novelty of the perspective forced the use of novel empirical methods and these represent an important contribution to the area.

This thesis is divided into four main parts. It starts with this introductory chapter and ends with Chapter 8 that discusses the findings and their implications and draws what conclusions are appropriate from this. Between these two chapters are two sections, Section I and Section II. Section I examines the current state of programming understanding and technology and identifies the general nature of the sort of
programming tool that will best be able to meet the specification described above. Having done that, it identifies the main areas that exist in the psychological issues pertinent to such a tool and that are likely to contain the major issues. In Section II are three chapters that each take one of these areas and explore it in more detail.

Because this is not a thesis in the traditional mould where a question based within a particular theoretical paradigm is asked and empirical evidence presented to establish an answer, it requires some explication, especially with regard the methods used. This chapter, therefore, explains the goals of the thesis and the ways in which these goals are achieved. It points to the questions that will be raised during its course and the extent to which the reader can expect to have them answered. The chapter starts, then, with a more reasoned consideration of the situation typically faced by company managers in the current computer climate, before considering the nature of the problem that is being considered and the methods that are appropriate for addressing it.

1.2. The Dilemma Faced by Company Managers

The main problem that is focussed on in this thesis is that of computer users who are frustrated by the need to generate applications that will support specific tasks but are unable or unwilling to learn to program. Licklider (60) points out that this is often true of military commanders and scientists but, for the purposes of this thesis attention will focus rather more on the company manager. The assumption behind this is that one cannot consider the detailed activity of all such classes of users and that insights obtained will be likely to generalise to other domains than those of primary concern.

In the early days of computerisation in companies a group of employees would exist whose sole purpose it was to handle the information needs of the managers by acting as an intermediary between the computers and the information users (McLean, 89). They would generate the applications required and maintain and improve them. But this resulted in delays for the managers who needed information rapidly and who became frustrated with dealing with programmer intermediaries. In addition to this, either the managers could not describe their needs in detail, or else their needs were changing because of changes in the company or the environment. This resulted in a situation where the ability to adapt systems and tools rapidly was vital, either when managers discovered a need of which they had not been aware when giving requirements to their programmers, or that had developed since then. These problems still exist. It is important, therefore, to have systems that can be developed rapidly and adapted easily and, ideally, the managers must be able to do as much of this as possible themselves (see also Watt, 87 and Bouchier, Horton & Williams, 87).
Given this, it is useful to examine the computing needs of managers. These needs can be distinguished from the need to maintain the basic functioning of the company (e.g. process sales, prepare letters, etc.) in that they are more concerned with the support of decisions (McLean, 89). In particular, managers need "information" and not just "data" and to be able to ask particular questions that the computer answers by analysing the data in the appropriate way (Eason, 73, Benson, 89). Another, related class of tool identified by Benson is those that enable the modelling of complex situations so that "what if" questions can be asked. These represent even more complex problems for the programmer to predict and allow for in the design. One final category of computer tool used by managers but not discussed by Benson is those that enable the manager to plan and organise. These are essentially tools for structuring thought, such as time managers or project planners, although they can be extended in many ways. The battery of particular tools that a given manager can work with, or requires for a particular project, will vary (Kelly & Chapanis, 82).

The task types identified tend to have one thing in common: they all involve very particular needs that are not easily supported by generic tools. Indeed, it may well be that the best way to achieve tools that will provide the support managers need is by means of systems that can be customised to suit each individual manager (Martin, 73). The only way this can be achieved realistically, especially given the changing needs of most managers, is for managers to generate their own tools, with the facility to program in new features as they discover the need for them. This is likely to be true of all classes of tool described above. This forces one to focus on the manager and consider the scope that is available for him/her to do this. A description of the typical manager would include the attributes listed in Figure 1.3.

Managers require systems, therefore, that offer high ease of use with minimal need for training. Even if they could manage to attend training sessions or read manuals, they would not remember the material for long, if their computer use remained intermittent. It is because of this combination of putting very high demands on the computer tools that will serve them, and of requiring ones that are very easy to use, that managers are described as the hardest group of computer users to cater for (Eason, 73).

This results in a situation where two sorts of managers emerge: those who do not take advantage of computing power and those who accept the cost of learning to program. Early studies of managers as computer users (e.g. Eason, 75) predicted that the computer use of managers was too intermittent for them to develop their own
applications. This has not been commented on recently but it is likely that a considerable number of managers do not use the available computer technology to any great extent at all, because of the problems outlined above. More recent studies (eg. Benson, 89) have found managers who have been forced to become programmers, at least part of their time. This latter study was of the computer use of managers in the years 1982/1983 and consisted of interviews with over 65 managers in twenty locations in the USA. Thirty-four of these managers did some programming in at least one of the areas described above using advanced application developer tools. One company that was surveyed had found more than twenty of its managers were spending over 50% of their time programming, but this was seen as a rare occurrence. It was also found, however, that the managers were concerned about the extent to which they did have to become programmers, as this seemed to change their job-image.

This survey demonstrates both the reality of the need managers have to develop their own tools, and the cost that is currently incurred in doing this. It would be extremely valuable to develop systems that allow managers and other similar users to construct tools. The reason why current systems fail to provide this facility is explained in the next section.

1.3. Application User Versus Application Developer

To understand the dilemma facing the designers of computer applications who want to provide tools for managers, it is helpful to know something of the development of computers. Initially all interaction with computers was by constructing detailed instructions of all the steps the computer had to go through to perform the desired task. These were the first programs and were written in a very basic, abstract, and machine oriented language; machine code. To do this required a good knowledge of how the computer worked and considerable expertise. Also, the computer users had to predict...
everything that could happen in the course of the program so that contingency actions could be built in (Licklider, 60). In order to make life easier for themselves, programmers wrote other programs called "compilers". Using these it was possible to write instructions in a higher level language, that could then be "compiled" into machine code. Thus, commonly used groups of commands from machine code were given labels that could be referred to in the higher level language. The concept of the compiler was gradually developed until programs were written that provided the user with a basic repertoire of possible actions, but where each action initiated a powerful sequence of computer behaviour. These were now called "applications" and could be marketed to, and used by, non-experts. The result of this was that computer users could be divided into two camps. There were those who could only use applications written by others (end-users) and there were the programmers, who wrote them. Such applications did not require the user to predict every step of the interaction in advance and could support a more exploratory approach (see eg. Young & Simon, 87).

In the present day most users of computers, whether end-users or programmers, work with languages that are far easier to use than machine code. While the difference between the means of interacting with the computers seem qualitatively different, this distinction can be seen to be less a question of kind than of degree. The main dimensions involved are the breadth of available commands that can be given (including any possibility to combine individual commands to create more specific ones) and the power of the available commands. A tool that offers a large vocabulary of basic commands will offer the user maximum flexibility but require a large investment of work. These are the tools used by programmers. A tool that offers few commands, but where each command is very powerful, considerably reduces the flexibility but allows the user to achieve a great deal with little effort. These are the tools that are used more by end-users. In between these are tools that can be used by either group.

What is clear from this description, however, is that there is a trade-off between the effort that must be invested in achieving goals, and the flexibility available. The sort of computer tool required to support managers must, however, offer great flexibility but demand little effort. It must provide programming power to the non-programmer. This thesis explores the potential for this goal to be reached and identifies the issues that must be understood if it is ever to become a possibility.

Having identified this goal, it is necessary to clarify the terms used. This will locate the domain of interest in a more precise manner and provide a useful basis upon which to
build. The next section examines the term "non-programmer" and the meaning of "programming" is looked at in the next chapter.

1.4. The Characteristics of the Non-Programmer

Users such as managers can be characterised by their lack of computer knowledge and by their unwillingness, or inability due to insufficient resources, to acquire it. These two aspects are discussed below. The term "non-programmer" will be used from now on to refer to this class of user.

Non-programmers can be distinguished from computer professionals by their lack of a detailed knowledge of programming languages and concepts. They may also have little understanding of computers in general. The sorts of abstractions that go on when processing information in a computer will not be familiar to them. When programming in the traditional manner, one has to extract the essential aspects of the information one wants to manipulate, and convert this into the sorts of variables and objects that can be handled by the language. Thus, to solve a problem about working out a wage rise, one must create memory locations to hold the numeric values that correspond to the salaries and formally specify the manipulations that represent what must be done to give them the increase. The results are then converted back into concrete terms as actual salaries. These sorts of ideas about the ways in which computers solve problems are, therefore, unfamiliar and difficult for non-programmers to handle.

Other characteristics of the non-programmer will be a lack of familiarity with the tool in question and how it achieves goals. There will, however, be quite a degree of expertise in the domain of the goal, as the user understands it. Essentially, one can assume that such users will have considerable expertise in their own field but not in that of computing. Also, it is important to remember that the need that the intended program is to meet will probably have been met before. This will have been by means of other tools, however, computerised or not. Thus, the conception of the goal will be in terms of tools and methods that may not map neatly onto those available in the programming tool. For example, if one is considering an electronic diary tool, the user will have kept details of appointments in a number of forms, in paper diaries, on calendars and possibly in other computer-based diaries. Finally, the non-programmer is likely to be used to communicating about goals and tasks (at least to humans) in the domain in question. It is likely, therefore, that a vocabulary for the domain will already exist, although it may be specific to the context (such as a company's jargon).
It is worth remembering that the level of experience and knowledge of computers in general that one can expect to find in the average computer user is increasing with time. This is both because schools are introducing computers into the curriculum at an increasingly early age and because computers are involved in more and more of everyday life. This does not mean, however, that one can assume that the user will have developed a positive attitude or a wholly accurate understanding of how a computer works. Rather, that certain elements of interacting with computers will have become very familiar, such as using a keyboard and extracting information from a screen.

Non-programmers can be distinguished from those actually intending to learn to program because they are not likely to be motivated to invest the effort to become experts and will probably not be using the tool enough to retain the details of what they have learnt. The problems of novice programmers relate to the best method of teaching accepted programming languages and of producing a rapid learning curve. For non-programmers, this will be a wasted effort, for the most part, as they will not build up experience over time owing to the intermittent usage. They will want a minimum of time to be spent in learning about the functioning of the tool they are using and will be impatient with struggling with novel concepts and abstractions.

It is now appropriate to consider the goals that can be set in relation to this problem. The next section does this.

1.5. A Statement of the Goal of the Thesis

This section describes the goals of this thesis, firstly by locating the general domain of interest, and secondly by stating the questions that it will address. The domain is identified by a focus on psychological rather than technical issues and a search for design questions alone rather than both questions and answers.

One continuum along which any piece of work in the area of Human-Computer Interface (HCI) design must fall is that between a focus on technical issues and a focus on cognitive psychology. Research at the former end might be in the realm of Artificial Intelligence or new interaction methods such as speech input. It would aim to push back the barriers that currently make programming prohibitive to non-programmers by means of technical innovation. Research at the latter end, however, could involve a consideration of the way users conceive of, and communicate, their needs for tools. It could result in statements about the improvements in interfaces that will necessarily precede the achievement of this goal. It could also identify the central areas that will
need to be focussed on if a system of the type described is to be usable. Clearly work of both kinds is necessary if any design goal is ever to be reached. The work described here will focus on the psychological issues involved in this area.

Closely associated with this decision to focus on the psychological issues is the question of whether the search should be for an actual design solution or whether to limit the goal to an understanding of the issues involved. Thus, one can be looking for answers to the design questions, or just to be able to articulate what these questions are. The approach that one chooses will reflect both the current state of understanding of the area in question, and the desired breadth of applicability of the results. These issues are discussed below, with the conclusion being drawn that the goal should merely be to identify the questions and not to find answers to them.

While a large body of work exists that has focussed on the psychology of programming, the bulk of this is aimed primarily at understanding the programming process as it currently manifests itself, with the implicit goal of improving on this in various respects. The improvements that can result from this are, however, of an incremental nature, leading to the development of slightly better tools each time. One of the arguments of this thesis is that, in order to achieve substantial improvements in the programming process, one must engage in a more fundamental study of the processes involved in the specification of requirements and design. It is not sufficient merely to observe how programmers currently use languages to construct or communicate a design to a computer. Rather, there is a need to explore these processes as they occur in general, if one is to be able to identify what possibilities there might be for fundamental improvements in human-computer interaction in this area.

Having said that, it will be shown that there is remarkably little work that has been done in this area. While considerable anecdotal material exists, owing to the fact that requirements elicitation is an everyday experience of many consultants, etc., very few studies have tried to actually observe these sorts of dialogues in situ. The situation is, therefore, one of minimal existing understanding or knowledge. There are no basic, accepted models of how people communicate requirements, or the different processes that occur in these dialogues. In the absence of this, the first step must be to improve on a debate that has little structure, so as to arrive at one that can articulate the main concepts and provide some models that can be tested. One must, therefore, aim to articulate questions rather than start by suggesting answers.
If one were in a position of having to produce a concrete product at the end of the work, as a goal in itself, then one would clearly be interested in turning the ideas resulting from the work into designs and applications. In particular, one would start to focus on the ideas that were most relevant to the domain at which the product was to be aimed and to see them in terms of the hardware/software architecture one must ultimately use to implement it. While the final product could be said to be a "demonstrator" of the issues and conclusions that had been addressed during the course of the project, the need to actually generate solutions to the problems identified would be likely to limit the scope of the comments that could be made. Given that this thesis represents a very early attempt to articulate the issues of this area, it would seem premature to combine that with actually identifying solutions or prescribing how this problem should be solved. In addition to this, the value of this work will be much greater if more general concepts or properties can be identified, than if a more domain-specific set of concepts are focussed on.

Finally, it seems possible that keeping the focus general could mean that the insights obtained will be relevant to other domains of "intelligent interface" than just those of programmable systems. The questions that one can expect to be able to answer within this piece of work will include, therefore those listed in Figure 1.4.

It is also to be expected that this work will include the identification of a number of research methods and tools that will be useful for more detailed investigation of the areas identified.

1.6. Research Methodology in Human-Computer Interaction

Having established the goals of this project, one must turn to considering the methods that are appropriate for achieving them. Given that the area of interest is not one that has been looked at explicitly before, there will need to be a considerable amount of examination of research and other literature from a wide range of disciplines. This will be in order to identify any insights that have emerged from related work that might be relevant. While the espoused topic of this thesis is programming, it actually turns out that the most valuable source of material is not from research focussed directly on programming. This is essentially because most of the research that has been done on programming has been technology-specific and has focussed on the interaction between expertise and specific notations or environments used. It has not looked at the more general properties of the early stages of the programming process when programmers are constructing an understanding of their requirements.
What approaches are currently being adopted to develop interfaces to programmable tools for non-programmers, and how effective are they?

How fundamental must any re-design of the programming process be for it to become accessible to non-programmers?

Are there any general properties of the process of generating requirements for the design of a program, and if so, what are they and what are the critical steps in this?

What are the issues when attempting to design an interface that will support the user's activity and communication at these critical steps?

What can one anticipate being the main problems in the development of tools of this type?

What general features of human information processing can be identified from this analysis that will be relevant to other "intelligent interfaces"?

Figure 1.4. The questions that may be answered in this thesis

A considerable body of work is described in this thesis and this requires more justification. This will be given in two sections. In the first, the dilemmas facing scientists wanting to expand our understanding of human-computer interaction are articulated. The intention in this is to make clear to the reader that much of the classical methodology of experimental science cannot be used in this area. In the second, the particular needs of this project are discussed.

1.6.1. The Formality/Informality Continuum

One important dimension along which studies can be located is that of formality of paradigm. This refers to a number of the aspects of the design of the study. Formal studies attempt to test specific hypotheses while studies at the more informal end of this continuum are more appropriately identified as information-gathering exercises, with no clear hypothesis being proposed. This dimension is closely linked to the experimental design chosen and the data that is collected.

Experimental design in more formal studies consist of attempts to observe the effects on a pre-defined set of dependent variables of some independent variable. One can, therefore, test hypotheses about the relationship between these variables. Central to the ability to perform studies of this type are the factors shown in Figure 1.5.

In addition to these factors, the formal approach is most useful within a scientific paradigm where the goal is to achieve increasingly accurate models of the processes in question, within some generally accepted framework.

An idealised example of such a study would be testing a hypothesis about the nature of memory. The situation would be one where a model exists that accounts for the data obtained to date and the goal is to put it to a more rigorous test. To do this, one must
find some piece of behaviour that the model predicts would happen in some specific way. For example, this may be a description of how recall behaviour is affected by some aspect of the conditions of recall. This, then, is the hypothesis. The experiment would compare recall under two or more conditions where as many aspects of the conditions, subjects and tasks as possible were identical, apart from the factor one is interested in. The subjects would perform some recall tasks and quantitative measures of their performance are taken. These are then subjected to statistical analysis to determine whether the data support the original hypothesis, i.e. whether there is evidence for the effect predicted, or for a different effect. This data can then be used either to support the model or to improve it.

* One or more clearly articulated hypotheses derived from an existing model of the process of interest.
* The ability to manipulate the independent variables without affecting other variables so that effects observed can be attributed to the independent variables.
* The ability to measure the resulting independent variables in a way that enables statistical analyses to be performed, so that some measure can be obtained of whether the data support the hypotheses.
* The focus of interest on common properties of the processes observed with individual variation in the data being seen as "random errors" against which the strength of the experimental effect is compared.

Figure 1.5. The factors central to the use of the formal approach to empirical research.

In studies at the opposite extreme of the formality/informality continuum, the lack of clear hypotheses being tested make it inappropriate for the design to involve the comparison of tightly controlled conditions. Instead, the goal of the design is to make some observations where the events observed have maximum relevance to the subject of interest. The lack of clear hypotheses makes it difficult to predict in advance what measures are likely to be most informative. This tends to result in any measures that can be taken being used, including informal interviews, videos of sessions, etc., as well as possible quantitative measures. This, in turn, makes the use of statistical methods, apart from descriptive statistics, difficult. There is never a total lack of hypotheses and one is always constrained to some extent by one's preconceived ideas about what behaviour or events are important. This determines, to a great extent, the experimental tools used. Data analysis becomes more of a process of sifting, manipulating and recoding in a search for representation that results in the identification of some part of the phenomenon that can be commented on in a meaningful way.

For example, one can consider the work of someone interested in the effect of a new system of directing traffic on driving behaviour. There may be no model that is relevant to this and no clear hypothesis. The interest is solely on what effects can be found and, possibly on what problems emerge. Thus, an observational study is set up
where the design consists merely of setting up the directions, instructing drivers in their usage and telling them to negotiate a system of roads. There will be an attempt to gather as much data as possible about how cars negotiated the area, possibly taking videos of this, as well as questionnaires and interviews afterwards. While the experimenters are attempting to notice anything and everything, they are likely to be particularly quick to pick up on any evidence of difficulty or accidents. Many different metrics can be taken from the videos and questionnaire results and this data can clearly be analysed in many different ways. It is not, however, possible to do more than use statistics to describe the patterns of behaviour seen, or frequencies of accident or error types.

Closely related to this continuum of formality is the question of whether the studies are lab-based or involve the observation of people's behaviour as it happens in context. This has been discussed by Curtis (86) and Soloway (86a) who both argue for the need for more of the latter. The reason for this is mainly the need for increased external validity of the results obtained. If one studies a non-representative sample (e.g. undergraduate students) in an artificial setting performing unrealistic tasks, then one cannot claim that the conclusions drawn are universally valid. Artificiality tends to be a feature of the more formal experimental type, owing to the need for control over the possibly confounding factors.

Given the above, should HCI research adopt the formal approach? The main measure of quality of much research, especially in experimental psychology, is the extent to which the formal method has been used. For a piece of research to be seen as valid and valuable it must have all the required elements. There should be a good clear progression from a testable model to a hypothesis to a well-controlled experiment. The experiment should clearly reduce the uncertainty about the model and make effective use of statistical techniques. In the minds of many people this criterion should be transferred to the domain of research in HCI. Certainly there are large numbers of papers that have attempted to do this (e.g. Waern, 87, Perlman, 84, McKeithen, Reitman, Rueter & Hirtle, 81). There are, however, substantial arguments against this approach being the most appropriate, both for HCI in general and for the domain of interest of this thesis in particular. The arguments are concerned with the issues mentioned in the previous paragraphs. For the sake of clarity of exposition, hypotheses within the domain of HCI are divided into three categories. These are hypotheses about the factors that contribute to good design of human-computer interfaces, the processes of human-computer interaction, or methodologies for
producing good interfaces. The prospects and problems for each type are discussed in the following sections.

1.6.2. Design Hypotheses

The first type of hypothesis that could be tested by research in HCI includes those that suggest that a given feature of an interface has a given effect on user performance. A number of such statements exist, mostly in the form of guidelines that suggest design criteria for making an interface more usable. For example, it is generally agreed that one should make changes in the state of the computer clearly evident to the user. One might suggest that amassing a comprehensive collection of such statements is the goal of HCI research, so that new interfaces can be generated that conform to them and are, therefore, better than any alternative. Certainly there is a considerable body of research and writing that seems to have this as an underlying assumption (e.g., Perlman, 84, Karat, McDonald & Anderson, 84). This may be because it would seem superficially that these sorts of statements are the type that are most open to formal testing. There are, however, a number of problems with this approach.

Among the problems with attempting to test a hypothesis of this type in a way that gives the conclusions a useful domain of validity. An example of this sort of hypothesis could be that:

"Menu-based interfaces are better than command-based ones for casual users."

The interaction between a user and a computer is immensely complex. Many different interdependent factors come into play and the resulting system is very hard to control. The evidence seems to be that many attempts to design guidelines follow one of two routes. One is to look for specific guidelines that will be detailed and useful but that end up being overly limited to a particular combination of user/task/tool features. Thus, one may be able to say that for a given user group, on a given type of task, some feature of the interface is better than some other. Even this can be hard to do in practice simply because the different elements of the interface are also interdependent and one often cannot change one without affecting the others. The value of this guideline is limited to situations when one is dealing with that particular combination of user/task/tool. Working with guidelines at this level must also necessarily result in an unworkable number of them, making the identification of ones relevant to one's problem impractical. Finally, this approach would mean that research was always
lagging behind technology because it would have to perform the same tests on every new interface type.

The other option is to carry out a battery of studies with diverse combinations of users, tasks and tools and hope to find some general nugget of information that is valid across all situations. This approach seems to result in guide-lines that are too vague to be of any real value. Certainly they appear to be very hard to implement and designers describe them as hard to use. Reviews of detailed work often end up like this (eg. Milner, 88).

It has been said that in the design of computers it is not possible to predict the effect a change to the interface will have in terms of the tool's usability, etc. (Rosson, Maass & Kellogg, 88). If this is the case, then there seems to be very little point in attempting to develop an understanding at the level of individual features as this will always be negated by the very complexity of the rest of the interface, its context, user group and task characteristics. While it is difficult to prove this statement, one can see that the work required to develop any sort of useful understanding of how the features of an interface interact with each other would be extremely time-consuming and unlikely ever to be possible. Even if it were attempted, the problem of advancing technology would rapidly make any understanding redundant.

A number of criticisms of the work that has been done in HCI have been written (Curtis 86, Weissman, 81). Many of these comment on the fact that formal, experimental rigour has been achieved at the cost of external validity. The subjects used tend to be students rather than real computer users, the tasks subjects have to perform are deliberately simplified to make them manageable within time constraints, the context of the investigation is artificial. The conclusion of many of these workers has been either to improve on the choice of materials, subjects, etc., while maintaining the formal approach (Brooks, 77), or to switch to studying computer use in context, and abandon lab-based studies (Curtis, 86). It would seem, however, that there is still value in lab-based studies if one is willing to pay the price of changing the data gathering approach. Instead of aiming exclusively at quantitative data that, by its nature, is simple and makes little reference to subjects' strategies and other sources of individual difference, one could consider more complex data collection that could lead to a deeper understanding of what the subjects are doing. It does, however, rely on being able to analyse & interpret the resulting data appropriately. This should result in more externally valid results, if the factors that make the lab-based study artificial are taken into account.
1.6.3. Process Hypotheses

The second category of hypothesis includes those that aim to describe the processes of human-computer interaction. These aim to develop models that will allow one to make predictions about how new technologies could be used. For example they can predict with varying degrees of accuracy the time taken to perform tasks using particular tools. These have their own methodological problems and dilemmas. The first involves the selection of the appropriate level of detail for the models that are being constructed, and the second stems from this and involves the data that is obtained from empirical work.

It has already been pointed out that the system of a user interacting with a computer is an immensely complex one. Not only is one wanting to model the basic human side of the interaction consisting of fundamental properties of the cognitive resources available, but one must also include the higher cognitive processes such as learning, planning and the use of strategies. In addition to this, one must model how all this interacts with the myriad of different elements of the computer and task. Any model that attempts to encompass all this must either be incredibly complex or else simplify the way the system is described considerably. This basically means identifying an appropriate level of description. Unfortunately, however, no such model currently exists. Process models in HCI tend either to be very detailed descriptions of how the user's knowledge is converted into basic behaviour such as key-presses, or else to be extremely sketchy.

There are problems for models that fall at either end of this continuum. The very detailed models such as GOMS (Card, Moran & Newell, 83, and others, eg, Command Language Grammar, Moran, 81) suffer by not taking account of strategic knowledge and tend to oversimplify the way knowledge is used. While this results in some useful results when evaluating interfaces, they are very crude and are limited to considering the more formal aspects of the system such as complexity of an interface, transfer across applications or consistency, rather than the human perspective. Models that are less detailed demonstrate the extent to which HCI is still in its infancy as a science. These still consist of little more than the articulation of the different sub-processes that make up a larger process such as programming, with no real claim to be describing the way these sub-processes interact or how the user moves between them. What is needed for a science of HCI to be effective is an understanding of how strategic knowledge and skill is used to make use of both cognitive limitations and the constraints of the interface. Models about this are rare. To develop these, one must start with observational studies that could suggest where one might look. Once one can start to construct such models, one might be able to make HCI less technology driven and start to predict how a novel interface would be used. Equally valuable, however,
may come the assertion that such models are essentially impossible to generate as they would be so complex.

Another problem with the work that is focussed on generating models of human-computer interaction lies in the nature of the empirical work that can be done. Very often the data collected in formal studies are very crude measures of the subjects' performance. It is up to the ingenuity of the experimenter to design a series of experiments that allows conclusions to be drawn about the underlying processes from such data. Thus, measures such as time taken and number of errors are the most frequent. By showing that more errors occur under certain circumstances one can suggest an interpretation of this data in terms of strategies used, etc.. This approach is in marked contrast to that adopted in other disciplines such as discourse analysis where quite a different body of data is collected covering many aspects of the observable behaviour. The interpretation of this data is a major problem but it does provide a greater number of clues to the designer about the interaction. Clearly there are both positive and negative effects of both approaches but it may be possible to find reasons for adopting one approach rather than another under particular circumstances.

In the case of HCI, the less traditional approach would seem to be an important one to consider because interest is in higher level activities and the use of strategies. One will expect to see considerable individual difference in the behaviour of the user because of the higher nature of the activities. This is in comparison with experimental psychology where one might be aiming to look for truly fundamental properties of the mind that are common across different subjects. In this case, individual difference is treated as "random error" and the statistical techniques used are intended to compare the strength of the general effect with that of these sources of variation. In HCI, one is wanting to observe both the common aspects of the interaction and the differences. This is because of the interest in strategies used that are more likely to be individually acquired.

Problems with this approach arise when one is attempting to perform analyses on the data obtained. It is much less amenable to statistical test, being less quantitative, and requires the imposition of categories onto the observed behaviour that can seem artificial. Thus, one could note the different behaviour patterns but, to be able to summarise or communicate one's findings one must be able to describe them. This involves classifying the patterns in ways that enable one to count the occurrences of each type and describe the sequences in which they occur.
One final method for identifying the processes of human-computer interaction that is currently being developed is "Claims Analysis" (Carroll, & Kellogg, 89). It represents an attempt to reconcile the need for more objective approaches with the rich sources of data already available: existing designs. The approach consists of identifying designs that have been found to be successful and extracting insights about user psychology from them. This is described in more detail in Chapter 3 but the main reservation about this approach can be summarised by the observation that it is not clearly proven to be any less subjective than other approaches. Having said that, an adapted version of the approach is adopted in that chapter for the analysis of currently existing programming tools.

1.6.4. Methodology Hypotheses

The final category of hypothesis that can be made in the area of HCI is that which contains statements about good design practice in terms of the methodology employed when moving from a specification to a product. Increasingly this is being seen as a major area that needs consideration. This stems for the most part from the frustrations with the inadequacies of both process and design hypotheses to support the design of computer systems that are usable. These, however, seem to be the hardest to examine using the formal approach as one is talking about processes that are even more complex and high level than human-computer interaction and where the possibilities for controlled experimentation are minimal. This has resulted in the work that is being done on this topic being mainly observational (eg. Curtis, 88, Rosson, Maass & Kellogg, 88, Gould & Lewis, 85).

This review of research in HCI is intended to provide a background for the choice of approach adopted in this thesis. The next section goes on to describe this approach.

1.7. The Approach Adopted in this Thesis

There are three main areas into which the work done in this thesis can be allocated. The first is in the way in which reviews of the literature are used within the text. The second is in the way existing tools and an imaginary programming tool are used. The third is the use of empirical work. The problems with this in HCI have already been described and the solution adopted by this work needs to be stated.

Before doing this, it is important to reiterate the goals of the work. Were this a thesis in the classical mould, it would start with a literature review that would be used to establish an area that requires investigation. This would be followed by a series of
studies that would take the reader from an initial exploration of the area to a statement of a hypothesis or model of the domain that could be tested empirically and then to a final series of studies that would test it. The results of these would then be interpreted in terms of the hypotheses made and conclusions drawn. This thesis does not represent a contribution to an existing paradigm, however, in the sense that Kuhn (see Chalmers, 82) uses the term. Rather it consists of a particular new view of what is in any case an underdeveloped area of research. This results in a choice of approach that will differ considerably from the classical. The nature of the thesis, itself, becomes more of an articulation of issues and, as such, verges on the philosophical, rather than aiming for the justification of a position, supported by a body of empirical data. Having said that, empirical data has been collected and this has enabled certain positions to be taken. In many cases, however, these will require further testing and will be, to some extent, speculative. It may only be possible to test them fully once an attempt is made to build the sort of system that has been described.

1.7.1. The Use of Literature Reviews

The main point about the use of literature reviews in this thesis is that there is relatively little reference to work on programming. The reason for this stems from the fact that most of such work assumes that programming will remain much as it currently is and that it is focussed on how to improve existing tools. As will be seen, a more utopian approach is adopted in this thesis such that most of what constitutes "programming" is discarded in favour of a more radical approach. This results in the absence of an existing, relevant body of literature describing the main areas of interest, from which a topic on which to focus can be carved out.

This results in a much more all-embracing approach to the literature. Rather than attempting to identify every paper written on the psychology of programming, as one might in a more traditional piece of work, it has proved necessary to look to other areas of scientific endeavour for relevant material. In addition to this, the literature found has been used, not as a basis from which to generate interesting gaps and hypotheses to test, but as a source of ideas, inspiration and insight. This is a much less well-structured process and the literature reviews presented are correspondingly more reflective than argumentative.
1.7.2. The Use of Artifacts

There are two main ways in which artifacts are used in this thesis. The first is in an ecological study described in Chapter 3, and the second is in the use of JEEVES, an imaginary computer.

Given the problem of making programming more accessible to non-programmers, it is important, in the first instance, to explore what steps have already been taken in this direction. This was done by means of an informal review of the existing tools that make claims of this sort. The informality of this review was an inevitable result of the nature of research and evaluation in HCI in general. What was desired was to obtain insights into the aspects of these tools that were particularly productive in this respect. Chapter 3 describes the issues in selecting the approach taken in more detail. An analysis of Carroll's "Claims Analysis" is described that argues that to achieve such goals by any formal means is impossible, within the context of a realistic research budget. The best approach is seen to be one that makes the maximum use of existing empirical evidence and understanding to generate plausible hypotheses about what factors are most important in a given tool's performance.

The second use of artifacts in this thesis is as a tool for thought and communication about the overall goal. While the goal of the project was not to generate a design for a computer that would solve the stated problem, it might seem useful to build a simplified version or simulation of one that could be used to test out ideas generated. Indeed, it is likely that it would be much easier to think about the issues and to communicate with others if such a concrete artifact were available to refer to. The danger with this approach, however, would be that the artifact could end up driving the exploration, rather than facilitating it. To develop such an artifact, it would be necessary to select a medium and technological substrate for its construction, whether a simulation or simplified version. This would require certain decisions to be made that might rather be left open. By fixing such issues, others could be influenced and the general direction of the work affected. Finally, the investment of time and effort in generating such an artifact was seen to be excessive. As part of the work that was done towards this thesis, a number of attempts were made to construct exemplars of tools that would be providing users with programming power in various ways. It was found, in fact, that the problems described above were found to hold and that little of value was obtained from them, certainly in comparison with the effort invested in their development. Because they have not contributed much to the thesis, they are not described within it.
It was proposed, therefore, to take a middle road between the construction of an artifact and the rigorous avoidance of any reference to how the goal might be implemented. This road involved the use of a hypothetical computer that was identified as JEEVES (after the fictional butler created by Wodehouse, eg. Wodehouse, 53). This was intended to provide the advantages of a simulation, in that it was possible to refer to it and imagine it, indeed to consider how it might be designed, without any limitation imposed by available technology or the need to construct it. JEEVES will be referred to frequently in the course of this thesis. The intention in doing this is to facilitate the communication of concepts that might otherwise be excessively abstract, rather than to suggest a design goal of the work. Thus, at certain points, statements are made about how JEEVES would have to be designed, or what decisions must be made before some aspect of JEEVES could be specified. These will only represent, however, some investment into imagining how a tool such as JEEVES could work, in order better to identify the issues involved. The thesis can be seen, therefore, as a series of forays into the design process followed by a stepping back to observe the issues and questions that arose from them. Once again, these can be seen to stem from the goal of the thesis as the articulation of issues rather than a more concrete goal such as a design or a well-supported hypothesis.

1.7.3. The Use of Empirical Research

The goals of this thesis are not to test a particular hypothesis about programming, nor are they to answer definite questions about the design of programming tools. Rather they are to identify, articulate, and explore issues that will make it possible for hypotheses to be identified that need to be tested and to facilitate the process of designing the programming tools of the future. Because of this, the success or failure of the thesis in meeting these goals will not depend on the extent to which the empirical studies undertaken can provide sufficient support for some argument about the programming process or desirable design. The role of empirical studies in this thesis is, therefore, unusual and requires some examination.

Given the goal of articulating as thoroughly and usefully as possible the issues that exist for anyone aiming to provide non-programmers with programming power, one must consider what empirical work will be most useful and how this can interact with any exposition of the overall work. Given that formal experiments are likely to be inappropriate at this stage of the research endeavour, one must employ less formal approaches. The goals with these will be to gather useful information about behaviour that can be used to stimulate ideas and representations rather than in a strong way to support arguments for certain positions. If one is interested in a domain about which
one has no prior information or understanding, one must perform data gathering exercises that will be sufficiently general for one to manipulate the data itself in order to identify possible patterns and trends. One cannot use this data to make strong statements about causes and effects but one can build up a set of possible events that can occur. One could then make hypotheses about these events in relation to what may have some causal relations to them.

For example, one may know nothing about the behaviour of dogs and want to build up a science of this. The first step must be to observe some dogs under a variety of conditions and get some measure of the variation in behaviour. Having done that, one can start to construct taxonomies that one believes, in an admittedly subjective way, may be significant. Having done that, one can return to one's data for clues about what led to the different behaviour patterns happening when they did. It is at this point that one can generate some hypotheses that can be tested by further empirical investigation of a more formal nature. If one's goal at the outset is slightly more specific, perhaps to observe dogs with a view to seeing whether they could help blind people (still assuming the lack of any existing science of canine behaviour), then the approach to the data will be more focussed. Not only will one want to classify behaviour patterns into those that could be, in some sense, valuable, those that are neutral, and those that are positively harmful, but one will want to identify clues about the cause and effect relationships that could suggest how to elicit the positive and eliminate the negative behaviour patterns.

As one performs a number of such observational studies, one will start to build up a list of important areas that will need to be investigated further as well as a basic idea of whether guide-dogs for the blind is a workable proposition. This can then be turned into a body of text describing the findings and explaining the conclusions drawn. The descriptions of the observations made will be presented alongside a considerable body of speculative material containing the arguments for the various positions identified as plausible. A number of crucial concepts and issues will be identified and these will be a result both of the empirical investigations carried out and the reasoned speculation and consideration of the researcher. As such, there will be few statements that are presented as supportable positions in relation to the issues identified but many points where the possible positions that could be taken are articulated.

This, essentially, is the nature of this thesis. Considerable quantities of empirical observations are presented but these are used more to facilitate the identification of concepts and issues than to support specific positions in relation to these. As a result,
there are a considerable number of threads that have been left hanging with the intention that further work to pin them down could easily be done.

1.8. Summary

In summary, this chapter has described the problem of company managers who require the flexibility to construct their own computer tools to meet their complex, varied and changing needs. Currently, to meet this requirement, users must program their computers in some way but such users are not in a position to learn to program. This represents a major problem for the software industry as packages tend to demonstrate a trade-off between flexibility and ease of use. The defining characteristics of users in this position are described as including the absence of specific computing skills but the presence of considerable domain skills and knowledge. The area of programming that is of interest is that of generating tools for and by individuals that will be used more than once to perform some task on the computer. The goals of this thesis are specified as consisting of an investigation of the psychological issues that result from this problem and will not include technical considerations or any attempt to find design solutions. A list of questions is given that will be answered by this work.

Finally, methodological issues are discussed and the need for a wide literature review and review of existing tools identified. While no example program will be generated to encapsulate the ideas of this thesis, a hypothetical one, named JEEVES will be used when communication or thought will be facilitated by the use of a concrete artifact to which to refer. Finally, it is argued that empirical work in this area must be informal and should not be seen as intended to test specific hypotheses. Rather the goal is to stimulate the articulation of important issues and provide clues about likely problem sources.
SECTION I

THE GENERAL NATURE OF THE SOLUTION

The first goal of this thesis must be to establish what must be changed about programming if it is to be made accessible to non-programmers. If improvements in teaching practice are all that are required, then the focus of the research must be on learning processes and the shortcomings of current programming courses, manuals, etc. If there must be a change to the tools provided for programming (including the languages used), then one must make a decision about where the changes are most necessary. This chapter and the next explore these questions. Having identified the general area that needs to be addressed by the work, it is then possible to examine it and locate the central issues.

In this section are three chapters. In Chapter 2, the nature of programming expertise is considered and the potential for improvements in teaching practices to make this easier to acquire are examined. This is found to be limited. In Chapter 3, programming tools are investigated. These are found to be limited in their scope and it is concluded that a more radical approach using automatic programming and a more informal approach to the specification of requirements should be investigated. Finally, in Chapter 4, this approach is explored and the areas of most importance identified, within a structure for representing the user-JEEVES dialogue.
CHAPTER 2
THE NATURE OF PROGRAMMING EXPERTISE

Abstract to Chapter 2

This chapter introduces the search for an identification of the best area to look for a solution to the non-programmer's problem. It starts by stating clearly what is to be understood by the term "programming" in this thesis. From there it moves to a consideration of what constitutes programming expertise. This is basically addressing the question of why programming is not an easy activity for the non-programmer. The final part of the chapter explores the potential for making programming more accessible by improvements in teaching methods, based on what has been discussed previously. It concludes that this will never, on its own, be sufficient to overcome the barrier that non-programmers experience when faced with a programming task. What is required is a change in the tools available, which are discussed in the next chapter.

2.1. Introduction

As a start to this search for the most promising avenue for investigation, this chapter looks at the nature of programming expertise. The goal in doing this is two-fold. On the one hand it is important to establish clearly that programming does require considerable expertise and that it is unlikely that non-programmers will ever be able to acquire this without a large investment of time and effort. On the other hand, it is important to understand as much as is possible about what this expertise consists of. This will facilitate later discussions about tools that aim to reduce the need for expertise. The chapter starts, however, with a discussion of exactly what is meant by the term "programming".

2.2. The Nature of Programming

The approach taken in this section is to search for a definition of programming that meets the needs of this body of work. It is found that this must involve a statement of the boundaries of the concept, and these are described. In particular, the distinction is made between a definition that defines "programming", the process, and a definition that defines "program", the end-product of the process.
2.2.1. Attempts at Definition that Define "Programming"

Programming is an activity that is very hard to define. Most rigorous definitions either include activities that one might want to exclude or exclude activities one might want to include. This results in definitions either suggesting blurred boundaries or making arbitrary choices according the interests of whoever is making the definition. An example of this is the definition of programming that distinguishes it from "coding" (Ledgard and Marcotty, 81), so that programming becomes just one part of the process of generating a program. One can define it either in terms of the effects on the computer, the process or the final product - the program. In the literature, there is more of a tendency to define the programming process than the program (eg. Ledgard and Marcotty, 81, Barr and Feigenbaum, 82). The different ways in which this can be done will now be evaluated.

One way in which one might try to describe the programming process is in terms of what the programmer is doing to the computer. Unfortunately, using this approach, it proves very difficult to distinguish programming from using a word-processor or entering data into a data-base. This is because of the fuzziness of the boundary between data and program. Entering a command causes the computer to do something, but so does entering data. When using a word-processor, one can insert a piece of text either by typing it in from the keyboard or by using some sort of "paste" command that takes the text out of memory and enters it. Any distinction between what they cause it to do are likely to be domain, tool or task specific. Certain sorts of data might, according to the program involved, not only cause a change in the contents of memory but also result in the initiation of some other activity. This is because programs consist not only of statements that manipulate the contents of memory but also of statements that control the execution of the program itself. These latter can be predicated on the value of certain items in memory as in:

```
if size is greater than 10 then
    put "It is too big!"
end if
```

The extent to which this distinction is arbitrary is also demonstrated by programming languages such as LISP that are designed so that programs can, themselves, be treated like data by other programs. In reality the distinction between programming and entering data is one that resides within the mind of the user rather than within the computer. That is not to say that the distinction is meaningless. In the definition of
programs that follows, it is argued that a program is an entity that has certain properties in the mind of the user and this is used to define it.

An alternative to a computer-centred definition is a user-centred one. One might be able to argue that programming can be characterised by what the programmer is doing when programming. If one attempts to do this, one can be in danger of excluding some of the more novel approaches to program generation. Certainly, one must be very careful at what level one describes the activity. A number of years ago it would have meant generating punched cards but nowadays it can mean drawing pictures, moving "objects" around on a screen, filling in electronic forms, or entering numbers and equations into boxes. At some levels, the activity is similar and many of the basic steps required are seen in generating all these sorts of program, but at a level of the observable activity they are very different. In the definition that is suggested, certain aspects of the process of programming will be included but these are the more general ones.

2.2.2. Attempts at Definition that Define "Program"

If one cannot define programming in terms of the observable activity, then one must try to define the concept of "program". Once one has done this, one can define programming as any process that involves the generation of a program. The dangers here will include those described above of being too specific, as programs can take many different forms (as will be seen in Chapter 3). The suggestion that is made that a program is an entity with the characteristics listed in Figure 2.1.

- It is a piece of software intended to run on some computer. That is to say that it exists at some level as computer-executable instructions even if this is not its representation to the user.
- It has a name or identifier of some sort that can be used to initiate whatever activity may be associated with it when entered into the computer in some manner (i.e., start the process of communicating commands to the computer for translation).
- It is seen by the user as being an active entity with regard to the computer, controlling its functioning in a direct manner.
- Consideration will be limited to programs that are constructed by one single user rather than a team ("programming in the large", Soloway, 86a).

Figure 2.1. The characteristics that distinguish a program from other similar entities.

The rationale behind these points is discussed in the following paragraphs.

Programming is defined by Barr and Feigenbaum (82) as the process of specifying the method for doing something the computer can do in terms the computer can interpret.
For the purposes of this thesis, this definition is too broad. This point can be illustrated by distinguishing between programming and using a command-driven application such as the Qedx editor (a primitive word-processor). One could argue that each command made while using Qedx is a short program. For example, the command to substitute "water" for "wates" in the current line of text:

\[
\text{s/wates/water/}
\]

This is a valid suggestion and the length of the program certainly seems an arbitrary method of distinguishing things. If, however, one wanted to repeat the effect of the command, one must repeat the written command as before. I would argue that the command only becomes a program when it is given a name. In this way, creating macros and abbreviations is programming just as much as building novel applications or setting up number-crunching routines. This rules out simply re-configuring an application (such as changing the colours used or menu options available) unless one puts a name to the new configuration. Generally, however, one starts to program when one decides either that one is going to want to make use of a certain group of commands or configuration of objects more than once (hence the need to name it for future reference) or because the goal is complex and one wants to approach it in a more planned manner than just opportunistically by making a series of steps.

It is clearly not sufficient to stop here as data files have names. The difficulty of distinguishing between programming and entering data has already been mentioned. The answer must lie in the user's attitude towards the object named. As far as the user is concerned, a data file is an inert entity that is acted on by the computer in some way. When word-processing, one can be writing a letter or writing a program. As far as the word-processor is concerned, there is no difference. However, the user is aware of being engaged in programming because the goal is to create something that will perform a task controlling the activity of the computer (as against a letter that performs a task by controlling the activity of the client, Gas Board, etc.). Not only does this exclude data files but it also rules out creating novel fonts, screen designs or icons unless they are actually combined with novel functionality.

The fourth characteristic is to indicate the particular area of interest of this work, defining a sub-section of the activity that could be called "programming". Soloway (86a) distinguishes two sorts of programming research; that which looks at the work of individual programmers, often in laboratory conditions, and that focussed on the work of teams of programmers engaged in large projects. Focussing solely on programs
created by individuals is an admittedly arbitrary boundary determined partly for ease of presentation and partly because, at present at least, programming in the large is an activity engaged in by professional programmers. At the end of the thesis, the question of how many of the conclusions would be relevant to programs generated by teams will be considered. Rather, the focus of the body of the thesis will be on programs that are constructed to act as tools to perform tasks that will:

- be too complex or detailed to be tackled by a sequence of commands within an application (such as analysing a large data set),
- be repeated (such as data presentation tools for monthly stock reports) or
- continue in use over time (such as time management tools).

The intention is to make the conclusions of this thesis as non-domain-specific as possible.

Having clarified what is to be understood by the term "programming" it is possible to move on to consider what is known about the nature of programming expertise. This is done in the next section.

2.3. The Nature of Programming Expertise

Much of research into the psychology of programming is focussed on specific questions that relate to specific languages. In addition to providing answers to these specific questions, however, it is also possible, after examining programming practice using a variety of languages, to build a picture of programming as a general activity that has properties that will be evident whatever the language being used. A good understanding of programming must involve, therefore, a combination of:

- knowledge about the processes of programming in general, and
- knowledge about how the properties of languages interact with this.

To date, the understanding obtained falls far short of this goal. However, some understanding can be said to exist in both areas. In this chapter, the understanding about the first of these is explored, while the second is left to the next chapter.

The goal of this chapter, it must be remembered, is to answer the question of how easily programming expertise can be acquired. The analysis of current work on the psychology of programming offered is very much geared to this. Four main arguments
could be said to exist for the intrinsic difficulty of programming. The first relates to the diversity of tasks that are involved in programming. Each of these will require some measure of skill from the programmer. A description of the different tasks is given and the nature of the relevant expertise examined. The second relates to the nature of programming as a process of finding solutions to problems. The programmer must, therefore, have access to candidate solutions and be able to manipulate them in the necessary ways. The next section explores the nature of the knowledge that enables expert programmers to do this. While a number of models of programming knowledge exist, the concept of programming "plans" provides a useful tool for thinking about what form this knowledge must take and this concept is focussed on because of this. A good understanding of the machine being programmed will also greatly facilitate much of the work that must be done while a poor understanding can lead to many errors. Finally, the complexity of the programming process necessitates the programmer having to make numerous meta-decisions about what to do when, etc.. The problems of managing this process are discussed in the final section in the light of recent work suggesting that it is much less well structured than many earlier workers assumed.

2.3.1. Programming Expertise Includes the Ability to Perform Many Different Tasks

Many people have described the different tasks that make up the programming process. One of the most comprehensive of these has been presented by du Boulay (88) and this is used as a basis for the description given below. While these different processes are typically presented as stages, there is no suggestion that the order is rigid. Rather there is an iterative process of cycling between the tasks as the programmer moves from an initial understanding of the problem to attempting to code it, to debugging it, with the possibility that he/she will return to an earlier task at any stage. The tasks are listed in Figure 2.2.

- Establishment of the problem.
- Comprehension of the problem.
- Choice of language and machine.
- Specification of the method.
- Coding.
- Testing.
- Debugging.
- Documentation and maintenance.

*Figure 2.2. The sub-tasks of programming (from du Boulay, 88).*
Establishment of the problem. Here the programmer is actually making the decision to attempt to achieve a goal by the creation of a program. Included in this decision is the question of how much of the problem can reasonably be expected to be amenable to solution via a program. This process is not one that novice programmers experience during a typical programming course because the exercises given implicitly define the problem and it is assumed that it can all be dealt with through programming.

Comprehension of the problem. Before one can start to think about the nature of the program solution, one must be sure of understanding the problem. This means decomposing it into sub-problems and identifying the main objects or processes. It may be claimed that this and the earlier task are independent of the actual language or machine that is to be used. In reality it is likely that the aspects of the problem that are identified as being accessible to programming and the way the problem is decomposed owe a lot to one's knowledge of programming and the machine one has in mind to use (see eg. Gilmore and Green, 88).

Choice of language and machine. Expert programmers are often in the position of having access to a number of different programming languages and tools. Having established the general nature of the problem, it is possible to identify features that will make it suitable or unsuitable for any given language.

Specification of the method. Having decided on the language, etc., one can start to represent the problem in terms of what is known to be available. The programmer starts, therefore, to make a conscious effort to see the problem in terms of the language that one is intending to use. Sumiga, Siddiqi & Khazaei (88) suggest that one is looking for ways to break the problem down into units that one believes to be achievable within the given language. In a sense, one has established one's goal and is looking for methods to reach it. These methods will not, however, be specified in great detail and will not be represented as lines of code.

Coding. The next step must be, therefore, to turn the methods into code. This basically involves the generation of units of communication that the computer can run. Green, Bellamy and Parker (87) have suggested that this process involves processes of generating items of code, reading these, and then fitting further code around them.

Testing. At any stage during the process of writing code, one can attempt to run the program. This will involve the use of some imaginary data and enable the programmer to determine whether the program is behaving as he/she wants it to. Having run the
program, the programmer must determine whether it has done what was required. Problems in the code can result either in the computer not being able to run the program, in which case error messages are generated, or in a run that does not produce the desired output. For a novice, it may not be entirely clear what sort of output would have been correct. When the programmer reaches a stage where he/she believes that the program is basically working as is required, the testing can be taken further to include more difficult data. This is necessary if the programmer wants to ensure it will work in a variety of different circumstances. A simple example is that a program may be fine when running on a list of 5 items but when the real data with 300 items is used it could fail. It may also be desirable for the program to be able to cope with erroneous inputs. Trying out unusual inputs can determine this to some extent. In general, testing at this stage is not something that novices are aware of and they will often not attempt to do this.

Debugging. If the programmer detects a problem with the program, then this must be corrected. Error messages combined with examples of program behaviour are the clues to help with this and a number of different strategies can be used to locate such bugs. Once the programmer believes that the bug is found, it must also be corrected. This can involve further thought and the program will need to be tested after the correction is made to ensure that the bug is gone and that there are no side-effects.

Documentation and maintenance. Many programs are used over long periods of time with the possibility that they might need to be adapted or improved on. This is known as "maintenance". If the time between original coding is long or the person available to perform the alterations is someone other than the original programmer, then there will be a need for good documentation to explain why the program is the way it is. This can be in the form of an explanatory document that is kept with the program or of comments in the actual code. Clearly, the ability to interpret such documentation is another requirement of the programmer.

This list gives some measure of the complex variety of tasks facing the programmer. In many cases these must be performed by the programmer with very little support from the tool or environment being used. This results in there being a considerable onus on the programmer to have a great deal of skill at the tasks and this involves many different particular elements of expertise. As will be seen in Chapter 3, many tools have been developed to support the programmer in these various tasks and their success in this is assessed in that chapter. For the time being, however, this list is very indicative of the
amount of expertise that must currently be acquired to be able to program and, as such, suggests already that the solution may not lie just in improving on teaching practices.

While there are a great many different tasks to be performed for a program to be generated that will achieve the goals required, many of these tasks relate to the basic need to find a solution to the problem identified. The nature of the knowledge that enables this is now discussed insofar as it is understood.

2.3.2. Programming Expertise Includes Knowledge of Many Candidate Solutions

Over the last thirty years the understanding of the programming process has developed considerably. One of the first ideas that was expressed about programming knowledge derived from studies of expertise in other areas such as chess playing. It was thought that expert programmers could perceive and manipulate programs in terms of larger "chunks" of code than novices (Brooks, 77, Tracz, 81). This was seen to confer on the expert many benefits including the ability to handle much larger programs before being overcome by problems of complexity. This led to research that attempted to identify the nature of these chunks, again using techniques derived from other domains (e.g. McKeithen, Reitman, Rueter & Hirtle, 81, Ehrlich and Soloway, 87). It is still true to say that a large proportion of the psychology of programming literature is aimed at establishing the existence and identifying the nature of the these units of programming knowledge. This was combined with models of how such knowledge is used in the various programming tasks. The most influential approach to this problem was to suggest that programming knowledge consisted of a battery of programming "plans" combined with some stylistic rules (Soloway & Ehrlich, 84). These plans consisted of stereotypical pieces of code that were associated with specific programming goals. A typical example of this is the plan for "reading in a stream of integers, summing them, and stopping when a sentinel value is input" (Soloway, 86b), as shown in Figure 2.3.

```
initialise a running total
ask user for a value
if input is not the sentinel value
    then
        add new value into running total
        loop back to input
```

Figure 2.3. An example of a programming plan.
These plans are not language dependent. The programming process was seen to consist of breaking down and refining goals until each could be associated with a number of known plans. These plans are then converted to code and combined to create a program (see Figure 2.4). Thus a programmer's expertise is seen to consist of a battery of plans combined with rules for their usage and this is developed through experience with programs and identifying common algorithms for solving problems.

Figure 2.4. The concept of programming plans used in programming.

This concept stimulated a large body of work that explored the way plans were used in the different programming tasks (Spohrer & Soloway, 86, Soloway, Pinto, Letovsky, Littman & Lampert, 88, Wiedenbeck, 87, Soloway, 86b). Alongside this work, however, has been evidence that plans do not represent the entirety of an expert's knowledge (eg. Widowski and Eyferth, 86). Rist (86) attempted to deal with evidence of this sort by suggesting a variety of different types of plan that could support different aspects of the programming process. His model was based on verbal protocols and discussions with programmers which gives little direct evidence of how plans are used in programming. An approach that did explore how plan knowledge interacted with programming tasks was used by Gilmore and Green (88) who performed an experiment to see how plan knowledge was used when locating bugs in programs. The manipulations they employed made it possible to examine both the language dependence of programming plans and the extent to which they are the basic knowledge used in all programming tasks.
The programs used were written either in Pascal (the language most frequently used by Soloway and collaborators) or in BASIC. The programs varied in the extent to which they contained additional clues as to their structure. These clues were either about plans (colour highlighting) or about control (indentation). Programs were presented that had one or other of these, or neither, or both. The bugs varied in their being related to plans or control flow, or both, or just being surface bugs. It was found that, while clues about plan structure supported the identification of plan-related bugs in Pascal programs, they did not in BASIC programs. Also, they did not support the identification of control flow bugs in either language, while indentation did, in Pascal.

While it was not possible to identify whether the differences between Pascal and BASIC were due to teaching method or language features, some important points could be made. Firstly, plan information was helpful under some circumstances providing some more direct evidence of its value as a concept in understanding programming expertise. Secondly, knowledge of plans is clearly not the only representation of programming knowledge as it does not support working with control flow, and it may be language dependent. From this, Gilmore and Green conclude that the concept of "plan" is insufficient for an understanding of programming knowledge. They suggest a more general model with three elements. These are:

* Knowledge about how the problem can be broken down and converted into methods;
* Knowledge about how pieces of code generate computer behaviour;
* Knowledge about mappings between the methods and the achievable computer behaviour.

This is in contrast to the concept of a plan that assumes all of these elements are present in one unit. This model is illustrated in Figure 2.5, in contrast to Figure 2.4, shown earlier.

The analysis of Gilmore and Green can be seen as questioning the notion that the whole process of mapping between a goal and a programming method is contained within a single knowledge unit. They are trying to partition this knowledge down further into constituent parts, each of which may or may not be present for any given programmer. This is an important issue as it seems to add to the knowledge that the novice must acquire in order to become an expert. Their analysis could, however, be taken even further by speculating on what knowledge must be present implicitly for a programmer to go from a goal to a piece of code, however it may be represented. An attempt is made in Figure 2.6 to represent this. This is based on the elements of the programming
process, as described in section 2.3.1, where the programmer has to perform a series of transformations on the goal to arrive at the final code. For each of these transformations to occur, the programmer must have knowledge of:

- The element to be transformed (El),
- The element into which is to be transformed (E2), and
- The mapping (E1 - E2).

Initially, one goal is identified (for simplicity's sake, although in reality there may be more). This consists of a number of sub-goals, which may in turn consists of smaller sub-goals. At some stage, however, the transformation changes from decomposition to a change in representation. Thus, a goal is related to a method and a method to more specific activities of the computer. Finally, this is related to some code. Clearly, there can be decomposition at other stages than when in the goal representation.

![Diagram](image.png)

**Figure 2.5. Gilmore and Green's model of programming knowledge (based on Gilmore & Green, 88).**

This allows one to review what was meant by the concept of plans and the analysis of Gilmore and Green. Plans are seen to be mappings that go from sub-goals to methods, and these map directly onto code units. The emphasis here is on the single unit that includes both elements and the mapping, as one unit of knowledge. The development of the plan concept extended this idea to plans at many levels of detail but with the same
basic assumption. The analysis of Gilmore and Green challenges this view by suggesting that knowledge may be less integrated. They suggest that programmers have knowledge of conversions between goals and methods, knowledge about relations between code and computer behaviour, and knowledge about mappings between these. This representation clarifies the uncertainty that exists in the data and opens up new possibilities that have not previously been considered.

![Diagram showing the relationship between goal, subgoal, method, computer behaviour, and code chunk]

*Figure 2.6. The transformations that must occur between identifying the goal and generating the code. For each transformation, the programmer must have knowledge of both elements and the mapping between them.*

While each of the different elements of knowledge described above could exist separately, it seems highly likely that, on many occasions, they will be grouped into larger units that enable much more rapid progression between goals and code. Thus, experienced programmers may well acquire "plans" in the sense used earlier if they notice the association between common problem types and corresponding code templates. Indeed, if one includes external memory concepts, they may keep code that they have written and employ it directly when faced with a given problem. It seems probable, in fact, that, if a chunk of code or method has been identified as a single entity, then it will probably be by its being associated with a common problem type. It may be that such chunks will not, therefore, exist in isolation from a mapping to a method, problem or sub-goal, as Gilmore and Green suggest they could.
It must also be true that such chunks do not represent the only part of what experienced programmers know. In novel situations, they will have to have the ability to perform the transformations in a step by step manner, using knowledge about particular methods that can be applied and adapted. It is this aspect of programming that requires considerable further investigation and is the repository of much of what constitutes real programming skill.

It would appear, then, that expert programmers have a considerable repertoire of programming information units, many of which will consist of ways of transforming a problem or goal into code directly, while others involve analysis and adaptation skills. This must represent a daunting challenge to non-programmers who must acquire all this if they are to turn their own problems into programs that solve them. Not only, therefore, does programming involve a considerable number of procedural skills, but it also requires a battery of more declarative knowledge. In addition to this, programmers have to have a good understanding of the machine that they are trying to control. This is discussed in the next section.

2.3.3. Programming Expertise Includes the Ability to Understand the Machine Being Programmed

Implicit in the discussion in the previous section is the assumption that, not only must programmers have all these knowledge units but that they must be able to manipulate them, identify when each is appropriate and adapt them in sensible ways. This requires that they have a good overall understanding of the way the different units of code interact and affect the resulting program. In addition to supporting these processes, a good knowledge of the machine being programmed is vital if program behaviour is to be interpreted for debugging purposes.

Presenting concrete descriptions of the machine being programmed to novices learning BASIC has been found to improve performance (Mayer 87). Mayer developed his model of the machine by use of a technique he calls "Transactional Analysis" and claims that this has close associations with the programmer's conceptual understanding of the machine. The analysis consists of establishing, for each statement type, the specific steps that it evokes in the computer. In practice there would seem to be a number of ways of describing the BASIC machine and that these should be chosen and explained in such a way as to maximise the novice's ability to transfer understanding from areas of existing skill. However, there is likely to be, for non-programmers, a considerable degree of novelty to any model presented as it consists of a very abstract way of dealing with information. Expert programmers have to have become able to
work with such abstractions as if they were concrete and tangible and this represents another considerable area where the non-programmer will have difficulty.

The last part of this exploration of programming expertise identifies the need for management skills in addition to these.

2.3.4. Programming Expertise Includes Being Able to Manage the Programming Task

Having suggested a simple model of programming as a series of transformations from goal to code units, it should be pointed out that this process is not as simple as it might appear, and that, in reality, this is not a true picture of programmer behaviour.

The first thing to notice is that, for a complex program, the transformation tree shown in Figure 2.6 would be very complex indeed. There would be many sub-goals and these would have to be further subdivided. For each sub-goal, there would have to be a method and the methods might suggest further breakdowns into sub-elements before being related to computer behaviour or code. Once down at the level of coding, one cannot just convert each item into text and enter it in. The different elements must be integrated into a whole which will involve merging code chunks. The process is, therefore, immensely complex. The second point that arises from this is that the process described is only one way of arriving at a solution. There are a number of different strategies possible. One could follow one sub-goal down to its method, to its code before dealing with another (depth-first approach), or one could break them all down before finding methods, etc. (breadth-first approach). These are represented in Figures 2.7 and 2.8. These two strategies are extremes and the programmer can select from any number of alternatives between these. Not only is programming complex, therefore, but it involves a number of decisions about how it is to be managed.

Figure 2.7. The breadth-first approach to programming.
One of the current trends in the understanding of programming is related to the realisation that structured methods and step-wise refinement, as described above, do not represent a good model of what actually goes on when programming. Sumiga (personal communication) points out that the process is much more opportunistic with the programmer looking for a solution to the overall problem and then, when one is found, trying to work all the other details around this. He also suggests that the process is one of looking for achievable goals and working around these. Certainly, there is a need both to understand the more complex processes of how the achievement of the different sub-goals is managed and of how the transformations from sub-goal to code are arrived at.

The processes of managing the different sub-goals and interleaving code units must represent still additional skills and draw on other knowledge. Programs requiring such activities are often where errors will occur (Spohrer & Soloway, 86) and these are used as further evidence for the validity of the plan concept. Also, the use of techniques such as running the program to observe its behaviour will require some skill.

What is clear from this is that programming knowledge must consist of a great variety of information units that are represented in different ways and at different levels of detail. The classification by Rist (86) represents a first stab at the articulation of these
but leaves a lot unsaid. In addition to the individual units, if they exist in this way, there must also be frameworks that enable the units to be manipulated and associated as well as just understood. In particular this must include a model of how the code interacts with the computer. Finally, experienced programmers must have many strategies and skills that enable them to manage the complex problem solving activities required and not become over-burdened by it all. While these strategies may seem to be counter-productive in some sense, because they result in the use of sub-optimum solutions (Sumiga, personal communication), they may still represent the best use of the cognitive resources available.

2.3.5. Summary

To summarise, therefore, the expert programmer's knowledge consists of:

- Skills in the many different tasks required for programming, including more peripheral aspects such as testing, debugging and documentation. While these will draw on the general knowledge of programs and the machine, they will also involve specific skills of their own.
- A large body of knowledge units referring to algorithms, goal-sub-goal breakdowns, and chunks of code templates, with information about how each can be associated with others at the same and different levels of abstraction.
- An understanding of the machine model and the domain of the task.
- An ability to manage the complexity of the task including techniques and strategies of many types.

This points clearly to the mammoth task that must be undertaken if non-programmers are to attempt to generate their own programs and indicates some of the specific areas that would need to be addressed by any approach to reducing this task. In the next section, the possibility of overcoming it by improvements in the teaching of programming are considered.

2.4. The Possibilities for Improving the Acquisition of Programming Expertise

From the discussion of the nature of programming expertise in section 2.3, it is clear that non-programmers have a lot to learn before they can be able to generate their own programs. They will have to learn a variety of skills combined with a battery of items of information, and to use these it would be necessary for them to achieve a good understanding of the abstract workings of the machine to be programmed. Given this, it is unlikely that improvements in teaching practices will ever be able to alter the time
and effort that must be involved in learning to program, given current technology. Nobody working in the area of teaching programming would disagree with this. At most they are using basic teaching concepts such as those that result in deeper processing of material (Mayer, 81), aiming to find ways of making the material more accessible, such as through the use of machine models (du Boulay, O'Shea & Monk, 81), or of teaching the problem solving skills required (Soloway, 86b).

There is, however, an additional reason why it is unlikely that the teaching of programming knowledge and expertise will ever be easy or rapid. It may well be that, ultimately, a lot of the work of acquiring it must come down to practice over time and that teaching programs can only provide an introduction that helps the novice acquire a basic grasp of the elements involved. There are two basic points to be made here. The first relates to the observation that it is not generally possible for novices to be taught to solve problems in the ways that experts do. In particular, providing novices with information that experts have will not help them because they will not have the mental structures and representations that would enable them to manipulate such knowledge. Thus, there are two reasons why novices do not have the plans of experts, that enable them to chunk program problems and cope with complexity. One is because they have not had enough experience to notice the existence of the algorithm involved, and the other is because they would not be able to deal with them as single units. This suggests that programming expertise can only be acquired through the gradual development of mental structures for handling the information required.

The second question is about the extent to which becoming an expert programmer is a matter of learning knowledge from textbooks as against being a matter of acquiring a skill, that involves practice at the various tasks (eg McKendree and Anderson, 87). In particular, it seems increasingly that the tasks that must be performed to acquire programming skill are programming tasks (Thompson & Ormerod, in press) so that many of the introductions to programming that use highly artificial tasks may not be providing the novice with the training required. What seems inevitable, therefore, is that the novice must spend time and effort in struggling with the particular tasks of programming on authentic programming problems. If this is the case, once again, it would suggest important limits on the potential for programming expertise to be imparted directly from an expert or tutor to a novice.

2.5. Conclusion

A programming language consists of a vast number of rules and methods. For programmers to be able to make use of these, they must have access to them and
knowledge about what they achieve and the way they can be used. While reference manuals exist to support this, there is no substitute for programmers acquiring an internal knowledge base of these items. It is only then that they can conceive of problems in ways that will suggest how they might be solved and make the transitory steps between articulating the problem and generating a coded solution.

Learning to program, much like learning any expert behaviour requires more than just learning a set of procedures, however. There must be a process of developing a mental framework that enables the procedures to be manipulated in an efficient and strategic manner. Even if it were possible to transfer an expert's knowledge into the mind of novices, it would be of no use to them because they would not understand it. There is a need to build up associations and structures that make the generation of complex programs possible, and this must inevitably take a considerable time and require much practice.
CHAPTER 3

WHAT CAN BE LEARNT FROM EXISTING PROGRAMMING TOOLS?

Abstract to Chapter 3

This chapter looks at currently existing and prototypical programming tools that aim to reduce the work required to program. In each case it is found that, while progress in this direction may be achieved, there will always be a resulting negative side-effect that either reduces the power or adds to the complexity involved. It is argued, therefore, that a more radical approach is required if programming power is to be made available to non-programmers. This involves users in specifying their requirements in an informal manner and requires of the computer that it be able to generate programs that meet these requirements. The need for this approach to be given a psychological investigation is established, to complement currently existing technical developments in this direction.

3.1. Introduction

The previous chapter has argued that programming is a complex skill with many facets and that it is extremely unlikely that the non-programmer will be able to acquire it in a short space of time. One approach to the problem is to look at existing programming tools and to ask whether there is anything there that implies useful design developments? In this chapter the potential of currently existing tools is examined with the intention of establishing how possible it might be that any one of them could lead ultimately to this goal. It starts with a description and justification of the approach taken in the evaluation of existing programming tools.

3.2. The Approach Taken

Programming languages are remarkably diverse in form and evaluations can be performed at many different levels and using different approaches. The approach taken in this chapter involves the identification of a number of key concepts in programming language design that have particular relevance to non-programmer-oriented languages. For each such concept a number of exemplars are described. The evaluation will consist of a discussion of how successful the concept has been in the context of the existing exemplars. From this its potential can be speculated upon.

This approach must necessarily involve a certain amount of informal analysis, often in the absence of good empirical data to support it. This must be, to some extent, an
inevitable consequence of the state of research in this domain. It is also, however, an approach that is being advocated by some as a valid way to achieve progress. Thus, while Green (in press) performs a similar analysis and uses this as an argument for the need for more research, Carroll & Kellogg (89) actually argue that the evaluation of existing technologies represents an important method of identifying "theories". This "Claims Analysis" of Carroll's is discussed in this section. Some weaknesses of his approach are identified resulting in a diminishing of the claims to objectivity that Carroll makes. Despite this, it is argued that this sort of approach is necessary in circumstances such as those prevailing in this area.

Carroll and collaborators (eg. Carroll & Kellogg, 89) have developed a technique that they call "Claims Analysis" for the evaluation of software tools (although their arguments for this approach suggest that this occurs informally in many disciplines, Carroll, 88). The origins of this lie in the difficulty in developing theories within HCI by means of standard empirical research, as described in Chapter 1. This difficulty, coupled by observations of how the design actually works suggests the concept of the design artifact as a source of psychological theories, or as a theory itself. The argument basically is that, given a design that is found to work well, it should be possible to suggest why it works well in a way that will enable this knowledge to be generalised to other design problems. The generalisation is achieved by articulating the reasons for a design's success as psychological statements about the tendencies of users within human-computer interactions. Carroll argues that, in doing this, one is extracting the implicit claims that exist within such designs. An example of this given by Carroll & Kellogg (89) is that of the "Training Wheel" interface which they analyse in this way. This is a technique for introducing novices to applications by making only the more basic functions available while the advanced functions can be seen in the menus but are not activated. An example of claims made by the training wheel interface is that exposure to full menus supports incidental learning, while reducing the number of system states makes it easier for the novice to make sense of what happens.

What Carroll et al. are doing is extracting hypotheses about user psychology from the observation that an artifact is successful. This must, however, represent an extremely subjective approach. It is subjective in that there seems to be little method in the observation of the way the artifact is used. All that is required is that one be able to distinguish between a successful and an unsuccessful artifact. This is far from the detailed recording of times and error rates that is usually associated with more formal methods. Having decided that a design is successful, the data associated with this conclusion do not seem to be used further. The researcher is left to examine the artifact
itself to determine why it is successful. This involves identifying "theories" that the researcher claims to be intrinsic to the design and stating that these are the aspects that led to the success of the artifact. At each of these stages, there seems to be considerable scope for bias of interpretation and judgement based more on the researcher's own perspective and cognitive set than on reality.

One of Carroll & Kellogg's arguments for using this approach is that it is less subjective than hermeneutics. This latter involves studying each user-task-tool situation subjectively and individually and developing an understanding from that. There is a danger, however, that Claims Analysis is presented as if it were, itself, free from subjective bias. It would appear not to be. This may not mean that it should be thrown out as a method, however. Rather, it would seem important to realise that in many situations subjective approaches are the only approaches for achieving progress. This is particularly true in design sciences such as HCI where the value of a theory or model is determined, not so much by its being supported by incontrovertible evidence, but by its being useful in generating good designs. An approach such as Claims Analysis opens up this idea for debate and allows one to identify what is necessary for one to claim a theory to be valid.

It would appear that Claims Analysis would be greatly strengthened if it could rely rather more on empirical evidence. Thus, one needs to find ways of turning observations of users into evidence that a given design decision contributes to the success of a tool. Also, it may well be that a lot can be learnt from a good design element that exists within a poor overall design and vice versa, rather than looking and assuming that all elements of a successful design are contributing to its success. There is a need, generally, for a more creative and reasoned approach to the identification of good design ideas that would probably involve the collection of a variety of types of data about artifacts and their usage, including what might, in other circumstances, be dismissed as subjective opinion.

It seems likely, therefore, that existing artifacts and knowledge about how they perform in supporting real tasks can represent, if not rigorous proof of a feature's value and of a theory's validity, at least clues about this. While it can be argued that the speed of technological change means that evolutionary processes cannot be relied upon to weed out bad ideas (Carroll, 88), it may be possible for designers and researchers to accumulate exemplars of approaches that can suggest whether an approach is likely to be worthy of development or should be thrown out. Once again, this will be subjective to some extent but there will be a forum for discussing such matters and the science
may develop along informal grounds in this way. Making use of empirical data and sound psychological insights as much as possible should add strength to the arguments put forward but it may never be possible to make categorical statements of fact about HCI owing to the unpredictable nature of the technological progress in this area.

This approach is now adopted for the study of programming language design. The bulk of this chapter will consist of the identification of a number of core approaches to programming language design and the evaluation of these in relation to their potential to support non-programmers. To some extent this will be based on an admittedly informal Claims Analysis of existing languages. However, this should not detract from its value as a contribution to the field given that it will represent an initial articulation of such ideas and be based on, at least, informal experiences of the languages in action.

The problem in many cases is that the software exists that demonstrates the approaches described but no data really exists about how well they work or how easy they are to use. Thus, a Claims Analysis is premature as one cannot even assume that the features embedded in the design are resulting in positive effects. Under these circumstances, one is left either to try the product out oneself or to speculate about how well it would work. Owing to the vast number of such products, it is not possible to perform even informal studies of how they work and many manufacturers are unwilling to provide data from their own tests. Given this, one must accept that some of the statements made are little more than provocations for debate and certainly not expert opinion.

The next sections discuss the various tools and approaches that currently exist. The first looks at those that could be said to support the general design process, the second those related to writing code, the third those that are focussed on getting feedback about what has been written and the last discusses some more general approaches that appear to disguise the programming process in various ways.

3.3. Aids to Program Design

Very little has been achieved in supporting the phases where the programmer must turn an, often vague, problem description into a set of known methods (Sumiga, personal communication). For the non-programmer this represents a major difficulty in the absence of a good knowledge of what is available and an understanding of how the programmable machine achieves goals. The most directly relevant tools are CASE (Computer Aided Software Engineering) tools that provide computerised facilities for working with existing diagramatic design techniques. Also discussed in this section are attempts to improve the machine model used. The need for programmers to acquire an understanding of this was discussed in Chapter 2 as being applicable to all stages of
programming. However, their main goal could be seen to be to help the programmer see how problems can be turned into methods and program behaviour.

3.3.1. CASE Tools

A number of diagramatic notations exist in the domain of systems analysis for representing the design of information systems, particularly computerised ones. These help structure the system analyst's thinking and make it easier to locate missing information or aspects of the specification that are inadequate. Data Flow Diagrams are an example of such notations (see Figure 3.1). CASE tools provide support for the design process by offering a number of such diagramatic notations for the programmer to use (Jones, 88). In addition to providing facilities for constructing the diagrams, the tools will often request information from the analyst about the different entities represented on them. Clearly these features force the analyst to define many aspects of the system explicitly which enables clarity of thought and removes much of the burden of structuring the whole system. This latter is achieved by means of a data dictionary where all the entities defined are stored, so that they can be drawn upon in other diagrams resulting in consistency and efficiency.

There are considerable weaknesses with these tools as far as non-programmers are concerned, however. Firstly, they must be able to understand and use the notations provided effectively. This is not a simple matter and can be a problem for those not trained in their use (Vitalari, 84). Ultimately, however, CASE tools might be developed that would be intended for those not trained in systems analysis with notations that might be easier to use. Secondly, they must be able to make the steps from their problem description to solutions expressed in terms of the diagrams. CASE tools do not take you from one representation to another. They merely provide tools that might help one to represent and manipulate (or sketch out) a solution. Often, however, they are only able to support the already well-structured aspects of the design task and do not even help the more intuitive behaviour that appears to go on in systems analysis. Finally, they often do not result directly in code but only in a useful representation of a program that must then be coded. Certainly they will support this process by offering a structured representation of the requirements but many of the problems of generating code will remain to be solved.

It is conceivable that future CASE tools could turn formally generated program diagrams into code. These would not necessarily represent a qualitative advance as the programmer will just need to learn to represent the solution in terms of these rather than in terms of standard notations. They would, therefore, result in the provision of a new
notation (see discussion below about this area) rather than a new approach to programming.

Figure 3.1. An example of a Data Flow Diagram (NCC 89).

The next section looks at the potential for improving the machine model with which the programmer must work. It may be that by substantial changes to this many of the non-programmer's problems could be considerably reduced.

3.3.2. Improved Machine Model

As was discussed in the last chapter, one of the problems when in the design phase of programming is the need to understand the machine that one is controlling through the
program. The machines that have been used in the past have taken little account of the needs of the user and been more a result of the demands of the domain or the technology (Wegner, 76). Even when they were intended to be easy, the measure of ease of use was the extent to which the language designer could understand them which is a poor measure of how graspable they are for non-programmers. It would seem important, therefore, to examine the potential for facilitating the programming process through the provision of more user-oriented machine models.

A number of approaches that have been taken to achieve this are discussed. These are:

- Simplifying the data structures,
- Trying to avoid the need to specify the sequence of actions when programming,
- Trying to make the fundamental unit of the program closer to concrete notions such as objects.

3.3.2.1. Simplifying the Data Structures

Programming essentially involves creating abstractions from some domain of reality where qualities of entities in a domain are represented by entities that can be manipulated by the program. These are the data of programs. Thus, programs can perform statistical analyses of the results of experiments or tests and provide information about these tests that can be applied to the questions originally asked.

Figure 3.20. The data structures used in BASIC. While many versions of BASIC now allow naming of variables and more advanced structures, the original idea still holds that the variables are independent addresses that can hold just one item of information. They are therefore very simple.
Because of this, the data structures used and the ways in which they can be manipulated represent the basic workings of most programming languages. When one talks about providing models of the machine being controlled, one is often referring to ways of explaining how the data is being manipulated. It is important, therefore, to attempt to generate data structures that will be easily worked with.

When considering how to represent the data structures, one should aim to make the representation such that it can be thought about by the programmer with minimal effort. This often means the use of metaphors that provide a sense of concreteness to what is going on (Tanimoto & Glinert, 89). In addition to this, one would want the structures to be sufficiently simple for the programmer to grasp their working. Unfortunately, this results in a paradox. Simplicity is achieved by reducing the different features and types of data structure. BASIC provides a good example of a language with relatively simple data structures (Figure 3.2a). Simple models can be provided to represent such structures that can have the appearance of being quite concrete. This means, however, that representing any problem in terms of these features involves considerable abstraction from the problem. To provide data structures that could handle such data better, while retaining some degree of flexibility, as COBOL does, results in their being very complex (Figure 3.2b). Naturally, one could provide data structures that were simple and concrete but they would have to be very domain specific, resulting in limiting the domain of the language or vastly increasing the number of possible structures available.

<table>
<thead>
<tr>
<th>Customer</th>
<th>Identity</th>
<th>Location</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Christian name</td>
<td>Address line 1</td>
</tr>
<tr>
<td></td>
<td>Surname</td>
<td>Address line 2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Address line 3</td>
</tr>
</tbody>
</table>

*Figure 3.2b. COBOL allows the programmer to create much more complex data structures. Thus, data items can be part of records that include a number of related items of information as with, for example, a customer record (from Maynard, 72).*
3.3.2.2. Alternatives to Programming as Generating a Sequence of Instructions

PROLOG (PROgramming in LOGic) represents the best known example of a new approach to programming that appeared in 1972. The approach is more radical than previous innovations as it represents a deeper rethinking of the programming process. Previous languages were "procedural" or "imperative", PROLOG is "declarative" and supports what is known as "logic programming". The meaning of this is discussed below. As well as its value to the builders of large systems, PROLOG programming was claimed to open up a way forward for novice programmers because in logic programming one is not actually specifying how one's problem is to be solved by the computer (Taylor & du Boulay, 87). Instead, one merely has to be able to articulate the relationships, etc., that make up the problem and let the method for solving it be left to the machine. It is now accepted in some quarters that logic programming is easier for non-programmers and it is used in other tools aimed at this group (eg. Spenke and Beilken, 89). This view of logic programming as providing a radically improved means of programming for non-programmers is questionable, however, as will be made clear below.

The Knowledge Base:

Elizabeth mother-of Charles
Elizabeth is-female
Charles is-male
x parent-of y if x father-of y
x parent-of y if x mother-of y
Philip father-of Charles
Elizabeth wife-of Philip
Elizabeth mother-of Anne

Interrogation of Knowledge Base:

is (Elizabeth mother-of Anne)
Yes

which (x:x father-of Charles)
Philip

is (Philip father-of Anne)
No answer

Figure 3.3. An example of a PROLOG knowledge base. Family relationships are often used when teaching PROLOG but many other domains of knowledge can also be used. In the interrogation, the system's responses are in bold type.
Programming in PROLOG consists, therefore, of setting up a knowledge base of facts about the situation one is interested in. Having done that, one runs it by entering instructions that interrogate the knowledge base (Figure 3.3). The description of the situation that is entered into the knowledge base must be in terms of very simple facts, relations and rules. These are logical expressions that enable the computer to extract information implied by the facts, upon interrogation. Complex facts or relations must be generated by combining simple ones. PROLOG requires, therefore, that any problem situation be expressed by the programmer in formal logic.

It has been suggested that logic programming is a psychologically valid form of programming (Kowalski 82) but, as Clocksin (87) points out, people do not actually think logically. One cannot, therefore, assume that it will be easy for non-programmers to convert their understanding of a system into a set of logical rules and facts. Even if it were possible for such users to express the facts and relations they could identify as important in this way, it might still not be adequate for creating a correct PROLOG program. This is because it is not enough to be able to articulate all the relevant rules, but one must be able to express them so that they combine to form a coherent body that will work together. This is not something that is a generally available skill and must, therefore, be learnt.

When comparing logic programming with more traditional procedural languages, it is not clear that a procedural approach is not the better for many situations. While certain aspects of a problem might well be most naturally expressed in terms of definitions of relationships, there are also likely to be procedural elements and having to convert these into rules would represent an addition to the labour of programming rather than a reduction of it. Indeed, this question of procedural or declarative approaches to programming is not clear-cut even within existing logic programming languages. Current thinking about how to teach PROLOG often involves the use of procedural models of its execution (eg. Brayshaw, 89). This is ironical given that it was originally intended to avoid the need for such an understanding. Also, when debugging a PROLOG program, one will tend to resort to the use of a tracer (such as the Transparent PROLOG Machine, see Eisenstadt and Brayshaw, 87, and Brayshaw, 89). This is a tool that provides some sort of representation (usually at great length) of what the computer did in answering the interrogation given. It is essentially, a procedural description. The need for this suggests that even experts do need to have a procedural understanding of this "non-procedural" language.
The basis for the logic programming approach is that the logic of a problem (the facts and rules that make up the problem domain) is separated from the control (how the facts and rules are used in the identification of a solution). An ideal logic programming language would present the user with a model of the machine that avoids the need to understand the control (Kowalski, 79). So long as the programmer can express the problem as facts and rules, then the computer would solve the problem. In reality, PROLOG and other logic programming languages fail in this regard and the distinction between the logic and the control is not complete. This means that the order of execution of a program is often important and affects the outcome (Davis, 85). In some respects this can be seen as an advantage and Kowalski (85) points out that rules can be deliberately expressed so as to affect the way the knowledge base is interrogated.

For novice PROLOG programmers, however, this means that they must understand how the computer actually processes an interrogation. Whether a perfect implementation is a possibility or not (Clocksin (87) suggests not), there still seem to be arguments for the user being given a model of how the machine works with the knowledge base. Once again, it would appear that PROLOG may not reduce the work of the non-programmer at all, whether the Logic Programming approach is more natural or not. A lot of work has been done on the nature of novices' errors when programming with Prolog (e.g. Taylor, 88, Fung, 87). The errors made seem to depend to a great extent on how it is taught - either procedurally or declaratively - but both result in problems. A declarative approach results in students creating programs where their errors suggest that the statements are being used like English statements as if communicating with a human. A procedural approach will result in errors due to misconceptions about how the programs are evaluated.

In conclusion, therefore, it would appear that it is not sufficient just to hide the workings of the machine from the user if the resulting interaction model is flawed. Also, one must be careful in the choice of model presented that it really is more "natural" and not based on a naive view of how people think.

3.3.2.3 Object-Based Approaches

One alternative way of representing the machine being programmed is that of object-based programming. This has been extensively developed in a number of forms and the understanding of "object" differs in each case. One specific subset of this approach is that of "Object-Oriented Programming Systems" (OOPS's) which defines objects in a very specific way, which is described first in this section. Other languages have been developed that make use of the concept of object but in slightly
different ways to OOPS's. Examples of these are LabVIEW* and HyperCard which are mentioned briefly at the end of this section.

The term "object-oriented programming" refers to languages based around a machine model where the program consists of a set of "objects" that interact by sending and responding to messages. It has been said that the concept of an "object" as used in OOPS's makes them more tangible and easier to understand (Kaehler and Patterson, 86, Thomas, 89). Also, the use of inheritance of characteristics is claimed to capture the natural mechanisms of organisation of concepts in human cognition (Wegner, 89). While such languages and tools have had a great deal of success among professional programmers and application developers in particular (Cox & Hunt, 86, Tesler, 86), it is not yet clear that they represent such a major advantage to non-programmers. Superficial descriptions and well chosen examples suggest that there is indeed a lot of scope for transfer from the real world to the OOP concept, however, when one probes more deeply, they turn out to be rather different to one's expectations.

Objects and messages occur at all levels even down to integers being objects receiving messages of "+". Thus when one writes:

\[ 3 + 4 \]

The "3" and the "4" are not the same type of entity; one is an object containing a variable and the other is an argument. This seems to add unhelpful complexity and is certainly not close to our intuitive understanding of arithmetic. It may be that this is the confusion that was noted by Lalonde, Pugh, and Thomas (85) who report novice users finding the distinction between messages with and without arguments confusing. They also report that more abstract classes are harder to understand. Thus, while certain concrete objects may well be well represented, many other types and domains of OOP are not.

As with PROLOG, programming in an OOPS involves representing the problem domain and then running it. In this case, the problem domain is represented by objects and the messages they can pass to one another (see Figure 3.4). It has been found that novices to OOPS's have difficulty phrasing problems in terms of objects (Kaehler and

* A list of known trademarks is included in the acknowledgements at the front of the thesis.
Patterson, 86). This may well be because the objects are not defined by concrete characteristics which would be familiar but in terms of responses to messages, etc.

This can cause problems as objects used to represent quite different physical systems might actually be quite similar because they respond similarly to certain message types (Lalonde et al.). Finally, Lalonde et al. found that ideas of message passing and inheritance were unfamiliar to programmers and took a while to get used to.

| Type of object, and description of hierarchy of objects (eg. object is a member of class "Dog", itself a member of class "Animal"). |
| Name of this instance of this type of object (eg. object is "Rover"). |
| State(s) (eg. object has a state "barksAloot" which can be true or false) |
| Messages to which it responds (eg. object responds to "talk", "bark", "beNoisy", "beQuiet"). |
| Code defining how it responds to messages (eg. when receive message "bark", run code that results on long "barking" if barksAloot is true, or short "barking" if barksAloot is false. Also, "beNoisy" and "beQuiet" change the value of barksAloot.) |

Figure 3.4. The defining attributes of an object in an OOPS. In addition to those that are specified for the particular object type, any others that exist for the super-class also apply.

Other work looking at how easy users found it to acquire OOP concepts has tended to focus on expert programmers, but even they, while finding the basic concepts easy to grasp have difficulty with some of the more subtle notions (O'Shea, Beck, Halbert & Schmucker, 89), or need to read lengthy descriptions of the concepts, and even when they can grasp the concepts can have trouble turning their understanding into programming practice. Basically, the objects of OOPS are not really like objects of concrete experience and some learning is required to discover in what ways they are similar and in what ways they differ. Also, it would appear that the strategies used to work with these objects and turn them into programs cannot just be transferred from common expertise but must be acquired through experience.

Two examples of possibly more successful uses of the concept of objects in programming are HyperCard and LabVIEW. In both of these systems, the concrete nature of objects is much more in evidence and the objects can be seen to relate to the user's experience of objects in a more direct and intuitive manner. In both of these cases, however, this has been achieved by means of the languages being built around a specific domain that lent itself to this approach. Because of this, they are discussed in
greater length later as their success cannot really be attributed solely to their use of object-like concepts.

Closely related to the use of objects in programming language design is a concept that HyperCard and LabVIEW share that also seems very useful. This was first pointed out by Green (in press) when he mentioned that in HyperCard, the programming process is the reverse of normal programming where code is written to generate entities and screen images. In HyperCard, the programmer starts by generating the screen entities using direct manipulation techniques and can go a long way in the program design before doing any coding. Once the screen entities are in place, programs are then written that are attached to them. This results in items such as "buttons" that can be "activated" by the user, by which is meant, their programs are initiated. This approach is also used in LabVIEW where the programmer starts by generating a control panel with all the desired knobs and switches for controlling the process to be modelled and all the data output devices for extracting information about it. Having done this, the program is written by creating the various links between these objects on a "diagram" that is imagined to lie behind the control panel and to determine its behaviour.

This approach allows the programmer to start with the concrete and familiar and to develop a design to a considerable degree until it is well established before moving to the actual development of algorithms and code. It would be interesting to obtain data about whether programmers in these languages take advantage of it. It may also be possible to take this idea further and to consider how environments could be designed that support it to an even greater extent. Thus, the programmer would be working in a direction starting with establishing concrete elements such as the interface, then slightly more abstract ones and so on until the code is added at the end.

3.3 3.3. Summary

The design stage of programming seems to be one where direct support is very hard to provide. Both CASE tools and the machine models presented offer support but only in a very indirect way. CASE tools provide the programmer with a set of structures and notations that could facilitate design but only after the overall goal has been broken down and converted into the rough methods that are to be used. Once the programmer has generated a sketch of the solution, they will support the more routine aspects of the clarification and tidying up that must go on. They may also support evaluating the solution and any subsequent alterations that must be made by making the effects of the solution clearer.
More natural machine models support the whole of the design and coding process by reducing the cognitive overheads and enabling transfer of existing expertise, but only to a limited extent. It would appear that they will be truly successful insofar as they present the programmer with constructs that map well onto concrete experience. And yet these cannot actually be too close to existing constructs or they will not offer anything new. If the programming language is to offer new solutions to the problems faced by the programmer, then developing these solutions must involve more than existing problem solving skills. The programmer must have to identify the new approaches to tackling the problem and this must involve the development of experience with similar solutions and other problem solving tasks using the solution elements provided.

In conclusion, it would appear that considerable improvements over conventional languages may well be possible by the provision of tools like CASE tools to aid the design process and the use of domain-appropriate machine models. However, the task of mapping between the goal and solution methods is likely, if one accepts the current approach to programming, to rely very much on programmer expertise that can only be developed through experience with the language in question.

3.4. Support for Writing Code

Having done some work in the general design of the program, the programmer must start to turn this into code. This is another complex task but one that is much more easily studied and, therefore, many more ideas have been developed for supporting it. The ideas can be divided into:

* those that reduce the programmer's need to plan,
* those supporting the programmers' identifying chunks of code to use,
* those that enable the programming task to be broken down into more manageable units,
* those relating to how the programmer can be protected from generating poor code.

The focus here is on the various tasks of constructing programs and the problems that they entail. This is distinguished from the next section where the focus is on extracting information about the program from the code that has been written to date. These processes are likely, however, to occur in parallel.
3.4.1. Reducing the Need for Planning

One approach to making the generation of code easier for non-programmers is that of incremental programming. The idea here is that programs are developed gradually with bits being added as the programmer thinks of them. The goal in this approach is to reduce the need for programmers to plan their programs. This planning stage requires a lot of skill and discipline and avoiding it would make programming a lot easier for non-programmers.

To date, little success has really been had in developing systems that support this approach. Prolog seemed as if it might but does not as the earlier discussion demonstrated. It is often claimed that with Fourth Generation Languages (e.g., Bouchier et al., Skrinde, 87) the programmer can use the first attempt at a program as a prototype so that it can be developed in situ. World-Builder and other systems where there is a program already working that the programmer is adding to often appear to support this process. In World-Builder, a system for constructing and playing a role-playing game, the game becomes more involved as the user adds more characters, locations, objects and code. In reality, however, there are limits to what can be achieved in this manner and programs generally require one to have an overall concept and to integrate the various elements of the program. The reason for this stems, essentially, from the minimal support that is really available to programmers in doing this. If they want to add some functionality at a late stage, then it is up to them to make sure that it is integrated into the whole. If the program is to be efficient and to do what is required, the programmers must plan how this is to happen in advance of any coding. This approach could become a possibility if "intelligent" support could be offered the user in the form of an interface that could incorporate code and instructions into an existing program.

3.4.2. Helping the Programmer Identify What Code to Use

One of the major problems for the non-programmer when coding is that of finding the code or methods to perform the desired task. The usual solution to this involves learning a vast vocabulary of statement forms and higher level chunks that can be mapped onto the desired goals or methods. This requires considerable learning and practice over time and would not be possible for the non-programmer. A number of approaches have been developed that facilitate this process of finding the code to use and their potential for solving this problem is discussed here.
One strategy that is used a lot by many programmers is that of creating new programs either by adapting old ones or by taking chunks from these to create new ones. Clearly, non-programmers are not going to be able to do this as they will not have a battery of programs that they have written previously to draw on. It is possible, however, for the programming environment to provide this for the programmer. In order to get an idea of the potential for this approach, an existing system that relies heavily on this, Smalltalk, is explored.

In Smalltalk, an OOPS, a vast library of existing object classes is provided. These consist of generic entities each with a battery of programs (methods). This provides a great deal of scope for code reuse and is the main way in which programming occurs. New object classes can be created by creating subclasses of existing ones but with altered or amended methods. Clearly, then, OOP works by modification not generation (Thomas, 89). A study has been reported (Lange & Moher, 89) of one expert programming in OOP and found that there was a great deal of reuse of code. Eighty-five percent of the code written was adapted from other methods which were used as templates. Eight percent just consisted of simple methods.

This approach has certain repercussions on what is easy and what is hard. On the positive side, the properties of inheritance and reusability of code save coding time (Tesler, 86, Cox and Hunt, 86). In reality, however, one needs to do a lot of typing to set up a system, despite the inheritance. In the study by Lange & Moher, it appeared as if the use of existing code that worked meant that the programmer did not have to understand the code at a detailed level but could just work with the aspects that needed to be adapted. This seemed to a great extent to be due to the programmer’s familiarity with these methods (she was one of the main developers of the system). Making good use of this facility requires a degree of planning as one searches for generic functions and methods that can be used by a number of objects (Thomas, 89). Finding existing classes that are close to what is wanted can be hard. One often needs to spend a lot of time browsing, looking for them (Tarumi, Agusa and Ohno, 88 and Lalonde et al., 85). Just navigating the system can be very hard as mature OOPS’s are very large (O'Shea, Beck, Halbert & Schmucker, 89). Learning to program in an OOPS involves a lot of learning of the library’s contents. Again, in the study by Lange & Moher (89), the subject had a detailed knowledge of the available classes and their methods and this would not be available to the non-programmer.

It can be hard to decide whether a given object or class is suitable. This is because many of the characteristics of the class are inherited from super-classes and are not
immediately present within the definition of each class. Also the name of a class may be misleading as the actual properties may not be directly related to what it is simulating (Lalonde et al., 85). Having found something that seems useful, one must decide how to adapt it. It may be unclear what aspects are crucial to the design. This comes down to attempting to understand why the original class designer made the choices made. Basically, the evidence is that reuse can be valuable but only after considerable experience and with a good system for organising the library so that useful programs can be found (O’Shea, Beck, Halbert & Schmucker).

Other solutions to this problem include example-based systems and menu-based systems. The former allows the user to type code in but offers a library of programs that the programmer can call up to examine and copy from. Studies of these find programmers using these mainly for ideas and as a way of getting information about the language (Neal, 89). The latter provide the units of code only through menus. This approach is often associated with visual programming systems (eg. VIP, Matrix Layout, LabVIEW, etc.) as other input techniques have not been developed to date. One final, similar approach is that offered by KBEmacs (Rich & Waters, 88b) where useful chunks of code are available in a library for copying directly into ones program. All of these approaches are limited by the fact that the programmer must become...
familiar with the libraries on offer. Currently, such libraries, while they may be well organised require the programmer to find the item required by searching for it. What would provide a much more valuable resource would be one where programmers could state their code needs and be taken to the appropriate part of the library. Until then, they do not represent any solution to the problem of the user having to work with a vast number of possibilities (see Figure 3.5, for an example).

3.4.3. Breaking the Task Down

Programs can run into many thousands of lines of code. This will inevitably mean that programmers cannot manage all the information involved at one time. As was mentioned in the previous chapter, expert programmers become able to handle more information as they identify chunks that can be grouped together. Even so, real programs for real world problems represent a lot to take in. For non-programmers who will not have been able to acquire this ability to chunk code, the problem is considerable. A number of strategies have been developed for dealing with this problem. These include increasing the power of the vocabulary, breaking programs down, and allowing the programmer to work with small bits of code at a time.

Early programs had many more lines of code than ones written in current languages simply because every step of the process had to be spelled out. It rapidly became clear, however, that by grouping commonly occurring steps together and giving them a single name, the length of programs could be considerably reduced. This inevitably reduced the expressive power of the language, however, but this did not matter so long as the groupings were well chosen to support the goal in hand. This approach has been taken even further in many languages where very powerful commands can be given using very few lines of code. This is often the case in fourth generation languages (4GL's - see Figure 3.6). This solution will only work, however, if the language is designed to be limited to a given domain. Otherwise, it just results in a massively increased vocabulary as all the specialist activities that need to be expressed are given a name. If one is willing to accept a limited domain of program, then the length of programs can be very short indeed. At the end of the day, this results in applications that do not appear to involve any programming because programs consist of single line commands that initiate established routines. The power is lost, however. What is required is a compromise that suits the needs of the user while remaining sufficiently simple.

The other problem that then arises, however, is that powerful commands require more understanding. 4GL's are a case in point. They offer the user considerable power with little code to write but they can result in very poorly structured programs that are hard to
maintain and not very efficient, unless the programmer is careful and makes sure that what is being worked with is fully understood (Sudwarts, 89, Haviland, 87). While high level languages can reduce the length of programs they do not necessarily reduce what the programmer must understand. This latter element depends more on the choice of the routines that are provided. If these are designed to incorporate good design understanding so that it is not possible to create poor programs, then the programmer can be spared some of this effort. Also, if the chunks are well chosen so that they map well onto the programmer's understanding of the domain, then little detailed understanding may be necessary. This is rarely achieved with any degree of success, however, as the tools that are provided do not really embody any great intelligence and the programmer is left with the burden of responsibility as much as ever.

```
PROMPT-FOR Salesrep.sales-rep
FIND sr USING s-f NO ERROR
IF NOT AVAILABLE sr THEN DO:
MESSAGE "Now created - add info"
CREATE sr
ASSIGN sr
UPDATE ....
ELSE DO:
MESSAGE ...
UPDATE ...
END
```

Figure 3.6. Some procedural code from a 4GL (PROGRESS). This is equivalent to 150 lines of COBOL code (Watt, 87).

The second approach involves ways of breaking the programs down. There are two basic ways of doing this so that programmers do not have to deal with anything very complex at any one time. Either the program consists of a number of separate program that are, themselves, kept short, or else a hierarchical structure is employed where programs are written that are used as procedures or functions by other programs, and so on up to the overall program. The best examples of the former approach are HyperCard and OOPS's. In both cases, whole programs are really systems of smaller programs that interact with each other to varying extents. Thus, in HyperCard, a program will be one or more stacks of cards, each of which will have buttons and other objects with programs associated with them (see Figure 3.7). These can be coded independently to a certain extent and will often be used independently. While this can simplify the programmer's task because only one button needs to be considered at any one time, there is still a need to have some idea of the overall organisation and how each button will work alongside the others. This is demonstrated even more clearly in
OOPS's such as Smalltalk. In these, a program will consist of a number of object class definitions, each with methods and instantiations. The programs here are far from independent, however, and the programmer must maintain a clear idea of how the whole system is to work and how the objects are to interact.
The other approach to breaking programs down is by means of sub-procedures and function calls, of which most programming languages support the use. This enables the program to be modularised and both reduces the size of any one program and reduces the need for duplication of code (see Figure 3.8). As with the use of many small programs however it does not result in the programmer being able to deal with each individual part on its own and there must be maintained an overall understanding of the whole. In complex programs this can represent a major strain on mental resources of the non-programmer.

3.4.4. Generating Safe and Readable Code

There are two main concepts when aiming to protect programmers from generating poor code. The first is to avoid system crashes and other negative experiences. The second is to encourage good program style so that the resulting program is readable. Languages of the former type are what Green (in press) calls "scruffy" and the latter ones are "neat".

System crashes and the like tend to occur when programmers have been using facilities that affect the computer in fundamental ways. In general, languages aimed at non-programmers or novices do not let the programmer access such low level machine commands. These languages must inevitably sacrifice some degree of power in doing this and limit the scope for programmers but that may be the price that must be paid for such protection if the programmer is to create programs without any intelligent monitoring of the resulting code. Clearly, if there were some means of assessing each program before it was run to identify problems, then a limiting approach might not be so necessary. Currently, however, this is beyond the scope of the available technology.

The other sense of protecting the programmer is less simple to deal with, however. This involves helping programmers create code that is well structured and consequently easy to read, debug and maintain. An example of a language that was bad in this respect was the early BASIC. In this language, the control structures provided included the GO TO command that allows the execution to follow whatever sequence programmers chose. The use of the GO TO command was the main method of diverting the course of execution. Its use was unrestricted and allowed the programmer to instruct the computer to move to any statement desired. In addition to this, calls to subroutines used a GOSUB command that took execution to the part of the program
with this name, carrying the values of variables with it. One of the consequences of this was that following BASIC programs could be hard (O'Shea and Self, 83, Stewart, 84) owing to the freedom of structure available to the programmer. Unless one was disciplined, the flow of the program could be extremely complex. Basically, the ability to use GO TO and GOSUB statements freely can result in programs that jump around from one part to another with considerable danger of bugs associated with keeping track of variable values being missed (see Dijkstra (68) and Ledgard and Marcotty (81). for a more detailed criticism of this and Figure 3.9). More recent versions of BASIC

Figure 3.8. An example from Logo of the use of sub-procedures to simplify programs. The first program performs the overall task, while the later ones define the meaning of user-generated procedures within it, and each other.
have attempted to deal with this problem by encouraging a structured programming approach (eg. Stewart, 84).

```plaintext
11 LET I = 0
12 LET S = 0
13 LET S2 = 0
14 LET I = I + 1
15 IF I > 100 THEN GO TO 25
16 READ D(I)
17 LET S = S + D(I)
18 GO TO 14
25 LET M = S/(I - I)
31 LET I = 0
32 LET I = I + 1
33 IF I > 100 THEN GO TO 40
34 LET S2 = S2 + ((D(I) - M)**2)
35 GO TO 32
40 PRINT "SUM", "SUM OF SQ DEV", "MEAN"
45 PRINT S, S2, M
48 DATA ....
50 END
```

Figure 3.9. An example of BASIC program that is hard to follow because of its use of the GO TO command (from Nolan, 69).

This solution is not as successful as it might appear. In addition to languages being designed to encourage this approach, programming environments have also attempted to do this (Giboin & Michard, 84), assuming that the provision of the appropriate tools will guarantee the adoption of a disciplined approach and the generation of good code. The evidence is that programmers do not work in this way (eg. Sumiga, personal communication). This is becoming increasingly realised in the psychology of programming literature as ideas about how programming should be are replaced by more accurate ones about how it actually does occur. This raises a dilemma for the designer of languages aimed at non-programmers. To what extent should they be forced to be disciplined in their programming? If they are forced in this way, it will be extremely unnatural and require considerable adaptation if they ever are to succeed in adopting this approach. If they are allowed to be scruffy in their programming and environments supporting this are developed, then how can one be sure that they will be able to generate well-managed programs unless they develop considerable skill in working with them? Tools are being developed to support scruffy programming (Green, in press) such as environments where the program can be run in a way that attempts to ignore any bugs and interpret what the programmer wanted it to do and where code can be used in many different ways. Some of the ways that have been developed to support good coding practices are discussed in the next section.
3.4.5. Summary

In summary, there are a number of aspects of generating code that require either considerable skill or knowledge of the language. To date the support that is available for these does little to overcome the barriers to non-programmers. This is mainly seen to be due to the lack of "intelligent" tools that could act in a pro-active manner and interpret the user's requests or cope with disordered code generation.

3.5. Support for Reading Code

When considering the processes of reading programs rather than writing them, questions must be asked about how easy it is to extract an understanding of the program from the code itself and about other ways of finding out what the program will do. Thus, there is a need to consider the notation with a view to general readability, ease of information extraction, the ease with which programmers can switch from writing code to running it, the nature of error messages, and other aids for obtaining information about the program. These will provide valuable feedback for programmers in determining whether their programs are good enough or not and are discussed in the following sections.

3.5.1. General Reading of Program Code

The choice of notation is often aimed at general readability. Papers, etc., on programming language design describe the many issues that go into creating the syntax of a language that could help novices (O'Shea and Self, 83) but none of these seem to represent major benefits on their own. What is needed is a single idea that could revolutionise the way programming notations can be used. In the absence of this the non-programmer will still have to learn a whole new language, even if small changes in notation are used to improve readability. Two such ideas have emerged to date. The first is the use of graphics or visual notations rather than textual ones. The second is the use of English-like (or French-like, etc.) notations.

3.5.1.1. Visual Programming Notations

It was suggested that CASE tools might be developed such that programmers would merely have to represent their program designs in diagram form and the code would be generated automatically from these. It was pointed out, however, that the advances that this represents might basically be either a change from a textual to a visual notation or a higher level of language. In this section the potential benefits of changing to a graphical
or visual notation are considered by an examination of currently existing or prototypical languages of this type.

Visual qualities are increasingly exploited by textual languages with the indentation of Pascal probably being the best-known example, taken up by many others such as HyperTalk. This certainly suggests that there are qualities available that could be exploited successfully. Myers (89b) suggests that visual programming could represent a way of making programming easier and more accessible to non-programmers. The reasons given for this are that they can use more than one dimension (programs can run across as well as down a screen) and people are good at exploiting this. He also claims that pictures represent a higher level of abstraction so no syntax is involved. The logic here appears to be confused. Certainly, the graphical notations experienced by the author have not involved syntax as it currently exists but there was a need to provide parameters to the various entities involved and these required some understanding of what was appropriate and what was not. Also, the notion of a higher level of abstraction is only helpful insofar as it generates a meaningful organisation of the domain and this can be advantageous without being at a higher level. Visual programming is distinguished from program visualisation which can use graphical techniques to help one understand programs (Myers, 89a, Green, in press). Also, it may or may not result in programs with strong visual content. Finally, Glinert (89) talks about "iconic" languages where one is working mainly with small images and minimally with text. In particular he cites PICT which was a language aimed at making programming more accessible to novices.

The developments in this area are such that a number of basic issues and problems have been found. Glinert's point about iconic languages is to distinguish them from languages where the visual element is rather incidental to their design. There is, in reality, a continuum in this respect. Visual Interactive Programming (VIP) and Matrix Layout are languages where boxes are used purely as an alternative representation for standard program statements. With VIP, it is possible to get a print-out of the program in purely textual form much like any other language. The visual element is enhanced by the fact that the boxes are organised using flowcharting conventions so that program structure with loops, etc., is clearly visible (see Figure 3.10). Flowcharts have their own problems however (Green, 82). Similarly, BLOX uses statements like traditional languages but employs a jigsaw metaphor to deal with syntactic aspects (Glinert, 89). In LabVIEW, the icons represented are quite different from statements in a program and
they are linked by the data that passes between them rather than by passage of control. Green (in press) also makes this point, that the diagrammatic notations used should be chosen with care if they are to represent something valuable that goes beyond what text can offer. Diagrams allow one to represent many different qualities in quite different ways to text (using shape, symbols, colour, etc.) and these opportunities need to be taken advantage of. This is being realised by people working in this area (e.g., Myers,
89a) but there is the equal danger that these possibilities are used to enhance a language but do not contribute to its ease of use because they do not carry inherent meaning.

This is one of the major dangers of visual languages. While providing the language designer with a wealth of possibilities for ways of representing programming concepts using various codes of icons, one must remember that all codes are purely based on convention, that conventions must be learnt, and that icons require interpretation (Myers, 89a). The strength of a language will not lie in its being visual but in how well it makes use of the visual possibilities to convey meaning to the programmer. This relies on two factors. The first is the appropriate choice of entity to be represented and the appropriate choice of organisational structure. These must be meaningful to the programmer and map well onto existing knowledge structures. The second is the choice of representation of the entities so that what they refer to is clear. Many workers in this area are aware of this and Myers (89a) points out that the visual images used need to be close to the user's conception of the domain. Tanimoto and Glinert (89) suggest how to design a good visual language. They argue that one needs to start by bringing out a mechanical or physical model of behaviour that is familiar so that the language can rest on some system that is already well understood by the programmer as a metaphor. This is likely to vary from domain to domain. One must then consider the representation of control and data flow. Finally, one must design good icons that link well with the metaphor used and are meaningful to the programmer. They suggest the use of stylisation to highlight important aspects of the images used.

![Figure 3.11. A small LabVIEW Diagram. This is the way the program is represented in LabVIEW, with the lines representing flow of data rather than control.](image-url)
VIP and Matrix Layout represent poor examples of this and LabVIEW a good example. In the former, the entities represented, as has been mentioned, are program statements much like any other program statements and are as hard to grasp as any language. While the organisational structure is clear, it seems (but this must await empirical validation) no more informative than standard textual program structure. In addition to this, the representation of the statements is arbitrary and one has to open boxes up, one at a time, to see what the statements actually do and what variables they use, etc. The distinction between box types is mainly textual. In LabVIEW, however, the entities

![Diagram](image)

*Figure 3.12. The Select or Case structure. This is one of four similar structures in LabVIEW (the others represent Iterative (For) loops, Repetitive (While) loops and sequence. Here the large boxes contain the actions the selection of which is determined by the small units feeding into the box structures.*

are either "virtual instruments" which can be understood as data collection or process control devices such as are used by the engineers for whom it is designed (Bhaskar, Peckol, Fluke, & Beug, 86), or data manipulators such as adding or multiplication boxes. The representation of mathematical calculations is strange, however (see Figure 3.11). There is a special function that can be used to write formulae in but this is not
very easy to use. The main weakness lies, however, in the representation of flow of data and flow of control which can be confusing and appears somewhat arbitrary. Flow of data is by "wires" linking virtual instruments, which seems to map well onto concrete concepts, and no instrument will activate until it has all the data it requires. This, however can result in programs of which the behaviour can be hard to predict. The representation of the boxes seems to work well with the purpose of each box appearing relatively clear. However, the looping structures, etc., are represented by

Figure 3.13. A common strategy for dealing with large, complex visual programs is to provide an overview facility where the structure can be seen even if some of the detail is lost. This example is from VIP.

larger boxes that can contain all the actions to be repeated, but this does not map well onto any existing understanding and appears to be harder to assimilate (see Figure
It is clearly the case that visual representations are better for some domains than others. The experience with LabVIEW suggests that mathematics may not be well represented in this way. Myers (89a) and Green (in press) point out that some specific types of program are more easily represented graphically such as concurrent ones, real-time control, or programs for creating graphical interfaces. It can be hard to represent parallel processing in text and one must usually have to learn that a given notation indicates parallelism. In visual notations such as Prograph, this can be much more intuitive.

Other problems with visual notations relate to how well they represent large, complex programs. Diagrams rapidly become too large to fit on a screen of standard proportions and this can make working with them very frustrating (Myers, 89b), so that overview facilities are required (see Figure 3.13). The other problem is with the tendency to become very messy. Links by wires, etc., rapidly become very confusing and difficult to follow (Myers, 89b). For example, LabVIEW diagrams can be confusing (see Figure 3.14). Both of these problems can be true of textual notations but they are indicative of the fact that visual notations do not simply mean an absence of difficulty. LabVIEW clarity relies on use of hierarchy (see Bhaskar et al., 86). There is a need, if visual notations are to take off, for improvements in inputting and outputting technology in general. Visual systems are currently menu-driven which has problems as outlined in section 3.4.2.

Few empirical studies exist comparing visual and textual languages and there is very little formal evidence of the worth of such systems in general (Myers, 89b). Ones that have been done have found some benefits for comprehension in using visual notations (Cunniff and Taylor, 87) with small programs. Given the difficulties with large programs, it seems particularly important that such languages are not just tested on "toy" programs as the problems may not be able to be scaled up. The successes claimed to date are in helping to teach programming and in allowing non-programmers to enter information in limited domains or construct animations and simple computer aided instruction applications. This suggests that visual programming is very dependent on its success on the choice of domain. One of the problems with studying visual languages is that there is an absence of any formal means of representing them comparable with the Backus-Naur Form (BNF - see Figure 3.15) for textual languages.
Given the above discussion, it would appear sensible to treat visual programming as a
technique like any other that should be taken advantage of but not used just for the sake
of it. C² is a programming package that combines text and graphics in parallel so that
the same program can be manipulated in either notation and changes made are visible in
both. This sort of idea may well prove to be the way forward in this respect, and other
workers are interested in how to combine text with graphics in general (Myers, 89b).
This would include providing ways for the programmer to add comments and other
documentation.
3.5.1.2. English-Like Notations

An alternative approach that has had a lot of interest focused on it is that of making the notation mimic "natural language" sentences such as English. There are now a number of languages and systems that have taken this approach assuming that there will be less to learn because it is close to a language that is already familiar to the programmer. Examples include COBOL, HyperTalk, and certain 4GL's. COBOL (COmmon Business Oriented Language) was intended to be a machine independent language for business data processing problems. One of the main emphases in the design of COBOL was that programs should be easy to read and maintain by non-programmers such as management personnel. With this goal in mind, the syntax of COBOL was "designed to be as much like ordinary English language as possible while containing those elements required by the computer system in use" (Maynard, 72). HyperCard was released at the end of 1987. It is an application builder that runs on Apple Macintosh machines. Programming in HyperCard can mean cutting and pasting objects with programs already attached to them, or actually writing code in the HyperTalk programming language. The most striking thing about this language is that the actual vocabulary and syntax is very English-like. This is designed with the intention of making programming more accessible to non-programmers (Durham, 88).

Both languages allow the free naming of variables, values of variables, functions and procedures. They also have syntaxes that result in statements that mimic English closely, in a superficial way. Examples are:

"ADD BONUS TO BASIC GIVING GROSS."

from COBOL and:
I put "Beethoven" into composerName from HyperTalk. In both languages, however, there are many reserved words that must not be used as names of variables, etc..

Both languages have some aspects that are optional or where a number of different ways of expressing the same statement exist. Thus, in the COBOL statement:

```
READ MASTER-FILE RECORD INTO WORK-AREA
```

the word RECORD can be omitted and:

```
ADD SOCIAL-FUND, PENSION GIVING DEDUCTIONS.
COMPUTE DEDUCTIONS = SOCIAL-FUND + PENSION.
```

both mean the same thing. Figure 3.16 lists five different ways of expressing the same command in HyperTalk that can all be used interchangeably.

The freedom offered by these languages clearly has some positive implications for both writing and reading code. The programmer can chose names that are meaningful. Thus, a value that refers to the annual salary should be called ANNUAL-SALARY or annualSalary. This, combined with the syntax, can result in statements becoming very readable and the meaning can be very clear without the need for programmer comments. Also, the programmer can express statements in ways that are more meaningful. There are also negative implications, however.

Writing code in COBOL can be much slower than it is in other languages (Research & Educ. Ass., 86) and programmers may not make as good use of the variable naming freedom as they might. Ledgard and Marcotty (81) argue that having different ways of writing the same statement adds complexity to the programming process and, if
different forms are used in the same program, can make reading more difficult. Small differences in language produce important changes in meaning. For example, the following two lines of code look very similar:

```
ADD SOCIAL-FUND, PENSION, BUPA TO DEDUCTIONS.
ADD SOCIAL-FUND, PENSIONS, BUPA GIVING DEDUCTIONS.
```

but they are equivalent to, respectively:

```
D = D + S + P + B
D = S + P + B
```

This basically goes against the basic language design guide-line that differences in meaning should be accompanied by clear differences in code (REA, 86) and is likely to result in bugs that are hard to detect. Similar problems exist in HyperTalk. While HyperTalk aims to make its statements very readable by mimicking English, it is also true that some commands can be very un-English-like. For example:

```
set the visible of card field 1 to not the visible of card field 1
```

Also, in order to make assignment statements readable, HyperTalk makes an odd use of "put", "it" and "get". For example, while one can assign a value to a variable by a statement such as:

```
put "cheese" into shoppingList
```

one must remember that this will actually replace the current contents of shoppingList and what is needed is:

```
put "cheese" after shoppingList
```

to add "cheese" to the end of the contents. One can also write:

```
p
```
and this opens up a Message Box on the screen with "cheese" written in it. This can be very useful for debugging but jars with the English. HyperTalk has a special variable called "It". Values can be put into "It" by using "get" as in:

```
get theDate
```

which can then be followed by:

```
put It into card field "date"
```

This seems to be a dangerous construct and liable to result in mistakes unless one can work out a way of understanding what is going on that is not confused by the language used. The freedom of expression cause problems also as there are a number of rather arbitrary rules associated with it. For example, the rules for using "the" in property names are shown in Figure 3.17. These are immensely hard to learn and make no clear sense. Errors resulting from this are also very hard to identify because the statement is likely to read well and the error message will not be sufficiently explicit.

* The word "the" is never required in a "get" or "set" statement.
* If the property has parameters, "the" may be included or omitted.
* If the property has no parameters and appears in any statement other than a "get" or a "set", "the" is required.

```
Figure 3.17. The rules for using or omitting "the" (from Shafer, 88):
```

COBOL never really succeeded in getting executives to read programs (Weinberg, 71). The success of HyperTalk in reaching non-programmers is yet to be formally evaluated. The use of English-like structures, while apparently facilitating certain reading tasks such as getting a general understanding of the program's goal and activity, can make others such as debugging and detailed comprehension much harder. Certainly there are advantages of HyperTalk in avoiding strange symbols or names (Green, in press) but it is debatable whether this could not have been achieved without mimicking English. It is also important to note that COBOL is basically a language for manipulating often numerical data using simple arithmetic methods. English is not a language that is well suited to expressing such methods. It would seem to make more sense to design the language to be such that these methods could be expressed easily as well as being one that will be familiar to most intended readers. The notations of algebra would seem to be the best candidate for this and English is clearly not. This is
demonstrated by the observation that in normal human-human communication one will often need to resort to algebra rather than English.

Another factor that was very clear in the analysis of the languages was that the freedom was combined with rigid restrictions of an extremely arbitrary nature. This is very different from the way English is used and may well result in programmers being confused by the apparent freedom of expression into making bugs that are then hard to locate. It has also not been formally established that COBOL or HyperTalk programmers really are aided by the possibility of transferring existing language knowledge to coding although some have suggested that they are and that the rules can be acquired relatively easily (Green, in press). One clear indication that HyperTalk does not fully achieve its goal of making programming immediately accessible to non-programmers is the size of the books and manuals that have been written to teach it. While the language may well facilitate the expression of statements once the programmer has mastered what the language can do and how it works, it may have little advantage for the novice.

It is worth noting that studies of errors made by novice programmers often appear to be due to the inappropriate use of "natural language" concepts or approaches to communication (Kurland and Pea, 83, Taylor, 88, Bonar & Soloway, 85). This could be used as an argument for the development of languages that are closer to "natural language" so that such approaches will result in less errors. It is not clear, however, that this is really the case. It seems more likely that such languages will encourage the programmer to attempt to employ existing, inappropriate knowledge to coding in the mistaken belief that the computer is closer to another human being. What is needed here is a study of whether novices and non-programmers can make accurate guesses about how to express things in these languages or whether they make as many errors as in other languages.

3.5.2. Extracting Other Information About Programs

Programming notations only make certain aspects of the program's nature explicitly visible. This makes it difficult to extract certain information that will be useful for debugging or understanding the program in general. The following sections discuss techniques that aim to provide other sorts of information to the programmer.

One of the main concepts in this area is that of program visualisation tools. These allow the programmer to obtain different perspectives on the program, so that different questions can be asked. Such tools can generate such visualisation automatically while
others rely on the programmer to construct other representations by hand that can then be used for analysis. Often the aspects to be visualised will be data so that one can see how the variables are changing as the program runs (Brown, 89, Myers, Chanhok and Sareen, 88). Also, the program structure can be visualised in various ways. Both of these representations can be animated or static so that one can actually see the program in action, or else see the state at different points in the process. Many of these tools are used for debugging. One of the most obvious is the provision of error messages after programs are run. These can often be cryptic and not help the programmer identify the location of the bug or the sort of error to look for. There seems to be no easy way of dealing with this as no system has error messages that make debugging into a trivial problem. One must always interpret the message and search through the program for the actual bug.

What is needed in the design of program visualisation tools is a good understanding of the information the programmer needs. This is often lacking in the rationale of the existing tools and there is little empirical evidence that they really help (Green, in press). Also, with programs of any size, the tools just provide a different representation of the program that is as large and unmanageable as the program itself with no ability to focus on the likely location of the error. This must be done by the programmer and can represent a formidable task. The Transparent Prolog Machine is a good example of a tool of this type that has been developed extensively. It suffers from the problems of many of these tools, in that the programmer must become familiar with yet another notation and skilled at interpreting the resulting diagrams. Also, when the program gets big, the diagrams become either very big or very cluttered, or else one loses most of the detail.

One useful concept in this area is that of the browser. When one is dealing with a view of a large program, one will need help in finding one's way around it and returning to parts viewed earlier. Browsers would support this navigation process as part of a program visualisation tool or just for viewing the actual code. The understanding of this area and the extent to which they can really help the programmer are unclear as yet.

3.5.3. Facilities for Observing Programs Running

As well as making programs easy to read, the other way of making it possible for programmers to establish what they do and whether they do what they are supposed to do is by getting them to run. What is desirable here is the ability to test out small code fragments to check on their behaviour and to test the whole program as it currently exists. Interactive programming is a technique offered by many current programming
systems (eg. Logo, BASIC, etc.) where the programmer can run single lines of code to observe their behaviour. This provides a valuable aid to learning the language and enables the programmer to construct a good understanding of the effect of the various statements used. Seeing one's code run is valuable way of getting feedback about the extent to which it is correct. It is helpful both for the programs to run after minimal time and for the programmer to switch easily between writing and running code. Both avoid wasting time by encouraging rapid and early checking of code. Few systems offer both of these options. Microtext is one that does. While the support that these facilities offer is important, it does not substantially alter the demands made on the non-programmer.

3.5.4. Summary

In summary, obtaining feedback about the code one has generated is very important and is supported by the notations in which the code is represented and the tools available for obtaining the feedback, particularly for getting the program to run. There are many problems with providing notations that will be easy to read from and these have been discussed. The use of concrete metaphors seem to represent the best approach to adopt but it certainly appears as if there will inevitably be some learning required.

3.6. Disguising the Programming Process

When considering new developments in tools supporting programming there is a final group that should not be neglected. A number of programming tools provide programming power in a way that subtly disguises the fact that the user is programming at all. It may well be that, if programming is to become accessible to non-programmers, then it would have to be hidden in this way owing to the current ideas about programming. This was discussed briefly in Chapter 1 where it was mentioned that many managers who find they have to do a certain amount of programming do not like the image that this generates. This section discusses these approaches and considers their value aside from disguising the fact that the user is programming.

DiSessa and Abelson (86) suggest that, in the future, programming will be like writing and drawing is now. There will be the experts who will be generating large programs analogous to books that others read and use. These programs will be "scholarly" and advanced. There will also be the general public who will just write small programs for their own use or for specific purposes. These will be more sketch-like and not especially clever. Also, the public will be able to "quote" from the experts' programs in their own ones. They suggest that programming will be more like reconstructing a
computational medium, in the way that writing is. DiSessa and Abelson describe a system they have developed, named Boxer, that illustrates their idea. There is no suggestion in this, however, that might make programming any easier, other than those already discussed.

Another approach that is currently used to a considerable extent is what could be called "programming as enhancement". Here, the user is working with an existing tool that runs adequately with no programming but that can be customised in various respects. This can include setting up aspects of the operating system one is working with or creating macros that can be used like standard commands. The evidence currently is that macro-building facilities are often not used (Wandke, 89). Some work by a technique known as "programming-by-example", that is discussed later and this may be a successful approach that could be adopted. Basically, however, this general idea does not provide any suggestions for making programming any easier.

If the programmer does not want to have very much control over the programs that can be written, then simpler interactive devices can be used such as form-filling for defining the program. This could be seen as an extension of menu-based systems and will only be useful if the options are very limited indeed. Otherwise they will suffer from the same problems of other systems. Often, such systems will be such that a number of different levels of programming skill can be catered for. This provides the option, either of entering programming gradually, or of accepting a considerably reduced degree of control and is not really a solution to the problem of this thesis.

Although these approaches appear very compelling they do not really suggest viable ways of avoiding the fundamental problems that are faced by the non-programmer that are different from those already discussed. As such they do not appear to have anything to contribute of value to this discussion.

3.7 Discussion

Having examined a great variety of approaches to improving the programming process through the provision of novel tools and environments, it is now possible to assess the extent to which they will offer the possibility of making programming available to the non-programmer. In doing this, one is looking for general insights that will suggest something about the form that such tools could take in the future and their potential. Ultimately this can only be speculative as one can never really claim to know the shape of future technology. It may be possible, however, to identify the most promising avenue for further investigation. That is the goal of this discussion, therefore. It starts
by suggesting that, so long as the facilities on offer do not contain any real
"intelligence" or ability to act pro-actively in support of programmers they will never
really offer any hope for non-programmers.

3.7.1. The Scope of Building Blocks and Dumb Tools

In summarising the results of this investigation, a general point emerges. The
approaches described are successful insofar as they provide the programmer with
either:

- building blocks that are more easily manipulated, or
- tools that support such manipulations.

In each case, it is argued below, the potential for supporting the non-programmer is
limited.

The provision of well designed machine models, constructs or notations can allow the
programmer to transfer existing understanding to the programming task. This makes it
easier for these "building blocks" to be manipulated because the cognitive effort is
reduced (they do not require so much processing) and knowledge of the properties of
such items can be employed. It is only possible to transfer a limited amount of the
skills and mappings needed to convert problems into solutions from this existing
knowledge, however. Most of the mappings will be novel because the solutions
generated will be novel. In addition to this, programmers must know what building
blocks are available for the construction of a solution. The fact that those provided are
similar to familiar ones in real life could enable them to guess at what others might be
available, but this will only be successful to a limited extent and there is evidence that
encouraging such guess-work could result in more problems than it solves.
Programmers must have ready access to information about what building blocks are
actually available and, at present, the best way of providing this seems to be by their
learning this.

The tools identified in this investigation all had one feature in common. They were
essentially "unintelligent". This means that they rely on the users to know how to use
them (and this may not be simple), and their activities did not include any Artificial
Intelligence methods or algorithms. Very often, programming systems will provide an
impressive array of such tools with the claim that they make programming easy,
whereas in reality they add more complexity to the task than they remove for the
non-programmer (e.g., Neal, 89, Beck, 89). Program visualisation tools merely offer another representation, program libraries require the programmer to locate the desired piece of code, CASE tools do not really suggest the solution to the programmers' problem, however much they may facilitate their finding it.

Such tools can be compared to a rowing boat that can be used to take someone to their destination but only if that person learns to row and does all the work. What is wanted is a power-boat where the skill is just in directing it to the destination and the work is negligible. Of course, there is no denying that programming could be made vastly more accessible if the systems offered optimised both of these elements so that everything possible was done to support programmers. If this was combined with well designed teaching material and on-line help systems, then life could be made a lot easier for those who have to program. However, it would appear that the potential for making programming available to non-programmers will always be limited insofar as the tools and facilities are incapable of acting in a pro-active and "intelligent" manner. While it would be possible to continue this thesis with an investigation into the ways in which some of these aspects could be developed, it might be more fruitful to explore the possibilities for "intelligent" programming tools. In particular, therefore, the area of "automatic programming" should be looked into. This may or may not yield a solution but it would appear to be the most promising direction.

3.7.2. Automatic Programming

A candidate for an alternative solution must involve more "intelligent" tools. This intelligence must extend, not only to program visualisation and the location of useful pieces of code but to the whole design process because of the many tasks involved. What is being discussed is essentially what is known as "automatic programming".

"Automatic programming" or "program synthesis" tools have been defined as those that:

* Automate of some part of the programming process (Barr and Feigenbaum, 82).
* Decrease the amount of effort required to obtain a correct and executable program (Biermann, 76),
* Enable the programmer to specify tasks at a more natural level (Simons, 83),
* Allow the user to describe what wants doing and have the program automatically generated (Johnson, 88),

87
These definitions are ordered from top to bottom in increasing ambition. The first and second definitions would include compilers and any tool that might support the programmer. The third definition would include any high level language. It is the fourth definition that is relevant to this section. If users could just describe what is required to the computer and have it generate the program for them (truly "automatic" programming), then there might be some reason to believe that this could make such programming open to non-programmers.

Automatic programming tools have been classified along a number of continua (see, e.g., Simons, 83, Barr and Feigenbaum, 82), but the most significant one for the purposes of this study is that of the method of communication between user and tool. Generally, there are four broad categories of approach:

- Formal specification,
- Interface definition followed by automatic code generation,
- Example-based programming
- Informal requirements specification.

The first of these does not help the end-user much as the language of specification is as formal as any programming language, although most automatic programming systems work in this way (Johnson, 88, Guicho, 83, Manna and Waldinger, 79). The second is currently available in some systems such as Prototyper. This allows the user to work with the aspect of the program that is most familiar - the user interface - and to define the program in terms of it. This will have advantages as discussed above, but ultimately still relies on the user to learn a vocabulary of terms and how to turn these into a formal program description. The third and fourth of these, however, provide a more realistic approach and these are discussed below.

Myers (89a) distinguishes two sorts of example-based programming. In one, users give examples of the input and output and the computer has to infer what would have happened in between. In the other, users actually goes through what they want the computer to do and it just mimics them, with no inference involved. The former of these approaches involves both sophisticated inferential capabilities on the part of the computer, but also the ability on the user's part to chose suitable examples that illustrate all the aspects of what must be done. This is not likely to be possible if one has only a simplistic model of the inferential process. Biemann (76) points out that implementing programming by example in this way is very hard, for anything but the simplest tasks.
The alternative, where the user essentially shows what is wanted to the computer might prove very useful in many situations. It could be seen as the ultimate in Direct Manipulation programming where the user takes the hand of the computer, as it were, and leads it. Already it has been implemented in HyperCard, as a Macintosh accessory (MacroMaker), and in some spreadsheet packages such as Wingz. In the former, users can set up buttons to link two cards by moving between the required ones. In the latter two, one can set up macros that will repeat, rigidly, any actions of the users. While these can be useful, they will only represent a serious option under certain circumstances. The task that one wants to program must be something that one can perform at the interface. This does not seem to be a candidate for very many problem domains. Also, without inferential abilities, the computer could never perform the same task on slightly different data, etc. To do this would require another walkthrough.

Comparatively few systems have been attempted that work with informal requirements specifications, however. This is partly because it is, technically, immensely difficult to achieve. Such a system represents the most fundamental rethinking of the programming process and requires least from the user and most from the tool. A number of tools have focussed on different parts of the informality of such requirements. Johnson (88) describes a tool that accepts ambiguous input and attempts to interpret it using contextual clues. Cohen (83) describes a language, GIST, that aims to take the user from an initial, incoherent specification to a model of the intended system that will make clear any inconsistencies or incompatibilities. Heidorn (76) describes four systems that accept "natural language" requirements and build systems based on them. Upon examination, the input is, in fact, very well structured and it is likely that the tools would have difficulty with authentic non-programmer input. Aside from the technical problems, however, automatic programming by informal requirements specification seems to represent the best candidate for an approach that might put programming power truly into the hands of non-programmers. This idea is taken a little further in the following section before being launched into properly in the next chapter.

3.7.3. The Next Steps

It has been identified, therefore, that the building blocks and dumb tools approach is limited in scope. Given this, it might be more valuable to open up a new approach to the programming process: that of automatic programming by informal requirements specification. In this paradigm, it may be possible that the programmer could generate programs with much reduced prior training and that the process of actually constructing
the program could be considerably shortened. It may be, of course, that this proves not to be the case, or else that the technical requirements for such a system are impracticable. In either case, the clarification of this will help those, either in the present or in the future, who are considering whether to attempt to build such a system, to decide whether it is worthwhile or feasible. In addition to this, informal requirements specification is currently a field that has not been explored in any depth so that an exploration of its issues should open up many that are relevant to other areas of human-computer interaction and applied psychology in general.

While there is a certain amount of work described in the literature that is on the technological side of the issue about how such a system could be constructed, there is none that has attempted to address the psychological issues. Whenever such considerations are involved, a very simplistic model of the user is employed or else the system designers appear to have tested their systems out on themselves (Heidorn, 76). The focus of this work will therefore be on the psychological issues involved and on identifying the central aspects and problems that will need to be resolved.

In order to make the focus of attention clearer, it is worth pointing out exactly what is intended by the notion of automatic programming by informal requirements specification. It is here that the concept of JEEVES is introduced. What is being considered is a system where the user can specify what is wanted in whatever manner seems most "natural", where the system then constructs a program that meets those stated needs and where further steps towards the final program occur by modifying the suggested implementation. This is analogous to many situations where a member of the public employs an expert (e.g., a fitted kitchen supplier, or financial adviser) to provide a service. In such situations, the expert will obtain a description of the client's needs (number of hobs, amount of savings possible), offer a possible solution and they will then engage in further dialogue until an agreeable solution is found.

3.8. Conclusions

It has been identified that current approaches to making programming accessible to non-programmers are unlikely to be successful beyond a certain point. This is due to their being limited to offering little more than improved building blocks and " unintelligent" tools. These require novel skills to master and can be complex to use. Indeed, they often do not really offer the support that is required. Given this situation, it is seen as valuable to consider a more radical approach to programming using automatic programming by informal requirements specification. It is identified that this area is relatively new and that no work has been done on the psychological issues that
arise out of it. The rest of this thesis will address this lack with the goal of establishing a science of this sort of dialogue that could be relevant to the interaction between people and "intelligent" computers in general.
CHAPTER 4
WHAT CAN WE LEARN FROM HUMAN-HUMAN DIALOGUE?

Abstract to Chapter 4

The work that exists focussing on human-human requirements specification dialogues is reviewed. This is found to lack empirical testing and rely excessively on anecdotal experience. It is, however, an important source of insights for the issues of automatic programming by information requirements specification. To deal with this, a study was performed that observed human-human dialogues of this sort and a number of important points about their nature and possible variations identified. Later in the chapter, a brief consideration is given to the technical issues that arise out of our goal and a sketch of the way they might be handled is provided. Finally, the chapter ends with the presentation of a framework that enables the identification of the main areas of any dialogue between a user and a JEEVES-like computer. This provides a basis for the next section that examines the main areas identified.

4.1. Introduction

In Chapter 3, it was argued that the best way of investigating the issues relating to the potential for making programming power available to non-programmers is by examining how it could be achieved through automatic programming by informal requirements specification. The first step in this is to identify what work has already been done that would be relevant. This chapter, then, makes a break with Chapters 2 and 3 and initiates an investigation into an area that could appear to have little relevance to programming. This is a result of the "quantum leap" in programming practice that is being suggested as the best way to tackle the goal of programming for non-programmers. This has two effects. On the one hand, it means that the relevant literature is no longer that traditionally related to the psychology of programming. The current work in this domain is very much focussed on programming as it currently manifests itself and is, therefore, of little relevance to the rest of this thesis. Instead, there is a need to identify domains of research and expertise that do contain relevant insights and concepts. The second effect is on the empirical work that can be done. Rather than working with existing tools and exploring their limitations, it is necessary to develop experimental scenarios where the subjects are interacting in ways that might be closer to the ways they would interact with a JEEVES-like computer. This tends to involve studies of human-human dialogues rather than human-computer dialogues.
The closest analogy to the situation of a user constructing a program using a JEEVES-like tool is that of clients describing their requirements to designers or programmers. It seems appropriate, therefore, to examine what is known about such activity. This chapter starts by looking at what is understood about such requirements specification dialogues. In what follows the person specifying the need is referred to as the "client". It is found that knowledge about the nature of requirements specification dialogues is very limited. The extent of this knowledge is described and it is concluded that a more detailed analysis is required.

4.2. Current Understanding of Requirements Specification Dialogues Between Humans

An understanding of requirements specification dialogues should have three elements. It is important to have a good understanding of both the process of requirements specification and the characteristics of the specifications that can be expected. Having achieved these, it is possible to consider how best actually to elicit good specifications in ways that are maximally efficient. The following sections examine what is known about each area in turn. The goal of these sections is to attempt to identify some useful taxonomies or structures that could be used to initiate a consideration of the issues.

4.2.1. The Process of Generating Specifications and the Nature of the Resulting Specifications

Little has been written about how clients specify their requirements. The focus is mainly on the nature of the specifications that result or on techniques for eliciting requirements. This is indicative of the lack of research that has been done in this area. More has been said about the nature of the specifications that can be obtained from clients, but even this work is largely anecdotal. A good specification for a design or a procedure has certain properties. Most of the statements that are made about specifications generated by non-experts say little more than that they lack many of these qualities. This often leads on to a discussion of particular tools for dealing with specifications of this kind. Thus, they are claimed to lack the various attributes listed in Figure 4.1.

This work is supported by more empirical research into procedure specifications that were found to have many of the same properties (Miller, 81, Gould, Lewis and Becker, 76, Galotti and Ganong, 85, Onorato and Schvaneveldt, 86). The main addition to this list that was found in this work was the extent to which the manner of communicating the specification was often found to be adapted to varying extents to the perceived needs of the intended user. In addition to this are occasional statements about what
such specifications are like. Thus, Harwood (87) claims that they consist of the client's idea of what the solution is to his or her problem rather than a description of the problem. Parkin (80) also reports this happening.

The greatest attempt to interpret these observations is by Eason (88) who suggested the source of the difficulties experienced by clients in this activity. He pointed out that they would stem from the fact that they may not have a clear idea of what is desirable, what is possible, and what the future needs may be. They will also lack the technical understanding to provide the sort of detail and clarity that is usually required. Finally, when presented with a design solution, they may well not be able accurately to assess it.

The main conclusions from this review are that:

* The specifications that one can expect will have many aspects that will make them hard to turn into a formal design;
* The specification may take the form of a design solution rather than a statement of the problem;
* The reason for the first observation probably stems from the client's lack of domain knowledge.

This provides limited scope for any useful analysis of the requirement specification dialogues that will suggest a taxonomy of issues. Clearly, it would be possible, on the basis of this, to explore these conclusions and how they could be manipulated but there would be a danger of missing more important issues. To obtain further information, therefore, the work on techniques identified for eliciting specifications is now discussed.
4.2.2. Techniques for Eliciting Specifications

The problems with the specifications provided by clients have been identified. These have resulted in two main categories of approach to eliciting specifications in a way that will maximise their usefulness to the designer. The first of these consists of structures for improving questioning as a means of eliciting a specification. The second is to use the provision of feedback. These two approaches are discussed below. While the first area has not been investigated to any great extent empirically, some studies are described that support the arguments that are presented in relation to the second.

4.2.2.1. Direct Questioning

Techniques for improving questioning consist of a consideration of the questions to be asked and of the way they are asked. The designer must decide what questions to ask in order to ensure completeness of the specification and to iron out any inconsistencies, contradictions or ambiguities. The main way in which this can be supported is by the use of formal notations for representing the specifications obtained to date. These work, therefore, much like CASE tools that will highlight any gaps in the knowledge or clashes between requirements (Parkin, 80, Kirby, Fowler & Macaulay, 88). Because the designer must convert every requirement into the formal notation—they also mean that any ambiguity or lack of clarity is more likely to be noticed. Vitalari (84) has written a critical review of many techniques used in Systems Analysis. He points out that the process contains many elements that are not formalisable and that most of the existing tools only support the most routine aspects of the process. He argues that the danger with these sorts of notations is that they deny much of the creative, intuitive aspects of the process and may serve more to impede than to support the client-designer dialogue.

One of the dangers of using such notations is that the desire to fill in all the gaps becomes the controlling element of the interview. Parkin (80) suggests a number of guidelines for creating a successful fact-finding interview and these include careful management of the interpersonal relationships established. In particular, the client should be allowed to be quite active in the discussions and not just be seen to be a passive source of information. He suggests that, while the interview should have an agenda and structure, this should not be stuck to too rigidly. While Parkin acknowledges the use of notations and other formal structures, he recommends that they be used after the interview and suggests a number of strategies to overcome the danger of misinterpretation, such as checking facts back and asking for clarification of vague answers. The actual form that questions should take has been discussed by
Parkin (80). Once again, these recommendations have not been subjected to empirical study but are based more on expertise acquired through practice. It is pointed out that general abstract questions about what is required are going to cause problems to clients who do not have a good knowledge of the technology available. Rather than asking what alternatives are being considered, analysts should find out about the problem and then generate alternatives themselves. These can then be offered to the client as a basis for further questions.

There is only a certain amount of information that can be obtained purely by asking questions or letting the clients describe their needs. It is generally recommended in any text on requirements elicitation that feedback be used to obtain further information. This technique is discussed in the next section.

4.2.2.2. The Use of Feedback

Two main studies have been performed that studied client-expert dialogues, in different settings. Both of these paid particular attention to the use of feedback and identified it as a crucial element in the experts' repertoire of specification elicitation techniques. The first of these was work by Carroll, Malhotra and colleagues (Carroll, Thomas and Malhotra, 79, Malhotra, Thomas, Carroll & Miller, 80). This consisted of a study of the dialogue between a designer and a client relating to the design of a computer system for a library. The main observation from this was that the dialogue could be divided into a series of dialogue structures that the researchers name "cycles". These were units of conversation pertaining to some small sub-problem that formed part of the overall design problem. The cycles consisted of the client giving a description of the sub-problem followed by an interchange where the designer clarified the nature of the problem and suggested a solution. In this way, the overall problem seemed to be solved by a series of local solutions that had ultimately to be combined into a coherent whole. This seems to be a problematic approach as the individual sub-solutions could easily prove extremely hard to combine. It would seem to be preferable to wait until the client had articulated most of the problem before attempting to suggest solutions. It was surmised that the reason for this unexpected approach was partly that it would reduce memory load but mainly that it would provide the client with good feedback about what the designer had understood. Any misunderstandings would emerge from the solution suggested, resulting in further information being elicited from the client.

The second of these pieces of work is that done by Falzon, Amalberti and Carbonell (85). The study was of receptionists in a medical centre taking telephone calls from clients. The main observation involved the division of the receptionists into experts and
others. What was found was that the non-experts would take minimal control of the
dialogue, letting the client specify all the aspects of what was wanted before taking any
action. In contrast, experts would let the client give only a very brief initial description
of what was wanted before taking control. This consisted of them suggesting a course
of action (solution) to the client that could either be agreed with or lead to the realisation
of a misunderstanding and further information. The expert receptionist was basically
attempting to categorise the client's needs into one of a number of stereotypical
possibilities and then suggesting the course of action appropriate to this. This seemed
to be happening on the basis of very simple keyword categories. In most cases, this
would be successful, because most needs were standard, and arrive at a resolution
extremely rapidly. In the situation where the need was not standard, the conflict
resulted in the receptionist getting to an understanding of what was wanted more
rapidly than if he or she had just waited. In this case, a more complex interaction was
required to deal with the particular case. It was found that these expert receptionists did
seem to be dealing with clients more successfully.

It is generally acknowledged that there is a need for the naive client to be given
feedback about what has been understood. This was seen in the descriptive studies
described above and Eason (88) suggests a framework for this. He suggests that the
process of specification elicitation involves two iteration loops. In the first, the client
specifies what is wanted and is offered a number of options which are evaluated and
selected from. Then a prototype is constructed which is evaluated in various ways.
Basically, the client is presented with increasingly detailed design proposals. Eason
suggests that the client only gives the minimum of information required at any stage so
that the description becomes increasingly detailed over time, rather than insisting on a
full description at the outset.

The need for feedback as a central aspect of the specification dialogue has resulted in a
lot of work going into the representation of this feedback. Initially, it took the form of
flowcharts that could be quite hard to interpret (Eason, 88) although they formed a
useful bridge between a formal design specification as needed by the engineers and the
informal requirements of the user (Potts, 88). Various options are available but all have
their shortcomings. The most popular approach is that of using prototypes that attempt
to simulate the design dynamically rather than by use of static representations of
activity. These are very useful as they help clients to decide what they want and to
focus their minds (Potts, 88) although, again, the work on this is anecdotal.

97
4.2.2.3. The Structure of the Dialogue

As well as looking at the detailed strategies used for eliciting requirements, studies have examined the global structure of the dialogues that emerges. In the studies by Carroll et al. (79), the structure was one of iterations through a small loop, with a number of these cycles together making up the dialogue. What Falzon et al. (85) found was that there would be three distinct and different stages to the dialogue between receptionists and clients. These were:

- Query statement.
- Negotiation.
- Modification of agendas.

Thus, the client would give an initial statement of what was wanted and this would be explored with the receptionist until it was clear to the receptionist exactly what was required. Having established what the client wanted, this was implemented in terms of making appointments with the appropriate doctor, etc. This is a very similar structure to that observed by Coombs & Alty (1980). The context in this case was computer centres and the dialogue involved staff offering advice to students and other computer users about how to solve problems. The structure identified in these situations also involved three stages with the first two being almost identical. The client states the query in the first stage and the advisor questions the client about it in the second stage. In the final stage, instead of entering an appointment, the advisor just tells the client what to do to solve the problem.

The study by Carroll et al. looked at very large problems of computerising a library while the studies by Falzon et al. and Coombs & Alty were of very small problems. This may well have had an effect on the structure of the dialogues. The size of neither of these is really comparable, however, to that of a single-user program. Because of this it is hard to predict what sort of structure might be most appropriate for the user-JEEVES dialogue. Clearly there must be some initial statement of the user's needs and there must also be some elaboration on this and suggestion of a solution. How these are best organised within a dialogue seems to be a potentially important issue.

4.2.3. Conclusions

The distinction of specification elicitation techniques between direct questioning and the use of feedback seems to represent an important concept. The issues that arise out of the use of each are very different and would need to be investigated separately. Related
to this is the question of how pro-active the expert should be in the dialogue. Thus, it would be possible to let the client take control over the dialogue or for the expert to lead it by asking questions or suggesting design solutions. The question of structure is also clearly important in designing JEEVES and relates closely to issues of control and feedback, etc..

These are the sorts of concepts that need to be identified before the issues that relate to them can be clarified. While the work to date provides a basis for this, it is anecdotal, and requires validation, or to be focussed at some particular aspect of the dialogue so that others are neglected. Also, it would be valuable to obtain information about the variation in strategy that is possible both for the client and the expert, so that one could identify the variables that must be described in classifying individual examples, such as those observed by Carroll et al.. In order to achieve these goals, a more wide-ranging study is required that will enable the examination of strategies used, etc., to varying levels of detail. To this end, a study was performed that would serve as a basic information-gathering exercise where observations about the nature and variation of such dialogues could be collected. In order to make it particularly relevant to the domain of interest of this thesis, the dialogue was engineered to be between programmers and clients.

4.3. Programmer-Client Dialogue Study

Given the state of this field that has emerged from the above discussion of the literature, it is clear that there is a need for the claims that have been made to be validated and for more detailed study, in order to obtain a clearer picture of what JEEVES has to contend with. In addition to this, a further exploration of strategies that could be used in eliciting requirements would help identify other ways in which the approach taken by JEEVES could be manipulated in order to produce a suitable interface. Finally, there is a need to structure the area and to identify any other basic issues that need to be considered.

All of the above are best dealt with by a study of dialogues of this type between two subjects rather than between a subject and a computer. In order to identify possible sources of variation and not to obtain results that may be limited in their validity, a number of task domains are used. In order to make the results as closely relevant to JEEVES as possible, the dialogue is between programmers and clients with the goal to clarify the clients' needs for a program that the programmer is supposed to write. The client is relatively naive about programming and about what potential there might be to create a program to support the particular problem domain. In the real world of such
situations, the dialogue is lengthy, taking place over a number of weeks or months as the programmer comes closer to understanding what the client requires and to building the program. The cost of monitoring these dialogues, once one had found a client and programmer pair who were both willing to be observed, would be very great. To do this for a number of such pairs would be prohibitive. Indeed, the fact that this study is intended more to generate ideas and concepts than hard facts about such dialogues suggests that sufficiently valid results can be obtained from observing role-plays of subjects in such situations.

4.4. The Method

Very briefly subjects in the role of client were given the bones of a programming task which they had to flesh out as they wished and then communicate to subjects in the role of programmer until the latter was satisfied that he or she understood.

4.4.1. Subjects

Eight dialogues were observed, each with completely different pairs of subjects. The subjects were either in the role of the programmer or the client. Subjects for each role were recruited from different groups. The programmers were all engaged in occupations closely related to computers and programming. The clients were all undergraduate students in the Human Sciences Department - reading Psychology or Information Technology and in their first or second year. All subjects were given a questionnaire about their occupations or courses and computing and programming experience. The clients all had less computing experience than the programmers, measured in the number of both applications used and areas for which computers were used (see Table 4.1). Seven of the eight clients had actually done some programming, although for most this was minimal (i.e., one course of Pascal with no practical use of it). This would be comparable to the situation of managers who might want to use JEEVES. Seven of the eight programmers had had experience of dialogues of the sort simulated by the study.

<table>
<thead>
<tr>
<th>Applications used</th>
<th>Areas of use</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mean (s.d.)</td>
<td>Mean (s.d.)</td>
</tr>
<tr>
<td>Clients.............</td>
<td>3.14 (1.68)</td>
</tr>
<tr>
<td>Programmers........</td>
<td>8.00 (3.74)</td>
</tr>
</tbody>
</table>

Table 4.1. Computing experience of the subjects used.
4.4.2. Tasks

In designing the study, one could either have let the subjects choose tasks that they really would have liked to have seen computerised or to have specified what tasks they were to attempt to communicate. In the former case, there would have been a considerable burden on the subject to come up with a reasonable possibility and minimal control over the sort of item chosen. This could have resulted in a number of false starts due to them choosing inappropriate tasks as well as possibly excessive variation in the domain of task. The alternative of specifying a task for communication could lead to the criticism of the study as artificial because the clients had not generated the tasks out of real needs. In order to attempt to satisfy both practical considerations and the need for validity, descriptions of tasks were given to the subjects with minimal detail and they were given 15 minutes to flesh them out to meet their own particular needs or ideas about how they saw the task being performed. The tasks were chosen on the basis of their being relevant to the subjects' situation and such that they could easily imagine wanting to have a computer support them. Table 4.2 lists the tasks used for each pair (see Appendix I for a more detailed description of the tasks).

<table>
<thead>
<tr>
<th>Session Number</th>
<th>Task</th>
<th>Time Taken (mins.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Diary</td>
<td>30</td>
</tr>
<tr>
<td>2</td>
<td>Diary</td>
<td>60</td>
</tr>
<tr>
<td>3</td>
<td>Diary</td>
<td>27</td>
</tr>
<tr>
<td>4</td>
<td>Reading Expt*</td>
<td>38</td>
</tr>
<tr>
<td>5</td>
<td>Reading Expt*</td>
<td>12</td>
</tr>
<tr>
<td>6</td>
<td>Reading Expt*</td>
<td>27</td>
</tr>
<tr>
<td>7</td>
<td>Colour Blindness Expt</td>
<td>11</td>
</tr>
<tr>
<td>8</td>
<td>Auditory Perception Expt</td>
<td>14</td>
</tr>
</tbody>
</table>

* = Experiment subjects have actually run

Table 4.2. Programs to be specified and length of dialogues.

4.4.3. Procedure

The sessions were scheduled so that the client would arrive 15 minutes before the programmer. On the clients' arrival they were given written instructions which consisted of general instructions about their role in the role-play plus the description of a task. They were told to read the instructions, to make whatever notes they wanted and to ask the experimenter if there was anything that was unclear. On the arrival of the programmer, the experimenter introduced the two subjects. The programmer was also
given instructions about the role-play which did not give any indication about the nature of the task. The dialogue would then commence. It was allowed to run for up to one hour unless the programmer ended it before then. Table 4.2 also shows the length of the dialogues. An audio recording was made of the dialogues and transcripts made without using any formal methods. After the session, the subjects were given questionnaires about themselves.

4.4.4. Data Analysis

The data obtained by this study did not lend themselves to simple quantitative analyses. What was required was to transform the transcripts, etc., into formats that would allow the extraction of patterns or trends. This involved the experimenter analysing the transcripts informally and identifying clues to patterns that seemed to exist within them. Having done that, it was necessary to construct tools that would enable the patterns to be described formally, their existence to be established in a quantitative manner, and their nature to be communicated. Two main tools were used for these purposes.

The first of these was a taxonomy of all the speech acts in the transcripts. This was devised to distinguish the important actions of the programmers and to facilitate the identification of strings of related requirements from the clients. The taxonomy is outlined in Figure 4.2 and described in more detail in Appendix I. To use the taxonomy, the experimenter had to divide the transcripts into speech act units and then classify each. Subjectivity was clearly involved in these processes. The extent of this was assessed by getting someone not actually involved in the study to categorise a section from one of the transcripts. This was compared with the categorisation of the experimenter. The agreement between these two was only 52.6%. The main sources of disagreement (accounting for 54.6% of the disagreements) lay in the speech acts being interpreted as feedback or as question asking (so that responses to these become responses to feedback or answers). This suggests that any distinctions based on this will be very subjective and should be treated with caution. The other area of uncertainty seemed to be the distinction between the programmer giving information (PI4's) and giving feedback that added information (PF5's). The analyses resulted in large computer files consisting of a series of units of the form "CQ1" for a closed question from the client or "PF3" for some feedback from the programmer. A number of SNOBOL4 programs were then written to count the occurrence of the different items and to extract information about the patterns that existed within these lists (examples of these programs are given in Appendix I). The other tool for analysing the nature of the dialogues was the extraction from the transcripts of the information that was being
communicated explicitly. This resulted in a list of information units, each of which could be categorised in a number of ways, as described in the next section.

<table>
<thead>
<tr>
<th>Requests for Information (mainly questions, but could include other grammatical forms achieving the same goal, directly). Subdivided into:</th>
</tr>
</thead>
<tbody>
<tr>
<td>* Q1 = Yes/No answer required (&quot;Do...?&quot; etc.)</td>
</tr>
<tr>
<td>* Q2 = Option selection (&quot;Which ...?&quot; etc.)</td>
</tr>
<tr>
<td>* Q3 = Information generation (&quot;What ...?&quot; etc.)</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Answers (information given in direct response to these.)</th>
</tr>
</thead>
<tbody>
<tr>
<td>* F1 = Basic Responses (&quot;Uhuh.&quot; etc.)</td>
</tr>
<tr>
<td>* F2 = Repeating back immediately what is understood.</td>
</tr>
<tr>
<td>* F3 = Summarising.</td>
</tr>
<tr>
<td>* F4 = References to past information used to make a point.</td>
</tr>
<tr>
<td>* F5 = Adding information (eg. offering an implementation).</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Feedback (information given by the programmer consisting either of repeating back what the client had said, adding to it, or making suggestions about implementation.) Subdivided into:</th>
</tr>
</thead>
<tbody>
<tr>
<td>* F1 = Basic Responses (&quot;Uhuh.&quot; etc.)</td>
</tr>
<tr>
<td>* F2 = Repeating back immediately what is understood.</td>
</tr>
<tr>
<td>* F3 = Summarising.</td>
</tr>
<tr>
<td>* F4 = References to past information used to make a point.</td>
</tr>
<tr>
<td>* F5 = Adding information (eg. offering an implementation).</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Other Information Giving (either information volunteered by the client or more factual information from the programmer). Subdivided into:</th>
</tr>
</thead>
<tbody>
<tr>
<td>* I1 = Occurring immediately after an answer, relating to it.</td>
</tr>
<tr>
<td>* I2 = Information relating to the last piece of feedback.</td>
</tr>
<tr>
<td>* I3 = Spontaneous (not related to recent utterances in a direct way).</td>
</tr>
<tr>
<td>* I4 = Giving more, related to recent information.</td>
</tr>
<tr>
<td>* I5 = Giving information to help with a question.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Meta-comments (controlling or about the dialogue).</th>
</tr>
</thead>
</table>

**Figure 4.2. The taxonomy of speech acts used.**

### 4.5. The Observations and Discussion

#### 4.5.1. Introduction

The dialogues observed proved extremely diverse and a cursory investigation did not provide any clear evidence of patterns. The general nature of the transcripts can best be communicated by means of a short extract from one (Figure 4.3). For the most part, this discussion of the dialogues will make only indirect references to the raw data and will concentrate on the various techniques developed for representing the transcripts in more manageable ways.
1) Client: Ok. For starters I want the computer to always give me this type of information. Ok. A day, the actual name of the day, the date, the time and lectures to be attended for that day.

2) Programmer: So is the timetable the main function of the tasks that you're going to do? Ok. So your lectures for that day, that's the important thing, yes?

3) Client: Ok. Right, I want it to compute how many hours I've got free, my leisure time, okay, and how many hours I've got in which to attend lectures. And I want it to beep every time I have to attend a lecture, like 5 minutes before I have to be there to remind me to go.

4) Programmer: So you want an alarm really.

5) Client: Yes, an alarm.

...
This observation was expressed more formally by means of the taxonomy of speech acts. Acts where the programmer is describing a solution or implementation of the program were identified as type PF5. These were found to be concentrated towards the end of the dialogues rather than appearing regularly throughout. Thus, when the dialogues were divided into two halves, there would always be more PF5's in the second half, with the ratio of second to first being at least 1.5 (see Table 4.3). The frequency of occurrence of PF5's was, in fact, found to vary very greatly comprising from 5% to 36% of the total number of speech acts. The observation that the programmer only provided design solutions later in the dialogue was also supported by the categorisation of the dialogues into information units. These were labelled according to the subject who generated them. In plotting these it was found that, once again, there was a shift through the course of the dialogues. Initially, information was mainly from the client. Later, however, a lot of the information would come from the programmer, presumably suggesting possible implementation details. As with the suggesting of solutions, there was a lot of variation in this. In some cases, the programmer would give a lot of information, while in others very little. Figure 4.4 shows graphs of this.

<table>
<thead>
<tr>
<th>Session</th>
<th>First Half</th>
<th>Second Half</th>
<th>Ratio of Second to First</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>7</td>
<td>52</td>
<td>7.43</td>
</tr>
<tr>
<td>2</td>
<td>0.5</td>
<td>6.5</td>
<td>13</td>
</tr>
<tr>
<td>3</td>
<td>20</td>
<td>35</td>
<td>1.75</td>
</tr>
<tr>
<td>4</td>
<td>27.5</td>
<td>41.5</td>
<td>1.51</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>7</td>
<td>21</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>3</td>
<td>8</td>
<td>2.67</td>
</tr>
<tr>
<td>8</td>
<td>5</td>
<td>16</td>
<td>3.2</td>
</tr>
</tbody>
</table>

Table 4.3. Number of PF5's that occurred in first and second halves of dialogues, and their ratios.

The lists of information units communicated were also used to construct graphs of the course of the dialogues (see Figure 4.5 for an example of such a graph). To generate these graphs the program is broken into parts and the items of information communicated allocated to the different parts. The dialogue can then be plotted as changes in focus of attention between the parts. In addition to this, the number of items of information communicated when focussing on a given part is represented numerically (no number indicates just one item) and by the size of box. Finally, the
Figure 4.4. Graphs created by plotting proportion of F4's that came from the programmer (i.e., PF4's) out of successive groups of 15 F4's. Thus, of the first 15 F4's, only 1 may have been a PF4 and 14 have been CF4's. Later groups showed a higher proportion coming from the programmer.
subject who initiated the change of topic in each case is represented by the box shape (circle for client, square for programmer). Another aspect represented on these graphs is the initiator of the change of topic for each group of communications about the topic. This provides a measure of who was in control at any one time. It was generally found that the Client would be in control early on in the dialogue but that there would come a stage where the Programmer would appear more in control later on. The strength of this effect varied greatly between dialogues as the Programmer in the different cases would take control to varying extents. The dialogue represented in Figure 4.5 showed this effect particularly strongly.

Most of the shifts described so far refer to the changing roles of the programmer and the client. There was another shift that was observed earlier in the dialogues that referred to the way the clients generated their requirements.

The information units that came from the client were further allocated into two groups according to whether the requirement could be said to come from the notes that had been made before the dialogues. Once again, it was found that use of the notes varied during the course of the dialogue and, in all cases, would decrease until they were only being used sporadically. Graphs of the dialogues represented in this way are shown in Figure 4.6. This suggests an important distinction that exists in the information that is being given by the client. While some of it may well have been prepared beforehand, with some degree of thought, other elements are generated in response to questions or feedback from the programmer and may, therefore, be based on a much less substantial consideration of what is needed.

In summary, then, the dialogues appeared to start with a phase where the client was in control, generating requirements from notes with the programmer offering little feedback apart from indications of understanding and paraphrasing some of what had been said. Later in the dialogues, however, the programmer would be more in control, the information flow would start to come from the programmer who would be suggesting solutions, etc., and the information from the client would be more opportunistic and less from the notes.

In relation to these two phases it seems relevant to note that on a number of occasions the dialogues ended with an agreement to meet again after the programmer had constructed a prototype program. It was clearly intended that this would be used to generate further requirements and clarify remaining misunderstandings. Thus, the
Figure 4.5. A representation of session number 3 using one type of diagram. See text for explanation.

All the graphs for the other dialogues are given in Appendix I.
dialogues observed were seen as only consisting of a first step in the requirements elicitation process.

Given the existence of these two phases, there are likely to be quite different observations associated with them. This proved to be correct. The object of interest in the first phase was the specification generated by the client, as this was with little prompting from the programmer. In the second phase, however, it was more important to consider the specification elicitation techniques being employed by the programmer and how they related to the further requirements communicated. The observations made about these are described in the next two sections.

![Graphs of dialogues showing the decreasing use of notes in the requirements specification.](image)

Thus, the generally higher values obtained at the left hand side of the graphs indicates that a large proportion of the information items communicated by the clients are coming from their notes. Only four of the dialogues had the clients giving sufficient numbers of information units to generate such graphs.
4.5.3. Phase One: Requirements Specification as the Suggestion of a Design Solution

The main observation about the specifications generated by the clients was that they would tend to be in the form of descriptions of programs that would meet their needs. This and other, related observations are discussed in the following sections.

4.5.3.1. Clients Suggest Program Designs

The fact that clients were frequently presenting their needs in terms of descriptions of programs that would meet them was noticed in two ways. The first was in comparing the dialogues of this study with the observations of Carroll et al. (79). The second was in categorising the information the clients communicated.

1) Goal Statement.
2) Goal Elaboration.
3) (Sub-)solution Outline.
4) (Sub-)solution Elaboration.
5) (Sub-)solution Explication.
6) Agreement on (sub-)solution.

Figure 4.7. The six states used by Carroll et al. (80).

Carroll et al. broke their dialogues down into units of interaction that were then classified into one of six "states". These states are shown in Figure 4.7. In order to compare the dialogues observed in this study with those described by Carroll et al., they were analysed using these categories, as far as possible. This was found to be rather difficult and subjective as the goal descriptions and goal elaborations blended into one another, as did the speech acts describing the solutions. This is illustrated by an extract from a dialogue in Figure 4.8.

What also became apparent when trying to categorise utterances was that the distinction between a statement of the goal and a statement of a solution could be very unclear. This was because clients' descriptions of what they wanted did not consist solely of abstract goals but would often include a lot of concrete information about how they wanted it implemented. This would not be complete, however, in the sense of providing a full specification and the programmer's role would often be that of going into the solution proposed by the client in more detail as well as considering aspects of how it would be implemented.
37) Client: And also it's got to sort of allow for the possibility of when the person types in the response they get it wrong and make a mistake.

(Turn 37) shows the Client stating a problem (state 1).}

38) Programmer: Yes. But you merely want, or you're talking about, that they know what they want to type in but they mistype it. Yes, I think, well there are ways of retrieving that kind of error but it would mean one of two things. It would be relatively easy to retrieve that type of error if the subject realised he'd made an error before he hit the "Return" key, or something like that. It would be more difficult otherwise because it would mean that you would have to interrogate the subject via the screen. You have to put up some sort of statement or question saying, "Was that what you intended to type? yes/no" which would make it rather cumbersome. But then what are you trying to do, if you're trying to measure the correctness of responses against the number of error responses then of course that people make errors of that kind might not matter. No it wouldn't matter. It is a different sort of error, isn't it?

39) Client: Yes.

(Turns 38) and 39) seem to involve a dialogue about details of the nature of the problem (state 2).}

40) Programmer: I can't see offhand any quick way of actually asking if they think they got it right.

41) Client: Okay, I suppose I'd better just allow for it. Or actually as I don't want the computer to process the data so as I'm actually looking at the data I should be able to see obvious mistakes.

(Turns 40) and 41) involve the Programmer claiming not to have a solution and the Client suggesting two that he might be able to use (state 3?).}

42) Programmer: There is one way we might tackle that problem, just to think about it, and that is perhaps if one could tell the subject that he could use a particular symbol, like perhaps just letter "e" or something like that to indicate an error and then type what he intended. I think probably an alphabetical letter would be a bad choice because that could easily be part of a response word but there are other symbols on the machine like exclamation mark for example, so perhaps, for example, if he wanted to type "du" and he was typing "tu" instead, he could type "tu", which is the mistake, and then "oh, dear, I've made a mistake." "du" and then we could program the computer to filter out all the symbols occurring an exclamation mark before the material is put on disk.

(In turn 42) the Programmer finds a solution and offers it. Exactly how to divide this into states 3, 4, and 5 is unclear and, indeed, the process seems rather more one of the Programmer constructing the solution and testing it as well as communicating it to the Client.}

43) Client: Yeah. If he decides he gets a question wrong, he could actually have the computer again but I don't suppose there is any way you could actually do that with a tape recorder, get them to actually play the sentence again. Although, no, I wouldn't want that anyway because I don't want the person to have two chances of listening to the sentence. Yeah, so, okay, I just want them to type in the response again.

(In turn 43) the Client tests out another idea but ends up accepting the Programmer's solution.}

Figure 4.8. An example of a piece of dialogue (from session number 8) where Carroll's states 3), 4) and 5) are not clearly distinguished. The different parts of the dialogue are commented on by the text in ( ) brackets.

Thus, for example, clients discussing experiments would describe the sequence of interactions between the imagined subject and the computer, but would often leave many details open. The programmers would then interrogate the client about these details, as they became apparent as ambiguous. Figure 4.9 shows an extract from a
dialogue where the programmer is merely obtaining a description of what is wanted, rather than clarifying the problem in order to suggest a solution.

The corollary to this observation was, therefore, that both clients and programmers could talk about both goals and solutions. This is in contrast to Carroll et al.'s observations of the client talking solely about goals and the designer talking about solutions.

28) Programmer: I suppose I ought to ask you about the speed at which the data is presented to the subject and the speed at which they're expected to respond.

29) Client: Um, right. Their sentence is played then it carries on at the user's pace.

30) Programmer: Right.

31) Client: So when he's typed something in, press "Return", carries on.

32) Programmer: Yes, there's no problem there.

Figure 4.9. An example of a piece of dialogue from the same session as Figure 4.8, where the programmer is only obtaining a detailed description, rather than attempting to construct a solution or implementation.

A further analysis was performed on the lists of information units communicated. This involved their being categorised using the classification shown in Figure 4.10. Having categorised the dialogues in this way, it was found that type (3) information units made up 82.8% of the total, on average (minimum was 69.2%). References to goals or requirements that did not explicitly suggest how the program was to run were rare. Items of type (1) (comprising between 1.52% and 17.1%) were, for the most part used in conjunction with an item of type (3) to justify it. Type (2) items comprised between 0% and 4.76% and type (4) between 0% and 21.2%. The main point about type (4) items is the distinction between those aspects of the program that are apparent to the user and those that are not. These were clearly not used to any great extent, by either client or programmer. It was possible that the programmers were "mirroring" the way the clients were describing the programs, using the same information types as the clients used. Evidence for this would have been found if dialogues where the client used a high proportion of information type (3) had programmers also using lots of type (3). This hypothesis was not supported by the data obtained however (correlation of proportion of type (3) units generated by programmer and by client was -0.151 - not significant).
1) Goals, reasons, requirements in the abstract sense, comments on importance of design elements.
2) Overviews of the program.
3) Descriptions of how the program should work, including what one should be able to do, what is to be visible on the screen, what the user is to do to make it work, etc.
4) Descriptions of the data structures, underlying processes or aspects of the actual implementation.
5) Speculations about how the program might be used in reality, given the design being considered and other evaluative comments.
6) Background information (eg. explanation of terms) to help communication.
7) Meta-information about the dialogue process.

Figure 4.10. Taxonomy of information items communicated.

Aim - Diary on Computer

Meetings )
Appointments )  DAY + TIME
Deadlines )
etc. )

PROGRAMME with a number of files.
FILE Split into no. of files - each file is a month.
Name files - J.F.M. name f1 Jan f2 - birth. f3 etc.
Month at top of file
Each file into Columns - week 1, 2, 3,
Each Column into a chunk Mon Cl M4 (Monday first week)
Tues Cl F4 (Tues. first week)

Day split into hours.
009 - 0010 Week - no times just
at beginning - day tasks to remember capitals
Deadlines in Red.
App. + Meetings in white
Open to anyone to put things in
Rcd - record info.
Rpt—ra
Desc. - Describe info

Give programme own name A. Name therefore easy for people to contact you.
Check
Also file for birthdays/addresses/shopping lists

Figure 4.11. An example of the notes made by the client in session number 1 before the dialogue
(transcribed as faithfully as possible).
The final piece of evidence in this respect was the form of the notes made in advance by the client. An example of these is given in Figure 4.11. These clearly consisted of descriptions, at a concrete level, of the program that the clients required, focussing on the aspects that would be most immediately salient, i.e., the interface. There are only occasional references to the underlying data management processes or requirements.

From this description of the dialogues, it is clear that the process was one of the client communicating the superficial (interface) elements of an imaginary program to the programmer. Having established this as the basic quality of the specifications obtained, other aspects are now described that contribute towards a fuller picture.

4.5.3.2. Detail of the Specification

Rich and Waters (88a) make the point that an informal requirements specification cannot be evaluated as to its completeness because the level of detail that it achieves is arbitrary and there are always going to be aspects that have not been specified. While it may not be meaningful to examine the completeness of the specification, it seems relevant to consider whether a consistent level of detail was being reached across the different elements of the programs specified.

What is clear from the graphs of the dialogues by topic changes is that there is very little consistency both between sessions and within a session, between elements of the program being specified and the amount of information given about each. See Figure 4.12 for an example of such a graph showing the total number of items about each aspect along the right hand side. Clearly, some parts are focussed on in depth while others are only given a cursory definition. This was a general problem with the interpretation that clients were describing programs that would meet their needs. The actual level of description of these programs varied enormously and no general statement can really be made about this. In some cases, the description was of little more than the general features that would be wanted, while in others, certain of the functions would be described in great detail including reference to the underlying data structures. This variation in the depth to which different parts of the program are specified was true for all the dialogues (see other diagrams in Appendix D).

4.5.3.3. Implications of this Approach for the Programmer

The fact that the client was describing a program rather than communicating needs had four major implications for the task of the programmer. The first relates to the fact that the program was being described without any organising framework, the second to the
Figure 4.12. Graph of session number 6 showing the total number of information items communicated about each part of the program discussed.
variety of levels of detail, the third to the fact that the client may be using a faulty set of concepts in generating and communicating the program, and the fourth to the order in which it is described.

There was no initial agreement about what sorts of structures were available or any sort of framework into which the desired program was to be fitted. This meant that the clients were generating their own structures to suit their needs and according to their experience with computers. The clients' descriptions of the desired programs would include references to these structures and the relations between the elements but these would often not be clearly articulated from the start. The dialogues are characterised, therefore, by the programmer having to accept information both about the details of each event or screen and about how they relate to one another, with no prior framework into which the information can be fitted. This has two effects on the programmer's behaviour. The first is that a significant proportion of the dialogue involves obtaining explicit descriptions of the relations required and the second is that the programmers' representation of these must be extremely fluid and open to continual change. This can be seen in the notes made by the programmers which often consisted of a series of structural representations of the program, either as flowcharts or as screens with arrows, etc., relating them, with numerous alterations. This would appear to be a major problem for programmers and computers in this role.

The programmers must be able to cope with the many different ways in which the clients describe the program desired. This includes different levels of detail and degrees of abstraction from a concrete description. This requires the programmers to fill in the implicitly left gaps in the description themselves. If they cannot do this, they may well ask for more detail, or for a fuller explanation of the reason for wanting the feature.

The second problem, that exacerbates the first, is that the client cannot be assumed to be employing a sound set of programming concepts. Thus, in some of the dialogues, the client appeared to be describing a program in terms of a certain set of structures and the programmer was able to accept this and understand it, but then it would become apparent that what was being understood was at variance to what the client has in mind. This could represent quite fundamental misconceptions stemming from the misuse of computer-related terms.

An example from one dialogue illustrates this (see Figure 4.13). In this, the client describes the desired diary as a number of "files". When this was initially interpreted,
it was assumed that this was a reference to the underlying data structure that was required. It rapidly became apparent, however, that this was not the case. This was because the client continued to describe the files in terms of orientation on screens, etc. What had appeared to be a description of an underlying structure was really nothing more than a description of the interface using terms that were inappropriate. This illustrates the fact that, if the clients are communicating their requirements by describing imagined programs that will meet them they must rely on their experience and knowledge of computers, of how programs can work and of what programs can do. If this is in any way faulty or lacks an awareness of a potentially useful facility, then a programmer who just accepts the client's description and creates a program around it will not be providing a good service. However, while a human programmer may be able to suggest that the client's design could be improved upon, it is harder for a computer to do this without putting a considerable strain on the user-computer relationship.

---

13) Client: I could have a number of files and say file 1 is split into different months and might give a general overview of all the things that I have to do... just a long list.

14) Programmer: yes.

15) Client: Complete... so I can just scan through it. Then if I split each month into a column of a week, say column 1 for week 1, column 2 for week 2, etc., and then each column into a chunk of a day so that I could put "month, Monday, cl" so that that would be Monday of the first week.

16) Programmer: Each column... into days.

17) Client: yeah

18) Programmer: Running vertically.

19) Client: yes with... um... on the left hand side I'd have all the times from 9 to 10, 10 to 11, throughout... um... up to about 8 or 9.

...
their own terms and in whatever order suited them best. In only one occasion did the programmer attempt to direct the dialogue by asking a series of questions and this was abandoned after the questions being asked were found to be unhelpful.

This resulted in a situation where the client would initially describe the program in a fairly structured manner, from the notes made. However, later in the dialogue, this order disappeared as information was offered in a more opportunistic way, as it was found to be required. On occasion this resulted in statements about fundamental or important aspects being left until late in the dialogues because the client had not realised that they needed stating. The programmers had further to adapt their understanding of the solutions to cope with these occurrences, therefore.

While most dialogues consisted of a series of groups of information units that related to a single aspect of the program being specified, there would also be a certain amount of apparently opportunistic jumping between such aspects. This was apparent in the graphs showing topic changes. Thus, it would appear that both the client and the programmer were initiating information communication in a manner that was ordered primarily by association rather than in a planned attempt to be thorough.

4.5.3.4 Summary

Having established the nature of the specifications observed, one can draw up a less anecdotal list of their main qualities that can be compared with that given in section 4.2.1.1. They were found to:

- Consist of program solution descriptions;
- Be inconsistent in their completeness and level of detail;
- Be poorly structured and planned, relying heavily on the programmer being able to work with them in an extremely fluid manner;
- Rely on the clients' having a good understanding of computer concepts.

The observations made suggest something about the process of generating a specification. It is clearly based heavily on existing knowledge of computers and ideas about what is possible. This must combine with ideas about needs to generate an imaginary program that will supply them. The initial period of preparing the ideas for this may well represent a common phase in such situations, resulting in some of the specifications deriving from some degree of planning while others are more opportunistic. This latter aspect must be inevitable as the client cannot be expected to
anticipate all the designers' requirements for detail (a point made by Eason, 80). This is admittedly sketchy and a more detailed model of how clients generate their requirements would represent an important basis for any further work in this area.

The other main area of interest was found to be the manner in which the programmer elicited requirements from the client in the second phase of the dialogues. This is discussed in the next section.

4.5.4. Phase Two: Use of Feedback and Questioning

There are three main areas of interest in the way the second phase was managed by the programmer. The first is in the overall structuring of it, the second in the manner in which the different elements of the program were dealt with, and the third in the way questions and feedback were used.

This phase of the dialogues seemed to be even less structured than the first. In some cases, the programmer would control the dialogue such that there would be a clear, thorough progression through the different elements of the program. In others, however, this progression would appear much more random and guided by associations and opportunism. This can be seen to some extent in the dialogue diagrams already shown. Attempts to represent these parts of the dialogues in a way that allows one to make simple statements about their form proved impossible. This is partly because there was very little obvious gross commonality between subject pairs and partly because of the great variety of events and patterns that could be found in the dialogues.

Not only was there a great deal of variety in the way the second phase was organised by the programmer, the actual manner of examining each aspect focussed on was found to vary considerably. The programmer could be obtaining a more detailed description of the aspect or else considering problems identified in how it could be implemented. In the former case, feedback and questioning would be used to obtain specific items of information or to test possible approaches for acceptability. Similarly, in the latter case, the programmer would combine eliciting more detailed descriptions of the requirements with the proposal of solutions and commentary on their advantages and disadvantages. In this way, the dialogues could take on a form similar to that described by Carroll et al. (80).

Most of the information given by the clients in the second phase of the dialogues was in response to the programmer either asking questions or offering feedback. This
suggests that it is important to understand the processes that contribute to this. Data from this study provided a basis for such work but do not offer any answers. The data obtained are described in this section. Before describing this data it should be remembered that the distinction between questions and feedback appeared most open to subjectivity when the transcripts were translated in terms of the taxonomy of speech acts. Because of this, no attempt was made to compare the observations associated with these two behaviours as they may well not represent distinct activities but points on a continuum.

Questions were categorised by their "openness". This related to the control exerted over the information that could be said to represent an answer to the question. The breakdown was into Q1's which were "yes/no" questions, Q2's which offered alternatives from which the client had to choose, and Q3's which were totally open. For each dialogue, the proportion that each type made up of the total questions was found and these values averaged between dialogues. This data is summarised in Table 4.4. Clearly, "yes/no" questions were used most frequently and seven of the eight programmers used them for more than 50% of their questions.

<table>
<thead>
<tr>
<th>Speech Act Label</th>
<th>Type of Question</th>
<th>Percentage of Total Questions of this type</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1</td>
<td>Yes/No</td>
<td>61.6</td>
</tr>
<tr>
<td>Q2</td>
<td>Offering Alternatives</td>
<td>10.2</td>
</tr>
<tr>
<td>Q3</td>
<td>Completely Open</td>
<td>28.2</td>
</tr>
</tbody>
</table>

Table 4.4. The relative proportions of different question types used by the programmers.

While questioning varied in how open it was, feedback would always be very open. When a programmer offers a client a piece of feedback, there is no boundary set on the responses that the client can make to it. In this sense, therefore, feedback represents the next most open type of question after Q3's. As described above, however, it can also be seen as similar to Q1's. Little in the way of patterns was found in different types of feedback.

The observations about questioning behaviour is in contrast to recommendations by Parkin (80) for the analyst to offer the client alternatives (i.e., to use Q2's). It may be that Programmers in this study were not motivated to expend the effort involved in generating lists of alternatives. An alternative reason for the use of totally "closed" questions in this way would be that they provide good support for the Client but also work much like offering feedback in that they may well generate conflict. This should
then lead to more information being elicited. This behaviour is, therefore, similar to that observed by Falzon, Amalberti & Carbonell (85). This led to an analysis of the observed effect of questioning and giving feedback on requirements generation by the Clients.

Questions were clearly more likely to generate at least one item of information than feedback, simply because the client was likely to provide an answer. Giving feedback did not guarantee any information and of the 632 times feedback was given, only 62 resulted in any response at all. This is not without value, however, as it is telling the programmer that the client has no objection to what is being fed back. For both questions and feedback, it was possible to identify the number of information units that were directly stimulated by each occurrence. This enabled the generation of tables of frequencies with which strings of information units of different lengths followed questions and feedback. On 42 occasions, the client produced strings of three or more units, but this represents a very small proportion of the occurrences of questions and feedback. It was, of course, possible that other information also resulted from the feedback but that the flow was interrupted by the programmer in some way, breaking the pattern and making it impossible for the SNOBOL4 programs to recognise it.

This data would appear to suggest that the stimulation of information by use of feedback is limited in its effectiveness and supports the importance of actually providing the client with a better indication of what is being proposed than a verbal description. In retrospect, it would have been interesting to have obtained data about the beliefs the clients and programmers had about the program that was finally agreed upon so that misconceptions could have been identified. It would then be possible to locate the failures of the use of feedback and questioning in accurate communication. This must remain an area for further research.

4.5.4.1. Summary

In summary, the qualities of Phase Two were much harder to characterise. This was partly because of the large amount of variation that was observed and partly because so much of it seemed to rely on opportunism and association. Thus, the value of questions and feedback would often relate to their ability to stimulate further information, but this appears to be a very poorly understood process, requiring further study.
Before taking the observations made further and examining their implications, it is important to make some evaluation of how much external validity this study can be said to have. This next section looks at this.

4.5.5. Validity of the Study

The use of role-plays must always carry weaknesses in terms of the extent to which conclusions drawn can be generalised to the real world. Certainly, if one is intending to obtain hard data about phenomena, then any role-play must be backed up by some observations of real situations. It may be, however, that role-plays allow greater control over the scenarios and greater scope for detailed analysis of observations made. This was found to be the case in this study.

Two points need to be made about this study. The first is to consider the extent to which the observations are supported by or in contradiction to those made by others. It would appear that, for the most part, they are supported by the anecdotal material presented earlier in this chapter, but that they are more detailed than any of them. In particular, the observation that clients communicate their problems by describing design solutions was supported by other work. This is important as there may have been some aspect of the instructions that led the clients to do this (cf. Gould, Lewis & Becker, 76).

The main contradiction between this study and any other is in the way the programmers’ offering of solutions differs from that observed by Carroll et al. (79). It would seem useful to compare these studies and to consider the factors that might have contributed to this difference. One important factor may be the size of the problem being dealt with. The study by Carroll et al. involved a major problem of computerising a library. It may be that this would be too large to consider as a whole so that the strategy of generating solutions to the different parts is more manageable. This would explain their being closer to the observations of Falzon et al. (85) & Coombs & Alty (80). Developing these sorts of taxonomies that will determine how best to structure the overall dialogue will represent an important issue in the design of JEEVES-like computers.

The second point in relation to the issue of validity refers to the goals of the study and the claims being made. The goal of this study was not to obtain definitive data about the nature of programmer-client dialogues. Rather the intention was to identify the main elements that make them up. The observations made could then be used as clues about the important processes that must be understood if such dialogues are to occur.
between users and computers. Given this goal, there is no intention to make claims about the data obtained as being generalisable to actual practice. It may be that certain of the observations are generalisable and that the work has identified some useful techniques for analysing real-life dialogues, but this is not the main aim.

As it is, this study has provided a basis for considering how one might expect a JEEVES-like computer to function and for structuring further work in identifying the central issues involved in this area. The next sections discuss these matters and conclude this chapter, laying a solid foundation for the next chapters that look at the main areas identified in more detail.

4.6. The Search for a Way Forward

The study just described has generated some useful observations combined with a general framework for understanding the process of requirements elicitation. It has also opened up a large number of issues and offers only a suggestion of how they might be resolved. This results in a choice of paths that can now be taken.

One possibility would be to perform further studies of this type, looking at requirements specification dialogues, possibly in real life, and taking them right through from start to finish. The questions that could not be answered by the study described could be addressed in such a study and a better understanding of the mechanics of these dialogues obtained. The resulting observations and conclusions would also have greater external validity. Alternatively, the studies could be repeated in a more controlled manner with different data collected to supplement the transcripts. Manipulations could be performed on the program specifications and the instructions given to observe their effects. Both of these options would generate further valuable contributions to the area of requirements elicitation dialogues between humans.

They would not, however, necessarily take us very much further in the search for issues relating to intelligent front ends for programming systems. It is important, at this stage, to use the results of this study to provide a scaffolding for further building rather than as an end in themselves. Further understanding of dialogues of this type between humans, while valuable, would not provide the focussed perspective required. The next step must be to use the observations made and the insights obtained to propose a structure for our understanding of the tools in which we are interested. This will enable the identification of critical issues that will need to be addressed if tools like JEEVES are to be built.
This section takes this step. It starts with a necessary consideration of what any tool of this type must provide and an evaluation of some of the overall design issues that will underpin any work that can be done to locate the central human-computer interaction issues. Following that, a structure is described, derived from the study described, that will provide the required scaffolding for thinking about what needs to be looked into in more detail. This is then used to make a tentative suggestion of where the important issues must lie.

4.7. How Might One Expect JEEVES to Work?

In order to make it possible to use the idea of JEEVES as a basis for thinking about the issues involved in automatic programming by informal requirements specification, it is helpful to clarify some of the main elements that will be involved and to consider something about its nature. These consist of a consideration of the nature of the dialogue, in particular how the user communicates a specification, a basic notion of how specifications can be turned into programs, and an assessment of the question of how general-purpose the computer will be able to be.

4.7.1. Dialogue Management

It has been argued that JEEVES must be able to accept a specification for a program and implement this without the user having to know in detail what implementation facilities are available. Thus, the user must be able to specify what is wanted without having to acquire a fixed vocabulary that will relate to available constructs. This means that the computer may have to accept inputs of a "Natural Language" nature. The nature of this must, however, be carefully investigated if the dialogue is to be successful.

In addition to the basic nature of the inputs from the user, it will be important to decide the best overall structure and locus of control over the dialogue. The disordered nature of the specification that can be expected and the variation found in how programmers controlled the dialogues observed, suggests that a number of solutions will need to be considered. Workers looking into similar systems such as Intelligent Front Ends for complex applications have identified a number of possible approaches to these problems (eg. Bundy, 85, Gershman and Wolf, 86, Morris, 87). These are based on very little understanding of the individual processes that go into the various activities that are being marshalled. It would seem to be more urgent, therefore, to arrive first at a firm basic articulation of these.
4.7.2. Libraries of Program Cliches

A number of good ideas are described for going from the users specification to the desired program. The most promising is by the use of domain "cliches" (Rich and Waters, 88a, 88b). These consist of code fragments of varying sizes that represent very commonly used units that the user is likely to want. Techniques for moving from a specification to a program based on available cliches would probably make use of an idea proposed by Falzon, Amalberti & Carbonell (85). The computer makes the assumption that the specification is maximally stereotypical and attempts to match it with as large a cliche as possible. If this fails, or if adaptations are needed, then more complex processing of the specification occurs to perform the necessary adjustments or construction.

4.7.3. Special Purpose vs General Purpose

The final issue that must be addressed about how any tool that supports this kind of automatic programming is the question of how general it can be. This refers to the breadth of different kinds of program that can be generated by the tool. While it would be ideal if it could support a wide variety of program construction, there are a number of reasons for believing that this might not be possible. These are discussed in the following sections.

Effective communication between the users and the tool must be one of the most important criteria for success in the sort of tool that is being envisaged. Rich and Waters (88a) and Balzer (79) make the point that communication relies on both parties having some understanding of domain of discussion and the ability to understand the specification. In the absence of this, all the concepts used in communication must be explained in more general terms and this would require the user to work at a very low level. This is similar to the points arrived at in the investigation of programming languages where general languages would either have to provide a massive vocabulary or force the user to program in terms of very general concepts. Barr and Feigenbaum (82) and Rich and Waters (88a) also point out that the likely nature of the specification, consisting of partial information and inconsistency, will require the computer to have the ability to fill gaps and interpret ambiguity. This likely to involve the use of knowledge that is very specific to the domain of the program being specified. The final argument for the need for the computer to have domain-specific knowledge and communicative abilities stems from the need for it to provide explanations to the user. Once again, in the absence of these, the explanations will have to be couched in terms that are too general or abstract to be meaningful.
Barstow (83) and Balzer (79) also suggest that efficient and effective identification of programs that will meet the specification will require domain knowledge. In particular, it will be important to reduce the space of possible programs that might be appropriate so that a workable subset can be analysed in more detail.

Finally, in the interests of making the system manageable, the fact that it should be provided with program cliches at a high level of complexity, such as complete, stereotypical programs will limit the scope of programs that can be offered by a system with finite memory.

4.7.4. Summary

The main reason for making any JEEVES-like computer domain specific is to limit the demands on memory by reducing the amount of domain specific knowledge and cliches that will be required. Unlike a traditional programming system, increasing the scope to a wider domain of possibility will involve more than just making a number of extra program chunks available. In principle, however, so long as the computer can cope with the communicative needs of any domain and have the cliches and problem solving skills associated with it, then there is no reason why it should be restricted to a given domain. In reality, however, it is more realistic to consider JEEVES and others like it as domain-specific.

Having established some important aspects and issues in how clients describe their requirements and how JEEVES might be built, it is now possible to structure a more detailed examination of the central issues. This will involve a framework for considering the user-JEEVES dialogue combined with an identification of the main aspects that need to be investigated within such a framework. The next section provides this.


While the dialogues observed in the programmer-client study were very varied and appeared to have minimal structure, an important distinction could be made between two phases. There was an initial requirements generation phase where the client was in control and providing the specification, with the programmer doing little more than clarifying meanings and providing basic feedback. As the dialogues progressed, there was a shift in control until the programmer was doing more of the work and the requirements were being elicited by means of questions and feedback. It was apparent
that in real life this phase would blend into one where the programmer providing prototypes of the program, that would elicit further requirements.

This division of the dialogues seems to be more than just a separation into two phases that may or may not be well separated. It also seems to represent an important distinction between two quite different psychological processes in terms of the requirements specification process. Thus, even if the dialogues had the two activities overlapping to the extent where both occur almost simultaneously, there would still be an important distinction to make. In terms of understanding the dialogues and working towards the issues that are central to implementing them in human-computer interactions, this distinction is, therefore, very relevant.

Having done this, it is useful to suggest a way of envisaging the dialogue between a user and JEEVES that will highlight this distinction. This is intended to serve to support the consideration of the two processes rather than as a definitive design suggestion. The dialogue is seen as consisting of two distinct stages as will now be described.

4.8.1. Stage One

The basic elements of the first stage are shown in Figures 4.15 and 4.16. At the start (Figure 4.14), JEEVES has a library of cliche program chunks (identified from now on as the Procedures Library) but presents the user with what is effectively a blank slate. The user must then generate a description of what is wanted, probably as some sort of program description, possibly with other elements of different sorts. The program described by the user at this stage is imaginary, based on the user's knowledge and ideas about what is possible. For this reason, this program will be referred to as the "Virtual Program", distinguished from the "Program Proper" that JEEVES is ultimately supposed to generate. The items specified will need to be related to one another in some way so that later in Stage One (see Figure 4.15), the user is incorporating the items into what has been describe of the Virtual Program at that point. At some point it is decided by either the user or JEEVES that enough information has been given so that Stage One ends and Stage Two is initiated.

4.8.2. Stage Two

Stage Two is shown in Figures 4.16 and 4.17. JEEVES must first process the description of the Virtual Program and then construct an image of a program that, based on this, should meet the needs of the user. This will be referred to as the "Proposed
Figure 4.14. The first stage of the interaction between a user and JEEVES.
Figure 4.15. The main activity of Stage One of the user-JEEVES dialogue.
Figure 4.16. The first part of Stage Two of the user-JEEVES dialogue.
Figure 4.17. The main activity of Stage Two of the User-JEEVES dialogue.
Program", once again, distinguishing it from the "Program Proper" that will ultimately be implemented. This is then presented to the user in some way (Figure 4.16). The user can now comment on this and tune the Proposed Program until it is close enough to what is required (Figure 4.17). This involves the processes of identifying some aspect that is unsatisfactory and communicating this to JEEVES who must then alter the program accordingly.

At the end of this stage, the program would be implemented and made available to the user.

4.8.3. The Areas of Interest

Having divided the user-JEEVES dialogue into two stages, one must then decide what areas of these warrant further investigation. This must be done by considering the steps that are most likely to cause problems or to determine the overall effectiveness of JEEVES as a tool - the "rate determining steps". Clearly, one question might be to determine how the two stages are to interact. For example, one must decide when Stage One is to end. Issues like these, however, will rely on a good understanding of the issues pertaining to each stage separately. For this reason, such matters are not considered to be as crucial and are left for later investigations.

Looking at Stage One, there are three steps that go into the process. The first is where the user must generate a specification item, the second where the item is converted into a communication and the third where it is incorporated into the existing specification of the Virtual Program. In reality, the last two steps can probably not be distinguished as the nature of the communication is likely to be affected by the process involved in relating it to other items. There are, therefore, two main areas for consideration in Stage One: the source of the specification and the means of communication. If JEEVES were to be designed, these would need to be well understood and the relevant issues identified. These are looked at in chapters 5 and 6 respectively.

Stage Two is considerably more complex. A number of points can be made, however, that can simplify matters. Firstly, the process of communicating a change to the Proposed Program, once the user knows what he or she wants to change, should not require as much effort as communicating requirements in Stage One. This is because one can either employ the same means as in Stage One (a sensible option if complexity of the interface is to be kept low), or make use of the fact that the specification to be communicated is to be in terms of an existing structure and allow the user to make reference to this in the communication. The contrast between Stage One and Stage Two
in terms of the means of communication is basically that between describing an item in
terms of the imagined Virtual Program and directing attention to a more concrete
Proposed Program.

What is left, therefore, are the steps of understanding and evaluating the Proposed
Program. Figure 4.17 suggests that this process starts, as it were, at the identification
of some item that is not satisfactory, leading to the identification of something that
needs changing. This is not something that is actually being claimed here. The arrows
are merely intended to provide a useful means of communicating roughly what is going
on. In reality, it is not clear where the process of evaluation starts or how it works. It
is not, therefore, appropriate to break it down and examine it step by step. Rather, it
should be dealt with as a whole. This is what has been done in chapter 7.

4.9. Conclusions

It has been argued that there is a need to structure the possible dialogue that can occur
between the user and JEEVES in order to make it more manageable in terms of how the
issues can be addressed. It is suggested that this can best be done by dividing it into a
first stage where the user gives an initial description of a Virtual Program and a second
stage where he or she is responding to a Proposed Program offered by JEEVES. This
follows quite closely the pattern observed in the study of Programmer-Client dialogues.

The main concept, then, is that programming using JEEVES will involve the separation
out of three distinct program-related entities: the Virtual Program, Proposed Program
and Program Proper. When programming in a traditional paradigm, the distinction
between these is minimal. What the programmer describes is what becomes the
Program Proper and there is no intermediary stages involving the computer interpreting
this description or the evaluation of some proposed implementation. The closest
analogy would be between the code generated, the program behaviour and the compiled
program. Thus, the programmer generates code that is interpreted by the computer.
This is returned to the programmer as program behaviour that is evaluated and results in
changes. Finally, the end-product is the compiled code.

In addition to this, areas of concern within these stages have been identified that require
examination. Two of these are in Stage One and consist of the processes of generation
and communication of specifications. The last is in Stage Two and is the whole
process of evaluation of the Proposed Program. The next three chapters will cover
these areas.
The results of this study reinforce the idea that JEEVES is going to have to cope somehow with the fact that the user's program description will consist of a number of association-driven requirements that may not be well ordered or structured. These and the other qualities of the requirements specification will require great flexibility and generally present serious difficulties in terms of the technology of the processing involved. However, the data obtained in this study also point to useful concepts and tools that might suggest a possible way forward.
SECTION 2

THE ELEMENTS OF THE INTERACTION

In Section 1 the various options that could be focussed on in this thesis have been assessed. It was argued that programming power will never become accessible to non-programmers purely through improvements in teaching practice. Similarly, there is evidence that simply developing more tools of the sort already in existence will not be sufficient to reduce the work required to program to a sufficient extent. It was concluded, therefore, that it might be most fruitful to investigate the possibilities of automatic programming where users are able to specify their requirements in an informal manner.

This is an area that has received little attention from psychologists and it was found necessary to perform some initial work to clarify where the main issues are likely to lie within it. This was achieved by means of a study of human-human requirements specification dialogue. One of the main results of this was an ability to suggest a general framework within which any requirements specification dialogue will exist. This consists of two main stages. One where users present an initial description of their needs, possibly in the form of a description of a "Virtual Program", and one where the system is generating "Proposed Programs" upon which users comment. The areas of concern relating to these were introduced in Chapter 4 and will be examined in detail in Chapters 5 to 7 of this section.

In Chapter 5 the processes and issues of generating requirements are considered. A number of empirical studies are presented that provide some understanding of what must be involved and the problems that this raises for any tool where users are not given a list of available commands. Chapter 6 looks at the issues that exist relating to the actual communication of requirements. Questions about whether "natural language" is appropriate or not are gone into and alternative requirements for the means of communication considered. A study is described that looked at what happens when novices are presented with a means of communicating that forces them to structure their language. Chapter 7 deals with Stage Two of the dialogue. It considers the issues that exist around the need for users to evaluate the Proposed Programs and describes a study that observed end-users evaluating a computerised tool.
CHAPTER 5
GENERATING A SPECIFICATION

Abstract to Chapter 5

This chapter examines the process of generating requirements. It starts with a discussion of the options available with respect to who controls the design process. A full Task Analysis performed by the tool is found to generate problems. This leaves the alternative option of users deciding what is appropriate as a requirement. The central question for this process is identified as being the need effectively to communicate to the user what can be done by the programs that JEEVES can generate. This is broken down into three specific information requirements: the task domain, the form of programs and the available facilities. This involves an understanding of how information from JEEVES and any manuals, etc., that are provided will result in requirements that are either appropriate or inappropriate. Two studies are described that investigate this matter. A key observation from these is that information provided is not used in any formal way but rather to stimulate ideas for requirements. This has the additional effect of requiring the system to check requirements to ensure that they really are within the program range. The implications of this and other observations are discussed for the design of the Stage One dialogue.

5.1. Introduction

Stage One has two main areas that need addressing. One is the generation of requirements and the other is their communication. This chapter looks at the former while Chapter 6 looks at the latter. The processes of generating requirements must represent the first step of this investigation into the issues of the tools under consideration. This will underpin many of the aspects of the way the dialogue is designed and the extent to which users will need information about what the tool is capable of.

The chapter starts with a consideration of the different ways in which users' requirements could be elicited by JEEVES. These are found to relate closely to the degree of control that JEEVES takes over the dialogue, with a number of implications.

There are three ways in which the user could communicate requirements to JEEVES:
As a description of the tasks to be supported, as abstract requirements and goals, or as a description of an imagined program that would meet the requirements.

One or more of these could be used by JEEVES to extract a specification of the Virtual Program. An important dimension that must be examined first, however, is that of the extent to which JEEVES takes the control from the user.

5.2. Complete Task Analysis - JEEVES in Control

The starting point for this analysis will be in the understanding that has already been achieved about how users can be expected to generate their requirements. This is obtained from the study described in the last chapter and from other work on obtaining requirements from users. The focus must be, however, on the techniques and understanding associated with obtaining requirements without the use of feedback, as this is the subject of analysis in another chapter. Basically, the question refers to the stage of the dialogue before any feedback has been provided and where the user is giving an initial description of the requirements.

The consensus from the literature is that end-users are notoriously bad at describing their requirements. This is, essentially, the reason for the extensive use of feedback and prototyping in designer-client dialogues. Other techniques do exist, however, for obtaining information about the needs of the user. The most frequently referred to class of methods are those known as Task Analysis. These work on the assumption that users are very poor at locating their needs within the existing tasks that they perform. Indeed, they form a subgroup of a wider policy type in these situations where the expert takes over the role of deciding what is needed. In these situations, the dialogue is guided by the expert who is in the position of questioning the client about his/her circumstances. This could be in the context of a designer-client dialogue or of an interview between a public information service and someone with a query. In both cases, a number of strategies can be adopted that vary in the extent to which the expert allows the client to describe his/her needs as against finding out as much as possible about the client's situation and deciding on what the needs are in a more paternalistic manner. This raises a crucial issue for the design of any computer that is to be worked by non-experts. To what extent should it be designed so as to make decisions for the user, as against letting the user make the decisions. There are a number of separate matters that are relevant here.
There is no doubt that this approach assumes the minimum as far as the user is concerned. There is no requirement for users to be able to do more than answer questions about their context. They do not have to design programs or be clear about their requirements. While this has advantages from the users' point of view, it can also have drawbacks as the next sections describe. These derive either from the resulting need for the computer to become fully acquainted with the users' situation to be able to make a decision, or from the expertise that the computer must embody to serve the user in this way, or from the fact that the user may not like to be treated in this way.

In techniques such as task analysis, the user(s) of the intended system are interviewed in depth about the tasks that they engage in. An attempt is made to represent these tasks formally in order to enable the systems analyst to identify appropriate aspects that can be computerised in some way. Basically, rather than asking the users to tell the analyst what they want, and the analyst then building it, the analyst attempts to understand the users' tasks as fully as possible so that he/she is in as good a position as possible to tell the users what they want. In many respects this is an ideal. It means that the programmer is in a very strong position regarding any decisions that need to be made about the system design. The clients have given as complete a body of information about their needs as they can so that the tool can be built with reference to this, rather than to some partial description. While the outcome of this process may well seem to be ideal, however, the process itself must be unsuitable. Obtaining a task analysis is a lengthy process. Usually it involves interviewing a large number of people and then considerable compilation of the results, followed by further interviews to identify errors or misunderstandings. Given that the goal of JEEVES is to reduce the time and effort involved in generating programs, this sort of dialogue may raise problems.

In addition to the length of the dialogue involved, performing a thorough and meaningful task analysis is a process involving considerable knowledge and expertise. JEEVES must be able quickly to grasp the overall structure of what is being described and to home in on the important questions. This is a subtle skill that is not easily automatised and requires considerable interpersonal ability (Vitalari, 84). In addition to skills in the area of extracting such a task analysis, JEEVES would have to have the ability to turn it into a meaningful body of data that could then be associated with existing program facilities. It would probably be necessary for the system to have a set of stereotypical tasks and situations available that would be likely to come up (Shepherd & Hinde, 89). These could then be used to "understand" the information obtained from the user. Having done this, it may then be possible to associate particular program...
features that are available with aspects of the task, so as to generate a program that will meet the user's identified needs.

While this may prove possible with more advanced computers than are currently available today and a better understanding of domains and tasks, it is certainly likely that this would result in tools that were very restricted in their domain of application. Not only would it only be possible for the computer to understand and generate programs for a specific domain, this domain would have to be one that could be described and represented relatively easily and in terms of manageable stereotypical units. Many tasks faced by managers and other professionals would not fall into this class. Also, most would not be that easily compartmentalised so that a full understanding of the needs of the user would only be arrived at by performing a full task analysis, rather than one restricted to some small subset of the user's activity.

The final element of such an approach would be the resulting power and control relationship between the user and the computer. Users would be placed in a position of having passively to describe their work situation, probably in elaborate detail and with reference to aspects that may be of no relevance to the intended program. This, alone would be a very frustrating experience unless the computer was able to handle it with considerable aplomb. In addition to this would be the fact that it is the computer that is in the position of choosing the program to be generated. While this technique would not preclude the user refusing the program offered, or suggesting changes, it is putting a dangerous dynamic into the situation. This would need to be dealt with in a very careful manner and after much testing.

These problems suggest that task analysis may have serious drawbacks as a technique to be adopted by JEEVES. The next section considers approaches that demand more from the user.

5.3. The User Controls the Process

It may be desirable, therefore, to give users more of a feeling of control over the requirements communication process. This will require them to make decisions about what programs are required and what aspects of their life or work are to be supported by them. Given this approach, there are two main issues that must be raised:

* In what general form are users to communicate their requirements?
* How is the user to know what constitutes an appropriate requirement?
The extent to which the second question is important will be predicated on the answer to the first question, so this must be dealt with first.

As has been said, the user could describe the tasks to be supported, the abstract goals to be met, or an imaginary program that could meet the goals. What must be asked is what form is most likely to be used? The evidence points to users communicating mostly by means of imagined programs that will meet their needs. The detail that goes into the description was seen to vary greatly from a very concrete description of how some functions would work to more abstract descriptions of general tasks or facilities that should be supported. This was found in the study described above and reported by others anecdotally. This is clearly not an exclusive approach and others will also be used, as needed. What seems to be true, however, is that the use of concrete artifacts represents a valuable means of expression in these circumstances.

JEEVES would seem to need to accept the user's requirements in the form of some sort of a description of the imaginary Virtual Program that will meet them, possibly combined with some descriptions of the tasks the user wants supporting and the abstract goals that are to be achieved. These latter could be used when communication directly in terms of the Virtual Program proves unsuccessful so that other information must be used. If this is the case, then there is a need to understand how users will generate such Virtual Programs and to consider the best means of supporting such descriptions.

5.4. Observations from the Programmer-Client Study

A number of observations were made in the programmer-client study of clients' generating designs for programs. Thus, it was found that they readily used the prior time of note-making to generate a design that would then be communicated to the programmers. It was found that the designs appeared to be relatively superficial with the dialogue often serving to establish what was wanted in greater detail. This involved the clients generating further information, therefore, in a more opportunistic manner, as it was requested by the programmers. Finally, it was clear that the clients were using their understanding of computers to generate the designs.

It was clear, from this study, that the process of generating a design solution in an unfamiliar domain relies heavily both on information obtained about that domain and on pre-existing information. If the user of JEEVES is to generate a coherent Virtual Program, then this will involve both the identification both of individual features that will go into it and some unifying structure. There may also be levels of complexity
between these that group a number of features and that are, themselves, combined to create the overall structure. Finally, the specificity with which the user can describe requirements, or the appropriateness of the design submitted will clearly depend on the extent of detail of beliefs about what facilities are available. In the absence of information to support appropriate beliefs, requirements will either be very vague and abstract, be based on inappropriate structures and concepts, or not be stated at all.

This suggests that the most important element of this part of the dialogue will be the user's beliefs about what JEEVES can provide. It is not desirable, however, to communicate this in the form of a list of available features as this leaves the user in much the same position as a traditional programmer. What is required, therefore, is an alternative approach to communicating what is possible. This is, therefore, a central issue in the design of any tool that is to meet the requirements of this thesis. It also represents a major source of concern for any tool that is to be used with minimal prior training.

The rest of this chapter is therefore concerned with this matter and with investigating the various issues raised by it. The general questions that seem to be most important are about what information should be communicated, how it is best communicated, and when is best to communicate it. These are explored using what is available in the literature. This is found to provide few useful leads and the need for empirical research focused on this is identified. Two studies are described and the observations that are made enable a more detailed examination. The initial discussion is started with a consideration of what is to be communicated.

### 5.5. What Information is to be Communicated and How

Three basic aims can be identified for the information that is to be communicated. These provide a structure for considering what is to be communicated and how it could be communicated. The aims are to inform the user about:

- the task domain,
- the form the Program Proper could take, and
- the available facilities.

These essentially represent the information that is acquired by users of traditional programming systems when they learn the language, but that needs to be communicated
in other ways to the users of JEEVES. The next three sections describe these in more
detail.

5.5.1. Informing the User of the Task Domain

The most basic goal of any information that is given to the user, in the context of a tool
where there is not a fixed set of available functions, is to communicate the domain of
possible programs that can be generated. This is usually presented in standard
applications in the form of brochures, demonstrations and tutorials. These will be
designed to provide an overview of what can be achieved using the package. As is true
of each of these types of information, the goal is to provide the user with as much
information using as small a volume of material and involving as little effort as possible
for the user. While advertising executives and the marketing division of software
companies seem to have a lot of expertise in this area, there is no material in scientific
journals that discusses it. There is no sound empirical basis, therefore, for designing
information to communicate the task domain of a product. The efficiency of the
communication will rely on the choice of concepts used. These should be chosen to
enable the user to infer as much as possible about what the tool is likely to be capable
of. This use of verbal communication could be supplemented by images and examples
that make the ideas more vivid and concrete. This seems to be the particular strength of
advertisers and the designers of brochures.

Le Courtier (Gershman and Wolf, 86) is a "Conversational Advisory" system for
giving advice about stocks and shares in the Belgian stock market. It has many of the
problems faced by JEEVES in that the user is not expected to have learnt a list of
available functions and to be able to interact with it using "natural language". It copes
with the problem of defining the task domain by giving an initial statement of the goals
of the interaction. The makers of Le Courtier report that, once users have understood
these goals, they tend to stay within their bounds. They will also accept the fact that the
system cannot handle requests that are not contained within these goals. This suggests
that the same might be true of JEEVES.

While the user will be helped by a knowledge of the domain of possible tasks, it will
also be important to know something a more concrete level about the nature of the
actual programs that can be generated. This is discussed in the next section.
5.5.2. Influencing the Structure of the Virtual Program

The previous section discussed the need for the information given to users to increase the likelihood that the tasks and needs that are brought to JEEVES are appropriate. It was seen in the programmer-client dialogue study that the subjects in the role of client would generate program designs that varied considerably in their basic structure. The programmers were seen to accept each as a basis for the programs that they would, fictionally, produce. This may have been, however, because they did not have any particular target environment in which the program would be generated. In the case of JEEVES and similar tools, the Procedures Library will constrain, not only the domain of possible programs, but also their form. The Virtual Program the user describes will either map well onto those procedures or it will not.

If it does map well, then the task of JEEVES is considerably simplified and the Proposed Program need not look very different from the Virtual Program. If it does not, however, then JEEVES cannot just identify elements of the Virtual Program that could relate easily to the contents of the Procedures Library, etc. Instead, there must be a process of extraction of the underlying idea behind each aspect of the Virtual Program, that is then converted into something that will support a similar idea. In addition to this, the Proposed Program will necessarily be significantly different to the Virtual Program resulting in confusion and possibly a sense of loss of control as well as a greater effort required to understand it.

It is likely that this latter situation will be inevitable, to some extent if the user does not know precisely what is available and must generate a Virtual Program on the basis of this incomplete knowledge. Generally, the less the user knows, the more danger there is that the Virtual Program will not conform well to the available programming resources. While it may be that JEEVES will inevitably have to cope with some degree of discrepancy between the concepts and structures used and those available, minimizing this will be beneficial in many respects. Once again, therefore, there is a need to understand how information provided can achieve this goal.

The last aspect that would be learnt implicitly when learning a programming language is the breadth of available facilities that it contains. This is referred to in the rest of this chapter as the Program Range. The problems involved in communicating this to the user of JEEVES are discussed in the following section.
5.5.3. Informing the User of the Available Facilities

Possibly the most difficult aspect of the Program Range to communicate is the breadth of available functions and facilities that exist. This is a problem even for traditional software tools. It is a well-documented fact that most users do not take advantage of all the available commands, etc., of their systems (eg. Eason, 88). There seems to be no very innovative ideas available for dealing with this problem, however, and it may represent a fundamental dilemma for the makers of complex tools. Having said that, it is useful to consider what techniques are used to communicate the available facilities of a traditional system.

The most common ones involve the use of detailed descriptions contained either in paper-based manuals or on-line "help" facilities. These are accessed through either a contents page or index. These seem to support, with varying degrees of success, the activity of searching for a function that will achieve a desired goal. In addition to these referencing facilities are the introductory tutorials given that (it is assumed) the user will go through in a more thorough fashion. These serve a function that includes describing the task domain and giving examples of the available functions. None of these techniques provide the user with an overview of what sorts of facilities are available. Brochures represent the nearest that one ever sees to this when they summarise what the tool described can do. This is by little more than providing long lists of the names of the commands or functions.

JEEVES differs from these traditional packages, however, in that providing an effective summary of the available facilities without reference to how they are to be called upon may well be adequate. It remains to be tested whether this could be done in a way that would make sense to the user. One of the likely problems with this is that short labels may have little meaning to users who are not already experienced with these sorts of concepts and entities. Unfortunately, this does not remove the problem. The problem for users of a traditional package is that unknown functions will not be taken advantage of. This is also a problem for users of JEEVES. These have an additional problem, however, in that not knowing how to tell it to include a given facility in the program being constructed does not stop them from requesting it. The need to communicate the breadth of available functions to users of JEEVES is important both to help them make maximum use of it and to reduce the risk of the Virtual Program containing features that cannot be implemented in the Program Proper.

This suggests two possible approaches to dealing with this. One is to communicate what is available, and the other is to communicate what is not available. The latter,
however, is not so very easy, as Gershman and Wolf (86) suggest. They point out that such lists would be long and cumbersome, that people would not be able to remember them and that the need to learn them negates the advantages of such systems. Indeed, one could never compile a list that contained all the facilities that were not available.

Thus far, the emphasis has been on facilitating the communication of the Program Range by means of suitable choice of information. An alternative approach that would have to be considered in conjunction with this, however, is that of the choice of the domain itself. It is likely that sensible choice of the domain boundaries and the building blocks of the programs could significantly reduce the work demanded of the description of the Program Range. This is discussed in the next section.

5.5.4. The Choice of the Domain and Structures

It would clearly make sense to define a Program Range that would map well onto existing concepts. In particular, it might be sensible to map it onto task or goal-related concepts rather than functional concepts. This would make it easier for the user to grasp the boundaries of the domain and to identify likely tasks that could be supported. The other choice that must be made is that of the actual building blocks or structures that are used to communicate the form of programs that can be generated by JEEVES. Once again, it will be important that these be chosen to be readily accessible to the user. This is akin to the problem of the design of traditional programming languages where the desire is to make the machine model or programming units more concrete. The underlying aim is to maximise transfer from existing knowledge and experience to the use of the tool. There would clearly be valuable insights to be gained from studying existing languages in this respect.

Thus far, the focus has been on the relationship between material presented to users and their subsequent behaviour in generating a Virtual Program. It is clearly implicit in this that the material presented will influence behaviour by means of altering users' beliefs about what JEEVES is capable of. The goal of the material is, therefore, to allow the user to generate an appropriate and accurate mental representation of the Program Range. This suggests that there might be some useful insights to be obtained from work on mental representations in general. The next section discusses this.

5.6. Mental Representations Arising From Information Presented

There is, indeed, a considerable body of work that has been done to identify the nature of "mental models". On examining this, however, it is apparent that the term "mental
model" and the resulting field of interest is limited to mental representations of systems, or processes (e.g. see Gentner & Stevens, 83) rather than domains. Thus, the goal of such mental models is to enable the user to ask questions about what happens after a given event, within the system of interest (Norman, 83). For example, how does a process control plant respond to a given valve being closed. The mental representations that are envisaged in this chapter are more intended to answer questions about whether a given item falls within the domain or not. This has the effect that, while books exist with insights about mental representations (e.g. Gentner & Stevens, 83), the specific details of the observations have little to offer this thesis.

The work that is closest to the issues of interest here is in reference to supporting users of large databases or hypertext systems. Jih, Bradbard, Snyder and Thompson (89) studied the effects of two models of a database on query writing. No major difference was observed between the conditions. The work on hypertext systems has looked at how users can be helped in their navigation of such systems by the provision of information about the system or by other means. This is still in its infancy, however. Despite this, a number of valuable concepts have been identified that will have relevance to the exploration of this area and these are described below.

The concept of a mental model carries with it the implicit idea that the owner of the model has a single entity that embodies his/her knowledge of the behaviour of the system of interest. Thus, someone having a mental model of how a program runs implies that they represent the program as a single entity and that their efforts to understand or predict behaviour will be with reference to this single understanding. This is in contrast to an alternative view of one's understanding of a process or system as consisting of a number of discrete packages of information about it. While most of what is written assumes the former perspective, it is not clear that the evidence supports this. It is worth remembering, therefore, that the user of JEEVES may not be constructing a single understanding of the Program Range but rather just be building up a list of items of information about it.

The second main idea about models is that they are not acquired in isolation from existing relevant knowledge and concepts. Thus, work has looked at the ways in which expert understanding of mechanics, etc., is developed out of "naive" models (e.g. Larkin, 83, diSessa, 83). This is an important point that will be very relevant when considering the information given to the user of JEEVES. By stating that the tool is able to generate time management programs, the information that this communicates to the user about the likely facilities that are available will depend heavily on what facilities
have already been experienced in association with such programs and tools. Similarly, the building blocks used to describe the structures of the programs will be interpreted in terms of the sorts of program concepts already available. It is therefore apparent that information given should exploit existing knowledge insofar as this is possible and avoid generating possible misconceptions.

Finally, a number of properties of "mental models" in general have been documented by Norman (83). Those of relevance here consisted of the observations that:

- Models are incomplete.
- Details of models can be forgotten.
- Boundaries can be confused between models of two different entities.
- Models are not necessarily rational.

These generate a picture of mental representations as highly unreliable entities but otherwise say very little. In particular, there seems to be very little known about how precisely, models are generated by information received or affect behaviour.

5.7. Communicating Information During the Dialogue

One final area that is worthy of consideration is that of the choice of when the information is given to the user. The options are basically to give it at the start, prior to the dialogue or to provide it during the dialogue as the need arises for it. There would be advantages in both approaches and the best solution might well be different for the different kinds of information. Thus, the task domain and the basic program form might be best communicated at the outset while information about available facilities could be made accessible to the user during the dialogue.

Techniques for providing information about what is available during the course of the dialogue have been developed by the designers of intelligent front ends to databases. These are intended to avoid the user having to take in a lot of information before being able to start on the dialogue, which is a goal shared with this work. Le Courtier (Gershman and Wolf, 86) and FLEX (Motto, 87) represent two examples of such systems. In both cases there are methods for determining whether a request from the user is within the scope of the system. If it is found not to be, then the system attempts to provide the user with information that will be of help. This is done by an analysis of the request to try to identify concepts that are familiar. This then enables the system to offer items within its repertoire that might be close to what the user is wanting. This
technique will only work for requests that are "borderline". Requests may be made that cannot be interpreted by the system and in these cases the system states that it does not understand and provides a simple restatement of the goals of the system.

Whether this sort of approach could be adopted by systems where the communication is about the design of a Virtual Program or not remains to be seen. It may well be that the problems involved will be considerably more complex than those for the systems described. This and other issues can only be considered in a speculative manner at present given the dearth of relevant empirical data. The next step must be to perform studies examining the attributes of dialogues of this type and the effects of different sorts of information on the behaviour of subjects describing Virtual Programs. Two such studies are now described that enable a fuller analysis of the processes involved and the implications for the design of tools such as JEEVES. As there is no existing science of this area, it was necessary to start with small studies that would provide a basis for thinking about the area and the issues.

It is not possible to look directly at the effect of different types of information on the user's understanding of a domain. What is possible, however, is to observe the effects of information on the specification behaviour of subjects. Two studies are now described that do this. They have focused on two different types of information. The first looks at the possibility of providing the user with a description of some part of the program that is fixed, as a stimulus to descriptions of other parts that can be customised. The second looked at the use of generic structures as a means of communicating the Program Range and provided a more general examination of the specification dialogue with reference to domain constraints. The goal in these studies was not to compare in a formal manner different types of information. Rather, it was intended to achieve a basic understanding of the factors involved when a given type of information is used.

5.8. The Use of a Program Skeleton as a Basis for Specification Generation

One possible way in which JEEVES could work is for the programs that can be generated all to be based around one single basic program skeleton. In a sense, therefore, the user would be customising an existing program rather than creating one from scratch. This might have advantages, however, for the user in that it would provide the user with a concrete basis for understanding the form programs can take and could stimulate requirements and program features that will be used with this. In this study, therefore, no information was given at the start about what facilities were available with which to customise the program. This approach was examined in a
study that asked subjects to describe how they would like an electronic diary to be
designed to meet their particular needs.

5.9. Method

5.9.1. Subjects

Subjects were eleven adult diary users. They were obtained in two ways. Two
responded to an advertisement, posted around Loughborough University, inviting
people who were interested in an electronic diary, the Psion Organiser II LZ to have a
free trial of it and to give their suggestions about how it could be improved. They were
therefore motivated to evaluate it both from the point of view of someone considering
purchasing one and for influencing the design of further models. Another group of
subjects were obtained by contacting people who already made extensive use of diaries,
but who, while interested in electronic diaries, might not necessarily be considering
getting one.

All were in paid employment and the diaries they described related to this work (i.e.,
they were not social diaries). A questionnaire was used to obtain a rough profile of the
subjects. All but two had used computers for more than 100 hours and four had done
more than 100 hours programming. Six of the subjects were familiar with the concept
of an electronic diary and two actually owned earlier versions of the Psion Organiser.

5.9.2. Information Presented

The Psion Organiser II LZ is a complex tool with many different facilities. For the
purposes of this study, only the diary-related functions were focussed on (described in
Appendix IV where the script of a demonstration used in a later study is given). The
diary of the Organiser works by a combination of three screens that can present
information about dates and appointments in different ways. Associated with the diary
is a menu of functions. Dates, times and menu items can be selected by means of a
cursor. Subjects were provided with the description of the screens shown in Figure
5.1. They were given, therefore, a description of the ways in which information could
be presented on the Organiser. This was intended to form the basis for their
specification of what features they would want to see on the Organiser to meet their
needs. This was designed to be understandable but clearly suffered from the subjects
not having the opportunity to interact with these screens in order to understand fully
what was involved. This was felt to be a reasonable way of doing this as the study was
of the effects, in part, of limited experience of the basic program skeleton.
Daily Diary

On the screen, one can have part of a day’s schedule. This will include any blocks that have been specified for appointments, etc. and the remaining free time. Both of these consist of blocks of time specified to the nearest quarter hour. Each appointment has a textual comment associated with it that can be longer than just the available space, and can have an alarm associated with it. Not all the day’s schedule can necessarily be shown on the screen.

Sun 5 Aug 1989 Wk31
10:00a CHURCH
11:30a <FREE>
1:00p LUNCH WITH

In the example shown, on Sunday 5 August 1989, the screen is showing some of the items in the diary. It is showing an item from 10:00 to 11:30 am and another starting at 1:00 pm. There is some free time between these.

Week-at-a-glance

One the screen one can see a week’s appointments, very simply, from Monday to Sunday. Each day is divided into 4 blocks. These blocks cover the periods 12 midnight to 6:00 am, 6:00 am to 12 midday, 12 midday to 6:00 pm and 6:00 pm to 12 midnight. The blocks that have something booked are shown in larger black blocks. Thus, one can see free time and roughly when one has some activity.

Mo Tu We Th Fr Aug89
31 01 02 03 04 05 06
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-
-

In the example given, the screen shows the week from the 31 July to the 6 August 1989. Monday is free, Tuesday has something(s) in the morning, Wednesday has a very busy morning, starting quite early, etc..

Monthly Calendar

One can see a whole month on the screen at a time but with no information about appointments. Not all the month is visible on the screen at once.

Mo Tu We Th Fr Aug89
-- 01 02 03 04 05 06
07 08 09 10 11 12 13
14 15 16 17 18 19 20

In the picture, one can see part of August 1989.

Figure 5.1. Descriptions of the Psion Organiser diary screens provided to subjects.
5.9.3. Procedure

The study took place at the subjects' homes and took just over one hour. The purpose and an overall description of the experiment was given and subjects were informed that their comments would be sent to Psion who were providing the Organiser used. The sessions started with an informal interview about the subjects' current diary usage and their ideas about what electronic diaries could offer them. This was to help the subjects think about diaries and the tasks for which they used them.

Subjects were then briefly shown the Organiser and its basic hardware was described. A description of the screens of the diary of the Organiser was then shown to the subjects and they were told to imagine that they were in the position of designing one for their own particular needs. They were told that they were limited to the information displays shown but could request any functions or commands that they felt they would need. The subjects then went through the screens saying whether they would want each one and describing their specifications for it. Detailed notes were taken of all comments made.

The experiment ended with a short questionnaire about the subjects.

5.10. Observations

Focussing on the process of subjects specifying how they would design the Organiser to meet their needs, the first set of observations refer to the dialogue between the experimenter and the subject. Having discussed this, the nature of the specifications will be described.

5.10.1. The Dialogue

The most immediate observation about the dialogue was the difficulty that was had in clearly explaining to the subjects what was wanted of them. In particular, conveying the concept of "command" or "function" required considerable thought. This was made concrete by asking subjects to describe what buttons they would want available or what menu items. Even this was not enough for some and examples had to be given.

Having communicated what was required, some subjects found it extremely hard to generate ideas about what was wanted. In these cases, the Experimenter had to be careful not to offer subjects features to which they could just give "yes" or "no" responses as the object was to get subjects to generate features themselves. In addition
to this, certain aspects of the descriptions of the program skeleton had to be clarified before the Subjects could comment on them.

### Day Display

- enter,
- edit,
- delete,
- alarm
- nature of alarm - fixed limited or recurring cancellable
- copying entry weekly/daily with time period specified
- spread of entry - how long it lasts
- negotiable (time reserved for writing) entries distinguished from non-negotiable (meetings with others) ones.
- priority indicator
- access to a database - categories of items and have questions about each one so that this can then be accessed when one looks at the item - more information about them, eg. where it is, how long, what to take, ...

### Week-at-a-glance Display

- spread of demand transferred automatically from day
- distinguish mandatory and reserved time - would be useful
- cursor moving around to go to day diary

### Month Display

- cursor to move to diary
- either start of day or first entry
- inverse video for commitments

*Figure 5.2. An example of a list of features and comments made by a subject.*

### 5.10.2. The Specifications

While the comments made were generated as a fairly continuous stream of dialogue, it was possible to arrange them into references to a number of different aspects or features (see Figure 5.2 for an example). It was then possible to categorise these features and make comments about their nature. The first method of distinguishing comments was in terms of the information they carried. Thus, comments could be divided into:

- those just indicating whether some aspect of the screens shown was useful or not,
- those describing how an aspect could be used and what it could be used for,
- those describing specifications of what the subject wants with this facility.

These categories were found to overlap to some degree but it was generally possible to make a decision about each comment. Thus, for example, comments about usefulness
were often associated with how an aspect would be used and comments about how an aspect would be used often ran into what changes were needed to it. Overall, there were 22 comments on a feature's usefulness, 16 comments about how features could be used, and 85 specification comments. It was also noted that comments could request features that involved a change to the basic screen design rather than the addition of a function, etc. Comments were then categorised according to whether they were appropriate, in this sense, or not. A count indicated that, of 85 specification comments made, 47 were appropriate and 38 inappropriate.

5.10.3. Relation Between Comments Made and Subject Characteristics

Having some data about the subjects made it possible to see if any relations existed between their experience and knowledge of computers and electronic diaries and comments made.

Subjects were categorised in the manner used in the description of the subjects. This resulted in four elements of a profile:

- Used computers for more than 100 hours (C) or not,
- Know about electronic diaries (E) or not,
- Programmed for more than 100 hours (P) or not,
- Own the earlier version of the Organiser (CM), or not.

Comparisons of subjects who were positive with those who were negative, for any characteristic, was not possible because the groups would not have been matched on the other elements. It was found that this resulted in a scale of some sort rather than a number of binary divisions within the subject group. Table 5.1 shows that all subjects fell into one of five categories. This results in a scale where each subject type is negative on one more element than the one above. This enables one to look for correlations between such "points" and number of comments. Points obtained in this way represent a purely ordinal variable, limiting the analysis that could be performed on it to non-parametric tests.

Spearman’s rank correlation coefficient was used, therefore, to calculate correlations between the profile of points and subjects' behaviour in the tasks. Such correlations proved significant for both number of appropriate specification comments (rho = 0.706, p<0.05) and number of inappropriate ones (rho = 0.746, p<0.01). A positive correlation was found for both suggesting that experience with computers and
electronic diaries is associated with generating both appropriate and inappropriate ideas. The ratio of appropriate to inappropriate comments was calculated for each subject and a correlation test was performed to determine whether this ratio was related to experience (see Table 5.2). The data did not support this hypothesis (rho=0.539 p<0.05). There is no evidence, therefore, that experience helps subjects distinguish between these.

<table>
<thead>
<tr>
<th>G</th>
<th>E</th>
<th>P</th>
<th>CM</th>
<th>No. of Subjects</th>
<th>Points Allocated</th>
</tr>
</thead>
<tbody>
<tr>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>Y</td>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>Y</td>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>Y</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>3</td>
<td>1</td>
</tr>
<tr>
<td>N</td>
<td>N</td>
<td>N</td>
<td>N</td>
<td>2</td>
<td>0</td>
</tr>
</tbody>
</table>

**Table 5.1.** Showing how subjects were allocated points according to which of five categories of experience they fell into.

<table>
<thead>
<tr>
<th>&quot;Points&quot;</th>
<th>No. of Appropriate Comments</th>
<th>No. of Inappropriate Comments</th>
<th>Ratio of Appropriate to Inappropriate</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>2</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
<td>3</td>
<td>1.33</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
<td>4</td>
<td>1.25</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
<td>2</td>
<td>0.5</td>
</tr>
<tr>
<td>3</td>
<td>9</td>
<td>4</td>
<td>2.25</td>
</tr>
<tr>
<td>3</td>
<td>2</td>
<td>5</td>
<td>0.4</td>
</tr>
<tr>
<td>4</td>
<td>11</td>
<td>6</td>
<td>1.83</td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>3</td>
<td>2.67</td>
</tr>
</tbody>
</table>

**Table 5.2.** Numbers of appropriate and inappropriate comments made, organised by "Points" (see Table 5.1).

5.11. **Discussion**

The results of this study suggest two processes in generating a specification for a Virtual Program. One is where the subjects use the information given to generate ideas about features they would like, and the other is where the features are monitored for their appropriateness. This is quite distinct from a model of specification generation that suggests that skeleton program description, such as that provided in this study,
allows the user to move directly to a set of specifications using more formal methods. The reason for this analysis is described below, along with a consideration of its implications.

The main observations of the study were firstly that specifications could be generated from the basic program skeleton description, and that these seemed to be associated with ideas about how useful aspects of the program were and how they could be used. In particular, it was found that computer experience was associated with the identification of features to specify. Upon examining the specifications, however, it was found that many of these involved requests for changes to the program skeleton. The extent of these inappropriate specifications was not associated with computer expertise, however.

What seems to be happening, therefore, is that the aspects of the program skeleton are used to stimulate recall of tools and experiences that allow the subjects to state whether the aspects are useful or not. Very often, this leads them to identify changes that would need to be made to the aspects that would improve their usefulness. These constitute the specifications. The data support a hypothesis that generating such specifications is facilitated by computer experience. This is a plausible hypothesis as, in the absence of computer experience, it would be very hard to generate design ideas to make the skeleton more useful.

This process of generating specifications is, therefore, very informal and unstructured. In particular, the existence of many inappropriate specifications suggests that it does not derive from a clear approach to using the skeleton. The problem seems to be one of distinguishing between specifications that represent additions of functionality, as was intended, and ones that constitute changes to the skeleton. This would seem to constitute an important secondary process to the overall activity of generating specifications. The evidence is that, either there is no attempt to distinguish between these, or the distinction is failing. Whatever the case, computer experience was not found to be facilitating this process.

The best interpretation of this data is that the subjects were not sufficiently familiar with the concepts involved and the program skeleton to understand the distinction that was being made. In a domain that is familiar to subjects, with a skeleton that could be well understood, and where the distinction between what could be changed and what could not was more concrete, one would expect them to make fewer inappropriate specifications. The observations made are extremely valuable, however, as they force
us to consider such matters if this approach is to be adopted. Thus, the need for users to achieve a good understanding of the program skeleton is clear. This will have implications for the training and preparation time that must occur before they can start designing on the basis of it. Further studies would be needed to determine the most important aspect to be understood and how best to communicate it. The danger is, of course, that the best way will be found to be via experience of using the item, and this suggests an inevitable training period.

While this study has provided an important insight into the processes of generating specifications, it does not go into much detail with respect to the dialogue involved and the way the subjects use past experience to generate ideas. It would also be useful to have some information about how subjects would respond to the provision of a different type of information about the Program Range. The next study described goes into more detail and explores a slightly different type of information in a different domain.

5.12. The Use of Descriptions of Generic Program Facilities and Examples

The task of this study was essentially the same but many of the elements altered to make it more valid. Unfortunately it was not, therefore, possible to maintain the situation where subjects were engaged in the task out of a belief that their comments would affect future designs and where they were in a position of actively wanting a tool of the type they are specifying.

An additional feature of this study was the engineering of the preparatory stages prior to the subjects' specifications so that it becomes possible to see a little more of the process of generating requirements out of current and past tool use and ideas about what electronic tools could offer. In this way it was intended to obtain further insights into the process of generating requirements as a whole, not just the elements of responding to the information presented.

5.13. Method

5.13.1. Subjects

In this study seven subjects, all currently involved in research either as University Lecturers, Research Assistants, Technicians or Research Students, were used. As the first part of the study, subjects were asked to describe referencing and bibliography systems that they either currently used or had used in the past. These could include
card files and any other ways in which they made use of references. Subjects were also asked about any other experience they had had of tools or systems used for similar purposes. This resulted in a rough profile of their experience with such systems that was used in the analysis.

5.13.2. Procedure

In the second stage, after the subjects had described their current and past use of referencing systems, they were asked to describe how they would like an ideal electronic referencing system to be designed if it were to meet their particular needs. Finally the explanation of a basic framework for building a referencing system plus generic descriptions of what could be added (see next section) was given and subjects were asked to specify how they would want such a system to be designed.

Users of JEEVES will be generating specifications from a combination of their relevant experiences and the ideas they have about what electronic tools ought to be capable of. Because of this, the three-stage design both simulates the situation of someone coming to JEEVES with a need in the domain of referencing systems and provides data on the basis upon which subjects generate their specifications.

5.13.3. Information Presented about HyperCard-

Having explained what would be happening in the last stage of the study, subjects were then given a description of the basic framework around which they were to imagine the referencing system would be built. This was a limited sub-set of the HyperCard system. This will be called HyperCard- (pronounced "HyperCard Minus") to indicate that it is less than the full HyperCard package. They were told that they could design a referencing system that would meet their own particular needs, using certain elements of the HyperCard system. The system they could construct was limited in that it could only consist of a single stack of identical cards with the subject's choice of fields and buttons. The description involved providing definitions of a number of generic objects (stacks, cards, buttons and fields) and the relationships that could exist between these. The definitions given are shown in Figure 5.3. For three subjects interviewed early in the study (A, B, D), the descriptions of the framework did not consist of any more than this. These found the concepts hard to grasp.

Because of this, it was decided to supplement the description with an example of a stack of the type being described. This stack was not a referencing system, however, but a product catalogue of an imaginary computer hardware firm (see Figure 5.4, a &
b). Subjects were shown this so that the concepts of fields and buttons could become more concrete and it could be explained how the system could have many different cards that were identical except for the information in the fields. The example was presented on paper so that subjects were not able to explore how it worked.

General

This framework is not exploiting the full potential of HyperCard. It is a limited subset of that. HyperCard has stacks of cards as its applications or files. The cards are analogous to cardfile cards and are ordered within a stack. The stack is is purely linear. The system you are designing will have just one stack. Each card will hold together information about a different book or article or whatever. Cards have all the same elements and design apart from the information held. The only exception to this is the Introductory Card.

Fields

Each card has a number of fields. Fields are where you enter information. They are essentially labelled spaces for specific information. Each card will have different information in its fields. So one card will have information about one reference in its fields and another will have information about another reference. Thus, there is no central database, the information resides solely on the cards.

Buttons

The other type of object that can be on cards are buttons. Buttons are the way to do things in HyperCard. Buttons, when clicked on using the cursor and mouse will perform a function. Functions can include moving within the stack or carrying out tasks on the information on the currently visible card or another card. There are the same buttons on each card, apart from the Introductory Card.

Entering the system

There can be a card that works as the first in the stack that has no fields but some buttons for getting to references of interest, etc.

Setting up a new reference

When you construct a new card for a new reference, it will be identical to the others except that the fields will be empty to start. You can then enter the various bits of information about it.

Navigating the system

Buttons are used to move around between cards. These can use the information in the fields to help find cards wanted. Alternatively, they can just move you to the next card, or to the previous card, using the order of the stack.

Figure 5.3. The information given to the subjects about the HyperCard framework. Explicit statements of constraints that are contained in this are printed in italics.

The use of the example resulted in further problems for the subjects, however. What emerged was that, while subjects could understand something of what the example consisted of and how it worked, they had difficulty understanding how they could use it as a basis for designing their own systems. All four of the subjects presented with
Figure 5.4a. The example stack illustration - Introductory Screen.
Macintosh Plus
Motorola 68000
1 megabyte RAM (expandable to 2 MB)
the example had trouble understanding what aspects of the example were fixed parts of 
the system and Subject C misunderstood its purpose completely.

This reinforced the observations of the previous study in that the example was 
analogous to the program skeleton. In both cases, there was a need to communicate 
clearly how the item was to be used. In particular, clear statements about what is fixed 
and what is not will be necessary. The danger with this is that it results in lengthy 
explicit descriptions of the Program Range that negate any value of the framework or 
example. It may be, however, that it is worthwhile having these simply to make the 
description more concrete.

5.13.4. Data Collection

Audio recordings were made of the interviews. These were transcribed in the same 
way as in the study of programmer-client dialogues.

5.14. Observations and Discussion

5.14.1. Stage 1 - Past Experience

There was a great deal of variation in subjects' experience of referencing systems and 
databases in general. Most, however, used ones that revolved around the same basic 
information being stored, but with specific features that supported specific tasks. Each 
was different, however. See Appendix II for details of these.

5.14.2. Stage 2 - An Ideal Electronic Referencing System

There was a wide variety both in the approach taken to generating an imaginary "ideal" 
system and to the resources available for specifying how it would work and the 
facilities that were desirable. Five of the seven subjects started with a database they had 
experienced and found ways of dealing with their frustrations with it by reference to 
what they knew or believed ought to be possible (see Appendix II for details). In many 
respects, therefore, their behaviour was analogous to that of subjects in the previous 
study, except that they generated their own program skeleton and did not have any 
constraints on how it could be improved on. The remaining two (B & F) appeared to 
have a less clearly defined starting point for their design, although in reality there was a 
continuum of the extent to which subjects were clearly basing their ideas on one single 
existing database.
5.14.3. Stage 3 - The Specification Generation Stage

In attempting to characterise this stage, a number of observations can be made. The clearest one, that provides an organising framework for the others, is that five relatively distinct activities can be seen to be occurring. At any one time at least one of these would be in evidence. Initially, the experimenter would explain something about what was to happen next and what was expected of the subject. Dialogue of this nature can be labelled "meta-communication" (M) and may occur at other points than at the start of the stage, if difficulties arise. The second element that appears is that of the experimenter explaining the basic concepts and structures that make up the HyperCard-framework with which the subject must work (C for "concept"). Once again, this can involve further clarification dialogue later in the stage as well as an initial description.

S: Well obviously it would have to have a lot of information on each card and the information would have to be categorised in various ways. It would be good to have an abstract of the article on the card as well, but you wouldn't necessarily have. I suppose you'd have to pull the same card up every time would you, or could you get bits of cards?

Ex: No, you can only pull up a whole card.

S: Oh, right. So you only get one at a time. That would be a bit laborious if you wanted to quickly get a list of...well, would it be able to print out just the names of, or, yes, just the names of people, or a number?

Ex: Well, let's think about the tasks that you'd like it to be able to do then. So you're saying...

S: Well you see, if you wanted, I mean it's all very well saying I want to look at one article, that's fair enough, but if you wanted a list it wouldn't be able to give you a list, would it? Give me all the articles on a certain subject so I can then get them out and read them.

Ex: Well, can you see any way around that?

S: Well I can only think of the computer having another database apart from the stacks of cards that has data in a different form, so that you can have two ways of looking at it. You can get printed lists out but you can also look at abstracts. But I don't see. I never, I either wanted to look at the article itself or just have a very quick idea of. I don't necessarily want to look through a stack.

The next three activities are not so clearly located within the dialogue. These are communications about the boundaries of what is possible within the HyperCard-framework (B), finding ways of achieving the desired goals within this framework (S for "solution"), and communicating such goals (P for "problem"). Examples of
extracts from dialogues of each of these types are given in Figure 5.5. At any one time, one or more of these can be seen to be the focal intention of the discussion although others may be occurring implicitly. It will be seen that this analysis allows one to make certain useful points about the way information helps the generation of a specification and the ways in which JEEVES will have to work. Each element is now gone into in more detail and further observations made. See Appendix II for graphs showing the progression of the dialogue in terms of these different activities.

Subject: Would you be able to, once you've done, designed it, and you suddenly realise that another article has some more information, another category that you'd completely forgotten, so you want to make this new category on all of the cards, like you suddenly want to do a number reference in the system where you previously hadn't done it, would it be flexible for that, or would you have to go back to the designer?

Expter: As far as what I'm talking about is concerned you wouldn't really support that, although HyperCard would in fact. What I'm basically saying is that I'm trying to separate out the design phase from the working phase.

5.14.3.1. Communication of Constraints and Specific Capabilities

The description of the HyperCard framework included explicit and implicit statements about the boundaries of what was possible. In addition to these descriptions at the start of the dialogue, further discussions of the boundaries occurred during the course of the actual specification. These could take two basic forms. On 7 occasions, subjects asked whether something was possible and were told either that it was or that it was not (e.g. see Figure 5.6). Alternatively, a subject could specify a requirement that the experimenter realised was outside the boundaries of possibility and communicated this (e.g. see Figure 5.7). This only happened four times. It should be noted that the remaining cases of subjects generating elements of specifications did not involve the experimenter to any great extent. There were 25 of these, insofar as it is possible to distinguish them. In most cases where the experimenter had to inform the subject that a specification was outside the Program Range, he would try to offer an alternative that was not. If that did not happen, then the subject would look for such an alternative, as was shown in the example of Figure 5.7.

These situations differ in two ways. The first difference is in the beliefs of the subject. To ask whether something is possible, the subject must believe that it might not be, for some reason. If, however, the experimenter had to point out that a specification was
outside of the boundaries, then the subject must have made the specification on an implicit assumption that it would be possible. This suggests a number of options, as laid out in Table 5.3. The cells of interest are (2), (5) and (6). The difference in the subjects' beliefs affect the required behaviour of the experimenter. If the situation is as in cells (5) or (6), then the subject will ask about the aspect, and the experimenter must provide an answer. If, however, the situation is as in cell (2), the subject will not have interrupted the flow of specification and it is up to the experimenter to identify the problem.

Subject: Well, from the sounds of it you're talking about a title screen and behind that a succession of cards which, from HyperCard I'd say was probably one per screen, which I'd find awkward. Because it's going to be that type of system and not a windowing system, I think I'd like to create links between relevant cards very easily very quickly so that I put a button on that card that will take me to all the others in that particular group. So it's as if I'm flipping through a stack of paper ones that already sorted.

Expter: OK. I should have said that you can't, again a further constraint is that all the cards will have the same buttons on, so you wouldn't be able to, you'd have a design phase which is where you put the buttons on. After that you can only add information that appears. And that's a further constraint.

Subject: Will the buttons pick up information from the fields?

Expter: Yes.

Subject: Well I'd like a field then that, if you put the keywords on and the keywords would act as links between the cards.

Figure 5.7. An example of subject E giving a specification that is outside of the specification and the experimenter pointing this out to him.

In both cases, the experimenter is having to compare the subjects' descriptions of specifications with knowledge of available facilities. The ease with which this can be done will depend to a great extent on the way the specifications are described. The specifications are often given at a superficial level of description (as was also seen in the programmer-client study), so that there is only a very vague implication about how they are to be implemented. This means that the experimenter must actually work out whether there is any way of implementing the item, in order to determine whether it lies within the Program Range (see eg. Figure 5.8). This was found by the experimenter to be a complex and effortful task and on at least one occasion, an impossible specification was not picked up on.
Subject: Well I mean I'd want to have essentially the fields that you have on your. Now, I'll tell you what, I'll describe what I. Maybe if I start saying the things explicitly. So, I want a number, a card number, which is going to cross refer to my physical reference. In fact I'm now going to describe just the, the basic reference card and I suppose, I mean talking about implementing the, if we implement that first system that I talked about, which isn't a bad system, how would that happen? That would. Now, you have to advise me. I want to be able to. If I were to implement that first system I would have to have cards which actually had people's names on so that I could actually enter a field with somebody's, no I wouldn't need that.

Exper: There is in principle a problem with that because you're requiring three or more sorts of cards.

Subject: Yes, you probably wouldn't need that with this system, or you need something which has empty fields and you enter the fields that you want. So if you would, I mean, I suppose I'm now going to describe just the basic search paradigm and a database so that is to say you have empty fields on author, year, journal, volume, publisher, summary, conclusions and, in my case, tasks used, measures and various techniques and then my own comments, key points.

Figure 5.8. An example (subject F) showing that the experimenter must imagine how a specification would be implemented to determine whether it was within the domain of possibility or not.

5.14.3.2. Generating Requirements

In the previous study, the process of generating requirements was characterised by the subjects looking for associations between the program skeleton and past experiences, that could lead to their finding uses for it. These would then involve the identification of ways of altering the skeleton to make it support these uses. In this study, a slightly
more detailed analysis of this process was obtained. The underlying idea of this was that the process of generating a specification could be seen to lie between two extreme approaches. Eason (88) pointed these out, stating that one can generate a specification, either by starting with a task and finding a tool to support it (task-to-tool) or by starting with a tool and looking for a use for it (tool-to-task). Carroll (88) also mentions a similar phenomenon that he identifies as the task-artifact cycle. This occurs when a need results in a tool being generated, that, in turn, causes the identification of further needs and further tools. On the whole, specification generation behaviour seemed to be divisible into these two categories. They are used, therefore, to structure the observations made. Before this, however, another observation is made about the implicit content of the specifications.

The majority of the specifications generated were clearly associated with major tasks that the systems were to support such as locating groups of references or organising them in various ways. It was remarkably rare, however, for the very basic facilities that would be required to support the more mundane tasks would be mentioned. Thus, few subjects stated that they wanted to be able to delete cards, edit cards or move from one card to the next. The suggestion seemed to be that the system they had in mind had a number of facilities that were so basic that they did not require description. This is not a surprising observation. If one is describing the car one requires, one does not specify that it need have four wheels and an engine. If JEEVES is to generate programs around users' specifications, it will therefore be necessary for it to know what is assumed to be obvious.

In a number of cases, it was clear that the subjects had established certain tasks for which they used their referencing systems and for which they would want to use any computerised system. Thus, in one case the subject (B) described mainly wanting to use references when writing papers and generating reference lists, and when identifying newly discovered papers that he wanted to read. The system that he designed was clearly intended to support these two tasks. This emerged from the description of the "ideal" systems where subjects were imagining systems that could do what they wanted in one way or another.

When they were presented with the HyperCard framework, the situation became one of looking for a solution to a problem within a specified solution domain (i.e., by means of the various generic objects) of which they had only limited knowledge and experience. In this situation a number of phenomena were observed. Ultimately all subjects were able to identify ways of achieving what they required, but a number of
routes were taken to reach this goal. The routes consisted either of the subjects finding a solution to the problem by adapting old solutions to varying extents, or of the experimenter being consulted about a solution. Three examples of these are given in Figure 5.9. There was no obvious way of categorising these as any number of combinations of events could occur and they seemed to blend into one another to a considerable extent. Thus, in some cases, previous structures or designs could be adopted wholesale with no alteration, while in others they had to be adapted or scrapped completely in favour of a new design. As with the programmer-client study, there was also a vast degree of variation in the detail with which a solution or specification was described.

D) The subject suggested that the cards could be filed in alphabetical order but that they could be switched to subject order. This would be useful for getting all the items related to a subject and noting their titles, etc. The experimenter fed this back to the subject as the desire to sort the cards and to flip through them.

F) The subject specified the fields to go on the cards. There would be one for the date, including month and year, page reference, book or journal reference, volume reference, page reference, title field, a few fields for keywords, and then a field which contained information about what was inside the paper.

B) The subject asked for a way of keeping records of references that are to be recorded but that have not yet been found and read. The way he thought this would work is by means of a separate stack for these. They could then be transferred to the main stack as they are found and read.

Figure 5.9. Three examples of subjects working out a specification from a goal. In the first one, the experimenter rephrases it. In the second, there was a much clearer conversion from needs to a specification, with no intervention from the experimenter. In the last, the specification was actually outside the possibilities of the framework, but the experimenter did not notice this.

Basically it is a matter of finding ways of achieving abstract goals when the previous method no longer works. This tends to involve quite a step of lateral thinking as one must avoid being fixated on the current solution and be fairly conversant with the new solution domain in order to be able to see how it might generate solutions. Subject F tried to employ a strategy using specially designed cards for locating the standard reference cards but this was not possible within HyperCard-. The card index idea seemed to be such that subjects were able to see possible solutions some of the time but also the lack of specificity about what functionality was available meant that they could employ any ideas they had from their general computing experience. This makes the problem extremely difficult for the subject to solve given the lack of experience in the solution domain. That is why in some cases, the experimenter would be better able to propose solutions.
Generally it seemed that subjects were looking to adapt old solutions as little as possible rather than developing entirely new ones (eg. Figure 5.8). There is clearly a danger, in doing this, of coming up with sub-optimal solutions. For example, in one case, the subject (B) wanted to have enough fields for each author to be placed in a separate field. The experimenter asked why this was and the subject described the need to search for each author separately. The experimenter told the subject that the search facilities could cope with this with the authors all in one field, and that this would actually making searching for authors easier.

This study found that many of the specifications could be seen to support tasks or needs that had been articulated previously, when describing either the current system or the "ideal" system. Appendix II has diagrams indicating the relations between the needs and specifications articulated. It was presumed, therefore, that these involved a process of going from a need to a specification that would support that need (task-to-tool). Sixteen specifications were observed, however, that did not seem to support any previously expressed need. In these cases it appeared that the process was more one of the information about the framework stimulating the identification of other needs that could be included within the tool being designed (tool-to-task). This was true both of the stage of designing an "ideal" system and one based around the HyperCard-framework. An example of this was where a subject described facilities for searching for items by keywords and other fields, but had described current systems that did not offer this option. Thus, the implicit question being asked was "How could I use this tool?" rather than "How can I design a tool to support this task?". This is a similar process to that found to occur when shoppers are browsing in the midst of a purposeful shopping trip. The implication is that there are needs that are not articulated because they are not known to be supportable by the tool. Novel information about what the tool can offer can, however, stimulate the articulation of these.

The danger with this approach (Eason, 88) is that the needs stimulated may be very peripheral and constructed from an incomplete analysis rather than derive from real problems that really could be solved by the tool. Thus, one may well suggest that one could do one's home accounts on a computer to justify buying one. This need may prove to have been generated to suit an underlying desire to have a home computer rather than because it is important. At the end of the day, it may prove not to be easier to do the accounts on the computer for any one of a variety of reasons.

If, however, this approach is to form part of how users of JEEVES arrive at a specification, then it needs to be taken account of. The most important element of this
is the process of identifying needs that could be met by the programs. The information provided must, therefore, support such stimulation as well as design, as was mentioned earlier. This and other conclusions are drawn together now in the next section that discusses what has been examined in this chapter and considers its implications.

5.15. Discussion

The studies described have provided a valuable insight into the processes of generating specifications under the circumstances envisaged for users of JEEVES. What is clear from these is that it is more appropriate to examine the issues in terms of the processes that must go on in this activity. Three main processes have been identified. The first is that of stimulation. One of the primary uses of the information provided to Subjects in the studies was seen to be that of the stimulation of ideas about uses of the programs being designed and features that could be included within them. How this can be maximised will need to be thoroughly established. The second process is that of design. The fact cannot be avoided that, if users of JEEVES are to communicate their needs by means of Virtual Programs then they need to be able to design them. The information and structures provided need to support this. What seems fairly clear is that the design of tools and the stimulation of tasks they can support work hand in hand following a Task-Artifact Cycle similar to that described by Carroll (88). Finally, the user must be provided with information about the constraints that exist on the programs that can be generated. Precisely how this is to be achieved needs careful thought. The next three sections discuss each of these areas in more detail.

5.15.1. The Need to Stimulate Ideas for the Virtual Program

There are two main areas within the notion of stimulation. One is that of the stimulation of needs that could be supported by the tool. The other is the stimulation of beliefs about what facilities are likely to be available within its Procedures Library. These will combine to give the user a good basis for generating appropriate program designs.

The first of these will probably be best served by information about the intended task domain of the tool, probably in the form of example programs and scenarios in which they could be used. Problems with this approach have been identified in the studies as users may not understand which aspects of the examples are mandatory and which optional.
Examples could also serve to stimulate the identification of likely facilities that could be expected to be available. They would demonstrate that certain classes of program and program feature exist in the Procedures Library that might allow users to guess other ones. Other approaches to the communication of information about the breadth of available procedures and objects could include a hierarchical referencing system that provides an overview of the contents of the Procedures Library, organised in terms of goals and tasks as well as functionally. This would be available to the user at all times and could be accessed to different levels. Thus, if users wanted they could find out exactly what was available and generate a Virtual Program from this. Alternatively, they could get a general idea and work from that. This would be the ultimate goal of the tool, so that users could include items that they do not actually know exist.

The success of any set of incomplete information about the available procedures and the task domain will depend, ultimately, on users' experience of computers and related tools. If they have none, they will not have exemplars of the general groups of procedures or the ability to recognise classes of items instantiated in an example. Another area that will also rely heavily on computing experience in general is that of the need for the user to actually design the Virtual Program. This is discussed next.

5.15.2. The Need to Support Design of the Virtual Program

It has been seen in the studies described that one aspect of the task of generating specifications is that of finding a way that a given task or need can be supported using the available facilities. This is, essentially, a problem solving task in that a solution must be found from within a limited solution space. This allows existing knowledge to be employed in thinking about this.

One of the most basic principles of problem solving is that it consists of the adaptation of solutions to previous problems. This was seen in the second study where Subjects attempted to generate designs for the HyperCard- referencing system out of old or current system designs. This means that, if users lacks previous experience of designs to support a given task, then it will be hard for them to generate one using JEEVES.

Clearly, this may be easier in an unconstrained programming system where, if necessary, old solutions can be imported wholesale. However, this would not provide the user with much support and would make the task of converting the Virtual Program into a Proposed Program much harder. The alternative, therefore, is to provide users with a set of building blocks and constraints that will lead them to generate Virtual Programs that are close to what can be generated. This will only support users,
however, if the items involved map well onto existing designs, so that they can be employed, possibly in adapted form, to generate new ones.

What is not clear from the studies is whether there are any other forms of information that could support this design process. It may be that by providing a machine model, as one is recommended to do in teaching traditional programming, some of the problems can be reduced. It is likely that this would require the user to build up an understanding of it, for it to be of any value and this requires experience and training. This might well, in fact, be true of all the different items of information that the user would be given as the user has to develop an understanding of how to use them. In general, in fact, the increased load that is being placed on the user by these different elements may well result in a tool that requires some considerable degree of experience before it can be used to anything like its full advantage. This is clearly a problem, given the goal of JEEVES being available to users without this delay. The alternative to this is to reduce the constraints on the user but this will increase the difficulty for the machine, as is discussed in the next section.

5.15.3. The Need to Constrain the Virtual Program

The picture that is now emerging is that of providing the user of JEEVES with information to perform two goals. One is to stimulate the identification of program features that would be both useful and available. The other is to facilitate the conversion of these into the design of the Virtual Program. There is still the danger, however, that some of the aspects the user generates are close to the boundary of what is possible or else actually outside of it. In these cases, there is a need for more information. This section discusses the issues involved in this.

5.15.3.1. What Determines the Boundaries of Possibility

The first thing to be clear about when considering the boundaries of what is possible is exactly what it is that defines these boundaries. One might suppose that they are defined in terms of task domains, so that any given version of JEEVES will be designed to support the generation of programs associated solely with a given domain of tasks such as time management or data analysis. If one considers the nature of behaviour associated with the boundaries, one finds that the matter is not so clear.

Such behaviour requires one to deal with a specification about which one is asking whether it lies within the Program Range or not. When in this situation it has been observed that the basis for an answer lies not in an examination of whether the
specification is appropriate for the relevant task domain but rather in the potential of the available resources to implement it. The boundaries of possibility would appear, therefore, to be defined by the contents of the Procedures Library. This is certainly the situation in traditional programming. Whatever the intended domain of a programming language, if programmers can use the available units to construct programs to perform tasks outside of that domain, then there is nothing to stop them.

With JEEVES, however, the question of whether a specification can be implemented or not does not depend solely on the contents of the Procedures Library. As was made clear in the last chapter, the constraints on the domains of possibility will derive as much as anything from the vast body of domain-specific knowledge that it must possess if it is to serve the user effectively. In particular, therefore, if a specification is given that could, in principle, be implemented with the contents of the Procedures Library but that lies outside of the understanding of JEEVES, then it will not be able to interpret it and convert it into a Proposed Program or Program Proper. This clearly represents a frustrating situation for users and it may be useful to allow the possibility that they teach JEEVES about this feature in some way.

The conclusion to this must be, therefore, that the Program Range is determined by a combination of the task domain (the boundaries of JEEVES' expertise) and the contents of the Procedures Library. This results in a boundary that will probably be complex to communicate. The issues involved in this are discussed in the next sections. There are three basic ways in which constraints can be used. They can be provided before the dialogue, forced upon the user when a specification breaks one, or made available for the user to interrogate during the dialogue. These are discussed in turn in the following sections.

5.15.3.2. Providing Constraints Before the Dialogue Starts

The first question to raise is about the appropriateness of attempting to communicate the constraints to the user before the dialogue starts. Gershman and Wolf (86) decided to make use of the two different types of constraint described above to use two different methods. Information about the task constraints and goals was provided at the outset (user herding), but could be reiterated at any point in the dialogue. More detailed constraints about the limits of the system's abilities were only described when the user made requests that went beyond them (user lassoing). They report finding that, once users understand the task constraints, then they did not tend to violate them. This seems to be a sensible approach given that the extent of the constraints is likely to be unmanageable and considerably extend the time required before starting the dialogue.
5.15.3.3. Unwitting Violation of Constraints

The "user lassoing" involves, therefore, the system dealing with the user's lack of knowledge of the boundaries by monitoring whether the requests made fall outside them. There are three main issues about how JEEVES would do this. The first refers to the process of recognition and the second to the action taken once such an event is identified. It turns out that these are closely related to one another. The third concerns the way JEEVES communicates the problem to the user.

The need to recognise such events was identified in the second study as involving considerable knowledge over and above knowledge of what is possible and the domain of applicability of the programming facilities. Indeed, it is not at all clear that such events would be recognised by any method other than that of actually attempting to implement every specification given. In some cases, it may not be necessary to extend much effort in attempting to implement them before it is realised that it cannot be done.

In others, however, this will be less clear. In particular, there will need to be a stopping rule of some sort. This is because the problem is very similar to that of proving the truth of a statement of the form "one cannot make a cat bark.". It would require the prover to employ any imaginable piece of behaviour near a cat to demonstrate that one could not make it bark (even then, one may have to perform the same test on all cats). For JEEVES to have established that the contents of its Procedures Library could not be combined in some way to implement a given specification would require a very considerable period of testing. What will be needed are heuristics for identifying the appropriateness of a given specification. This, however, involves the interpretation of the specification in terms other than just those of the available code.

Related to the question of how specifications outside the boundaries of possibility are identified is that of when the identification of such a specification is communicated to the user. There are two basic possibilities here. One is that JEEVES is continually monitoring the specifications and, when an unimplementable one is identified, the user is informed immediately. The other is that JEEVES does not monitor the specifications during the first stage but that, upon processing them to generate the Proposed Program, it identifies those that cannot be implemented and this is communicated to the user then. Clearly, JEEVES could be monitoring during the specification but not communicate until the end, but there would seem to be little advantage in that.
The problem with leaving the communication of problems to the end is that many of the aspects of the program design may have been based on the assumption that the problematic aspect is okay. Indeed, if there is a fundamental misconception involved, then the whole program design may be based around it. It would be much more helpful if this was pointed out early on, when the aspect was first communicated. The choice that is made about this will depend, however, on the allocation of resources as this latter approach involves JEEVES in much more work during the dialogue process. What may be possible is some sort of half-way measure so that a superficial monitoring goes on. Using this approach, blatant problems are identified as they occur but subtle ones may still get through and only be picked up at the end.

The final issue for dealing with this sort of problem refers to the way JEEVES communicates the boundary violation to the user and the support that it offers the user to find an alternative specification that is within its boundaries. This is very similar to the situation of the database systems referred to earlier (Motro, 87, and Gershman and Wolf, 86). In these, it was possible to distinguish requests that could be understood but not answered, from those that could not even be understood. In the former case, it was possible to offer the user advice or alternative items of information that might be useful. In the latter, the only option was to state that the request was outside the domain of what could be offered. A similar distinction would probably apply to JEEVES. In some cases, the specification given could be understood to the extent that an alternative specification can be identified that is within the boundary of possibility. In others, however, so little could be interpreted from it that JEEVES would be able to do little more than offer a polite invitation to the user to try again.

5.15.3.4. Giving the User Access to Information About Constraints During the Dialogue

Finally, it may well be that users will, on occasion, identify uncertainty about the constraints and boundaries that apply and want to investigate them themselves. This can take one of two forms. One is where the user just wants a straight statement of the boundaries and this could be accessed much as the list of contents of the Procedures Library is accessed, in an hierarchical manner in increasing detail. The other is more complex and involves similar problems to those were discussed in the last section. This is where the user has an idea about a specification but is not sure that it would be implementable. As was seen in the second study described, the user will want to ask JEEVES whether the specification is possible. It is at this point that the problems arise.
Basically, the task is much as that of monitoring specifications. JEEVES must make a decision about whether an item can be implemented. Whatever techniques were found to support this will be applicable for this situation also. Indeed, if one could be sure that users would always ask before requesting something risky, then the burden on JEEVES would be much reduced. Unfortunately, users cannot be assumed to know what is near to the boundary and what is not, so this is not a possibility. The issues about how the answer to the question is generated will be the same as for unwitting violations of the boundaries.

5.15.4. Can Valuable Models of the User be Provided?

One final point that is related to this concept of "model" is the question of the extent to which JEEVES needs specific information about the user - a "user model". This need arrives out of the realisation that users' specifications will be shaped by their understanding of the Program Range. This means that the interpretation of the Virtual Program will be greatly facilitated and made much more accurate if JEEVES has an accurate model of the user's understanding of the domain.

While a lot has been written about the theoretical value of such models (eg. Gilbert, 87, Payne, 89, Benyon, Innocent & Murray, 87, Murray & Benyon, 88) and about how they can be distinguished from other types of models (Whitefield, 87, Murray, 87), there is really very little evidence that models of this type can be constructed in a way that makes them useful. This can be seen more clearly if one considers the possible avenues for generating them. There are two main alternatives that represent one of the fundamental choices open to Artificial Intelligence in general. When generating an "intelligent" system one can either try to mimic human intelligence or expertise in the domain in question, or generate a more computer-appropriate representation or knowledge base. Which of these is the more appropriate is not being discussed here. It is useful, however, to try to examine each to see whether either could really allow the construction of useful embedded user models.

One of the basic models of advanced "intelligent" human-computer dialogue, as was discussed in Chapter 4, is human-human dialogue. This may or may not be a good model for such interaction. It is clear, however, that humans have many different means of communicating and that they can select from among these in a fairly effective manner. They must, therefore, have some basis for selecting an appropriate means of communicating and interpreting messages received. In this sense, therefore, they are successful systems with regard to interacting with "novices" (i.e., people they have not met before). One might argue, therefore, that computers need merely to be taught how
humans do it to be able to do it as well themselves. There are two questions about this, however. One is the identification of how humans do it and the second is that of getting computers to do the same. There is very little theory of human-human communication in a form that enables it to be formally modelled. This, however, is what is needed if computers are to be able to mimic it. It is debatable that psychology will ever get so far as to be able to understand how humans do this sufficiently for this to form a basis for getting computers to do it. As to whether computers could be programmed to do this once the understanding was available, it is an open question as one would need to know the way humans represented the knowledge to be able to decide whether computers could represent it in a similar manner. It seems likely that this transition from human knowledge to computer knowledge might not be as straightforward as one would like. This is seen currently in the development of Expert Systems where the expert's knowledge has to be transformed into a very clear and well-understood form for the system and this results in behaviour that may not mimic the expert in a very close way.

The second option for designing embedded user models is to use our understanding of human psychology. The argument here would be that we need only understand human psychology to be able to teach computers how to understand their users so as to understand their inputs and generate understandable outputs. There are three problems with this approach. Firstly, there is, once again, the issue of the extent to which human psychology will ever really be understood to the level of detail required. It seems likely that generalisations will be all that will ever be possible given that so much of human activity is shaped by experience and the development of strategies. At many levels, human activity may prove fundamentally unpredictable. Secondly, even if we could achieve this understanding, one must ask how this would help us design computers that could truly understand their users. Clearly, stereotypes might be identified as a basis for categorising users in as intelligent fashion as possible but this will only achieve a limited understanding. Indeed, the more detailed the understanding to be achieved, the more information is required about the user. This could necessitate lengthy interactions where the computer is building up a picture of the user and his/her knowledge before any useful interaction is possible. Thirdly, having understood the user, in some sense, how is this information to be used to generate outputs and interpret users' inputs? Given the likely weaknesses of the model of human psychology and the general categorisation of the user that can be expected, it is very unclear exactly how this is used to allow the computer to act intelligently on the interaction. It would not be hard for the computer to have a number of different languages that were available with a translator between the interface and the information to be communicated, but beyond
this, levels of subtlety and true interpretation of users' communications seems highly unlikely. Indeed, the current realisation with regard to human computer interfaces is that all the knowledge of the user group does not allow the designer to generate an interface that will be guaranteed of success. There is no reason to think that computers will be able to do any better unless we can provide them with better information.

It may well be, therefore, that sufficiently sensitive and useful embedded user models will not be available for a considerable time, if they ever are. In the absence of such models, the interpretation of the Virtual Program by any computer system will be very crude and rely heavily on stereotypical ideas about what is meant by different aspects of Virtual Programs. This will mean that it will demand a lot of the users in terms of checking the interpretations are as they intended and in engaging in clarification dialogues.

5.15.5. Trade-offs and Undecidable Issues

On considering the overall implications of the observations made and their implications, it becomes clear that a dilemma presents itself in the way this first stage of the user-JEEVES dialogue would be managed. This results from the relationship between the amount users are expected to know about the available facilities and the difficulty of interpreting their specifications. Closely related to this is the underlying desire to generate programs that are efficient and well-designed, resulting in a need for JEEVES to take increasing control over the process, as the user is allowed to be less skilled.

Specifying what is required would appear to be a relatively complex task for users to undertake. This can be supported by giving the whole responsibility and control over to the computer. It extracts all the necessary information about the user’s situation and constructs a program that best meets the needs identified. This places an enormous burden on the computer and requires a vast knowledge base. It would only be practicable for well-understood domains with clear boundaries, so that the information that is needed can easily be converted into program types, etc.. In most cases, this is simply not likely to be implementable, even as computer power increases, unless task domains of users can become much more clearly described. The other difficulty with this approach is that of the trust that is being put in the computer. The inevitable feeling of lack of control experienced by users of this sort of this tool could seriously impede its acceptability.

The alternative to this, then, is for the computer to have some expertise along these lines and to be able to advise and direct the program design process but where the user
is much more clearly in the driving seat, defining the needs to be met and, to some extent, how they are to be met. In this situation, however, one must consider how the user can be supported in what has been found to be a complex task for those unused to articulating their needs in this way. The observations made in this chapter suggested that the provision of information about what is available will not be sufficient for this. Ultimately, the user is in the role of designer and this requires skills and extensive knowledge of program designs. The approach of subjects in generating designs based on information provided suggested that they lacked many of the skills of managing this process and of using the information provided effectively.

What must be established is the extent to which the computer could support the design process as it occurs in the non-expert. This would appear to represent a major difficulty for computers without considerable domain expertise and would require subtlety and complex interplay of control and information that would necessitate considerable advances in the fields of Artificial Intelligence. Ultimately, it may be better not to attempt to expect the user to design without skill and rather to be satisfied with generating a battery of advanced tools well suited to supporting design by moderately trained individuals.

5.16. Conclusion

On the assumption that users of JEEVES will want to communicate their requirements in the form of a description of a Virtual Program, the main issues were identified as being related to the communication of information about the Program Range of the programs that JEEVES could generate. The existing empirical work in this area was found to be very scant, so that only a very superficial analysis of the problem was possible. Two studies were described, however, that provided a basic understanding of the processes involved in the generation of specifications in circumstances analogous to those of the users of JEEVES. The results of these suggested the distinction between the different goals of the information that is to be communicated.

The first goal is to stimulate the user into generating requirements and elements of the Virtual Program that are appropriate to the general task domain. The second is to support the user in constructing a design of the Virtual Program that will meet the needs identified. The third is to communicate the inevitable constraints that will exist on the programs that JEEVES can implement. The issues associated with each of these goals were discussed and the scope for finding solutions to the questions raised examined. The most serious problem for the user is likely to be the need to understand the available program facilities sufficiently to be able to design with them. The most
serious problem for JEEVES is likely to be that of identifying aspects of the Virtual Program that are outside of its Program Range.

Finally, it was found that this stage is fraught with difficulty mainly encapsulated in the dilemma of deciding who must ultimately design the program. If the computer does, then it must incorporate a vast knowledge base and the user must trust it. If users must design, then it may be necessary to train them and the support of the computer can only be limited.

The next chapter continues this investigation into the first stage of the user-JEEVES dialogue. It explores the issues involved in the actual means of communication from user to JEEVES.
CHAPTER 6
THE COMMUNICATION OF THE SPECIFICATION

Abstract to Chapter 6

This chapter looks at the means of communication between users and JEEVES. Two approaches to this problem are considered in this chapter. The first is "natural language" communication because it is often seen as the means of interaction that, if only it were possible, would solve the problems of novice or intermittent users. Problems with this are identified in terms of its appropriateness as a means of communicating what is to be communicated. The second is the use of some form of structured notation that might be better suited to what is to be communicated. Some consideration is given to how this might work and a study performed to test its feasibility. It is found that novice users of the notations had to develop new problem solving skills and strategies and that many of these were error-prone. It is concluded, therefore, that there is likely to be a trade-off between appropriateness of a language and familiarity, with no single option that solves all the problems.

6.1. Introduction to the Chapter

In the last chapter a number of issues were identified relating to how users of JEEVES could be helped to generate appropriate specifications. This is, however, only half of the problem for the design of the Stage One of any user-JEEVES interaction. The other half is the actual means of communicating the specifications generated. As has already been mentioned, in traditional languages, the user will learn the language at the same time as the units of which the specification is to be comprised. With JEEVES, however, these two aspects are more clearly separated and can be investigated more or less independently. In this chapter the actual language of communication is examined. This is an area that, in traditional systems of any complexity, requires considerable training to master. If users of JEEVES are to avoid this, careful consideration must be given to the nature of the language used.

Before starting this, however, it is useful to review what is known about how the nature of the specifications that users of JEEVES are likely to want to communicate. The first section covers this.
6.2. What Users are Wanting to Communicate

The underlying theme of the previous chapter was that users of JEEVES can be expected to want to communicate their requirements in the form of the description of a Virtual Program, possibly clarified with reference to goals and tasks. The main issue that emerged from that was the need to find a way of bridging the gap between a specification that was not rooted in any particular means of implementing the program, and the contents of the Procedures Library and JEEVES' domain knowledge. The solution should not involve, however, the user having to acquire a detailed knowledge of such program building blocks. Similarly, it should not require the user to learn a vocabulary of terms relating to such building blocks.

Information available to users would describe the task domain, the contents of the Procedures Library using an hierarchical information structure, and a description of the basic framework within which programs could be constructed. Thus, it would be likely that users would know, in varying degrees of detail, what facilities were available. It would be desirable, ultimately, for users to be able to specify features in more detail than that to which they were actually known. For example, a user might have discovered that various printing facilities existed, but not the details of the options and how they work. This should be sufficient, therefore, for the user to describe the specific printing requirements that should be associated with the desired program. In this way, therefore, one can conceive of the user's task as consisting of communicating the general generic type of facility required (printing, button, field, etc.) based on the knowledge obtained about what is available, combined with descriptions of the specific details that are to make up the facility.

Having established in slightly more detail the nature of what is to be communicated, it is useful also to review what has been learned about how users are likely to communicate it, in general terms. It has been found that subjects varied greatly in the detail with which they described the various aspects of the Virtual Program. Some would be very vague, while others would include considerable reference to underlying data structures. Specifications seemed likely to be generated in a fairly disorganised manner, with no particular order, especially the aspects that were not pre-planned. Related to this, it appeared that the descriptions might not be totally coherent and involve some interactions that were not fully specified.

These aspects combine to form a rough picture of what the language of communication is to support. The next section discusses whether "natural language" would be a suitable candidate for this.
6.3. Is "Natural Language" The Best Means of Communicating With JEEVES?

When considering the design of a computer tool that is intended to be particularly user-friendly, the quality of "naturalness" is frequently invoked. When the computer being envisaged has considerable processing power, or the intention is to conceive of futuristic solutions, then this quality is extended. In particular, it is often assumed that the most "natural" interface would be the one that allows the user to interact with the computer using "natural language". The term "natural language" is understood to mean the same language that humans use in normal discourse (Obermeier, 87).

A number of examples of computers where the goal includes that of a "natural language" interface are described by Heidorn (76), Wahlster et al. (83), and Bobrow et al. (77). One example of the extent to which the use of "natural language" is seen as important in the development of future user-friendly computers is the fact that the Japanese Fifth Generation Computers research and development initiative includes a spoken "natural language" as one of its main aims (Simons, 83). Finally, Morik (83) carried out a survey of people's opinions of "natural language" interfaces for computers and found that most people think they would be beneficial in some way. In fact 88.6% thought that such systems would be put to good use. While this suggests that "natural language" is seen by many to represent the best language of human-computer interaction, there are others that disagree. This section investigates this issue and assesses the extent to which "natural language" is truly the Holy Grail that many claim it to be. It starts with a consideration of the nature of this form of interaction before investigating the advantages and problems with the approach.

6.3.1. Some Properties of "Natural Language"

Many articles discussing the benefits and problems of using "natural language" are very unclear about exactly what they mean by it and assume that it is a generally well-understood term. This is partly because they have not considered the true nature of human-human discourse and have, therefore, a simplistic notion of what it consists of. Before it is possible to establish whether "natural language" would be a desirable basis for user-JEEVES communication, it is necessary to be clear about its properties. There

* The term "natural language" will be used in quotation marks throughout as it is felt inappropriate to consider it as language that is natural, whatever this might mean. The reasons for this will be made clearer later in the chapter.
are three main properties that have been identified by the author. The first is the difference from "textbook" language, the second is the multiplicity of the languages involved, and the third is the multiplicity of channels that can be used.

The first thing to notice about the way people communicate in normal human discourse is that it rarely follows what one is taught about the language in a textbook. Transcripts of spoken English do not read like children's stories where everybody speaks in complete, grammatically correct sentences. The relation between the rules of language and the actual communicative acts observed in the real world is unclear but it is certainly not simple. In fact, it is more accurate to consider the use of language as being governed by conventions that can be agreed globally or locally and to rely on interpretation rather than literal translation. This means that communications of this sort are inevitably ambiguous and rely on the listener's ability to interpret them. This would also apply to any computer that must accept "natural language". It would not be possible to provide it with a dictionary that links words or even phrases and sentences to meanings. Interpretation would need to take place with knowledge of, or at least clues about, the domain of discussion and the user.

Another crucial observation about the means of communication between humans is that it actually involves more than one language and each language will often be quite well-developed for a particular purpose and quite different to others. Thus, English (and other textual languages) is used in a variety of "variants" in different situations (see eg. Bryen, Hartman & Tait, 78). Thus, written English is very different from spoken English, and one talks very differently if one's listener is an adult or a three-year-old. In addition to this, language as it is "naturally" used in human-human dialogue involves more than just textual language. One can use body language and images such as diagrams.

The fact that different languages exist and that these can be used interchangeably in the course of human-human communication suggests an important point about the concept of "natural language". To define "natural language" as the language used in normal human discourse would necessarily have to include all the varieties of language used. What is generally meant by "natural language" is, therefore, a subset of these, usually some particular English language interaction where the user can enter commands in an unrestricted text format to which the computer responds in similar fashion.

The variety of languages available is associated with the fact that communicators will constantly be making choices about which to use. Thus, certain dialects or conventions
are adopted as they suit the listener's ability to interpret them or as the needs of the message that is to be communicated vary. That people do adapt their language in different circumstances and according to their beliefs about the receiver of their communications has been well documented (Gould, Lewis & Becker, 76, Richards and Underwood, 84, Kennedy, Wilkes, Elder & Murray, 88). What is not so clear, however, are the rules and knowledge that communicators employ when choosing a language. The evidence suggests (eg. Galotti & Ganong, 85) that they do not take full account of the needs of the receiver so much as develop some understanding of what is likely to be understood, based on stereotypical ideas and feedback. This is then used to affect the nature of the language used. There are, equally, no definite boundaries between languages, and people will tend rather to adapt their language in various small ways according to their beliefs about the receiver (eg. Richards & Underwood, 84).

One final point about human-human language is that it frequently involves more than one channel of communication. Ones spoken messages will be reinforced by one's body language and expression. Buxton (87) argues that this ability to use parallel communication channels is rarely exploited in the design of computer interfaces. An interface that works purely by "natural language" may well not be exploiting this, either.

These three elements alone demonstrate that language between humans is much more complex than most "natural language" interfaces. In particular, it is not necessarily the basic language of "natural" communication and must be seen in a more dispassionate light. The next section assesses "natural language" in this light and considers its appropriateness as the language of human-computer communication in general and of user-JEEVES communication in particular.

6.3.2. Is "Natural Language" Appropriate for HCI in General?

Another source of confusion with regard to discussions of "natural language" is the absence of any clear definition of what sort of human-computer dialogue is actually being referred to. In order to consider its advantages and disadvantages, one should attempt to highlight the distinction that exists between different interfaces with "natural language" elements. In particular one should not confuse an interface that allows free use of English (or whatever other language) by either the user, the computer or both, as part of the interface and one that attempts to mimic human-human spoken discourse. Many of the comments that apply to the former will also apply to the latter, but mimicking human-human discourse can also raise further issues. Also, one must
remember that what is referred to as "natural language" need not be used by both parties in the interaction, and may not be suitable for all communications made.

Bearing this in mind, the next sections look at the advantages and problems of using "natural language". Finally, the whole concept of "natural language" is evaluated.

6.3.2.1. The Advantages of "Natural Language"

If one cannot argue that "natural language" is the language that is used in human-human dialogue, making it the most "natural" choice, one must ask what the advantages really might be of using it in human-computer communication. Morris (87) lists some advantages and disadvantages of "natural language".

The first and most obvious of these is its ease of use. The argument basically is that humans already have a language with which they are familiar. If they can use this to communicate with the computer, then they are saved the labour of learning a new language. Closely related to the ease of use is the increased expressiveness of the language. The fact that the user will already be familiar with an extensive language means that he/she can readily express a vast variety of different concepts and needs. Two other advantages cited for the user of "natural language" is the increased ability to provide good explanations and to model the user. These seem to be bi-products of the necessary complexity of the whole system and not a result of using "natural language" per se. Finally, if the interaction mimics human-human communication, then there is the potential for great flexibility in the structure of the interaction. Thus, either party could take control over the topic or add comments in a very unconstrained manner.

The next section describes the disadvantages of using "natural language". In addition to finding problematic features of this form of interaction, many of the advantages described above are further analysed and found to be flawed.

6.3.2.2. Problems with "Natural Language"

That "natural language" is ambiguous and imprecise is well-established (Gould et al 76, Miller 74). This can be compensated for, to some extent (Galotti & Ganong 85, Chin 84), but never fully eradicated. It is worth noticing that it is not just in human-computer interaction that a desire for precision has led to the development of "unnatural" languages. Hill (83) points out that professions such as Law have developed languages that are extremely unnatural to the uninitiated in order to enable the unambiguous stating of contracts, etc.. This ambiguity affects the user of a computer
either by resulting in undesirable computer behaviour or by the need for clarification
dialogue (Shneiderman, 87). Some ambiguity may have to be accepted as inevitable in
user-JEEVES interaction if users are to be able to phrase the details of their
specifications without the use of a pre-defined language.

Related to the ambiguity of "natural language" is its verbosity. Rich (85) points out that
such a means of communication will often result in very unnecessarily lengthy
dialogues when working in simple constrained domains. The commands that must be
given can often be phrased using combinations of short "words" rather than lengthy
sentences. In particular, "natural language" is a very inefficient means of
communicating between users and humans in that it does not take advantage of the
facilities available. The screens could be used to transmit information much more
efficiently than just by lines of text (Shneiderman, 87).

This represents one example of the general observation that "natural language" is often
not the best means of communicating about a topic. This was observed in human­
human dialogue where diagrams and gestures can be resorted to for certain messages.
Any system that solely provided "natural language" for communication would result in
many communications being very inefficient. It would be more valuable, therefore, to
start from an analysis of what is to be communicated and then to consider how best this
can be done. It is worth noting that the flexibility of the mixed initiative dialogue that
was described as an advantage above can also be disadvantageous if the topic of
discussion requires structure. Similarly, the fact that once something has been said it
cannot be unsaid can have negative consequences in many situations (Shneiderman,
87). Once again, the dialogue needs to be designed to suit the task rather than just to
mimic some dialogue mode that seems to work in other circumstances.

The observation that "natural language" is misleading for users has been made by many
workers (eg. Potts, 88). This seems to arise because the ability of the computer to
converse intelligently suggests to the user that it is intelligent. This can include beliefs
that it really does understand what is being communicated and that it can do things that
are outside of its scope. In all cases, computers do not really understand the concepts
they can use in the same way that humans do and they will only be able to interact with
them in limited ways. Thus, an automatic programming system that can solve queuing
problems involving trucks (Heidorn, 76) does not really know anything about trucks
other than that they are mobile and can form queues. In particular, it could not answer
questions that do not pertain directly to the queuing problem. The use of "natural
language" in this system does not communicate any of this to the user and, indeed,
suggests the contrary. This is, essentially, the flip side of the expressiveness advantage of "natural language". Its very expressiveness is likely to suggest to the user that the computer can understand everything that can be expressed.

If, however, one succeeds in communicating to users that the computer is not human in its ability to understand the dialogue, the language does not provide any clues about what can be assumed about it. "Natural language" assumes shared culture and, in the absence of this, one can be left not knowing what is possible.

There are clearly some serious problems with the use of "natural language" in human-computer interaction. There are also problems with the concept of "naturalness", in particular as it is applied to "natural language". The next section discusses this and suggests a better criterion for language selection.

6.3.2.3. Problems with "Natural Language" as a Concept

Given that "natural language" is not the Holy Grail that it seemed to be, how does one go about selecting the best language for a particular interaction? Buxton (87) and Rich (85) discuss the meaning of "natural" and their work can be combined to make the point that no language can be given the label "natural" in isolation from the topic of communication and the intended user group.

Rich makes the point that "naturalness" is not a property of a language in isolation but a property of the relationship between a language and a task domain. Buxton points out that "naturalness" does not suggest that the user of a "natural" language was always able to use it to communicate. There must always have been a time when it was learnt. Not everybody knows the same languages so that what is natural for one person may be very unnatural for another. Thus, "naturalness" is a quality of the relationship between a language, a topic of communication, and the experiences and knowledge of the users. Indeed, it is often important to consider the environment of the interaction when designing user-computer dialogues, also. Under certain extreme conditions, some forms of interaction may be totally inappropriate and, therefore, quite "unnatural".

Any analysis of the best language for an interface must, therefore, take account of how well it allows the expression of what is to be communicated and of how easy it is for the user to use. A language such as "natural language" may be easy to use, in some respects, but it has many weaknesses and may well not be well-suited to a given domain of communication. Similarly, traditional programming languages are well-
suited to communicating program specifications but require considerable training to be used. What must be found, therefore, is some sort of middle ground that is sufficiently easily grasped, yet suitable for the communicative needs of the situation. The next section explores the potential for this by examining one possible approach to the means of communicating.

6.4. What Language Would be Appropriate for Communication With JEEVES?

The question of the most appropriate means of communication between the user and JEEVES can be approached at a number of levels. This section explores these to varying extents and considers the potential of the various options available in each case. The first option to consider is that of designing JEEVES so that the user-JEEVES dialogue mimics that which is observed between clients and human experts. This could be said to represent the most "natural" approach and might require the minimum of learning for the user. A number of disadvantages of this approach have been discussed however and they will apply in this case as much as in any other. Users will need the dialogue to have structure while allowing some degree of mixed initiative. The dialogue should make it evident that JEEVES is not as capable as a human expert and should offer users clear information about what has been understood by the specifications. Finally, the dialogue should be designed to take advantage of the available communicative channels and this would probably result in quite a different dialogue being appropriate.

Having said that, there are likely to be elements of the client-expert dialogue form that are worth applying to the design of the user-JEEVES dialogue. These are drawn on in the following discussions that attempt to generate an understanding of the issues that determine what the dialogue should look like. The goal of this is not to identify actual answers but rather to identify the main areas that will need to be addressed and explore some key questions about how users might communicate their specification. The basis for this is the observation that users will basically be specifying the design of the Virtual Program. This suggests a useful framework for considering this dialogue, which is described in the next section.


The first stage of the user-JEEVES dialogue seems likely to consist of the user designing a Virtual Program. This suggests that one can consider the dialogue in the light of concepts that draw from more traditional programming tools. In particular, such tools embody two different languages of interaction. On the one hand,
Programmers are constructing the description of programs in programming languages of some sort. Thus, they must be able to represent their program designs in terms of these. At the same time as having to work with this language, programmers must employ an editor or programming environment for actually putting the design together. This represents another level of dialogue that they must master. This distinction is, essentially, the same as that between any language and the channel for constructing messages in that language.

Given this distinction, one can examine the particular issues that arise for each element of the interface design. Each will need to be designed both to minimise the learning required of the user and to facilitate the communication in general. This results in a number of areas that need to be considered in the design of each element. Finally, it is worth remarking that there will be an inevitable interaction between these two elements. For example, the design of the language of specification will influence how the environment for specification is conceived. These various factors are discussed in the next two sections.

6.6. An Environment for Specifying the Virtual Program

There are many decisions that must go into the design of any environment for supporting users constructing computer-based artifacts (program, text documents, drawings, etc.). It would not be appropriate to attempt to describe all of these in this thesis. Rather, the main issues are discussed insofar as is possible in such an abstract domain. It would only be when the details of the programming domain are clear that one could really hope to consider answering these questions. A number of people in the area of Human-Computer Interaction have attempted to articulate the important qualities that must be taken into account in the design of any interface. Thus, the user must be given feedback about what has been understood and clear information about the machine state. This is discussed in the first section. Following this, issues more relevant to the specific domain are considered such as the need for mixed initiative and the ways in which the process of specifying the Virtual Program design can be supported.

6.6.1. The Provision of Information About the Computer

Users of computers need good, clear feedback in response to their actions. This is generally accepted (see e.g. Gaines & Shaw, 84, Shneiderman, 87). This feedback can take a variety of forms, however, and vary in complexity. The most immediate feedback that is necessary is to inform the user that the input has been received. In
addition to this, it is likely to be important for the user to know whether it was accepted or not. Feedback about unacceptable input must carry information that could help the user to generate more acceptable input in future. Beyond, that, however, it may also be important for the user to have access to information about how the input was interpreted. This is particularly true in any environment where, as with JEEVES, the user is using terms that have not been given pre-defined associations with computer behaviour.

The degree of this sort of feedback will depend, to some extent, on the degree to which it is processed at the time of input. As was discussed in the previous chapter, the Virtual Program may not be examined for implementability until the end of the first stage of the dialogue. Other relevant issues are about how unacceptable inputs are dealt with by JEEVES, and how clarification is obtained of inputs that cannot be interpreted. These issues would need to be decided by empirical testing once a prototype of JEEVES was established.

A second guideline that is frequently quoted for designing interfaces refers to the provision of information about the current state of the interaction. In particular, the user must be given clear signals about:

* the state of the computer, possibly in relation to any modes it may have;
* the current activity of the computer, especially if the user must suspend inputting activity to wait for it to finish, and;
* the current expectations on the user and goals or tasks that are being addressed.

The requirement for JEEVES to provide users with information about its state and about its acceptance and interpretation of inputs may seem simple to implement. In reality, however, the matter is not quite so easily resolved. Both types of information need to be well-designed so that they are readily interpreted by users. Unless some sort of "natural language" means of communicating these can be developed, users will have to learn to be able to understand what each message means. Even if "natural language" could be used, it would still require some interpretation in the context of the interaction. There is clearly a burden on the user of the system to become familiar with such messages until such time as they can become interactive. At that point, users could demand clarification dialogue of JEEVES until they were happy with the content.
6.6.2. Control and Support for the Specification Process

One of the central issues in the design of an interface refers to the locus of control over the course of the dialogue. In traditional programming tools, the programmer is in control but can be constrained by the design of the environment. In tools aimed at end-users, the computer is often in control, asking the user questions or offering menus of choices, so that the dialogue follows a fixed course. It is generally recommended that tools aimed at non-expert users take over much of the control of the dialogue (e.g. Damodaran, Simpson & Wilson, 80). The argument for this is that such users will not be in a position to exercise control effectively and need to be protected from the effects of their lack of training.

The removal of all control over the specification process was discussed in the previous chapter and found to be problematic. It seems sensible both to guide and to constrain users of JEEVES insofar as seems appropriate, given their likely lack of expertise with JEEVES. It seems important, however, to allow users some control, in particular over the process of specifying the Virtual Program owing to the apparently disordered manner in which specifications are likely to be generated. In practice this issue may well prove to be rather harder to resolve than it might appear. If inexperienced users are to be given control over the process of specifying the Virtual Program, then they will need considerable support in this from the tool.

The nature of the support that can be provided was examined in Chapter 3 and found to be limited. Thus, menus could be used to facilitate the generation of objects to go into the specification, but these rapidly became unwieldy if too many options were involved. Examples could be made available but, again, the more there were the harder to manage. In the previous chapter, the need for users to be able to interrogate the contents of the Procedures Library in an hierarchical manner was mentioned. Similarly, it was found to be important for users to be able to ask about the constraints that existed on possible program designs. The difficulty with all these is that the interface rapidly becomes complex and hard to manage, rather than reducing the demand on users.

It would appear, therefore, that JEEVES would need to provide "intelligent" support of some form that would cater for all these needs in a manner that did not weigh the user down. As with the need to provide information about the state of the machine, JEEVES would have to offer the support through another interface that could cope with possibly informal requests for help, advice and clarification. The technology currently available has not really got as far as would be required for the users of JEEVES.
In conclusion, therefore, it would appear that, not only must JEEVES provide users with the ability to specify a Virtual Program without having previously learnt a programming language, but it must also support them through the whole process. This has already been seen in the problems of designing the Virtual Program, but it appears also to be true for communicating the design to JEEVES. The next section looks in more detail at the actual language that would have to be used for representing the Virtual Program and considers the demands that this will put on the user and the extent to which they can be avoided.

6.7. The Language of Specification

The actual representation of the specification could take any number of forms. One of these could clearly be unconstrained prose. This would have all the advantages of freedom of expression etc., but also a number of disadvantages. These relate to what is required of a good language for communicating something like a specification for a program. This section discusses these factors and suggests how they might be addressed in this case.

There are two immediate goals that the language chosen must achieve. One is to provide users with a clear idea of what is required of them, and the other is to facilitate the expression of whatever is to be expressed. If users had to describe the Virtual Program in straight prose, they would have no clues about where to start or what information was required. It would be likely that they would require more structure than this to help them formulate appropriate statements of their specification. Closely related to this issue is the need for the language used to support the expression of the particular concepts that are to be expressed (Jackson, 83). The very freedom of expression of "natural language" provides minimal support in this way and is remarkably ill-suited to expressing many things.

The question becomes, therefore, one of identifying what is to be expressed in the task of specifying the Virtual Program. It has been suggested that users of JEEVES be given information about the task domain of the tool, descriptions of a framework for programs and of generic objects of which they can be comprised, and have access to hierarchically structured information about the specific options that are available for these. This means that programs will consist of specifications of the objects required, their interrelationships, and the details of their functioning. The details of the generic objects could be specified in the same way as the whole program, by means of further generic objects, etc.
While prose could be used to describe the detailed functioning of the generic objects chosen, when this is not in terms of other generic objects, the interrelationships and choices of objects would not be well represented by prose descriptions. Structured notations would be considerably better at this. Evidence for this comes mainly from work on how people can use information presented in different formats to solve problems (eg. Wright and Reid, 73). This supports the view that structures such as tables or flow-charts are better at communicating certain sorts of information than prose. Also, some suggest that diagrams are less ambiguous to use than prose in this sort of domain (eg. Potts, 88, Buxton, 87).

This suggests, therefore, that one may be looking for a more structured notation than prose that will allow the explicit representation of the relationships that exist between the objects of the program. The particular notation used would depend on details of these relationships as well as on user familiarity. This immediately provides other important needs of the language. It must aim to make it clear what information is required of the user. The user must clearly indicate what objects are required, how they are to relate to one another and what the details of the functioning of each are.

Generic objects could be chosen from menus if there were not too many of these, and more advanced facilities for helping users choose the appropriate objects might ultimately have to become available. Programs could be specified at any number of levels of detail, with different detail given about different elements, and it all being specified in the order that suits the user. Finally, it would probably be necessary for users to be able to associate other information, such as descriptions of the tasks that certain parts are to support, etc..

Other goals of classical programming languages are to make certain aspects of the notional machine clear to users. Thus, they should encourage them to develop a good understanding of this machine and how it works, they should provide them with information about the available options and possibilities, and they should make clear how each communication will be interpreted.

These goals are only appropriate to a limited extent in the case of JEEVES. Much of the way JEEVES will be designed should be aimed to provide a minimal model of the machine that will support users in the development of programs. They cannot be expected to work with a complex or abstract notion of the machine that is being programmed as this would involve them in a considerable assimilation burden. Similarly, users are not expected to acquire a detailed understanding of all the available...
options. The language used would only hint at the possibilities and more specific information would be obtained by interrogating the Procedures Library. Finally, the language cannot guarantee to make clear how each item will be interpreted as this would suggest that there was a simple relationship between inputs and computer behaviour that must be learnt.

All this will have to be in the context of a tool that will, as has been discussed, provide the user with considerable support in the design process. In the absence of this, users will not be able to generate appropriate specifications for Virtual Programs, even with suitable notations. Despite this, however, users will have to acquire the ability to express solutions in such notations after limited experience with them. The problems with this have not been investigated to any real extent. The next section aims to redress this and to obtain some basic information about whether this approach could work.

6.8. Empirical Study of the Use of Structured Notations by Novices

It is suggested, therefore, that users of JEEVES could communicate the specification of the Virtual Program using representations of generic objects. These could be located within a network of relationships represented using some structuring notation. The descriptions of the details of the objects would somehow be associated with them, possibly including the notation to describe detailed aspects of each object. While this appears to provide a sensible means of communication of the information involved, it is uncertain whether it would be sufficiently user-friendly. One might argue that it consists of little more than a slightly more formal way of expressing the specifications that subjects have been found to give in previous studies. Given the proven ability to people to adapt to new variations on language, there might be little problem for users with such an approach. On the other hand, the step might be too great for users to manage easily, requiring considerable learning before they can use such a notation with ease.

It seems important, therefore, to examine the ease with which people can adapt to such notations and methods of expressing concepts. It seems likely that the best way of communicating a specification to JEEVES will involve some structuring of the description from a prose-based one. While this particular approach may not, ultimately, prove to be the best suited to the representation of the specification, it constitutes an example of the sort of representation that might be used.

The method of representing the Virtual Program described above is very reminiscent of many notations that currently exist to represent instructions or system designs such as
Flowcharting and the notations used in the CASE tools described in Chapter 3. With the appropriate choice of domain, it would be possible to use such notations as a means of examining how easily such an approach could be adopted by users. The use of the notation could be separated, therefore, from concerns about understanding the nature of the generic objects or the program structure involved. If such notations could be acquired with relative ease to represent familiar concepts then it would support the use of this approach in JEEVES. If, however, there are problems with this, then an examination of these problems will furnish valuable material for identifying the issues that still need to be resolved.

Work that has been done on the use of structured notations mainly explores the ease of information extraction associated with different ones (eg. Green, 82) or the extent to which they support problem solving (eg. Wright and Reid, 73, Sheppard, Kruesi & Curtis, 80). The conclusions from these studies tend to be that different notations are good for different tasks, and that none are good for them all. They then go on to consider what aspects of a notation make it suited to the extraction of what type of information. In particular, Sheppard et al. (80) made the important distinction between symbology (the language used to express individual statements) and spatial arrangement (the facilities provided to express flow of control). The subjects were professional programmers and were given extensive training in the different representations used before the tasks. The tasks involved answering problems about three different type of program relating variable values to parts of the programs' execution or giving the output associated with a given set of input values. They found that natural language resulted in longer problem solving times than more constrained forms of symbology, on three different task types. The spatial arrangements used consisted of a simple sequential arrangement (as used in prose), a branching arrangement (as used in flowcharts) and a hierarchical arrangement (as used in Hierarchical Task Analysis). The branching arrangement was found to be the best for all task types but the hierarchical arrangement was not significantly worse when the problem involved finding the states of certain variables at a given point in the program (i.e., tracing backwards).

Little has been found about the process of generating descriptions using these notations, however. The only exception to this is work by van der Veer, van Beek and Crut (87), that looked at how well novices from different backgrounds could use structured diagrams as a means of representing program algorithms. This combined the tasks of finding an algorithm for a program with representing it in the notation. Only the latter task is of interest here. Also, the focus was on how to help subjects with non-
mathematical backgrounds and did not go into any detail about how structures were generated.

A study was performed, therefore, that looked at this. In particular, it attempted to get some impression of how subjects new to a notation can use it and the sorts of strategies they use.

6.9. Method

6.9.1. Overview

Overall, the experiment consisted of subjects:

- being given a description of a notation,
- performing a series of short tasks that are intended both to measure some aspects of notation use and to provide practice in using the notations,
- carrying out a major final task where they have to generate a set of instructions in the notation.

There were two conditions: Flowcharting or Jackson Structured Diagrams (JSD) according to the notation that were used throughout.

6.9.2. Subjects

The subject group was deliberately kept small owing to the extensive nature of the analysis that was to be carried out on the subjects' responses. This was to be of a more detailed nature rather than general performance measures. It was decided to use a diverse subject group so as to be able to comment on the effects of various traits on performance. There were nine subjects in each group, allocated randomly. These were mostly students (5 undergraduates and 8 postgraduates) but with 5 others not attached to the university. A questionnaire was used to obtain a subject profile and ensure that the groups were comparable. Although the JSD group appeared slightly more experienced at computing, the differences did not appear substantial (see Table 6.1).

6.9.3. Notations

The study examined the use of Flowcharting (similar to the branching spatial arrangement used by Sheppard et al. (80) with unconstrained symbology) and a variation on the hierarchical arrangement known as Jackson Structured Diagrams (also
with unconstrained symbology). This was intended to explore further the different strengths of these two spatial arrangements within tasks more closely related to those that a user of JEEVES would have. An example of a small set of instructions in each notation is given (Figures 6.1a and 6.1b).

<table>
<thead>
<tr>
<th>Number of subjects (out of 18) who:</th>
<th>All</th>
<th>ISD</th>
<th>Flowcharting</th>
</tr>
</thead>
<tbody>
<tr>
<td>Had never used a computer</td>
<td>1</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>Had spent more than 80 hours on one</td>
<td>14</td>
<td>7</td>
<td>7</td>
</tr>
<tr>
<td>Had never programmed</td>
<td>5</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>Had programmed for more than 80 hours</td>
<td>7</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>Had not used any notations</td>
<td>3</td>
<td>0</td>
<td>3</td>
</tr>
<tr>
<td>Had never had to write instructions</td>
<td>4</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

Mean Cooking experience (subjective rating - out of ten):

<table>
<thead>
<tr>
<th></th>
<th>All</th>
<th>ISD</th>
<th>Flowcharting</th>
</tr>
</thead>
<tbody>
<tr>
<td>General</td>
<td>6.4</td>
<td>6.8</td>
<td>6.0</td>
</tr>
<tr>
<td>Cakes</td>
<td>4.1</td>
<td>3.9</td>
<td>4.3</td>
</tr>
<tr>
<td>Lasagne</td>
<td>3.6</td>
<td>2.6</td>
<td>3.8</td>
</tr>
</tbody>
</table>

| Table 6.1. The subject profiles as obtained from the questionnaire. |

Figure 6.1a. An example of a diagram in the Flowcharting notation. It represents the instructions: "Add the milk. Stir the mixture until it is smooth."
Finishing the mixture

Add the remaining milk

Is all the milk added?

Yes

No

Stir the mixture

Until the mixture is smooth

Figure 6.1.b. An example of a diagram in the JSD+ notation. It represents the instructions: "To finish the mixture off, add any remaining milk and stir it until it is smooth."

The Flowcharting notation consists solely of Process Boxes, Decision Boxes and arrows. Small round boxes with no contents were also used with the Decision Boxes when they were used as loops. These were due to the limitations of the computer package that was originally going to be used to generate the diagrams. The rules of using Flowcharts are simply to follow the arrows, sometimes with the choice of arrow being determined by the answer to the question in the Decision Box.

The variation on JSD made a strong contrast to Flowcharting. It has quite different properties such as the groupings that are allowed and the method of following a diagram. The particular form was chosen to maximise the ease of use. This was based on the considerations discussed in Green (82) and on a pilot study that identified problems with the form shown in Figure 6.2a so that the version shown in 6.2b was preferred. Also, it is nearer to Flowcharts, having Decision Boxes. This type of box was made more explicit by the use of different shapes. Decision Boxes actually work much like Flowcharts with the control being passed down, as if following the lines.

A more detailed description of each notation is contained in the rules the subjects were given (see Appendix III). This included information about how the notations were to be followed and gave examples of structures.
6.9.4. Tasks Used

It has been pointed out (Green, 82) that problems may not be apparent when solving problems with small structures but become major issues when dealing with really large complex structures. Wright & Reid (73) also found that the difference in performance between prose and structured information in solving problems only appeared when they were complex. This study looked, therefore, at Flowcharting and JSD with relatively large structures and problems of both reading and writing.

There were four main sets of tasks. The first two were intended to provide practice for the subjects in using the notation. The third set was to explore the subjects'
understanding of how the structures were to be read. The fourth was a larger task involving the conversion of a complete set of instructions into the notation.

The first task group related to a Flowchart or JSD of a recipe for making lasagne. These were of a considerable size covering a number of sheets of A4 paper. They are shown in Appendix III. Subjects were asked to imagine a scenario where they had a robot chef that accepted instructions in the notation specified. A scenario was suggested where the subject had noticed that the lasagne had not been coming out very well and that some changes were needed to the instructions.

1) Changing the order of execution
Some of the mistakes in the Lasagne are due to actions being performed in the wrong order. You will have to decide what the correct order should be. To indicate the correct order, circle the boxes to be moved and put an arrow to show where they should go. Also, add any links that need adding.

a) The first steps of the instructions constitute an ingredients list of sorts. This is rather disorganised. You want to arrange it so that your chef will request dairy products together and vegetables together. Thus, put milk, butter and cheese together and mushrooms, pepper and onion together.

b) You have told your robot to put the lasagne in the oven and to cook it before actually putting the layers of mince, lasagna and cheese sauce into the dish. Correct this.

c) You don't like the cheese sauce it makes at present. You think this might be improved by adding the cheese before thickening the sauce. Alter the instructions accordingly.

2) Adding some elements
Some of the mistakes are due to actions being missed out. You need to decide what needs adding and where. To indicate what you have decided, put a line through any boxes you want to delete and draw in any additional boxes with their links.

a) You remember that you were told, when making a cheese sauce, always to take the pan off the heat before adding the milk. Maybe this is the problem with the sauce, so add this step.

b) Another problem is that the mince sauce is rather tough. You trace this to the fact that the instructions do not include simmering it at all. This needs to be done once all the ingredients for the mince sauce have been added. It involves covering the pan and cooking on a low heat for 3 hours.

c) Well, it is hardly surprising that the dish is a mess! You haven't given instructions for turning the oven on (well in advance) or greasing the dish. Add these.

d) The last problem with the mince sauce is probably due to the fact that onions need longer to fry than mushrooms. Alter the instructions so that the onions are fried for about five minutes before adding the mushrooms.

| Figure 6.3. Instructions for Tasks 1 & 2. |

200
The first set of tasks (Task 1) involved subjects making changes to the order of execution of the actions, but not actually adding anything. Task 2 was similar but with subjects having to add steps. The instructions for these are shown in Figure 6.3.

Task 3 involved the subjects being given segments of Flowcharts or JSD's and writing down the order in which the boxes are examined and executed. The subjects were reminded of the rules they had been given about how the notations were to be followed and it was suggested that they use these. It was recommended that, for each segment, the subjects read through it before writing anything in order to make sure they understand how it worked. The subjects were provided with suggestions about how they might present the description of the order of execution. This was because subjects in pilot studies seemed to be having difficulty understanding what was expected of them. The suggestions came in the form of examples (Figure 6.4). The intention with these was to give some idea of how the flow of control might be communicated without giving too much away about how the flow of control is worked out. Because of this, they were only shown to the subjects at the start of the task.

---

### JSD Examples

- execute box 1
- execute box 2

  - answer question in Decision box 3
    - if the answer is Yes, execute box 4
    - if the answer is No, execute box 5

  - examine condition in Repetition box 6
    - if it holds move on
    - if not execute box 7
    - return to Repetition box 6 and repeat the process

---

### Flowcharting Examples

- execute box 1
- execute box 2

  - answer the question in Decision box 3
    - if the answer is Yes, execute box 4
    - if the answer is No, execute box 5

  - examine the condition in Decision box 6
    - if the answer is Yes, move on
    - if the answer is No, return to box before Decision box via box 7

---

*Figure 6.4. The examples given of how to describe the sequence of examination and execution of Flowcharts or JSD's, used in the Task 3. These were given without the corresponding diagrams.*
The first diagrams were taken from the lasagne instructions and had the actual instructions included in them plus random numbers for the identification of the boxes (see Appendix III).

In Task 4, subjects were given a recipe for a cake written in numbered paragraphs (see Appendix III). Subjects were to translate the recipe into the notation, a paragraph at a time, each on a separate sheet of paper, in a specified order. The order was merely to ensure that the more interesting paragraphs were actually attempted given that time was limited. If subjects completed all the specified paragraphs, they were to combine them into a complete set of instructions.

6.9.5. Procedure

The experiment started with subjects being introduced briefly to the experiment and its purpose. It was emphasised that it was the notation that was being tested and not the subjects. Subjects were then given an introduction to the notation which they read. They were told that they could refer to this at any time. Any questions were answered. Subjects then read the top sheet of the booklet containing the tasks and went over the instructions for lasagne (in JSD or Flowchart - see Appendix III). Subjects were encouraged to make as few mistakes as possible rather than perform as fast as possible when carrying out the tasks.

The different parts of the first group of tasks were carried out using different coloured pens. Subjects made changes on the actual lasagne instructions. Subjects would often be given feedback about their performance on these tasks but this was not done with any consistency of approach. This clearly weakens the possibility of comparing the groups. The time was limited to 20 minutes. The second task group was then carried out. They were given as many as they could do in 20 minutes (10 minutes on those with text, 10 minutes on those without). Finally, the third task group was performed which was allowed to go on for up to 25 min. At the end of the experiment, subjects were given a questionnaire about themselves and their experiences in the session.

6.9.6. Data Gathering

In order to obtain subjects, it had been necessary to state a maximum length of time that the session would take. This resulted in each stage of the sessions being strictly limited in time allowed. For that reason, times were not taken other than to ensure subjects did not run over. The aim in analysing the data was to obtain clues about the strategies being used, rather than to hope to find statistically significant differences between the
groups, etc. To this end, various specific measures were taken of each task, mainly from the subjects' written answers. This mostly consisted of judging whether the subjects' answers were correct and, if not, what the problem was, in terms of a number of categories (see below). The questionnaire asked about the subjects' experience with computers, programming, using notations in general, writing instructions, experience with the notation used, experience with cooking (including special reference to lasagne and cakes), and difficulties encountered. Notes were also made by the experimenter of the subject's sex, any questions asked and any problems encountered or comments that seemed important.

6.10. Results and Discussion

The results of this study took a variety of forms and required considerable analysis. There were some initial general observations that were made to put the other observed behaviour in context. There was a large body of error data from the tasks involving subjects generating or altering diagrams. Finally, there was the data from the tasks where subjects had to indicate the flow of execution of a series of diagrams. The performance observed could also be examined for associations with the factors of the subject profiles obtained.

6.10.1. General Comments

The questionnaire gave a profile of the subject group in terms of some relevant details. The results of this are summarised in Table 6.1.

Despite being told that they could refer to the description of the notation at any time, no subject looked at it again after the initial reading.

6.10.2. Patterns of Errors Made in Using the Notations

The errors made by subjects constituted the largest part of the data that was obtained. While each could be looked at individually in its context and some understanding could be achieved through this, it was only when categories of errors were identified that it was possible to generate observations that could lead to generalisable predictions. Before describing these categories, some basic frequency data is presented to indicate the extent to which errors were made.
6.10.2.1. Number of Errors Made

Before considering the nature of the errors made, it is important to establish whether the number of errors indicates competent or incompetent use of the notations. If the subjects were basically producing perfectly good answers with only sporadic slips, the situation would clearly be quite different from that where subjects were making errors every time they wrote something.

Table 6.2 summarises the frequency of occurrence of errors in the three task groups involving generation of structures, rather than reading in Task 3. The total number of errors made in all the tasks came to 79 for the JSD condition and 59 for the Flowcharting condition. Thus, for the final task, subjects were generating an error every 6.66 boxes in the JSD+ condition and every 14.25 boxes in the Flowcharting condition. While this suggests that Flowcharting is better because it produces fewer errors, both groups made errors at a level of frequency that cannot be ignored.

<table>
<thead>
<tr>
<th>Tasks</th>
<th>JSD+</th>
<th>Flowcharting</th>
</tr>
</thead>
<tbody>
<tr>
<td>1a</td>
<td>3</td>
<td>6</td>
</tr>
<tr>
<td>1b</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1c</td>
<td>6</td>
<td>4</td>
</tr>
<tr>
<td>2a</td>
<td>6</td>
<td>11</td>
</tr>
<tr>
<td>2b</td>
<td>8</td>
<td>3</td>
</tr>
<tr>
<td>2c</td>
<td>8</td>
<td>5</td>
</tr>
<tr>
<td>2d</td>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>43</td>
<td>26</td>
</tr>
<tr>
<td>Total</td>
<td>79</td>
<td>59</td>
</tr>
</tbody>
</table>

Table 6.2. Frequencies of errors made by subjects in different conditions.

Ultimately, the purpose of the study is not just to be able to state which notation is better but to understand something of how they are used. There may be aspects of each that result in problems and this information could lead to the identification of factors that would lead to a notation that is better than either. Also, given that neither is really a candidate for use in JEEVES, the aim is to discover generalisable concepts about notation use that could be applied to whatever specific problem is to be dealt with. The next sections describe, therefore, the different types of errors observed, and discusses what the data suggests about how the notations were being used. The data are, again,
drawing on the subjects' behaviour in all the tasks. The frequencies with which the
different error types occurred throughout the tasks is summarised in Table 6.3.

<table>
<thead>
<tr>
<th>Error Type</th>
<th>JSD+</th>
<th>Flowcharting</th>
</tr>
</thead>
<tbody>
<tr>
<td>Looping</td>
<td>14</td>
<td>9</td>
</tr>
<tr>
<td>Under/Adjacent</td>
<td>13</td>
<td>N.A.</td>
</tr>
<tr>
<td>Double Repetition</td>
<td>3</td>
<td>N.A.</td>
</tr>
<tr>
<td>Mislocation</td>
<td>1</td>
<td>12</td>
</tr>
<tr>
<td>Logic</td>
<td>11</td>
<td>9</td>
</tr>
</tbody>
</table>

Table 6.3. Summary of the frequency with which the main error types occurred across tasks 1, 2, & 4. Some errors were not categorisable into any of these types.

6.10.2.2. Looping Errors

One very common error for users of both notations (14 occurrences with JSD, 9 with Flowcharting) was associated with actions to be repeated or carried out over a period of time (labelled "Looping Errors", see Figure 6.5). In most cases of this, the problem occurred when there was a preparatory action (e.g. "add milk") that preceded action(s) to be repeated (e.g. "stir"). The error made was that of creating a structure where both the preparatory action and the action to be repeated were repeated. Another case when Looping Errors can occur was observed in the final task when the subjects had to translate:

"Bake the cakes for 25-30 mins. until the tops are brown."

Figure 6.5. Looping Error. Representation of "Mix the cocoa and coffee powder to a smooth paste with water just off the boil ..."
Nine subjects had trouble with this. The difficulty seems to come from the question of how it should be broken down. Its surface structure suggests a structure such as that shown in Figure 6.6. This is because there is an action and a condition for when to stop. However, the process is not one of cooking the cakes for 25 mins., checking whether the tops are brown and then cooking for another 25 mins., etc. In fact, to put this into a structure, one needs to think about how one would do it, oneself, given the constraints of what the notation can express.

![Figure 6.6](image)

This would indicate that subjects were not breaking statements of actions down sufficiently when converting them into the notation. They seemed often to have used a superficial analysis of the structure of the statement when selecting the representation of the instruction. Having done this, they could not really have given full consideration to what the resulting structure would mean and how it would be executed. The frequency data presented on the previous page suggests that this is a problem that is not specific to any one notation and is more a symptom of the general method used when writing instructions in this way.

6.10.2.3. Under/Adjacent Errors

The next most common type of error were labelled "Under/Adjacent Errors". There were 13 occurrences of this type of error but these were associated with only four subjects (and 12 occurred with only three subjects). The problem here is one of treating JSD's as a Flowchart-like notation with actions that are supposed to follow one another being linked with direct lines down rather than across (see Figure 6.7). The
The errors occurred in two categories of context. In three cases, the subjects were having to insert a single Process Box into a structure and these were placed wrongly with respect to the actions that were to precede or follow them. In all cases, this type of mistake was made only once, was pointed out by the Experimenter and not repeated in this context. This suggests that the subjects had not fully learnt the rules for using JSD's and needed them to be reinforced. The other context in which these errors occurred was when the subjects were generating large structures. In these situations, the task was to convert an instruction involving complex aspects into the notation. The resulting structures would end up bearing very little relation to true JSD's and were very hard to interpret. It was usually clear, however, that there was some intention for the order of the operations to follow the lines (as it does in flowcharting) rather than to use the JSD approach.

This type of error suggests that some subjects had existing rules for generating diagrams based on flowcharting-type notations. Thus, early on, when the rules for generating JSD's were not well established, they would occasionally slip into the old method. Such errors would be corrected, however, so that this possibly random effect would be reduced. Later, however, subjects would slip into using the flowcharting
approach if the problem was complex in some way. This suggests some sort of concept of mental load that can result in an increase in errors. This hypothesis was not supported by the questionnaire profiles obtained, however, as the subjects who made these errors did not seem to have been any more experienced with Flowcharting than the others. It is interesting to compare this with the observations of Bonar and Soloway (85) who suggested that pre-programming knowledge of "natural language" specifications were the cause of many bugs generated by novice programmers. They suggested that such knowledge was used when the novices did not know how to generate a solution in the programming language itself. This suggests a more conscious use of alternative sources of knowledge than was implied in the interpretation given here. The data do not provide any basis for selecting between these interpretations. Further studies would be necessary for that.

6.10.2.4. Double Repetition Errors

Another error that occurred only infrequently (3 times) but that added to this picture of how Subjects were using the notations was the "Double Repetition Error". The error could only occur when using the JSD notation and consisted of a structure that would not actually work, given the rules for following JSD's (see Figure 6.8). Owing to the Experimenter not having fully analysed the structure, however, it was included in the JSD's used in the second task group (where subjects were reading diagrams). Subsequently, in the final task group, some subjects used this to represent actions that involved two conditions. This was how it was used in the diagrams they had seen.

These three categories of error suggest that the process of generating diagrams to represent a given instruction follows a general pattern. Subjects appear to be acquiring a battery of structures (such as the looping structure or the double loop) combined with rules for when they can be used, in terms of the sorts of instruction statement that is appropriate. In generating a diagram, therefore, they perform some sort of analysis of the material to be represented and, based on this, generate a structure, or series of structures to suit. If the material appears to be of a standard type, then a stereotypical structure will be used. This strategy may fail if the instruction only appears to be of this type and, in fact, has a more complex nature. Finally, subjects will have rules for generating the notation in question but they may also have other structures in their repertoire that are derived from other notations and these can be used by mistake under certain circumstances. The object of training in the use of notations becomes, therefore, one of discouraging the use of inappropriate structures and the provision of appropriate ones combined with methods for identifying the rules for deciding when they are to be used.
When considered in this light, the distinction between learning structured notations and learning a traditional programming language becomes less clear. In both cases, the user is having to develop a battery of solution units that can be called upon in appropriate circumstances to combine into an overall solution to a representation problem. While, in learning to program, one is having to learn about the notional machine and there is much less freedom of expression, the differences may prove to be of degree rather than of kind.

6.10.2.5. Mislocation Errors

One group of errors was more strongly associated with Flowcharting than with JSD (12 out of 13 occurrences were from this group). These occurred when subjects had to insert an item into a structure, as they frequently had to in the earlier, practice tasks. The errors basically involved the items being inserted in the wrong place in the order of events. This confirmed that the groupings and headings that are used in JSD provide useful clues. This problem has been discussed by Green (82) and is one of the main criticisms of Flowcharting; that it does not force the writer to structure the diagram leaving the reader with few supports.
Other implications of the groupings achieved in a JSD-like notation were observed. In tasks where subjects had to move a group of Process Boxes in relation to others, they would often take the group as a whole and move that, using the group header. In some circumstances this was correct, but in others it resulted in more boxes being moved than was required. The advantages with the groupings appear to lie in the fact that the subjects were able to work at a larger grain of analysis, treating groups of items together and ignoring the details. As was seen, however, this had positive and negative repercussions.

6.10.2.6 Logic Errors

Two errors that occurred frequently but did not fall into any of the above categories have been labelled "Logic Errors". The first of these was when subjects inserted an instruction to "take the pan off the heat", as they were instructed to, without another to put it back on later. The second was when inserting instructions to turn the oven on so it could be pre-heated. These would often be entered as the very first box of the lasagne diagram, resulting in over three hours of pre-heating. Both of these errors reinforced the view that subjects were performing the tasks often with only minimal consideration of their effects. They would seem to use simple strategies or rules (such as to turn the oven on before any cooking) without considering the consequences.

6.10.2.7 Summary

The results described so far present a picture of the process of generating structures in both notations that has a considerable potential for error. Strategies and rules appear to be used that involve little mental effort. They can however easily result in structures that do not represent what the subjects were assumed to have intended. These rules seem to constitute an important part of the subjects' knowledge of the notation as they made minimal reference to the original descriptions provided and there was evidence indicating that they would pick up the rules from examples and feedback, during the course of their experience with diagrams. One of the questions that remains about the process of generating these diagrams is that of why they did not notice the errors they were making or the lack of accuracy of their representations. The next section explores the results of the task group involving subjects describing the flow of execution of diagrams to explore this issue.
6.10.3. Behaviour when Indicating the Flow of Execution

Owing to time constraints it was not possible to give this task to all subjects, but the results seemed to be fairly consistent. The Flowcharting group who attempted this (4 subjects) made no mistakes at all. The JSD group (6 subjects) all made some mistakes. The mistakes made could all be allocated to one of two types. Four of the subjects made mistakes relating to the Repetition Boxes. These consisted of incorrectly stating when the Repetition should be tested. In some cases, there was a single action to be repeated and the subject employed the wrong rule about when the test was to be carried out (see Figure 6.9). In one other case, there were many actions to be repeated and the subject said that the condition would be tested after each of the actions, rather than after them all. Finally, some subjects had difficulty with the double repetition structure, but, as has already been mentioned, this would not work anyway.

![Diagram of Repetition Box]

**Figure 6.9.** An example of a Repetition Box in a JSD+. The correct interpretation of this is that the test for stiffness is to be carried out both before any beating has been done and after each period of beating. The incorrect interpretation that was given was that the test was only carried out after the beating.

The other problem with flow of execution appeared in one particular task only (see Figure 6.10). This was a true segment extracted from the lasagne recipe but it does not make any sense in isolation from the whole. Five of the JSD subjects misinterpreted this diagram. The diagram itself was not particularly complex but subjects appeared to be ignoring this, to varying extents, and attempting to make sense of how it should be followed based on the box contents (see Figure 6.11 for an extreme example).
Execute box 4.
Examine box 2 if Yes Execute box 6.
if No " " 3.
From box 6 Examine box 1 if hold. END, if not
Execute box 5 Reexamine box 1. until box 1 holds end.
From box 3 Examine box 1. if holds Examine box 2
if not. Execute box 5, reexamine. box 1.

The absence of errors in the Flowcharting group suggest that Flowcharting strongly supports the interpretation of the flow of execution. This reinforces the findings of Sheppard et al. (80). The other observations provide some clues about the process involved when working with JSD's, which were not available in this earlier study, owing to the only measure being time taken. In that study, training was such that very few errors were made at all. All was that was possible was to state that the hierarchical arrangement did not support such rapid problem solving as the branching arrangement. The error data collected here enables a more detailed consideration of the processes.
contributing to this difference between notations. The strength of Flowcharting for facilitating the answering of questions about what occurs after what in a set of instructions has been noted by others in this area (e.g., Green, 82). Either through familiarity or some innate quality of the use of arrows in this way, users can usually easily follow such structures. This, alone provides few clues about how they do this, but, in conjunction with the observations from the JSD group, one may be able to go further.

Subjects in the JSD group were, for the most part, able to follow a basic diagram that involved no Repetition Boxes. They were more likely to get into difficulties, however, if such boxes were involved. Compared with Flowcharting, no explicit clues are provided about when the condition of the repetition is to be tested. This forces the user to rely on a good knowledge of the rules to interpret them. The difficulty that subjects had with these indicated that they found such rules hard to use in some way. Looking at the performance on the task shown in Figure 6.10, it would appear that subjects were relying heavily on the contents of the boxes and their knowledge of how recipes work to support their interpretations. There was no evidence that the Flowcharting group were having such difficulty and they did not get misled by the contents of the boxes.

The relevance of this to the above examination of how diagrams are generated relates to the extent of grouping of structures that occurred. We have seen that subjects appeared to be generating stereotypical structures to represent prose statements rather than always breaking them down into individual elements. We have also found that, in interpreting JSD’s, subjects are forced to consider loops as whole structures that are interpreted in the light of rules, rather than by following explicit elements of the diagram. The work on reading indicates that the process is very much expectation-driven and looks for large units rather than building up recognition by combining letters (Wright, 68). This approach is probably what is being observed here. Subjects are generating diagrams in terms of a number of large structures and reading them back in the same terms. Normally, they can do this with reference to the box contents so that they will know what form of repetition is implied. In the task where they had to make this explicit, their lack of real knowledge of the rules became apparent and they had to either guess from the box contents or construct rules as best they could. When reading back structures that they, themselves, have generated, there is likely to be a variety of levels of detail that they go into, with many structures just being interpreted as meaning just what the subject intended. Ultimately, this represents very little monitoring behaviour at all, but the problem is akin to that of programmers looking for bugs in their own code. Because they know what it is supposed to mean, it is hard to read what it
actually says (Myers (79) suggests this is why one needs to read one's program to someone else when looking for bugs).

6.11. Conclusions

This chapter has examined some of the possible approaches to language design for user-JEEVES interaction. The desired language should require minimal prior training but provide maximum support to the communication of program specifications. So-called "natural language" was found to have many problems as far as the latter criterion is concerned, while obviously minimising the learning required. The study described looked at two examples of an approach that could provide considerably more support for the communication process and examined the extent to which they could be acquired and used in a short time.

What was found in this study was that the task of converting a textual description of an instruction into a formal notation was not trivial for people unfamiliar with it. In particular, it appeared that subjects needed to develop a battery of appropriate structures that could be used to represent the different sorts of instructions that were to be communicated. This is likely to come only with experience and exposure to examples and feedback. That is to say that they will need training in the use of the notation. Having said that, subjects' use of the notations was such that they made a significant number of errors. This seemed to be an inevitable result of their effort-reducing strategies for selecting structures to use and the sort of monitoring behaviour used.

It was suggested that this means that, ultimately, the user of JEEVES, if using such a means of communicating a specification, must learn a new problem solving skill, albeit one that may make less cognitive demands than traditional programming. This appears to be a general trend that is running through much of the observations that have been made up to this point. The use of JEEVES does not remove the need to solve problems and generate solutions from the user but rather replaces the nature of the problems and solutions involved. In this respect, the inevitable novelty of the domain will require the user to develop skills and knowledge that were not previously present. These must include both batteries of solutions and the ability to apply them appropriately.

The particular notations had strong effects on the behaviour observed. The main sources of these effects were the groupings and headings of the JSD notation and the direct or familiar use of arrows to represent flow of execution in Flowcharting. A notation that combined these elements might well represent a stronger notation for the sorts of tasks used.
In conclusion, therefore, there appear to be major problems in the design of the means of communication from the user to JEEVES. Both the environment and language for specification seem likely to provide the user with major learning difficulties unless JEEVES can accept very informal specifications and communications with the likelihood that the structures used are either idiosyncratic or not representing what the user wants to communicate.
CHAPTER 7

END-USER EVALUATION: A CONSIDERATION OF THE PROCESSES AND REQUIREMENTS

Abstract to Chapter 7

This chapter looks at Stage Two of the user-JEEVES dialogue where the user is evaluating the Proposed Program. Before considering its nature, the importance of this stage is discussed. Two main areas of concern are identified. One is the way the Proposed Program must be presented to support both understanding and successful evaluation, and the other is the processes of evaluation as they are performed by end-users.

The former area is addressed and a number of factors identified that should be included in an ideal evaluation situation, through a literature review combined with a series of interviews with experts in this area. The problems with achieving these are discussed. Having considered the ideal conditions for an evaluation of a program design, with particular regard to its presentation, the actual evaluation process is focussed on. Information is described about how experts evaluate programs and designs in general. Skilled evaluation is found to involve the careful choice of criteria and metrics based on the goals of the design. Evaluation is, therefore, driven by the project goals. The skill and experience required for this is identified and found to be considerable.

A study is described of how end-users evaluate an electronic diary with respect to their particular needs. The observations made suggest a much more data-driven approach where the subjects were using elements and features of the diary to stimulate evaluatory comments rather than goals or needs. In addition it was found that many comments did not relate to needs that had been articulated in relation to the current diaries, suggesting that the presence of the electronic diary was allowing subjects to generate additional requirements. The chapter ends with a discussion of these observations and their implications for the design of tools where user evaluation of the computer's output is important.

7.1. Introduction

Having examined the processes involved in the specification of the requirements and identified some central issues relating to this, one must consider the other main activity that would occur in the user-JEEVES dialogue. This has been labelled Stage Two and consists of users working on the Proposed Program and identifying:
Before considering the nature of the evaluation process, the importance of this stage is underlined.

7.2. The Importance of Evaluation

There are three reasons for presenting the user with the proposed program for evaluation. The first of this is that the provision of a prototype program can, in the case of the non-expert, lead to the identification of other needs (Harris & Parker, 87). This is because it is often not until something concrete can be considered that the non-expert can really start to articulate what is required. It may be with JEEVES, therefore, that there will be a number of iterations required before an acceptable program is found.

In addition to this, in any dialogue between a human user and a computer tool there must be some assessment of whether communication has been successful. Failures could have been due to the use of inappropriate commands (user errors) or because the computer misinterpreted the user's instructions (system errors). Feedback is used in this way in any dialogue situation. While JEEVES can be expected to have considerable powers of interpretation and inference one cannot assume that the communicative link between it and the user will be infallible. Users must, therefore, be able to examine the Proposed Program to assess whether it really meets their stated needs.

Finally, users must assess the usability of the program. One might imagine that JEEVES could be designed with the expertise to generate only usable programs. The evidence is, however, that this will not be the case. The current guide-lines are not sufficient for generating usable computer tools (see eg. Lansdale, 83) and systems that can evaluate programs or interfaces are very crude and rely on formal descriptions of the qualities to be evaluated (eg. Tullis, 84, Back and Hietala, 84, Streveeler and Wasserman, 84). This is leading to a change in emphasis among Human Factors workers, as has been mentioned, towards the establishment of "usability by user-testing". Essentially the rationale is that good usability is a product of four factors: the tool design, the user, the context, and the task. This is too complex to manage in a predictive manner and requires the use of very specific studies, essentially a battery of these for each design. This means that, while JEEVES may be able to generate a
program that will follow good Human Factors guidelines, there will still be a need for the user to test this for usability problems.

Finally, it is important to mention that users of JEEVES should be able to evaluate the programs produced while they are still at the Proposed Program stage. If they can only identify problems when they have been turned into Implemented Programs then it will be much more costly to correct the errors or improve the programs. It is necessary, therefore, for Stage Two to be as effective as possible in weeding out problems with the Proposed Program.

This chapter examines the issues that emerge from this goal. It starts with an analysis of the elements that make up the evaluation process before going into them in more depth. A number of possible sources of mismatch between the Proposed Program and the supposed ideal program that would meet the user's needs have been identified. These will differ in type, therefore, but it is useful to be able to refer to the general class of such items. In the text that follows, the term "mismatch" will be used in all cases.

7.3. The Basic Elements of Evaluation

There are three processes involved in the evaluation stage. Users must understand the Proposed Program description, identify mismatches and communicate these to JEEVES in a way that will enable it to make the necessary changes.

In order to be able to identify mismatches, users must be able to understand what is being proposed. Given that they will only have been given a limited introduction to the available technology prior to Stage One, it is conceivable that this will prove problematic if the Proposed Program is at all complex. Related to this are issues of how to present a design so that it is most easily interpreted by the lay-person. Ideally users would be able to interrogate JEEVES about aspects that they did not understand, so a good ability to provide explanations will prove important also. While there are many issues relating to this area it is not proposed to go into them in this thesis. Essentially, the problems come down to matters of effective communication and this is an area that has been the subject of investigation for some considerable time already.

In general, dynamic prototypes are seen as the most powerful aid to the facilitation of both end-user understanding of a design and end-user evaluation (eg. Harker, 87b). Precisely why this is the case is often not considered by the writers of papers advocating prototype usage. Comments that have been made include the problem of interpreting such paper-based descriptions (Harker, 87b, Eason, 88, Potts, 88) because
they are either too abstract, or the language and terms of reference are not familiar. The other main point about prototypes seems to be the fact that users can interact with them directly and do not need to imagine what this would be like, resulting in an inevitable loss of detail.

Given that prototypes provide possibly the best combination of qualities that meet the various requirements described above, they seem likely to be the most effective representation to use for the Proposed Program. While the details of supporting diagrams, etc., will need to be investigated in particular cases, the use of prototypes looks like being a major element of the way users are presented with the Proposed Program for evaluation. The next section gathers together, therefore, the expertise that exists about the form this should take. This was done by an interview study of Human Factors experts who have used prototypes in end-user testing.

7.4. The Study

When designing computerised tools and systems the complexity of the interaction between the tool design, it is increasingly being found that end-user evaluations using prototypes are the best approach to adopt to ensure a good match with users' needs. This has resulted in Human Factors workers becoming experts in the domain of obtaining evaluations from end-users. In this study a number of such experts were interviewed in order to obtain insights into the conditions needed for end-users to evaluate programs. The study establishes the main features that apply in such a situation and their relative importance. It consists of a series of structured interviews with Human Factors consultants at the HUSAT Human Sciences and Advanced Technology Research Centre. The interviews went through a number of different factors, for each one asking the subject how important it was seen to be in achieving a valid and complete evaluation and then drawing him/her out about what aspects were particularly important and what further concepts were relevant.

7.5. Method

7.5.1. Subjects

Nineteen subjects were interviewed, all working at the HUSAT centre, often in a number of capacities. The subjects' work varied very greatly from small projects designing generic computerised tools such as in-car telephones or CAD packages to large databases for government institutions. Their input also varied from acting as main designer to the project to just performing the Human Factors testing. Some had
worked on only small numbers of projects (9 on 1 to 3 projects, 6 on 4 to 10), while 4 had been involved in more than twenty in their professional careers.

Owing to the fact that the subjects of this study were describing situations where they, themselves, were using subjects in a study, it is important to distinguish between these groups. In the text that follows, therefore, the term "evaluator" is used to refer to the end-users used to evaluate the tools. The term "designer" is used to refer to the subjects of this study, i.e., the designers or Human Factors experts. It was found that the programs designers had worked on varied greatly in size. Thus, while some were small tools for specific tasks, others were large systems to support the general activity of an organisation. In the description of this study, however, all such programs will be referred to as "tools".

7.5.2. The Structured Interview

The interview started with a few brief questions about the extent of the designers' experience in end-user evaluations. A number of different elements of the different ways of using a prototype were then explored, by asking the same questions about each and then encouraging further verbalisations or explanations. Finally, aspects of the dynamics of the evaluation process were identified by considering further details of how the prototyping session was structured.

The main body of the questions related to different elements of the methodology of using prototypes. These elements were taken from various references on the area (Harker, 87a, 87b, Wright & Monk, 90, Damodaran, 81, Eason, 88). In each case, designers were asked whether that aspect had ever been used in their experience in prototyping, how important this was to obtaining good evaluations and any other comments about its use. These questions were used to spark off a discussion of this aspect rather than as an extraction of specific items of information. The elements were:

1) Context.

2) Making sure that the evaluator is skilled in using the prototype - as against requiring evaluations from users with no experience or training.

3) Tasks used in exploring the prototype's functionality include aspects that make them immediately relevant to the intended use of the tool - as against just using a battery of stereotypical tasks involving similar functions.

220
4) Presentation including all functions and ways in which the tool will be used including infrequent aspects requiring maximum flexibility - as against just giving the core functions carried out in the most common manner.

5) Providing hands-on experience of the prototype - as against being shown a demonstration/video.

6) Giving the evaluator tasks to perform specified by the experimenter - as against being instructed to generate tasks that realistically represent how the tool would be used themselves.

7) Giving the evaluator a considerable period of free use of the tool - as against a specified session with it.

8) Allowing the evaluator to make changes to the design and observe their effects - as against having to imagine how it could be different.

9) Showing the evaluator more than one alternative - as against just having a single design to consider.

7.5.3 Procedure

Interviews were carried out at HUSA T, either in the designers' own offices or in free rooms such as the library. For practical reasons, the interviews had to be limited to an hour and a half which severely restricted the extent to which the different topics could be gone into. In practice this meant that the questions about static representation, measures taken and dynamic aspects were often omitted so that the other elements of the methodology could be focussed on. In some cases, once four designers had articulated a concept or method, this was incorporated into the questioning so that the time could be spent on identifying finer points.

7.5.4 Analysis of Data Obtained

The designers' answers were taken down in note form in an informal manner and then their arguments and comments reconstructed from these. This provided a knowledge base of concepts and descriptions of practice that could be compared and contrasted. In analysing this, both commonality and disagreement were focussed on. Also, there was little reliance on any measures of agreement as one designer articulating something that the others had not does not devalue it as it may just mean that it had not been noticed by
the others. This results in the data obtained being rather hard to quantify. The
description of this has attempted to give an idea of the degree of consensus observed
but this has not always been possible.

7.6. Comments Noted

The results of this study consisted of a body of insights and concepts that pertained to
the different aspects of presenting information about designs to end-users. Each of
these is discussed in relation both to the context of the designers and then with
reference to its relevance to the design of JEEVES. In many cases, this results in a list
of factors that must be taken into account when designing an evaluation session. The
final discussion pulls some of the concepts together and considers their overall
implications.

7.6.1. The Need for Hands-on Experience of the Prototype

It was agreed by all designers interviewed that this is vital if one is to get accurate,
useful feedback. When the prototype would be incomplete or not faithful, however,
the decision to opt for hands-on was less unanimous and some designers pointed out
that good information could be obtained from videos, especially if the right questions
were asked (3), or if the tool was very simple (1). This is clearly relevant to the design
of the evaluation stage for JEEVES. The factors that determine when a demonstration
is adequate could be used to decide between these options, if they can be identified.
Alternatively, it may be necessary for users always to interact with the Proposed
Program, which would have corresponding implications for the extent to which it must
carry all the functionality and the need for users to learn to use it.

7.6.2. The Need for Training

Nine of the designers interviewed found the need to train evaluators prior to the
evaluation to be a cause for considerable concern. If they were wanting to find out how
well the tool would be used by experienced users then the necessarily short training
periods would only simulate this situation to a very limited extent. The training periods
described were between one hour and two weeks. Having said that, seven of the
designers pointed out that one will either be working with a tool that is to be used by
inexperienced users, or be interested in just how easy it is to acquire the necessary skill,
or else be able to employ users with skills that will transfer readily to the new tool (eg
skilled word-processor users). In these situations, there is less of a problem.
The training issue is actually just a small aspect of the general problem of information overflow. If one has designed a complex tool, there is a very large amount of information to assimilate if it is to be evaluated effectively, whether one is trained to use it or not. The main strategies for dealing with this were to evaluate different parts at different times or to have extended evaluation sessions running into days and weeks rather than just hours. It is highly likely that users of JEEVES will not be willing to invest that sort of time in learning to use a tool that may ultimately be considerably changed in its final form, especially if a number of design iterations are to be gone through. This must represent a major barrier to users being able to evaluate the proposed tool effectively, if the data from this study are to be accepted. How one deals with this is unclear.

7.6.3. The Importance of Presenting the Prototype in Context

In the course of the interviews a number of different context categories were articulated as having a potentially important impact on the use of the proposed tools and, as such, require consideration when designing the prototyping sessions. These are listed in Figure 7.1.

* Computer Systems.
* Other systems
* Task.
* Physical.
* Social.
* Organisational.
* Motivational.

Figure 7.1. A list of the different contexts identified by designers.

Basically, as any of these factors can affect the way the tool is ultimately used, either the evaluation takes place within the actual context of intended use or else they should all be faithfully mimicked at some point in the session. Contextual aspects of the way the Proposed Program is evaluated are clearly important. While JEEVES can inform the user of this fact, however, this does not necessarily support him/her in the process of establishing the contextual features that are important in the case of the tool in question, nor does it ensure that efforts will be made to check such contextual features. It seems likely, then, that JEEVES will not be able to influence the user such that the evaluation will be carried out in the context in which the tool is to be used (above and beyond the extent to which this would be true by default). The most that may be possible is to provide a check-list of contextual factors that would be likely to interact in
7.6.4. Tasks Used to Test the Design

A number of aspects of the tasks used to test the design were discussed. The way these are used was seen to be very important in the identification of problems. Five designers commented that, ultimately, the whole tool be evaluated, on all the required tasks. This may, however, involve a series of trials focussed on different parts and ten of the designers referred to a set of "core tasks" on which the tool will be evaluated initially. These will be the more frequently occurring tasks but four designers pointed out that infrequent tasks can also be important (especially emergency support) and need testing. It was agreed by all designers that tools that are designed to support specific tasks need to be tested on these tasks. These are arrived at from the task analysis that was obtained as part of the Requirements Elicitation phase of the design. Three of the designers suggested that there may be a place for early testing of the tool using tasks that are less closely related to the intended ones. These will be valid when considering general usability, rather than usability in relation to specific tasks.

While much of the evaluation consists of the evaluator performing tasks prescribed by the Designer, the degree of this prescription was seen to be open to variation. Thus, while the evaluators will be told what tasks to perform, they can be given varying degrees of direction about how they are to perform them. The factors involved are shown in Figure 7.2. In relation to the user-JEEVES dialogue, there is the question of how the tasks to be used are identified. Ideally, they would result from a Task Analysis of the user's needs and the way the program works. It is unlikely that either of JEEVES or the user will be able to perform this, however. To deal with the need for thoroughness, it would be helpful if there were some way of helping the user identify areas of the program that are more likely to contain mismatches than others, or that are more important in some respect so that these can be focussed on.

Closely related to this issue is the possibility of giving the evaluator free use of the tool, often away from the designer. This presents problems with regard monitoring and interpretation of data but has many other advantages (see Figure 7.3).
Arguments in favour of a highly prescribed approach

• Base on task analysis so looking at what they would realistically do (2).
• Important if want to test all the functionality (1).
• Need for evaluator behaviour to be goal-directed (1).
• Hard for non-expert evaluator to play and explore an unfamiliar tool (1).
• Better for comparing different evaluators or designs (2).
• Produces easier data to analyse (2).

Arguments in favour of a less prescribed approach

• Find out unanticipated things (1).
• Better if tasks are sophisticated (1).
• Better if looking for imaginative solutions (1).
• Better if looking for a range of solutions (1).
• Better to let evaluator try to perform the tasks in the most familiar manner (1).
• Can only do this if prototype is sufficiently robust (1).

Figure 7.2. Arguments presented in favour of more or less prescription of the tasks used. Number of designers giving each argument is in brackets.

• Evaluators may be affected by the presence of the experimenter (some managerial users will be very reluctant to perform observed at all).
• Evaluators will probably not explore so much or else in a different way.
• Evaluators will be motivated differently.
• Evaluators will not be so bothered by repetitive tasks if they have only to do them a few times. In conditions of free use, they may prefer to use pencil and paper.
• The knowledge of the tool will be fresh in their minds where, in reality, intermittent users may have to attempt to reconstruct vaguely recalled procedures.
• Evaluators may end up using the tool to support a small proportion of the whole task, if it is discretionary in real use.

Figure 7.3. Arguments in favour of giving evaluators a period of free use of the tool.

7.6.5. Show More Than One Alternative or the Effects of Changes Specified

The designers had not, for the most part, had experience of showing evaluators more than one alternative. Reasons for not doing it related to factors not directly relevant to the situation of the user-JEEVES dialogue. Two designers saw the presentation of alternatives as a useful way of helping evaluators to make a more informed evaluation of the designs. Evaluators with limited computer experience would have little basis for criticising a design. Providing possible alternatives might help this. The opportunity to show the evaluator how the tool would work if the changes specified were implemented
was also considered. This was seen by seven designers as being potentially of use as an educational aid.

7.7. The Presentation of the Proposed Program

The results of this study indicate that the presentation of the Proposed Program to users in a way that will enable them both to understand it and to evaluate it effectively will represent a major difficulty to the designers of JEEVES-like tools. Many of these represent major difficulties when it comes to considering the locus of control of this stage of the dialogue. For a number of these factors to be included in the evaluation, JEEVES may have to take the role of the Human Factors experts interviewed and this will mean controlling the process rather than letting users perform the evaluations in the way that they want. This is a major problem and some of the issues that are emerging from this are discussed in this section.

In considering such notions as the context in which the tool is evaluated, the extent to which the user learns about the tool, the tasks used in the evaluation and the general procedure used in the evaluation, one must consider whether it is appropriate for JEEVES to take the same degree of control as the designers interviewed. There are, essentially, three options available. One is for JEEVES to direct the whole evaluation process, giving users instructions about every aspect. The second is for JEEVES to offer advice. The last option is for users to be left to perform the evaluation unsupported. In order to decide between these, one needs more information about how users would perform evaluations if left unsupported. The next section considers this and the whole process of evaluation in more detail.

One fact that came from the interviews was that designers would carry out a number of evaluations at different stages of the design process, focusing on different aspects of the tools. This meant that the fundamental design was accepted as good enough before the details of the interface were looked at. This would be a very efficient way to proceed as fundamental changes to the Proposed Program could negate the value of any earlier comments about the interface. Unfortunately, in the study described one of the Human Factors experts interviewed described finding that evaluators would often comment more on aspects of the interface, particularly items where the data used was inaccurate, than on the underlying functionality. She named this phenomenon the "dirty window" effect. This suggests that evaluators are easily distracted by certain aspects of the prototype. This and other evidence about the nature of the evaluation process discussed above suggests that asking users to evaluate different levels of the Proposed Program at different times may not work.
For such an approach to be successful, it would be necessary for users to evaluate fundamental, abstract aspects of the Proposed Program while ignoring the interface. It may be, however, that the interface will need to be cleared up to some extent before the fundamental aspects can be considered at all. Having said this, if there is a truly fundamental misconception about the nature of the program required, then users will probably pick this up early on. What would be desirable would be to find ways of presenting the Proposed Program to users in a way that focuses their attention on the fundamental details. This would, however, probably involve the use of representations abstracted away from the actual prototype with the corresponding problems described earlier.

Another issue about the way the evaluation dialogue is managed relates to the question of whether changes to the design are made after every comment or once the user has finished commenting. The former approach would both provide users with rapid feedback about the effects of their input (considered as potentially beneficial by the designers interviewed), and reduce the "dirty window" effect. If the mismatches are removed once spotted, then more fundamental issues can be examined. Finally, some of the criticisms may be quite fundamental and it would not be worth going on with the evaluation until they had been dealt with. Arguments in the other direction relate to the efficiency of having to stop and make changes too frequently. It may be more sensible for JEEVES to await a list of changes and then to implement them as a whole, dealing with the more fundamental first, and including all the information into the overall specification for the program.

7.8. The Processes of End-User Evaluation

While a number of factors have been suggested that should go into an "ideal" evaluation procedure, little has been said about how these goals can be achieved in the context of the user-JEEVES system. It has provided some ideas about how users do evaluate tools that suggest problems for the design of JEEVES but these need further examination. In particular, therefore, a greater understanding of the evaluation process is needed. This should be backed up by actual observational data of users evaluating a tool. The next sections address this by looking firstly at how experts evaluate and then describing a study of how end-users evaluate software tools. This starts by a consideration of the area that is closest to that of interest: that of how programmers currently identify problems in their programs.
7.9 Testing Programs

As part of the activity of more traditional programming, the programmer must, towards
the end of the task, test the program generated to ensure that it does not contain bugs.
This is known as "software testing". Software testing is analogous to the evaluation of
the Proposed Program that must go on in Stage Two of the user-JEEVES dialogue.

Unfortunately, no studies have been identified on software testing behaviour as it
actually occurs, tending rather, when any studies have been done (see Myers, 79), to
compare different testing techniques or technologies. One comment is that testing is
hard for novices because they will often not know quite what to expect from running
their program so may not be able to tell immediately if it is correct (du Boulay, 88).
This is particularly relevant to user-JEEVES interaction as this is exactly the situation of
the users. They will not have a clear idea of how the program is going to meet their
needs because they have left that decision to JEEVES, so that they will not be able to
perform a simple comparison between the behaviour of the proposed implementation
and what they expected. While there is no empirical data on software testing
behaviour, there is a body of prescriptive material to try to help programmers with this.
This material is exemplified by "The Art of Software Testing" (Myers, 79). This book
makes some important points about the problems of testing a program.

The first of these is that most programmers see testing as demonstrating that a program
does not contain bugs, rather than as finding the bugs that must inevitably be present.
He suggests that this subtly affects the strategies they use to test a program,
encouraging the use of tests that do not identify bugs. This raises important questions
for the evaluation by users of JEEVES of the programs it presents. Will they be
looking at them just to confirm they are okay or will they be more critical? How can
this attitude be influenced? Related to this is the fact that one can never guarantee that a
program is bug-free. Unless one runs the program on all the possible data that it could
possibly have to cope with, one cannot ensure that there will not be some data set that
would result in a crash or in faulty output. This raises the question of how the user of
JEEVES should, ideally, decide that he/she has evaluated the program to a sufficient
extent. The answer is not at all clear and there is a danger that, in the absence of a
stopping rule, the user will either feel the need to evaluate all aspects in great detail or
else perform a very cursory evaluation.

Given this danger, there is a need for the testing that is performed to be as efficient as
possible. For programmers performing software testing much of the activity of testing
lies in the selection of good test cases. One should aim for the subset of all possible
test cases that is most likely to lead to the identification of bugs. This brings us right back to the comments made by the user evaluation experts about the importance of selecting tasks that are representative of the actual tasks that are to be performed. These are the ones that will meet this criterion.

Aside from this need to identify good tasks with which to test the Proposed Program, there is a need to explore how well end-users can use tasks to identify mismatches. This has not really been discussed so far. The next sections describe a study that looked directly at this.

7.10. A Study of End-User Evaluation

No work has been found looking directly at how end-users evaluate programs. Systematic studies of user testing using prototypes are very hard to carry out owing to their being located within a context that makes them hard to manipulate in any sort of systematic fashion. Some data from an actual evaluation carried out by end-users would be very informative about the details of the evaluation process, particularly with respect to the way mismatches are identified. The tendency for evaluators to be easily distracted ("dirty window" effect) suggests a data-driven approach to the evaluation. They are looking for "interesting" features to pick up on, interest being defined in a number of different ways but with the underlying assumption that these will be less taxing to articulate than alternative, more fundamental problems. This study employed a number of ways of presenting a program to users to see how these might interact with their ability to identify mismatches between it and their needs. Thus, subjects were presented with an introductory demonstration of the tool, followed by a time of free use with the manual provided, followed by having to perform three short tasks using the tool.

Many problems exist in studying evaluation as one is wanting to find a situation where the subject is testing the proposed program against an existing set of needs, as would be the case for the user of JEEVES. This means that it may not be sufficient just to have subjects motivated by cash or credits. What is required is to be able to offer them a program that could really be useful to them, and, ideally that they might be considering obtaining for their own use. Failing this, one could make the situation closer to that of the user-prototyping session where the subject believes that the evaluation is to some genuine purpose and will be more motivated to perform a real evaluation rather than a cursory or false one in some way. An attempt was made to achieve these goals in this study using a product that exists on the market and with the cooperation of the manufacturers who had expressed genuine interest in the results.
7.11. Method

7.11.1. The Object of the Evaluation

The tool to be evaluated was the Psion Organiser II Model LZ. This is a hand-held computer that combines "personal organiser" functions with programming power and many facilities that would make it useful for remote data-logging, etc. For the purposes of this study, only a limited subset of its facilities were used. These were the diary, a notepad facility and a database that could be used for addresses, etc. It was explained to the subjects that the tool had many more facilities than these and they were given the opportunity to explore these after the main session. Appendix IV contains the text of the demonstration that was given subjects of the Psion, as a description of the functions involved. The Psion was on loan from Psion plc. They were approached about this and offered a copy of the report that would come from the study to help them in their product development. This had the double goal of reducing the cost of the study and of making it possible to tell the subjects that their comments might be used in the design of future models.

7.11.2. The Subjects

Nineteen subjects were used. These were obtained both by advertising and by direct approach. The advertisements aimed to attract people who were considering purchasing a Psion, promising them an opportunity to try it out for themselves. They also described the study as an evaluation of the Psion, rather than as a study of evaluation per se. Seven of the subjects volunteered for the study because of these. The remaining subjects were approached because they were believed by the experimenter to be diary users and, therefore, to be potential Psion customers. Apart from this, the subjects were extremely diverse in their occupation and level of computing experience, as was found from the questionnaire results, as well as in the extent and nature of their diary use.

7.11.3. Procedure

The study consisted of two sessions. In one, subjects were asked about their current diary use, using a structured interviewing approach, and then given a questionnaire about their experience of, and intentions with regard to, electronic organisers and their computer experience. In the other session, the subjects are exposed to a series of three experiences of the Psion after each of which they are asked to give their comments about how well they think it would meet their needs.
Subjects were asked to describe their current diaries and how they used them using a number of headings as shown in Figure 7.4. These areas were used to obtain as detailed an analysis of how the subjects used their diaries as possible in the time available (45 minutes).

* Describe the diary itself, including all the different parts used,
* Describe what information is kept in the diary, in detail, including sections other than the daily spaces,
* Describe how and when the diary is referred to,
* Give the average number of times it is referred to in a day,
* Give the maximum number of entries in the diary per day or week,
* Give the average number of entries in the diary per day or week,
* Describe any other diaries that are used and how this one relates to them,
* Describe the main strengths of the diary used,
* Describe the main weaknesses of the diary used and any ways in which it fails to help the subject,
* Describe the features that the subject would like to see on an "ideal" electronic organiser.

Figure 7.4. The aspects of current diary use asked about.

Subjects were told that they were to evaluate the Psion based on the demonstration, etc., with regard to how well it would meet their own particular needs rather than in some general sense. They were also told that their comments would be sent to Psion and that they may be used in the design of future models. Subjects were encouraged, in this manner, to take the evaluation seriously and, it was hoped, were being treated as experts in diary use.

There were three experiences of the Psion that were given. These were, in the order of presentation:

* **Demonstration.** This was a combination of a taped walk-through and the experimenter working the Psion. It lasted for ten minutes.
* **Manual.** In this, the subjects had ten minutes with free use of the Psion and the parts of the manual that referred to the programs focussed on. They were told that they could either attempt to look at it all briefly or focus on one part.
* **Tasks.** The subjects were then given three tasks to perform on the Psion. These were designed to be representative of real tasks for which it might be used and included descriptions of likely contexts in which they would occur.
The subjects made their evaluations by putting comments down under three different headings:

- **Needs.** Aspects of the Psion that do meet my needs.
- **Problems.** Aspects of the Psion that do not meet my needs.
- **Changes.** Changes that I would like to see to the Psion.

Interest here was on both the aspects of the tool that the subjects had noted, whether as positive or negative, and the ways in which they communicated these. In particular, whether subjects tended to describe the negative aspects purely by describing what they did not like or by suggesting a change to the design. A different evaluation sheet was provided after each experience of the Psion so that the relation between the comments made and the experience that generated them could be identified. The sessions had to be kept short so that subjects would agree to take part without the need to resort to financial incentives. It was hoped that this would result in the subjects obtained being mainly motivated by interest in the Psion.

Thirteen subjects gave the diary description before evaluating the Psion (Normal condition), while six had this order reversed (Reversed condition). This was to check whether describing the diary was affecting their evaluation performance. Both groups had the sessions arranged so that the second stage came at least three days after the diary description. This was for the Normal group, both to provide some distance between the two, so that the diary description would not be too immediately familiar, and so that the tasks specific to the subjects' diary use could be designed. In four cases, in the Normal group, the length of time between stages was forced to be more than ten days. This is summarised in Table 7.1.

<table>
<thead>
<tr>
<th></th>
<th>Short period between sessions</th>
<th>Long period between sessions</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Normal</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Volunteered</td>
<td>3 Subjects</td>
<td>0 Subject</td>
</tr>
<tr>
<td>Not Volunteered</td>
<td>6 Subjects</td>
<td>4 Subjects</td>
</tr>
<tr>
<td><strong>Reversed</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Volunteered</td>
<td>4 Subjects</td>
<td>0 Subject</td>
</tr>
<tr>
<td>Not Volunteered</td>
<td>1 Subject</td>
<td>1 Subject</td>
</tr>
</tbody>
</table>

*Table 7.1. The design of the study.*
7.11.4. Data Collection and Analysis

The data collected consisted of the descriptions of the current diaries, the evaluatory comments made on the sheets, and the results of the questionnaires about computing experience and interest in the Psion. It would clearly have been interesting to have also taken some sort of recording of the actual process of generating the comments such as an audio or video recording. A combination of factors including the desire to have the sessions in the subjects' homes or places of work to minimise the inconvenience and the vast amount of extra data that would have been generated by this resulted in a decision not to do this for the present study. It was thought that if the results of this study prove sufficiently important to merit follow-up, then methods such as these could then be considered. The variables focussed on are given in Figure 7.5.

<table>
<thead>
<tr>
<th>Independent Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>* The two conditions of describing diary before or after the evaluation (and also length of time between these).</td>
</tr>
<tr>
<td>* Whether subject volunteered or not.</td>
</tr>
<tr>
<td>* Amount of computer experience and extent of diary use.</td>
</tr>
<tr>
<td>* The different experiences of the PSION (demonstration, manual plus free use, tasks).</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Dependent Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>* The aspects commented on and the criticisms made.</td>
</tr>
<tr>
<td>* The detail with which subjects described negative elements of design and the extent to which solutions to these mismatches were proposed.</td>
</tr>
<tr>
<td>* The relation between evaluatory comments and current diary use.</td>
</tr>
</tbody>
</table>

Figure 7.5. The variables focussed on in the study.

Because of the uncertainty about exactly what phenomena were likely to be found, the design was such that as great a variety of data could be collected. This necessitated the use of a relatively small subject group. Consequently, the different groups would not be likely to demonstrate significant effects. The groups were, therefore, combined when comparing the conditions. Thus, when comparing the Normal and Reversed subjects, whether the time between the sessions was short or long was ignored. While it is not strictly valid to compare the different conditions by grouping subjects together in this way, this was intended only as a preliminary investigation that would open the way to more carefully designed studies where more subjects are used in the various groups. The small size of the subject groups also forced the use of non-parametric statistics for the analyses.
One final point about the way the data was analysed refers to one particular subject, AB. It was found, on a number of occasions, that the data demonstrated considerably greater homogeneity when the data from AB was removed. Clearly, with such a small subject group, one subject represents a larger proportion of the group than with a larger group, but it was still considered worthwhile, in a number of cases, to examine the data both with and without AB. On examination of the subject profiles, it was found, incidentally, that AB made exceptionally great use of his current diary and, while not being an extensive computer user had actually considered purchasing a Psion at one time. Such idiosyncratic features may account for his performance.

7.12. Results

As a first step, it was necessary to compare the groups of subjects that were used in the various conditions to check that they were comparable. This reduced the danger of random variables stemming from subject differences, although the grouping of conditions could still have an effect. Having checked that, certain general properties of the evaluations observed are described before a more detailed analysis of the data is gone into.

7.12.1. Checking that the Subjects in the Conditions Were Comparable

The only significant difference between groups was between the volunteers and those that did not volunteer was in the means of their expressed interest in Electronic Diaries (Mann-Whitney, T=17.5, p,0.05). Otherwise the variances for each group were not significantly different (F-test, p,0.05) for any measure. One might expect to find such a difference with volunteers and there is no desire to attempt to separate out these factors as they both indicate increased motivation and interest in the Psion.

The next section gives a general picture of the way the subjects evaluated the Psion, before embarking on the more detailed investigations.

7.12.2. The Overall Form of the Evaluations

Although no method was used to record the subjects' overt evaluation behaviour, the experimenter was able to notice the general pattern that this took. In the case of each experience of the Psion, evaluations were only noted after the experience, for the most part. The most common exceptions to this occurred in the Manual experience, where subjects were being guided in their exploration of the Psion by the manual and would occasionally note things they found at the moment of finding them. It is worth noting
that a number of the subjects expressed frustration at the short experience of the Psion that they felt they had had and the feeling that they could hardly comment on it under such circumstances.

Figure 7.6. Examples of comments made by one subject.

The basic data of the study was what the subjects wrote down as their evaluations of the Psion. These tended to occur as a list of points rather than as a general description of the whole thing. Figure 7.6 shows some examples of comments made. Table 7.2 gives the basic frequencies of comments made allocated into positive and negative ones. There were, in total, 13 cases of subjects repeating themselves, usually by making the same comment after two successive exposures to the Psion. These were omitted from the analysis so that only the first occurrence of each comment was recorded.

<table>
<thead>
<tr>
<th>Number of Comments Made</th>
<th>Mean Number of Comments per Subject</th>
</tr>
</thead>
<tbody>
<tr>
<td>Positive...122..............</td>
<td>6.42</td>
</tr>
<tr>
<td>Negative...160...............</td>
<td>8.42</td>
</tr>
<tr>
<td>Total........282...............</td>
<td>14.84</td>
</tr>
</tbody>
</table>

Table 7.2. Summary of the frequencies of different comment types.

Most of the comments made, both positive and negative were made by very few subjects (see Figures 7.7a and 7.7b). Only two comments were made by more than 10 subjects. There was no significant difference in pattern, in this respect, between positive and negative comments. Subjects' comments were, therefore, mostly idiosyncratic. Items where most agreement was found (i.e., items where a comment was made by more than 3 subjects) are shown in Figure 7.8. The data was confounded to some extent by the variation in specificity of the comments made. In some cases, subjects were making comments that could easily embrace more detailed ones made by others. Aspects of level of detail are looked at later in the analysis.
Figure 7.7a. Graph showing the number of positive comments that were made once, twice, etc.
Clearly, most comments were made just once.

Figure 7.7b. Graph showing the number of negative comments that were made once, twice, etc.
1) Comments on aspects that are very general such as a general dislike for the notepad,

2) Comments referring to a small number of key qualities that were found to be problematic:
   - Screen size,
   - Keyboard layout,
   - 15 min time blocks in diary,
   - Size and weight,

3) Comments about the alarm being useful.

Figure 7.8. Items where most subjects agreed.

7.12.2.1. Discussion of the Basic Frequency Data

There is some evidence here that, in the evaluation process, negative comments are more common than positive ones. The analysis of the evaluation will, for the most part, focus on the negative comments as these are what are central to the success or failure of the evaluation (positive ones do not lead to improved design, even though they can suggest what needs to be emphasised in future models). While some of the negative features seem to be commented on by many subjects and may represent those most likely to be causes of difficulty (but only one was mentioned by more than half), many more were only commented on by individuals.

Idiosyncratic evaluations have been observed in professional designers also (e.g. Hammond, Hinton, Barnard, Maclean, Long, & Whitefield, 84, Wright and Monk, 90). In this study, the apparent idiosyncratic nature of the comments could stem from three sources. Firstly, each subject was found to have different patterns of usage of his/her current diary. Thus, each may have genuinely different needs of the Psion. Secondly, it seems likely that each subject may have missed mismatches for some reason. Thirdly, there was evidence that some of the comments made were not an expression of a real need and may have been more of the nature of a response to the instruction from the experimenter to generate comments. Further, more detailed analysis of the data will shed light on these processes. It is worth recalling that software designers showed the same lack of consensus when evaluating programs in the study described by Hammond et al. (84).

Having identified some features of the evaluation behaviour as a whole, it was found useful to distinguish between two basic areas of evaluation. The first is the extent to which the Psion was found easy or hard to use and the second the sufficiency of the functionality provided. This distinction was found to generate an important division in
the analysis. The next section describes the reasons for using this distinction. The following sections examine the nature of each type of evaluation comments.

7.12.3. The Distinction Between Usability and Functionality Comments

In examining the data, it became clear that there were two general categories of comment; those about how well the Psion met the needs of the subjects in its functionality, and those about how easy or hard it was to use. This left a small number of other comments that did not fall easily into either category, such as those relating to its size, etc. The comments were consequently allocated into these three types. Many of the analyses subsequently performed on the comments were found to result in different data for each type suggesting that this distinction was an important one.

It was concluded that usability comments, as defined in this study, were really a sub-set of functionality comments, i.e., those that were specifically commenting on the ease of use of the functions rather than other aspects more directly related to what tasks could be performed using them. This point is elaborated upon later when functionality comments are sub-divided further.

More functionality comments (mean = 9.2 F-comments) were made than usability comments (mean = 5.7 U-comments). This difference proved highly significant (Wilcoxon, T=13, n=18, p<0.01). On the whole, therefore, subjects commented more about functionality than about usability, but all made at least one comment of each type. No more will be said about direct comparisons of these types until each has been analysed in more detail. The first type to be examined in this way are the usability comments.

7.12.4. Usability Evaluation

Having divided the comments into those referring to usability and those referring to functionality, the various properties of these different comment types were examined in detail. In this way, it was possible to establish some picture of the processes that lead subjects to making comments of each type. The data can be characterised in a number of ways and these are discussed in the following sections.

7.12.4.1. Patterns in the Types of Usability Comments Observed

The comments were divided into positive and negative and the totals for each were 23 and 85, respectively. AB was the only subject who gave more than two positive
comments about usability (he gave eight), and, of the three that gave two comments, two were Human Factors professionals. The rest gave either one or none. This difference between number of positive and number of negative comments made was found to be significant (Wilcoxon, \textit{T}=0, n=16, \textit{p}<0.01). No subject gave more positive comments than negative ones. This suggests that the evaluation of usability mainly consisted of noting problems that arose, or that the subjects anticipated arising, during the course of the experiences. It is difficult to see exactly how one would establish the criteria for making a positive comment about usability. As was mentioned above, the greater focus in the analysis is on the negative comments but this provides a useful first clue about how they were identified.

Many of the usability comments were described in a very general manner (e.g., statements that the Notepad was too hard to use). The thirty others, however, were more specific and could be considered to represent a general body of usability criticisms of the Psion. No subject noted more than one third of these. Over half the subjects noted less than three items out of the 30. Informally, it was recalled that a number of the comments made referred to problems that most of the subjects had had difficulty with (such as confusing pressing up-arrow instead of down-arrow to move to a later date in the diary) and problems had been observed by the experimenter that no subject had commented on. This strongly suggests that, while usability criticisms were only generated when problems were encountered, some problems were encountered without subjects commenting on them.

It has been remarked that the comments varied in their specificity. One of the problems subjects may have been having in commenting on problems may have been in articulating them sufficiently specifically. It was decided, therefore, to categorise the usability criticisms in terms of specificity. The categories used are shown in Table 7.3. It was occasionally hard to distinguish some of these, as the boundaries were, to some extent, artificial. In order to obtain a measure, for each subject, of the specificity of the comments made, points were allocated to the subjects accordingly (i.e., 1 point for a "I" type comment, etc.). This measure was then used to identify associations with elements of the subject profiles.

It has already been said that AB generated a lot of positive usability comments. The data without AB showed that nine of the 13 positive usability comments were in category 1, i.e., very general about the Psion as a whole. Many comments were made of each different type. AB did not stand out to such an extent in the production of
negative usability comments. He still obtained 25 points for negative items, the most of any subject.

<table>
<thead>
<tr>
<th>Frequency of Occurrence</th>
</tr>
</thead>
<tbody>
<tr>
<td>1) Comments referring to the whole Psion                     16</td>
</tr>
<tr>
<td>2) Comments referring to either a particular program within the Psion or some large part of it (e.g., the keyboard)  17</td>
</tr>
<tr>
<td>3) Comments referring to a specific problem with a program or some small part of the Psion (e.g., a key)           39</td>
</tr>
</tbody>
</table>

Table 7.3. Categories used in distinguishing specificity of comments made with frequencies of occurrence.

Clearly, many of the comments made were very general and vague. One interpretation of this is that there is a process of locating a mismatch within a program or tool with some being harder to locate than others. This also had implications for JEEVES in that it may be necessary to work with vague comments.

Having looked at the detail with which the comments were located within the Psion, it was also noted that a number of the comments contained suggestions for how the situation could actually be improved. Comments were then categorised according to this. In doing this it was found that many of the comments contained implicit reference to the sort of solution the subjects had in mind while not actually including an explicit reference to change the Psion in that way. Thus, three categories were used as shown in Table 7.4. Actual statements of solutions are usually quite concrete although can sometimes just refer to a change without indicating what one is changing to. Half-solutions, where subject is not actually making an explicit statement that the machine should be changed, tend to include many vague ideas. Basically they are problems that the user experienced with some indication of the source of the mismatch and a hint implied about how the problem could be solved. No solution items are just descriptions of mismatches that subjects encountered, for the most part.

Table 7.4 also summarises the frequencies obtained. Some subjects made three or four explicit suggestions for a different design and these tended to be computer buffs, ergonomists or AB. There was, however, no statistically significant relation between these measures and computer experience. Only 6 subjects made no suggestion of a solution. This data can be interpreted in terms of subjects going different distances.
along the path of generating a criticism, from just stating the mismatch, to suggesting why a problem occurred, to suggesting an actual solution. Subjects were observed giving comments at each of these stages, with roughly equal frequency. It is worth noting that to describe a change involves finding a solution to the problem identified which must be harder than just identifying the mismatch.

<table>
<thead>
<tr>
<th>Frequency of Occurrence</th>
</tr>
</thead>
<tbody>
<tr>
<td>Explicit suggestion for a different design of some element ......................................................... 16</td>
</tr>
<tr>
<td>Implicit information about what the subject would have preferred ........................................ 12</td>
</tr>
<tr>
<td>Purely a statement of the mismatch ................................ 14</td>
</tr>
</tbody>
</table>

Table 7.4. Categories of comment according to amount of information about solutions that was included with frequencies of occurrence.

It would appear, then, that subjects were having different degrees of difficulty in articulating the mismatches identified and in suggesting solutions. Clues for the factors affecting this difficulty, and hence to the overall evaluation process, could either come from differences in evaluation performance after each experience of the Psion, from the behaviour of subjects in the different conditions, or from individual differences between subjects. These three areas are examined, starting with the former.

7.12.4.2. Differences in Comment Types Related to Experiences of the Psion

As Table 7.5 shows, there was not a lot of difference between the experiences. A slight drop was observed but this proved not significant (Friedmann, df = 2, p=0.7). It was interesting that this was the case, suggesting that identifying mismatches may be more a question of time spent with the tool than of particular aspects of the experience. In the study of prototype usage, the designers interviewed suggested that hands-on experience was vital if one is to obtain good data about a design. This is not supported purely on a frequency basis. Subjects seemed to be able to give usability comments whether they had had hands on experience or not. It may be, however, that the comments differed in quality between the different experiences. This was investigated by categorising the specificity of comments given after each experience.
An informal examination of the data suggests that the comments after the demonstration were more vague than after the other experiences (see Table 7.6), but specific comments could be given after them all. A Chi-Square test could not be carried owing to the cells not being independent and the data did not warrant the use of a more complex analysis. There was no evidence of a pattern in the extent to which solutions were offered, however. This makes sense as there is no reason to suppose that hands-on experience facilitates the generation of solutions.

<table>
<thead>
<tr>
<th>Experience</th>
<th>Mean Number of comments</th>
</tr>
</thead>
<tbody>
<tr>
<td>Demonstration</td>
<td>1.74</td>
</tr>
<tr>
<td>Manual</td>
<td>1.68</td>
</tr>
<tr>
<td>Tasks</td>
<td>1.47</td>
</tr>
</tbody>
</table>

Table 7.5. Mean number of comments per subject after each experience.

Table 7.6. Frequency of occurrence of comments of each type of specificity, according to the different experiences.

<table>
<thead>
<tr>
<th>Comment category</th>
<th>Demonstration</th>
<th>Manual</th>
<th>Tasks</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 (very vague)</td>
<td>8</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>2 (less vague)</td>
<td>6</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3 (specific)</td>
<td>9</td>
<td>13</td>
<td>11</td>
</tr>
</tbody>
</table>

In interpreting this data, one must consider the way subjects were to identify mismatches in using the Psion. The Demonstration did not contain any instances of a user having problems so subjects would have had to imagine these to make any comments. In the Manual and Task experiences, they were more likely to work on actual problems encountered. The difference in specificity may have been related to this. If subjects were having to imagine problems when watching the Demonstration, their comments may be less accurate but this could not be assessed. It should be remembered that usability comments that are based on a short experience of the tool
may also not be an accurate reflection of problems that would be encountered after long term usage.

7.12.4.3. Differences in the Comment Types Related to the Conditions or to Individual Differences

The subjects varied in a number of ways. Differences could be divided into those relating to the conditions used or to the individual differences that were noted between the subjects. There was no significant difference between the groups in the various measures of the usability evaluations. Similarly, there were no significant differences between those who had described their diaries a short time before the evaluation and those that had done it a long time before. There was no evidence, therefore, that describing the current diary prior to evaluation had any effect on the subjects' ability to evaluate the usability of the Psion. This suggests that the process of evaluating for usability has little to do with comparisons between the tool and the current diary, and is purely focussed on what is to be evaluated.

There were no differences between the group who volunteered and those that did not, in terms of the usability comments. Similarly, the measures of subjects' interest in electronic diaries did not correlate significantly with this. There were few significant correlations between computing experience apart from consistent positive correlations between the measures of computing experience and the number of usability points allocated to the subjects (a measure of the detail of the comments made). Thus, Spearman's rho was found to be 0.666 between hours spent computing and usability points. This suggests that experience with computers in general can be an important factor in being able to give detailed comments about usability problems. In particular, the correlations (without AB) between computing experience and the number of usability points were also significant. There is a strong suggestion here that the influence of computing experience is over both the number of comments made and the level of detail possible. Also, there was a significant correlation between the number of packages experienced and the average level of detail of the negative usability comments made. The difficulties with the data and available resources suggested that, while a Factor Analysis might be the best way of ensuring that these effects are not spurious, the large number of such correlations can be taken as sufficient evidence of their validity. This suggests that experience with computers plays an important part in subjects' abilities both to articulate usability comments and to be precise about their location (but that other factors can be equally as important in the absence of computing experience). Programming experience, however, did not correlate significantly with these measures, and neither did the extent of the subjects' current diary use.
A picture is emerging of the process of generating usability comments that sees it as one of experiencing or imagining problems with the tool and attempting to articulate the mismatch, possibly with a solution. There are definite problems with this articulation process that may be reduced by experience of other computerised tools. There seems to be no evidence that this sort of evaluation involves any sort of comparison with other diary tools as it is not related to previously describing the current diary, or extent of diary use.

Having considered the process of generating usability comments, the other type of comment functionality comments are examined in much the same ways.

7.12.5. Functionality Evaluation

7.12.5.1. General Aspects of the Functionality Comments

The first analyses of the functionality comments were just repeating the approaches used for usability comments. Thus, their idiosyncratic nature, the relation between positive and negative comments, the information they contained and the relation to the different experiences are described first.

As with usability comments, there was minimal consensus in the functionality comments. However, this could be simply because different subjects had different needs. There is a significant tendency (Wilcoxon T=11.5, n=17, p<0.01) for subjects to give more good comments than bad about functionality. However, in practice, they did not give very many more (totals were 99 good to 75 bad). Subjects do not, therefore, seem to be using problems or difficulties as the sole criteria for identifying something to say. Their comments were either:

- that a given feature would/would not be used, or,
- that some aspect of the design affected how well a given feature supported goals they wanted to achieve.

Thus, it appears that subjects are looking at the features and making decisions about whether they are useful or not.

As with usability, functionality comments were found to vary considerably in their specificity. Negative comments could either dismiss programs wholesale or consist of descriptions of particular functions that would not be useful. Comments were also allocated into groups according to their information content in relation to the provision
of solutions. It was found that comments about why a need was not met would vary greatly in how explicit they were and in how much information was contained. This then provided different amounts of information about how it needed to be changed to improve it. There were two basic categories of reason given. One was to state that some function, requirement or goal was not supported (14 of these) and the other was to say that it was too hard to achieve (3 of these). See Figure 7.9. for some examples of these. Even when suggesting changes, subjects may not really be describing why they are needed. This suggested two continua along which a comment can fall resulting in a three-by-two grid of possibilities (Table 7.7). It was found that few comments gave minimal information about the mismatch, i.e., neither reason nor change but that 33 gave only one of these. Similarly only eight comments contained both a description of the change and information about why it was wanted.

* Wouldn't use note pads.
* visual field too restricted.
* not deal with diagram/symbols - need to get used to.
* It might prove difficult to separate appointment booking from 'time planning' for tasks. This almost needs two categories, one which is fixed, eg meetings, one which is flexible, eg work plan.

Figure 7.9. Examples of different types of reason for criticising aspects of the Psion.

<table>
<thead>
<tr>
<th>Amount of information about reason for change</th>
<th>Describing the change required or not</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Yes</td>
</tr>
<tr>
<td>1) None</td>
<td>3</td>
</tr>
<tr>
<td>2) Minimal</td>
<td>9</td>
</tr>
<tr>
<td>3) Considerable</td>
<td>22</td>
</tr>
<tr>
<td>4) Statement of strategies for using features to support tasks (EH)</td>
<td>2</td>
</tr>
</tbody>
</table>

Table 7.7. Grid of possibilities for different amounts of information given in functionality comments, with frequencies of occurrence of each type.
As with usability comments, it was seen as important to test for any patterns in the way the functionality comments elicited were distributed in relation to the different experiences of the Psion. There was a highly significant trend for subjects to give fewer comments after the later presentations than after the Demonstration (Friedmann, p<0.001). No definite pattern, with regard to the different types of functionality comments, could be observed as so few comments were given after the last experience and the comments after the manual stage had the same shape as those after the demonstration.

It would appear, then that the Demonstration was by far the best experience of the Psion for generating functionality comments. This may be an order effect, but this seems unlikely. On the other hand, informal observations suggest that the effect may be more due to the problems subjects had in using the Psion during the other experiences. Their lack of training meant that they had to invest a considerable effort just in following the instructions. They may well not have had any resources left for noticing features that are or are not useful. Certainly, however, there is evidence that demonstrations can serve a useful purpose, when evaluating functionality, if the evaluator is likely to have difficulty using the tool.

As with usability comments, functionality comments seem to vary in the detail and information that is included in them. They differ from usability comments in that difficulties in using the tool impeded their generation rather than facilitating it. The next section looks at a particular property of functionality comments. While usability comments can only be tested against difficulties subjects were observed to have with the tool, functionality comments can be compared with statements subjects have made about their actual needs.

7.12.5.2. Relation Between Functionality Comments and Subjects' Needs

It was noted that many of the ways in which subjects currently used their diaries could not be supported by the Psion. A list of such basic tasks was drawn up (see Appendix IV). It was found that, instead of tasks either being definitely supportable or definitely not supportable, it was more a case that, for some tasks, of the extent to which strategies had to be developed to find ways of making the Psion support them. Thus, it could not be said that the Psion made it impossible to keep an entry referring to the whole day because one could usually keep such items in the slots allocated to the night. However, it was possible to determine what the Psion had actually been designed to support and what tasks required some adaptation of the way it is supposed to be used if they were to be supported.
It was observed informally during the study, and also from some of the comments, that subjects were often attempting to find ways of making the Psion support their needs, when it did not do so directly. These seemed to be extreme examples of what they were doing all the time. Thus, subjects would be assessing a feature by determining how it could be used to support their needs. It should be noted, also, that this is related to the concept of a usability comment. This sort of comment is also saying something about how well a function supports a task and the effort required to use it, but in a slightly more specific way. Again, there is a decision about whether the effort required is acceptable or worthy of comment. Given that there is a need to find ways of adapting functions one might expect that some subjects would be better at this than others as it is, essentially, a problem solving exercise requiring skill and experience.

Given this list of tasks not supported by the Psion it was possible to compare the subjects' descriptions of their current diaries and identify ways in which the Psion would fail them (labelled as negative aspects). It was also possible to identify functions of the Psion that would be particularly useful to the subjects, aside from the very basic diary facilities (labelled as positive aspects). One could then examine the evaluation comments of each subject and note those that consisted of times when the subject had identified such aspects of the Psion that were commented on. This left another list of items that were not found in the evaluation.

Table 7.8 summarises this data for both positive and negative aspects. Roughly the same number of positive and negative items were found. However, there were many more negative items identified by the experimenter so that the proportion missed was greater. Subjects were found to have missed many points, both positive (69% of all positive points there were to find for each subject) and negative (83% of all negative points there were to find for each subject). The difference between positive and negative was not significant (Wilcoxon, T=26, n=13, p>0.1). The maximum percentage of negative items found was only 50%, while the maximum for positive items was 100%. This suggests that the evaluation process was not very successful as far as comparing the Psion with current needs was concerned.

While categorising the functionality comments in terms of the descriptions of the current diaries, it was noted that many of them bore no relation to these. Further categories of comment were developed based on the comments found and these are shown in Figure 7.10. Address books and notepads, etc., were not included in the exploration of current diary usage so that these were extras also. A distinction was made between these two categories of extra. Thus, comments that evaluated the
notepad or address facility were separated from those that commented on the fact that they were present as part of the diary.

<table>
<thead>
<tr>
<th>Category</th>
<th>Frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>Found -</td>
<td>22</td>
</tr>
<tr>
<td>Found +</td>
<td>21</td>
</tr>
<tr>
<td>Missed -</td>
<td>99</td>
</tr>
<tr>
<td>Missed +</td>
<td>49</td>
</tr>
</tbody>
</table>

Table 7.8. Summary of frequencies of occurrence of positive and negative items that were missed or found.

ex+ = extra functions provided by the Psion that were not discussed in relation to current diary. These can be aspects that did not exist in current diary such as an alarm, or aspects that were part of the current diary but not mentioned by the subject as beneficial.

ex- = extra criticisms of the Psion that were not discussed in relation to current diary, eg. time divisions when current diary did not employ any.

ex? = items identified as lacking, in the Psion, but that was also lacking in current diary. Only 2 of these were found.

unn = times when diary is criticised for not supporting an aspect that it does, in fact, support. This occurred only twice.

pro = a criticism about a function that the Psion can support but that the subject felt would not be used owing to other aspects such as usability. Four of these were found.

Figure 7.10. Other categories of functionality comments.

Having categorised the remaining comments in this way, the frequencies of the extra comments were compared with those of the comments relating to the current diary. This data is summarised in Table 7.9. Of the functionality items found, a high proportion were evaluating features that were not directly relevant to answering the question "How well does the Psion meet my current needs?". This was more noticeable for the positive comments made. Subjects gave many more positive comments about functionality than negative ones but, while a total of 22 positive points were made that related to their current diary use, 46 were made that did not.
This suggests that the process of evaluating functionality has little relation to actual diary use and is more focussed on extras. This is reminiscent of the observations about usability evaluation where prior descriptions of the diary have no effect on performance. To return to the idea that usability comments form a subset of functionality comments, it is now possible to see that functionality comments can be divided into three groups. There are those that are focussed on the current diary and current needs, those that are focussed on functionality but on that of the Psion, and those that are just focussed on usability. This represents a continuum of increasing focus on the Psion and decreasing focus on the actual functional needs. Table 7.10 includes all three to show the extent of this. Only 19.9% of the comments are actually about how the Psion relates to the current needs of the user. It seems likely that the extent of this relationship between comments made and stated needs would be affected by having described the current diary before performing the evaluation. This issue is examined in the next section, along with the effects of individual differences.

<table>
<thead>
<tr>
<th>Type</th>
<th>Percentage</th>
</tr>
</thead>
<tbody>
<tr>
<td>&quot;Relevant&quot;</td>
<td>34.0%</td>
</tr>
<tr>
<td>&quot;Extras&quot;</td>
<td>51.2%</td>
</tr>
<tr>
<td>Others</td>
<td>14.8%</td>
</tr>
</tbody>
</table>

Table 7.9. The percentages of the different functionality comment types.

The proportion of items found to all possible comments was significantly greater for subjects who had described their current diary before the evaluation (Mann-Whitney, p<0.05) but they did not produce significantly more comments in total (Mann-Whitney, p>0.05). This suggests that having described the current diary before evaluating the Psion was shifting the subjects’ focus from the Psion alone to the relation between the Psion and the current diary. No significant difference (Mann-Whitney, p>0.05) was

7.12.5.3. Relation between Condition and Individual Differences and Functionality Comments Generated

The proportion of items found to all possible comments was significantly greater for subjects who had described their current diary before the evaluation (Mann-Whitney, p<0.05) but they did not produce significantly more comments in total (Mann-Whitney, p>0.05). This suggests that having described the current diary before evaluating the Psion was shifting the subjects' focus from the Psion alone to the relation between the Psion and the current diary. No significant difference (Mann-Whitney, p>0.05) was
found between subject groups who described their diaries long before or recently in identifying problems with the Psion in relation to the current needs, but these two groups have been found not to be truly comparable.

There was little difference between the performance of the groups of subjects who did volunteer and those who did not. The one exception to this was that the volunteers missed significantly more items (Mann-Whitney, \( p<0.043 \)). This may have been accounted for, in part, by the fact that the volunteers did tend to have diaries with more points to identify in relation to the Psion, i.e., there were more points to miss for this group. This result was only significant at the 9.9% level, however. The other measure of motivation was the interest in electronic diaries in general and this also correlated with complexity of the current diary (0.710, \( p<0.005 \)) and with missing more items (0.513, \( p<0.05 \)). The same interpretation can, therefore, be applied to this result.

Related to the matter of motivation is the notion of how clearly the evaluator knows what he/she wants. One possible criticism of this sort of study is that the subjects were in an artificial situation where they did not really have any interest in or need for the tool being evaluated. The argument would be that subjects who were genuinely interested might have a much clearer idea of their needs and be more motivated to relate the tool to them. This data certainly provides no support for this argument. While having articulated one's current needs does seem to have an effect on focusing the evaluation on these, measures of genuine interest do not.

Only two significant correlations were found between functionality evaluation performance and computing experience and these were isolated cases and do not suggest any overall trends. Little change was achieved with the removal of AB. There seems to be no evidence that experience with computers facilitates the evaluation process.

Significant correlations with the extent of current diary use were associated more with commenting on extras (0.550, \( p<0.01 \), positive "extra" comments, 0.604, \( p<0.005 \), criticisms of "extras") than with identifying relevant items (0.181, \( p>0.05 \), negative items found, -0.079, \( p>0.05 \), positive items found). This suggests that more extensive diary use does not improve the ability to spot uses of or problems with the Psion, but it does seem to affect the tendency to comment on novel functions.
7.13. The Process of Evaluation

Certainly, the results of the study described reinforce the conclusion that end-user evaluation is data-driven and works opportunistically either on direct experience or imagination. A more detailed picture can be constructed from this, however. This consists of information about how the presentation of the Psion affected the evaluation, and the nature of the three types of evaluation identified: usability, relevant functionality, and extra functionality. These are described in the following sections.

7.13.1. The Overall Procedure

The observations made of subjects' behaviour combined with the data about the comments generated provides further information about the way different presentations interact with evaluation behaviour. There was considerable evidence that Demonstrations could have advantages over hands-on experience, if users do not have time to become familiar with the Proposed Program, although hands-on experience may be necessary for usability evaluation. There is a need, however, for a more detailed consideration of how demonstrations are to be designed to best facilitate the evaluation process. It was found that, when subjects did have hands-on experience of the Psion, they would tend to be driven by whatever material was available. Thus, if working with the manual, they would explore the tool by following the manual, rather than generate tasks to do and use the manual for finding how. On the other hand, they were happy to perform tasks when they were given some. The shortage of time and experimental setting could both have contributed to this behaviour. It seems possible, however, that users would have some difficulty in generating tasks for testing the Proposed Program unless they have time to become quite familiar with it. Evidence for the value of tasks relevant to the particular needs of the subjects was not found, other than informally.

7.13.2. The Process of Usability Evaluation

The evidence obtained about usability evaluation suggests a process of the general form shown in Figure 7.11. The process starts with the subject experiencing some degree of difficulty in using the tool, or in an observation from the Demonstration that leads to the subject imagining that some degree of difficulty would be experienced. In the latter case, this must presumably depend on understanding the Demonstration and relating it to the subjects' past experience of similar tools.
It would appear as if some difficulties experienced or imagined do not get articulated. This could be due to a number of factors such as beliefs about the subject's competence and what is an acceptable level of difficulty. Experience with computers is associated with making more comments but this could be interpreted in a number of ways. Comments that are made can vary in their specificity and in the inclusion of information about how to solve the problem identified. This suggests that both of these elements of a comment may be hard to generate. Again, experience with computers seems to be associated with locating the problem.

7.13.3. Two Types of Functionality Evaluation

It was found that a distinction could be drawn between two sorts of evaluatory comments. Some comments could be associated with the current diary and needs ("relevant" functionality) while others were not ("extra" functionality). These two types of comment appeared to have different characteristics. Generating more "relevant" functionality comments was associated with having described the current diary beforehand, but generating more "extra" functionality comments was not. The latter was associated with making heavy use of the current diary, however, which did not affect the former. This suggests that two different processes may be in operation in generating these comments. Consequently, they are considered separately in the following sections.

7.13.4. Generating "Relevant" Functionality Comments

As with generating usability comments, the process of generating "relevant" functionality comments must involve both detecting an item worthy of comment and
generating a comment about it. The second stage of this seems likely to have similar characteristics as when dealing with usability comments. This is because comments noted were found to vary in the extent to which they included a solution as well as just the problem statement and their specificity. It was also found that subjects would give varying amounts of information about the reasons for the change being required.

This information provides clues about what must be going on in identifying an item as worthy of comment. There were two forms that the comments given could take. In a few cases the comment was that, while a goal could be supported by the Psion, it was too difficult or inadequately supported. These comments are close to usability comments but seemed to refer more to the way the functionality was organised than to ease of use. In most cases, the comment was that a given goal simply could not be supported at all. It is interesting to compare these two comment types with the comments that did not include reasons. These consisted of statements either that some feature was excessive or inadequate in some manner or that it just would not be used. This suggests a continuum where features of the Psion are being compared to what is known about the subject's needs.

The process of identifying a functionality item worthy of comment must, therefore, be more complex than for a usability item. The main difficulty with the process of comparing in this way lies in the different representations involved. Thus, information about what is available is in terms of Psion functions while information about what is required may be in terms of abstract needs, but is equally likely to be purely in terms of the functionality provided by the current diary, or the tasks performed on it. Any comparison between these must involve a transformation of one or both of these representations. The data from the study described provide very little direct information about how this is done. A general model of the process is given in Figure 7.12. The evidence from the frequency of occurrence of extra comments and the extent to which aspects were missed by the subjects suggests that they are employing a data-driven approach. This would mean that they start with a given feature of the Psion and attempt to relate it to a need. This would mean trying to find uses for features and, when they fail, indicating that the feature is of no use. There was evidence for this approach in that some subjects commented that they would be able to support some task by adapting the Psion's functions in some way. In these cases they were clearly looking for ways of using the Psion to support their goals.
This is clearly not the whole story, however, as some comments involve the identification that a need is not met, rather than that a feature is of no use. It is likely, therefore, that the process is heavily driven by information about the Psion but that this can include the stimulation of needs that are associated with the functions seen. The most likely explanation is that subjects have knowledge about their current tasks and uses of diaries and that these carry implicit or explicit information about their needs in a more abstract representation. The more abstract the representation of the needs, the more likely it is that they can be related to functions of the Psion. Otherwise, the comparison will be between the diary and the Psion in concrete terms which would not make any sense. This allows one to interpret the observation that the proportion of "relevant" functionality comments increases with subjects who described their current diary before the evaluation session. Such a description would probably increase the extent to which the subjects have thought about their needs rather than just using the diary without thinking a lot about it. This would make it easier to associate aspects of the Psion with the current needs. The subjects' attention is likely to be more focussed on the current diary in this case also.

Many questions clearly remain about how subjects are able to compare features of the Psion with needs. The results of this study provide a starting point and indicate something of the difficulty of doing this. Further more detailed studies employing similar methods to those described here could be used to test some more detailed hypotheses. The next stage is to consider the evidence of the "extra" functionality comments to see how they contribute to the understanding of evaluation as a whole.
7.13.5. Generating "Extra" Functionality Comments

The statement of needs that were categorised as "extras" was not found to be any different from the statement of needs that did relate to current needs, in terms of the characteristics examined. However, the extent of this correlated with quite different factors, as has been described. While the articulation of "extras" was not related to having previously described the current diary, as one might expect, it was affected by the extent of current diary use. This, combined with the data suggesting that the process is similar to that of identifying items relevant to current needs suggest a similar overall process of articulating comments relating to "extras" as to relevant aspects. The process is shown in Figure 7.13. The diagram is essentially the same as that of Figure 7.12, for the "relevant" comments, except that the needs identified are "idealised" rather than "actual".

![Diagram of Battery of Idealised Needs Identified](image)

Figure 7.13. A general model of the process of identifying "extra" functionality comments.

"Idealised" needs would be those that stem from frustrations, etc., with the current diary. The interpretation of the correlation data is that heavy diary users would be more likely to have had such experiences, both because they put more demands on their diaries and because they just use them more often. They are also more likely to think carefully about their choice of diary, resulting them having a greater interest in extra features. For example, subjects may have found that they did not always remember to look at their diary and consequently missed appointments. This experience can remain little more than the memory of the problem or it may have led the subject to wish for some sort of alarm. In either case, when presented with a tool that has an alarm facility the subject can associate this with that need and this will trigger a comment.
It is interesting to note that subjects had been asked about their frustrations with the current diary and what they would like from an electronic diary. The comments they gave were included within the "relevant" comments division. This means that the "extra" comments represented aspects that were not articulated prior to viewing the Psion. This suggests that these "idealised needs" were only available to the subjects when they were presented with a concrete artifact. This supports the view that much of the requirements specification process relies on the presence of an implementation to criticise.

7.14. The Relevance to JEEVES

The studies of the evaluation process described have provided an initial picture of what may be going on when a non-expert evaluates a program or computerised tool. There are clearly many areas that need more detailed investigation but the data collected has made it possible to identify a number of important processes and concepts that will, in turn, facilitate the identification of the crucial issues for the design of JEEVES. These issues are discussed in the next sections.

7.14.1. The Evaluation of the Usability of the Proposed Program

The distinction between the different aspects being evaluated is a useful starting point for this. The first element is usability: the user must ascertain whether the program has any aspects that make it hard to use. The studies described have identified two main problems with this. Firstly, it is not at all clear how one can guarantee that all such problems will be identified. It seems unlikely that users of JEEVES will be thorough enough in their evaluation to guarantee their experiencing all the sources of difficulty. There does not seem to be any way around this unless one can make predictions about likely problem areas to which the user can be directed. There are a number of factors that should be included in a valid usability evaluation that are likely to be omitted by untrained users: use of actual tasks, particularly relevant ones, consideration of contextual aspects, etc.. It may be that users can be supported in this respect by the provision of advice from JEEVES about these matters. The extent to which this advice can be or is taken needs further investigation.

In addition to the danger that problems will not be identified is that, even if they are identified, they may not get articulated. There seems to be a certain proportion of problems that subjects experience that do not actually get mentioned. This issue could be approached from two angles. The first is to attempt to change the user's attitude, so that difficult experiences when using the tool are seen as both caused by the design and
open to change. There is anecdotal evidence that casual users of customisable systems will often not take advantage of this aspect and accept a poor interface rather than change it. Research is needed into how this approach can be altered. The second way this problem can be dealt with is by JEEVES monitoring the user's behaviour when using the program and trying to identify points where he/she seems to be having difficulties (pauses, actions undone, etc.).

7.14.2. The Evaluation of the Proposed Program in Relation to the Current Needs

The second way in which the program is to be evaluated is in relation to the current needs of the user. Essentially, this is making sure that the program proposed by JEEVES meets the needs the user has articulated to date. Once again, one would hope that JEEVES would be sufficiently well designed for this to be redundant but it is likely that there will be some misinterpretations of the user's requirements, as with a human-human dialogue of this kind, and the goal is to identify these.

The main observation that is relevant here is that the evaluation process has been found to be mainly data-driven. There are clearly problems with this approach and one could consider whether it were possible to influence the user such that a more knowledge-driven (or need-driven) approach be adopted. Ideally this would consist of the user generating a list of needs that are to be met and checking whether each one is supported by the program. The extent to which this is a possibility cannot be predicted from the results of this study alone. However, the evidence from the work on people describing their needs is that such a list would be hard to produce and that it might well be hard to use.

In the absence of such a list, if the user can generate tasks that he/she would be likely to want to perform on the tool, then there it seems reasonable that the needs will be tested to some extent. This is, essentially, a more concrete way of representing the needs. Again, empirical data is required about whether non-experts can generate such tasks in the absence of a more thorough Task Analysis. The danger would be that some tasks would be generated but that certain aspects of what the program is wanted for would not be recalled at this stage.

Given that the user is having some experience of the program, there is a need to support the processes both of understanding the various functions in terms of what needs can be supported and to make comparisons between what is needed and what is offered. A suitably documented demonstration of the program would include reference to the
different needs that the functions support. This sort of commentary would be important in helping the user to understand it. Any manual or description of a tool will contain some information about what the tool can be used for and this may map on to particular needs of the user. However, a more tailor-made description would be a valuable aid in this.

This sort of approach would hopefully facilitate both the understanding of the tool and the comparison between what it offers and what is required. This becomes difficult when, as was found with the Psion, the tool does not explicitly state that a task is supported but certain functions can be used for it. Also, there may be times when the user has a particular approach in mind for how a task is to be performed and the program deals with it in quite a different way. Both of these occurrences require the user to adapt implementations to fit with needs and this may prove to be difficult under certain circumstances.

7.14.3. The Evaluation of the Proposed Program in Relation to Extra Features

These features will be those that were not requested in the original specification but that are included in the proposed implementation either by default or because they form part of that sort of program in JEEVES' library of programs. The evaluation of these is not central to the user making sure that the program does what is required but it is worth recalling that these do seem to distract the user from what is supposed to be evaluated. It may be useful to attempt, in the presentation of the program, to make these aspects less salient than those that are responding to stated needs. In this way, the attention can be focussed on what is important.

7.14.4. Using the Comments Obtained

There are two main issues that remain about this stage of the user-JEEVES dialogue. These are about how JEEVES deals with vague comments and whether the comments need to be filtered in some way. These issues are discussed briefly in this section.

Some of the issues relating to how evaluation comments are dealt with by JEEVES were discussed in the previous chapter. Detailed information about the nature of the comments was not available at that point, however. The observations from the end-user evaluation study provide such information, however. The main point from this was that comments were found to vary considerably in the information that was contained in them. This led to the interpretation that more precise comments which included a statement of the problem, its cause, and a solution, would involve users in
considerable work and require experience that may be beyond the intended user group. JEEVES may, therefore, have to deal with a certain proportion of vague, ambiguous comments.

There are clearly a number of ways of dealing with this situation. In many respects, the problem is analogous to that of the whole user-JEEVES dialogue. If one is to expect the user to give instructions that are easier to interpret correctly, then more skill or knowledge must be required. In the absence of this, one must rely on inspired guesses on the part of JEEVES, feedback, and clarification dialogues to improve the likelihood of adequate communication.

One other comment that was made on the basis of informal observations from the above study was that subjects may have been generating comments out of a desire to find something to say. This, combined with the likelihood that the users of JEEVES will lack skill in discerning real problems with the Proposed Program from more transient ones stemming from a lack of skill in using it, suggest that JEEVES may need to perform some sort of filtering role on the comments before they are implemented. How this is managed is unclear and requires, again, a careful investigation of the control implications. It may be most appropriate to request further information from users about why they feel that the feature in question is problematic so as to assess the justification for the change. If it is found to be lacking, then users could be informed of this and left to decide whether it should be taken further or not.

7.14.5. The Need for Evaluation Revisited

The study of end-user evaluation has underlined two points about the need for evaluation. On the one hand it is clear that the presentation of a concrete artifact stimulates many more requirements than are generated by other means. The preponderance of "extra" functionality comments provides strong support for this hypothesis. It indicates that the presentation of a prototype program serves, not only to allow the identification of misunderstandings and minor implementation issues but may be vital as a means of obtaining a large proportion of the specification. If this is the case, then the second stage of the user-JEEVES dialogue would need to be presented in this light, so that users are not seeing their task as that of evaluating a potential design but rather of using the Proposed Program as a springboard to further requirements.

This is the positive side of what has been found. The negative side lies in the observation that end-users are actually very poor evaluators of programs. The predominantly data-driven approach observed where many of the subjects' needs were
not taken into account suggests that the Proposed Program, while being a valuable aid for stimulating additional requirements, may well not lead to the identification of misunderstandings. While JEEVES may well be able to advise and support users in this process, there is considerable potential for it to fail. This represents a major difficulty in any system that relies, like JEEVES, on users being able to distinguish between desirable and undesirable outputs. It is interesting to note that Wilson and Long (88) in their tutorial notes on evaluation suggest that what they call "Expert Evaluation", where an expert examines the program informally, is also likely to be incomplete. They suggest the need for formal metrics where the constraints or goals are converted into easily testable design attributes.

7.15. Conclusions

The evaluation of the program proposed by JEEVES is important for three reasons. Firstly, it may be that this program provides users with the first opportunity to articulate their needs. Up to then, the absence of anything concrete may have hindered this. Secondly, users must check that the program meets all the functional requirements. Thirdly, users must test the program's usability. This is a complex property of a program and cannot be guaranteed by even sophisticated design guidelines. It is only when users come to try and get the program to work that mismatches will be identified.

In this chapter, the way end-users carry out evaluations was examined. It was found that their approach was in contrast to the skilled, planned approach of the professional designer. Also, it was noted that the presentation of a design proposal was a valuable aid to the identification of further requirements. In the evaluation, however, they were observed to be predominantly data-driven. There seemed little way of guaranteeing a thorough evaluation of the tool in question. Given that end-users seem to be rather poor evaluators, the options for improving the quality of the evaluation process have been considered and a number found that might offer a way forward. There seems, however, no way of reducing the need for the user to do a considerable amount of work at this stage, so long as JEEVES cannot be trusted to produce a program that will meet the user's needs.
CHAPTER 8
DISCUSSION AND CONCLUSIONS

Abstract to Chapter 8

This chapter reviews the observations and results reported in the previous chapters and considers their relevance to the future both of programming and of the design of intelligent human-computer interfaces in general. It starts with a review of the thesis to date. This leads on to two main areas of discussion. The first is an evaluation of the research techniques that have been employed and an identification of the lessons that have been learned from them. The second looks at the body of observations and discussions and considers their relevance to the design of JEEVES-like tools, to the future of "intelligent" interfaces in general, and to cognitive science.

The chapter concludes the thesis with the observation that the original idea of providing programming power to non-programmers may prove to remain elusive and questions the value of pursuing this approach in the future. It suggests that other "Holy Grails" of Human-Computer Interaction also need such appraisal before investing heavily in the assumption that they are worth aiming for.

8.1. Introduction

This final chapter of the thesis performs three main functions. The first is to review the work done, the second to discuss its implications and the third is to draw conclusions from these. It starts with a brief review of the results presented in the thesis, both in the form of empirical data and methodological observations.

8.2. Review of Thesis

In Chapter 2 the nature of programming expertise was described, as it currently manifests itself. The point was made that this expertise had many different facets as programming:

- involves a number of tasks,
- requires considerable knowledge of the language and methods,
- requires a good understanding of the machine being programmed,
- involves the programmer in some complex task management issues.
It is argued that these elements can only be acquired through practice on the actual tasks of programming and that improvements in teaching methods alone will never solve the problem of the non-programmer who does not have the time to invest. In Chapter 3 the potential for improvements in tools or environments to achieve this goal was assessed. The different approaches currently under development or already available were examined. While most of these seem to represent useful advances, it was found that the features that made one aspect of programming easier resulted in another being harder, or else that the addition of the feature resulted in an undesirable increase in the complexity of the interface. The only occasions where ease of use was achieved to any great extent involved domain specificity or sacrifices in terms of programming power. The conclusion arrived at, at the end of Chapter 3, was that while some progress could still be made to make programming more accessible to non-programmers, the potential for this is limited. It was argued, therefore, that a more fundamental re-appraisal of the way users are required to communicate their requirements might be fruitful in yielding a more rapid step forward. Indeed, if programming power is to be made available to non-programmers, a fundamentally different approach to the whole process must be adopted in order to side-step the problem-solving nature of the task. This approach would be one where the user-computer dialogue is closer to that which occurs between a human client and a programmer consultant. Under these circumstances, users are not forced to communicate their requirements in the constrained language of programming constructs but can describe them in ways that are closer to those in which they are represented "naturally". This was identified as programming by requirements specification and the computer interface that accepts such communications was named JEEVES.

In Chapter 4 a study was described that generated a basic understanding of the processes that can take place during a requirements specification dialogue between clients and programmers. A number of observations were made about how people generate and communicate their requirements and about how the elicitation of such requirements can be managed. The main points of this related to the fact that the clients described their needs in terms of programs that would meet them, which highlighted the extent to which their descriptions were affected by their domain knowledge. In addition to this, the locus of control and the way feedback was used by the programmers were also noted. This study suggested two fundamental processes that could be used in considering requirements specification dialogues between users and computers. The first is that of the initial generation of requirements where the users are describing what is wanted, based on whatever knowledge or ideas they have about what is available. The second process is that of responding to the way this set of
requirements is interpreted by the machine and will consist of evaluating the output and
determining whether it is as required. Users must then give further requirements to
tune the program until it is close enough to what is wanted. These two processes were
then examined in more detail to obtain a fuller understanding of their nature. The first,
then, was that of generating the initial requirements.

It was pointed out in Chapter 5 that control over the requirements specification could
rest with the user or with the computer. It was argued that, unless the computer could
have sophisticated abilities in this area, it would be better left with the user. Given that
clients were observed to communicate their requirements in terms of programs that
would meet them, it was assumed that this approach would be the most suitable for
most of this part of the dialogue. A study was described that achieved a grasp of where
the central issues of this sort of dialogue lie. They were identified as being concerned
with the need effectively to communicate what can be done by the programs that the
computer can generate. It was shown that information is required about the task
domain, the form of programs and the available facilities. Two studies were described
that investigate how this information can be communicated in a way that allows it to be
used effectively in the requirements generation process. A key observation from these
was that the information provided is not used in a formal way but rather to stimulate
ideas for requirements. This has the additional effect of requiring the system to check
requirements to ensure that they really are within the program range.

The actual nature of the communication between users and JEEVES is a potentially
major source of difficulty for users. The immediate response to this question would be
simply to suggest that "natural language" be used as the means of communication.
Aside from the question of whether this will ever be a possibility, there are strong
arguments against the use of "natural language" in HCI. These were described in
Chapter 6 and stem mainly from its inappropriateness for many domains and from its
misleading nature. It is clearly desirable, however, to use a means of communication
that is as familiar as possible while providing users with the necessary support in the
generation of appropriate program descriptions. It was argued that notations such as
flow-charting represent a useful example of such an approach. A study of the use of
such notations after minimal training demonstrated the problems with them. It was
found that users still had to develop a battery of structures that could be used for their
different communicative needs and that they could not transfer sufficient expertise from
their knowledge of language and the domain to be able to use the notations without
training.
The second major process of generating programs using JEEVES would be that of responding to the implementation that is suggested to meet the needs communicated. This requires the users to assess such programs as to their suitability for the intended purpose. This may also result in the identification of further requirements. Essentially, the main activities involved are:

- understanding the Proposed Program,
- testing the program and identifying mismatches at the levels of functionality, methods and usability,
- communicating these to the computer.

The need for users to understand the programs generated by the computer represents another source of difficulty for the designers. There will need to be sophisticated explanation facilities available if this is to be at all successful. In addition to this, a number of elements were identified as important if the evaluation is to be successful in locating problems. While some of these can be dealt with relatively easily, others are more problematic involving issues about the extent to which the computer has control over this part of the dialogue. Thus, for example, it is desirable that users are presented with a prototype of the program and that they perform tasks that they will ultimately want to perform on it. While the presentation of a prototype is a purely technological matter, the question of how users are encouraged and supported in the testing procedures is not. This is likely to be a considerable problem as evaluation is a skilled process. In order to assess the scope of the problem, a study was described that looked at how end-users evaluate computerised tools. It was found that they would appear to be largely data-driven, rather than focussing on their needs and goals. This resulted in many comments about the usability of the tool and about features that it provided that were not supporting current needs of the subjects. One other area that was identified from this study was the dilemma about whether to provide users with hands-on experience or not. It was noted that, with the short opportunity to become accustomed to the tool, subjects found that it was not possible both to make it work and to comment on it. A demonstration was found to be more successful in eliciting functional comments because of this, but there was evidence that it did not generate as usability comments that were as useful. Basically, the time needed to perform any sort of thorough evaluation seemed to be considerable under these circumstances. If there is a need to go through a number of cycles of iteration with new programs being suggested each time, then this could be an extremely lengthy process.
As this review has indicated, a large number of areas have been addressed in this thesis and these have relevance, both to the design of any tools with similar goals to JEEVES, and to other, related, domains. This relevance is discussed in the next sections. The first aspect that is focussed on is that of the research methods employed.

8.3. New Research Techniques Developed

As was mentioned in Chapter 1, the work being done here does not fit in easily to any single research area. It has drawn on many areas and has involved the identification of appropriate empirical methods as well as the investigation of novel domains of study. The lessons learned from doing this are described in this section. It reviews the different approaches taken before offering some general recommendations about how to undertake the development of novel research methods.

The study described in Chapter 4 provided a broad view of how requirement specification dialogues are managed. There is much further work that remains to be done in this area that would have applied implications for the design both of tools that endeavour to replace the expert (as JEEVES would) as well as tools to just improve expert-client consultation. In particular, it would be important to observe dialogues of this type in real life rather than in role-plays as the validity limitations of the latter are obvious. For such studies to be possible, a number of research tools are required. While little innovation is needed to record such dialogues, considerably more is required to represent the transcripts obtained in a way that facilitates description and comparison of such dialogues. The analysis of the observations made in the Programmer-Client Study contained a number of approaches and comments were made about the author's experiences of their positive and negative features.

There are a number of stages to such an analysis. These are shown in Figure 8.1. Stages (a) and (b), where the raw data is broken down and categorised, can be very subjective even when one has developed apparently objective means of performing them. One must be careful, therefore, to assess how subjective the methods used are and to acknowledge this rather than try to deny it. While any subjectivity may appear to weaken the data obtained, it may be that it is necessary given the domain of interest. Indeed, one must be careful, even when having established that one's methods are reliable between different analysts, not to claim that this means that the categories used reflect reality. The reliability may just indicate that the analysts share a common perspective on the data. Indeed, the best approach is one where a number of data collection and analysis techniques are used that will confirm or lead one to question the conclusions drawn (cf. Good, 89). The later stages, (c) and (d), are less likely to be so
subjective and could, conceivably, be performed by a computer or unskilled assistant. The use of SNOBOL4 was found to be very helpful in this respect although it clearly forces one to be able to express the objects of interest in an unambiguous manner and one may not be able to do this for them all.

**Figure 8.1.** The process of analysing a complex piece of raw data. The first stage (a) involves breaking it down into suitably sized units. The second (b) involves categorising the units using an appropriate taxonomy. From this, it is possible to analyse the data in a quantitative manner (c). The last stage (d) involves representing this analysis in a meaningful way.

In general the work described in this thesis required the development of particular empirical methods to respond to areas of interest that were being identified. There are a number of general observations that can be made about this process that apply across the board. The first comment is to accept that, in dealing with a new area, one is
having to cope with a large body of material for the handling of which only minimal representations and structures exist. This inevitably results in a considerable cognitive load for the experimenter and also means that one must anticipate one's findings without having the background knowledge that is really required for this. The need for pilot studies is, therefore, all the more crucial than in more classical experimental work. On a number of occasions in analysing the work described in this thesis, important ways in which the studies could have been improved or strengthened were identified and more effective use of pilots might have avoided this. Closely related to this is the observation that one will be unlikely to generate a water-tight, fundamental, conclusive experimental design when working in a domain that is so unstructured. The second point is one that has been forming the undercurrent of much that has gone on previously and was made by Carroll and Campbell (89). There is a danger in research to place too high a value on the stamp of approval of more traditional scientists resulting in a desire to be seen to do "good science". The methods that arise from this, while being formally acceptable, will often not achieve the goals that one has set. If this is the case it is better to be realistic about this and to work with less formal approaches if they can be demonstrated to deliver the desired goods.

8.4. A Discussion of the Results of This Work

Having noted that there are a number of lessons that come from this work relating to research methodology, the relevance of the observations made to the different domains of interest is now explored. This is introduced, firstly, by a description of the domains to which the results are relevant.

The work described in this thesis can be taken at a number of levels as was mentioned in Chapter 1. At one level it represents a contribution towards the design of any tool that aims to perform the function of Automatic Programming by accepting informal requirements specifications from users and providing them with an implementation. A number of the most crucial issues that will have to be taken into account when considering this goal have been articulated and examined by both the collation of literature and the empirical investigation of the processes involved. As a result of this work, it is now possible to assess the extent to which the ideas embodied in the concept of JEEVES are useful. In particular, one must consider whether they are likely to support the non-programmer in achieving programming power and whether they can be used in other programming contexts.

The work described in this thesis also has wider implications. The idea of making programming power available without any cost is one of a number that exist about how
computers might be in the future. The observations made have implications, therefore, for the area of "intelligent" interfaces in general. Also, the work has looked at communication and there are some points that can come out of this that are relevant to wider issues relating to this.

Each of these areas is examined in turn to see what the work described can contribute, starting with the most obvious; that of the design of JEEVES-like tools.

8.5. Relevance to the Design of JEEVES

Many of the insights relating to the design of tools for Automatic Programming by informal requirements specification that have arisen from the work described have already been discussed. Each was examined in the chapter that focussed on it and it would not be appropriate to repeat the exercise here. There are, however, some issues that require an overview of the whole user-JEEVES dialogue if they are to be considered effectively, and these have been left to this point. Thus issues of the overall complexity and efficiency of the dialogue are only possible once the requirements of all the main components have been considered. It is also, at this point, important to review the scope of JEEVES as a whole and to consider its feasibility. Firstly, however, issues that relate to the user-JEEVES dialogue as a whole are considered.

8.5.1. Issues Relating to the Dialogue as a Whole

One of the problems that was noted when considering the various tools that already exist to support programming was that the help provided was generally reduced by the complexity their presence added to the interaction. Thus, while advanced debugging aids may provide the programmer with valuable information with which to locate bugs, they are often unwieldy and require considerable expertise to use. In considering the various aspects of the user-JEEVES dialogue, there would appear to be some danger of falling into a similar trap here. While JEEVES may be able to reduce the effort users must put into directly constructing the program, it does not allow them to withdraw from the process entirely. They must still provide a set of requirements at the start and assess the implementations proposed to identify whether they are going to meet them. The work described in Chapters 5 to 7 has shown that this is not likely to be possible without users having acquired relevant skills that must, inevitably take time. However well the interface is designed this must be the case.

Two basic areas of skill can be identified in any interaction. One must identify both what to communicate and how it is to be communicated. The arguments presented in
Chapters 5 to 8 lead to the identification of a number of factors that suggest both of these must require users to have achieved some level of expertise in the areas in question. Thus, for example, in Chapter 5 it was argued that users are likely to have to acquire some knowledge of the Program Range that will enable them to generate appropriate Virtual Program elements, while in Chapter 6 it was found that any means of communicating these would probably be either inappropriate or unfamiliar. This was on the assumption that the interpretative power of JEEVES is not likely to be qualitatively different from that of advanced computers currently available. If, however, it were to become possible to construct computers that really could understand the inputs of users better even than other humans, then one can only speculate about how this would affect ease of use. Some of the issues relating to such ideas are discussed in section 9.5. What must be decided, for the present, is whether such qualitative leaps in technology are necessary for JEEVES to become a reality. There would appear to be strong arguments therefore that without them such tools would end up by being merely complex in different ways from current programming tools.

The complexity issue relates mainly to the amount of expertise users must acquire before they can use JEEVES effectively. There is another aspect of the cost of using JEEVES that must also be considered, however, and that is the actual time taken up in the interaction. The risk here is basically that the distance of users from the actual program construction means that they must invest considerable time in making sure that the communication of the Virtual Program is effective and that the Proposed Program is what they want. The sum total of the time involved in these activities may not be much less than that of designing and coding the program directly.

The extent to which this is the case will depend greatly, as with the complexity of the interface, on the power of JEEVES to interpret the users' instructions appropriately without their having been extremely pedantic. If it can do this, then there will be minimal need for clarification dialogue and users will know that they have just to describe their needs, give the Proposed Program a brief examination, and tell JEEVES to implement it. If it cannot, then the users will become frustrated and disillusioned and prefer more traditional methods. The ability of JEEVES to perform in this way will depend, not only on its interpretative powers, but also on the contents of the Procedures Library. If the building blocks are well chosen then the programs that are generated in response to the users' instructions will be good candidates for ones that will meet the users' needs. In particular, if the larger building blocks can be used, then
there will be less reliance on the interpretative powers and less coding to be done than if
the programs must be constructed from smaller units.

It is also worth noting that clarification dialogues may occur at other levels than that of
the actual program description. If users are not using a set vocabulary, then every input
runs the risk of being misinterpreted, both requirements specifications and evaluatory
comments. Under these circumstances, there may need to be many iterations associated
with each instruction if JEEVES cannot interpret them correctly. This, combined with
the possible need to evaluate numerous Proposed Programs will also add to the length
of the actual dialogue.

One final point about the factors that might make users dissatisfied with JEEVES relates
to the program that they end up with. When programmers construct programs in the
traditional manner and they fall short of meeting their needs they have only themselves
to blame and, because of their inevitably intimate knowledge of the program, will have
some understanding of the source of the problem. They are also more likely to have
established more realistic goals for the program at the outset, if not during the coding
process. The users of JEEVES, however, by their being more distanced from the
programming process, may be in more danger of feeling dissatisfied with the
Implemented Program. Also, their poor evaluatory skills leave them more open to the
risk of ending up with a program that has some unidentified weakness in functionality
or usability. The implications for this shift of responsibility over the program cannot
fully be anticipated but do need to be taken seriously. It can be seen immediately that
the reality will be that the software manufacturers will be seen as much more
responsible for the programs their clients generate than they are now.

Once again, if JEEVES can correctly interpret users' needs and generate programs to
meet them, then there is less danger of this. This is questioned in Chapter 5, however,
when the issue of embedded user models is addressed. The problem with providing a
computer with the ability to interact with all users, and in particular with any first time
users, is that currently it only accepts a limited set of instructions. These must be learnt
by users before they can communicate effectively with it. Each potential user has,
however, a ready-made means of communicating provided the communicee can
understand it. This need not necessarily be spoken English but may be a particular
formal notation, etc. For a computer to interact effectively with a first time user on the
first contact, it must therefore be able to accept and correctly interpret the user's
notations. This must involve having the ability to generate an appropriate model of the
user that can be used to assist this interaction in both informing the computer of the
meaning of the inputs and in generating outputs that will also be in the user's notation. It was argued in Chapter 5 that the science of embedded user models is still a long way from the point where they will be really useful in this way. This means that one may never avoid misinterpretation and the need for extensive clarification dialogues in human-computer interaction. Even if computers could correctly interpret the users' descriptions of their requirements, there remains the fact that users may well not really know their needs until they have used the program for a week or two and found that it does not meet them. It is likely, however, that JEEVES will not be a paragon of this type but have failings and generate inadequate programs (as human programmers often do). Under these circumstances, users will have good reason to feel dissatisfied with it and seriously reconsider the traditional approach to program generation.

This section has pointed out that computers that aim to provide Automatic Programming of the form envisaged might not meet all the needs one would wish them to. Given this, it is appropriate to consider its likely scope of use. Where would such an approach to program generation be most appropriate, assuming that one does not have access to computing power that is qualitatively different from that currently available?

8.5.2. The Scope of This Approach

There are four basic continua along which one could locate JEEVES-like tools. These are those of the domain of use of the programs, the size of the programs, the expertise of the users and the number of users. Each of these will be considered in turn in the following paragraphs.

If JEEVES-like tools are to be acceptable, the users must not be likely to want to control the details of the programs too much. Related to this is the need for satisfactory stereotypical programs that already exist in the domain, combined with users having general familiarity with their structure and standard ways of describing them. If great control is required then users are likely to be frustrated with the distance that must inevitably exist between them and the programming activity. If, however, they have a basic set of requirements and just want a program that will meet them, then they are more likely to be happy to pass control over the design and construction of it to the computer. From the opposite perspective, as has already been discussed, JEEVES would be more likely to be able to generate a satisfactory program with minimal dialogue if large building blocks that are likely to be good enough can be provided in the Procedures Library. Finally, the choice of domain is likely to have an effect on the ability of users to specify their requirements. In some areas they may know very clearly what they want, while in others their needs are less immediately obvious. Each
of these three aspects should be considered, therefore, when deciding whether a given domain of use would be appropriate for a JEEVES-like approach.

The next aspect to consider is that of the size of program that is best generated by JEEVES. The extremes would be small macros that exist within larger applications for performing frequent but tedious tasks and the vast systems that are used by large organisations to handle their data. It would seem likely that in both of these cases a JEEVES-like approach would be inappropriate. In the former, the effort required to generate the macro by more traditional means would probably not be much greater than that for communicating the requirements to the tool, etc.. In the latter, the importance of the program and the need for control would necessitate a more direct approach and cutting corners would not be worthwhile. In between these, however, lies a large area of program needs that may usefully be amenable to a JEEVES-like approach, always assuming that the problems described in the last section are successfully addressed. For such programs the issues mentioned in the previous paragraph would be the best for choosing the appropriateness in each case.

Related to the size of the program is the number of people working on it. Programs for large systems are designed and coded by teams of Software Engineers. Would a JEEVES-like approach be appropriate for a team project? Always assuming that JEEVES could handle it, the reduced need for the team members to coordinate their inputs and the advantages of more people evaluating a program could indicate that this approach would work well in such a situation. JEEVES would need to accept requirement specifications from the different team members and to be able to cope with the inevitably inconsistencies and conflicts. This is, however, a whole new area that would require considerable investigation and would rely on JEEVES being a viable proposition for single users first.

The last question about the scope of JEEVES relates to the level of expertise of the users. Up until now it has been assumed that JEEVES would be a tool aimed primarily at non-programmers with minimal programming expertise. JEEVES-like approaches are being investigated, however, that are aimed, not at making programming available to non-programmers, but at reducing the work of the already skilled programmer (eg. MacApp). The examinations of the different stages of the user-JEEVES dialogue have identified a number of problems for the unskilled user and indicated that users may actually be required to acquire a considerable amount of skill to use JEEVES effectively. This suggests that it may not be such a success for non-programmers and may not meet the goal that was originally set for it, unless considerable advances in
computer interpretative power become available. Whether this is true or not it is also worthwhile considering whether it would be of use to those who were more willing to invest time in learning to use it and who may already have some programming expertise. The question in this case becomes one of assessing the benefits of this approach compared with existing methods. These have already been discussed and found to embody certain difficulties. In addition to these it may also be that programmers would not appreciate the reduced control that they have over the program generation process. This would need to be investigated and is related to the notion of trust that is discussed in the next section.

8.6. Relevance to the Future of "Intelligent" Interfaces in General

It is suggested, therefore, that computers are not likely to become advanced in the way that is envisaged by Licklider (60) and others within the foreseeable future. It is valuable, however, to consider the effects of trying to push computers to their limits of capability, however, in order to see what this would really mean for users and society in general. Thus, one can consider the implications for computers, like JEEVES, where users are not required to learn a language of interaction beforehand and where the computer takes over a larger part of the task to be performed. A number of such issues are discussed in this section.

It is clear from the discussion that has gone on that the interaction involved will involve a considerable amount of trial and error and misunderstandings. Interpretations will approximate to what was wanted and not result in immediate assimilations of the meaning the user intended. While this is, essentially, similar to human-human dialogue, it may never achieve that level of success. Whether this sort of dialogue really represents an improvement on current, more precise forms has already been questioned in relation to programming. It is a question that is relevant generally, however. Ultimately, it is likely that there will be domains of computer use where control and a need for precision of expression are not so important and where the costs of this sort of interaction are acceptable. In these areas "interpretive" dialogues may well become increasingly used. In others, however, they may prove a dismal failure. It is also likely that the choice will depend on the intended user group.

It has been noted in relation to JEEVES-like tools that users will require help if they are to be expected to generate appropriate requirements and to evaluate the tool's outputs. However, as other "interpretive" computer tools are designed, the area of communicating the scope of the tools and assessing their outputs will become increasingly central to the design of the dialogues. They may, eventually, replace the
concern over choice of command name or menu choice as the main issues of interface design, if such dialogues become dominant. This must follow directly from any such shift of interaction type as, in all cases, the situation will become one where the user is not given an explicit statement of the available functions expressed as a list of commands. As has been argued, the computer cannot be expected correctly to interpret users' inputs every time. This will result in users having to be more careful to assess that the output they obtain from the computer will really meet their needs. It has been found, however, that end-users do not respond in a predictable manner to information about the range of available facilities and that they may be poor at evaluating the computer's output. If this is the case, then considerable research will be required to identify how to cope with this situation and to manage the dialogues so as best to support users through it.

Closely related to these matters is the fact that users of such tools would be working at a greater distance from the actual provision of their needs. Rather than finding ways to achieve their own goals, they would merely be informing the computer of them and letting it decide what is best. For such tools to be worthwhile, users will have to accept some of their decisions without checking them. Otherwise, they will be spending as much time checking the output as they would have had to spend in generating it with a conventional application. Thus, while the positive side is less effort on the part of users in communicating their needs to the tool, the negative side is the necessity for quite a different power and control relationship between user and tool. Users will have to learn to trust their tools to an extent that is currently not common. This generates a novel design dilemma: whether to design the tool such that users will trust its output without checking, or whether to encourage and support users in checking. The question comes down to the extent to which the tool is trustworthy.

This is not a minor matter as it involves users relinquishing control to some extent. This is part of the trend where computers become able to replace humans in more and more tasks. The change is subtle and gradual but could be seen as the thin end of a very dangerous wedge. One could imagine a future scenario where more and more decisions and tasks are left to computers and computer users are less and less able to identify whether the computer's output is correct. This is particularly true as the areas of problem solving involved will increasingly be such that there is no single correct answer but come down to a matter of judgement (Partridge, 88). While one can ask a computer for a bank balance and check easily if it is correct, it is not so easy to check advice on investments. Even when the tool can be shown consistently to generate outputs that do meet the users' needs they may still not feel entirely happy with the
situation. Rare occurrences of writing that attempts to examine these issues exist (eg. Council for Science & Society, 89) where points are made about the increasing reliance on the way the tools are originally programmed and the entering of appropriate knowledge structures. However, there is very little evidence of this being considered seriously in the scientific community. The most common medium for expression of such fears lies in science fiction. Science fiction brings latent fears and dilemmas to life (Griffiths, 80) and forces one to think of issues in advance of the technology that will generate them, rather than waiting until they are upon one. It is worthwhile, therefore, to examine such material to see what ideas need to be considered.

In his chapter on robots and computers, Griffith (80) suggests that the main aim of stories with these as the central objects is to explore what makes humans human. However, he points out that there are, in fact, two general areas of such material. In one, robots, etc., are anthropomorphised considerably, while in the other they are seen as very much under the dominance of their instructions or rules. While the former class of story may well be exploring whether a machine can ever be called "human" or "alive", the latter seem more to be exploring the potential dangers of more realistic machines. Thus, in 2001: A Space Odessy (Clarke, 68), the "infallible" HAL 9000 computer starts to malfunction and things on board the space ship it is controlling go badly wrong. Asimov was the first to explore robotics seriously and probably his main contribution was the "Three Laws of Robotics" (Griffiths, 80):

* A robot may not injure a human being, or through inaction, allow a human being to come to harm.
* A robot must obey the orders given it by human beings except where such orders would conflict with the first law.
* A robot must protect its own existence as long as protection does not conflict with the first or second law.

The very need for such laws is indicative of the potential state of affairs that could prevail if robots become developed to any extent. However, Asimov goes on to demonstrate that such laws are easily side-stepped and that they present one with a myriad of further difficulties rather than solving them. Thus, for example, one must be careful to define "harm". How important is emotional harm as against physical harm? If this is taken to its extreme, robots would be forced to take over all human activities because they could potentially lead humans to harming themselves (eg. cooking, working, etc.).

275
Ultimately, the problems all stem from the need for the original programmer correctly to foresee the effects of each programming decision on the computer/robot's behaviour, which is an unrealistic expectation. In the absence of this, one cannot trust that the decisions the computer takes will be appropriate in the circumstances. As more and more decisions are handed over to the computers, more and more uncertainty must follow and more reliance on the original programmer. While Licklider (60) could foresee a time when computers do not need people to solve problems, there are likely to be considerable social as well as technical barriers to this ever being a successful reality.

8.7. Relevance to Cognitive/Social Science

The final area where the work in this thesis could be said to contribute is that of general cognitive science and social science. The relevance to the former will depend on insights generated about individual cognitive processes and to the latter on insights about human-human interaction and communication. The main areas of interest to cognitive science could be said to be those relating to the way information about a domain affected behaviour in relation to that domain, and to the processes of evaluation. The apparent preference to use objects and artifacts as a means of communication between experts and non-experts is something that could be seen as an underlying theme throughout the thesis and is relevant both to cognitive and to social science. These various areas are now discussed to highlight the observations made.

Two main points came from the studies described in Chapter 5 that looked at how information about a domain of possibility affected specifications of requirements. One was that the information provided is not used in anything like a formal manner but rather as a stimulus to the recall of possible solutions that could be adapted to meet the perceived constraints. Thus a description was provided that was intended to act as a skeleton upon which the flesh of the desired program could be added. However, this was used by subjects to stimulate previously experienced solutions of a similar type, or that could be related to it. Thus, the actual solutions suggested consisted of adaptations of the skeleton, often in ways that were not truly within the constraints imposed. This could be seen as a "generate-and-test" approach that is in contrast to one that relies on "analysis-&-synthesis". This leads to the second main observation about the process of specifying requirements for an item within an uncertain domain. It appeared as if it must necessarily be an interactive process between the solution generator and the holder of the domain information. In particular, there was a need to test proposed solutions constantly with respect to the constraints that existed, even when the constraints had already been articulated. Having said this, there still remains a lot of potential work that could explore the interaction between the information provided and the solutions.
suggested. As was pointed out earlier, as well as any benefit to less applied science, this will be increasingly necessary with the advent of more "intelligent" computers.

While some understanding of how one should evaluate programs and other artifacts is continuously being developed, the everyday occurrence of lay users assessing items about which they have much less expertise seems much less studied. The work carried out in Chapter 7 attempted to redress this and identified some useful concepts and issues. In particular, the competing tendencies of assessing the tool by comparing with actual needs, and just noticing features of interest that are much less associated with actual needs. It is not clear whether this represents a dichotomy or just two ends of a continuum and the factors that influence which end is selected are still very unclear. The very fact that these issues have been articulated, however, means that one can now go on and look more closely at them and start to build up a more detailed and realistic understanding of what is going on and how it can be influenced. Once again, this has both pure and applied importance and is worthy of much further study.

By far the most interesting matter that has arisen from this thesis relates to the use of object or artifact in communication. In many respects this is not a new idea and exists within the folk expertise of many domains. The prevalence of the use of prototypes in many design disciplines is just one example of this. However, as a psychological phenomenon, it does not seem to have been given much attention. Its relevance is very wide as it seems to represent the tip of the vast area of communication between people or agents with different perspectives and expertise. Thus, the use of artifacts can also occur when inter-disciplinary communication is involved, such as between psychologists and computer scientists, when working on human-computer interfaces. However, the use of concrete examples is a common phenomenon in most communicative settings when one of the parties perceives that communication has broken down using more abstract ideas. It would be extremely valuable, therefore, to understand this in more detail and to be able to articulate different ways in which it can occur.

The work that has come closest to this is that by Susan Star (Star & Griesemer, 89, Star, 89). She has examined the ecology of large research establishments involving workers with many different perspectives, such as research museums and hospitals. She has commented on the fact that the people working in these institutions all have different perspectives, goals, time horizons and expertise, yet they must cooperate in a way that results in each achieving his/her goals. This involves negotiation and the identification of common terms of reference. Star suggests two main conceptual
entities that are relevant to this. One is that of "Method Standardisation" and the other is that of "Boundary Object". The former basically involves disciplining all parties to follow certain procedures when, for example, recording a particular case, that, while being slightly in line with their individual goals also supports the goals of the other interested parties. Thus, while doctors may want to record the progress of their patients for their own records, it will be much more valuable if the records have enough detail and are laid out in such a way that researchers doing surveys can also use them. Boundary Objects (objects that can be used by people on different sides of a boundary) include such "Standardised Labels" (the forms, etc.). Other Boundary Objects have also been identified, however. These are Repositories (such as museums), Ideal Types (such as maps and atlases that contain mutually valuable, abstract information about a domain), and Coincident Boundaries (such as county frontiers, that can be used by all parties to delineate areas, even if the items contained that are of interest differ). Clearly the meaning of the term "object" is different from that used in the discussion previously. However, it seems reasonable to see the insights obtained in this thesis as adding to the categories of Boundary Objects that Star has identified.

It may be, however, that such objects are only used in particular situations. Thus, one could suggest that such artifacts are used mainly when the dialogue is between a client and a supplier or consultant of some form, or else between pure and applied scientists. It would be very valuable to identify the conceptual nature of the dialogue that must pertain for the use of this sort of object to be worthwhile. In addition to this, it would be useful to identify the way in which such objects facilitated communication. What seems to be happening is that the usual reliance on shared understandings of terms used is by-passed by the fact that each party can interpret the artifact in his/her own terms. As was seen in Chapter 7, it also reduces the reliance on imagination which can be lacking in those who are not very familiar with such items. It would be interesting to know whether there is more to it than this, however, and what the other elements might be.

8.8. Conclusions

The thesis started with a description of a problem combined with an image of a solution to that problem. The problem was that of company managers who want to make use of the facilities of their computers in ways that lie outside the possibilities of the standard packages, but that do not have time to program them. The solution was suggested as being in the form of an imaginary computer (labelled JEEVES) with which managers would somehow be able to work to generate the desired programs (Figure 8.2). Little more has been assumed than this about the nature of the solution, other than that it
should not be qualitatively different from present-day computers. The work of the thesis has been to explore the most likely source of a solution to this problem and to investigate the issues that will determine its nature and, indeed, whether it is ever likely to become a reality. The main thesis that can be proposed, therefore, based on the work described, is that it is unlikely that anything of a JEEVES-like nature will be forthcoming in the foreseeable future without major advances in computer technology and Artificial Intelligence in particular.

![Figure 8.2. The initial solution that was proposed.](image)

While this may seem to represent a rather negative conclusion, it is worth noting that John's problem may well be best solved by a human expert with a prototyping tool (Figure 8.3). This could provide the dialogue envisaged without the training needs for John. This suggests that the value of prototyping tools that already exist may well increase and that they will be found in new niches within the community of computer users.

It is hoped that the value of this work will go beyond this observation. The human-computer interaction industry has a number of similar "Holy Grails" that are assumed to be just a technical break-through away. The implications of these goals are not trivial and need to be addressed. Indeed, it is important that the HCI community examine the work that is going on in the development of advanced computing systems in general in an attempt to identify what other areas need to be examined in advance of the technology becoming available. In this way, it may well be possible to lay the groundwork for a psychological understanding of the issues that will emerge once the human-computer interface to these technologies has been developed.

279
Figure 8.3. The proposed solution for John.
"Oh, Jeeves," I said; "about that check suit."
"Yes, sir?"
"Is it really a frost?"
"A trifle too bizarre, sir, in my opinion."
"But a lot of fellows have asked me who my tailor is."
"Doubtless in order to avoid him, sir."
"He's supposed to be one of the best men in London."
"I am saying nothing against his moral character, sir."
I hesitated a bit. I had a feeling that I was passing into this chappie's clutches, and that if I gave in now I should become just like poor old Aubrey Fothergill, unable to call my soul my own. On the other hand, this was obviously a cove of rare intelligence, and it would be a comfort in a lot of ways to have him doing the thinking for me. I made up my mind.
"All right, Jeeves," I said. "You know! Give the bally thing away to somebody!"
He looked down at me like a father gazing tenderly at the wayward child.
"Thank you, sir. I gave it to the under-gardener last night. A little more tea, sir?"

From: The World of Jeeves by P.G.Wodehouse (Barrie and Jenkins Ltd. 1967), Chapter 1 - Jeeves Takes Charge p.18.
REFERENCES


Eason,K.D. (1973) The Manager as a Computer User, HUSAT Research Centre, rep. no. 44.


Appendices

Appendix I  Material from the Programming-Client Study:
A) The Task Descriptions Given
B) Detailed Description of the Taxonomy Used
C) Two Examples of SNOBOL4 Programs Used
D) Graphs of the Dialogues

Appendix II  Material from the Bibliographies Study:
A) Summarised Descriptions of Subjects’ Past Experience of Reference Systems
B) Summarised Descriptions of Subjects’ Ideal Reference System
C) The Dialogues between the Subjects & the Experimenter
   Represented in Terms of the Different Activities & the Duration of Each
D) The Relationships between the Needs & Specifications across the Different Stages, for the Different Subjects

Appendix III  Material from the Study Comparing Notations
A) The Description of Flowcharts & JSDs Used
B) The Flowchart & JSD of the Recipe for Making Lasagne Used in Tasks One & Two
C) The Diagram Segments Used in Task Three
D) The Cake Recipe Used in Task Four

Appendix IV  Material from the Study of End-User Evaluation
A) Demonstration of the Diary of the Psion Organiser II Model LZ
B) List of Basic Diary Tasks that Psion Could not Support.
Appendix I

Material from the Programmer-Client Study

A) The Task Descriptions Given

Diary

You want a tool to help with your time allocation and scheduling. By this I mean the way you divide up your time between the various tasks, interests and goals you have. This would include lectures, essays, projects, leisure, eating, revising, etc. You should be able to give it whatever information it needs and it would give you an output that might consist of a timetable for the day, week, etc.

Reading Expt

You want to carry out an experiment to look at the differences in reading method used by fast and slow readers.

Your experiment consists of an initial test of reading speed designed to divide the subjects into fast and slow readers. You will then want the subjects to be presented with three reading tasks involving text that has been degraded or altered in three different ways. Finally, the subjects will be given a comprehension test to see how much of the material of the three reading tasks got through to them.

The computer should be programmed to run the experiment more or less independently of you. It should provide the subjects with instructions, both at the outset and for each test, as well as the displays of reading material. It should make any measurements of time taken that you require and note the answers to the comprehension questions.

Think about what data you would want the computer to store and how you want it to give you access to it. Do you want it to perform any analyses or to display it graphically at all?

Colour Blindness Expt

You want to use a computer to carry out a test for colour-blindness.
This involves presenting the subject with a series of coloured displays with letters or numbers hidden within them. For each display, the subject should indicate the letter or number that he/she sees.

The computer should be programmed to run the experiment more or less independently of you. It should provide the subjects with instructions, both at the outset and for each test, as well as the display that will be in the form of coloured slides. The computer should note the subject's response for each slide. Certain slides are key colour-blindness testers and the computer should note what proportion of the subject's responses are correct, both for the key ones and for the rest. So long as the subject achieves more than 90% on the general battery of slides, a score of 50% or less on the colour-blindness slides means the subject is colour-blind. These latter are divided into two groups for the two types of colour-blindness - type A and type B - and the computer should, at the end of the testing period, be able to tell the subject whether or not he/she is colour-blind and to specify which sort.

Auditory Perception Expt

You want to run an experiment looking at how exposure to part of a sentence affects understanding of a partially heard word at the end. It may be that having heard a string of words that leads one to expect a certain final word, one is much more likely to hear a partially heard word as the one one expects. For example, having heard:

Seeing the stone hurtling towards John, Keith shouted out "Tu..."

one may be likely to hear the last part word as "Du..." rather than "Tu...".

You want the computer to present the subjects with the sentence but with gradually more and more of the final word. The subjects should basically write down what they hear. For each sentence, there must be a number of presentations and there will be a battery of sentences that you will design to test certain hypotheses you have.

The computer should be programmed to run the experiment more or less independently of you. It should provide the subjects with instructions, both at the outset and for each test, giving the sentences with less and less of the final word missing and allowing the subject to type in what he/she has heard and indicate when he/she has finished.
Think about what data you would want the computer to store and how you want it to give you access to it. Do you want it to perform any analyses or to display it graphically at all?
B) Detailed Description of the Taxonomy Used

Introduction

When attempting to categorise an utterance, first divide it up into units and then categorise each unit. The dialogues are mainly concerned with the communication of information. The units should be such that:

- the largest size of any unit is one complete utterance of one person only,
- a unit should include only the communication of one piece of information.

For example, if someone gives a list of facts about something, each item on the list is one unit. However, for the most part, the division by units is by phrase. Thus:

"I want a red balloon."

is one unit despite communicating two items of information (what is wanted is red and a balloon).

If an utterance is unfinished but interpretable, then attempt to classify it. If it is unfinished but the meaning or intention is unclear, then leave it unclassified.

In the examples, I have put in italics the part that would be categorised as described in that section.

Requests for Information

This category includes any questions or general requests for information. It may be a statement spoken with a questioning inflection or a command or invitation to give information. These actions can be divided into three main types according to the nature of the response explicitly requested of the other person:

Q1) Yes/No. These are such that the explicit request is for the other person to give an answer of the form either "yes" or "no".

eg. Do you want a print-out?  You need to be able to scroll through?
Q2) **Option selection.** These are such that the explicit request is for the other person to identify which of a number of stated options is the required answer.

    eg. Do you want it red or blue?

Q3) **Information gathering.** These are such that the explicit request is for the other person to specify the answer in his/her own terms. This category should cover all requests for information that do not fall into one or other of the other two. Here the questioner gives the least help about how the question is to be answered. This would include a request for the other person to draw something, etc..

    eg. What colour do you want it?  
    Tell me more about that.

**Multiple Questions**

It can happen that questions are strung together without the other person having a chance to answer each. If this occurs, it may be that:

a) the other person gave an inaudible answer, or the questioner answered the question for him/herself,
b) the intention is to rephrase the question or add to it in order to help the other person answer it or to indicate more precisely what information is required.

If this occurs, attempt to chose between these are for:

a) categorise each successive question separately,

    eg. Where do you want this? And this? = Q3,Q3

b) categorise each part separately but then label them together,

    eg. How do you want it? Do you want it like this or like this? = Q3/2

**Answers**
A) This includes any information given in direct response to a request for information.


Feedback

This category refers to the occasions when a speaker makes a statement that involves saying back to the other something about what the speaker has understood. This can be divided into:

F1) Response to piece of information given by the other speaker with no information or repetition.

eg. I want ... and ...
   Hmm.

F2) Repeating back immediately what is understood. This may involve a certain amount of rephrasing but should not significantly add to or alter the information conveyed.

eg. I want a time manager.
   You want a sort of personal organiser, yes.

F3) Summarising. This refers to when a number of statements of the speaker's understanding are made purely for recapping or ensuring that the understanding is accurate.

eg. So you want a page-a-day diary, a week-at-a-glance and a note-pad. Is that right so far?

F4) Making reference to past information conveyed with a view to using this as a basis for making a point or focussed discussion.

eg. A little while ago you said you wanted a ..., now it seems you are saying...
F5) Making an inference. Here the speaker is feeding back something of what the other said, but with some information added or made explicit. This can be either:

1) The speaker is pointing something out about the physical reality of what the other wants, or the implementation of the other’s request.

   eg. I want it to remind me of my lectures.
   So you are going to carry this around with you.

2) The speaker is referring to the implied goal of a given request.

   eg. I want it to bleep me.
   So you want an alarm to remind you of things.

3) Any other statement of this type.

Information Volunteered

11) These occur immediately after an answer to a question, and relating directly to it.

   eg. What colour do you want it? Green, well green on top but red underneath.

12) Giving information related to last piece of feedback. This could be (dis)agreeing with the feedback, adding to it, elaborating on it, etc..

   eg. So you want it green. Yes, green, possibly a dark green, if you've got one.

13) Spontaneous, i.e. not directly related to any immediately previous utterances.

   eg. Do you want it green?
   Yes please. I'd also like a print-out.

14) Elaborating on information given previously, within last five lines, i.e. giving related information.
Now, I want it green. In fact, I'd like it dark green all over.

15) Giving information that will help the other person answer a question (including suggesting a hypothetical situation).

What colour would you like? By colour, I mean the paint used.

Meta-Comments

M) These are comments that effectively control or area about the dialogue or topic of conversation.
C) Two Examples of SNOBOL4 Programs Used

Counting the Number of Pf4's

NEW B =
  X = 0
  PATT = LEN(3) . ACT

* OUTPUT = "ENTER THE NAME OF THE FILE TO BE ANALYSED"
INSTRING = INPUT
INSTRING = TRIM(INSTRING)
INPUT('DATA',7, ,INSTRING) :S(OK)
OUTPUT = 'COULD NOT FIND FILE' :N(NOVEL)
OK FPATT = LEN(6) . NAME REM . OTHER
INSTRING FPATT
OUTSTRING = NAME '9.DOC'
OUTPUT('OUT',8, ,OUTSTRING)

* A = 0

* NEXTG N = 0
  X = 0
  SEARCH = 'pf4'

* MATCH DATA PATT :F(FINAL)
* output = X ' ' ACT
  ACT SEARCH :F(CHECK)
  X = X + 1
CHECK N = DIFFER(N,15) N + 1 :S(MATCH) F(NEXT)
FINAL B = 1
NEXT A = A + 1
  STRING = "THE NUMBER OF pf4's IN GROUP " A +" IS " X

* OUTPUT = STRING
OUTPUT =
OUTP = STRING
OUTP =

* OUTPUT = DIFFER(B) 'END' :F(NEXTG)
NOVEL OUTPUT = 'DO YOU WANT TO DO ANOTHER FILE? (YES/NO)'
UPPERS = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
LOWERS = 'abcdefghijklmnopqrstuvwxyz'
ANSWER = TRIM(INPUT)
S = REPLACE(ANSWER, LOWERS, UPPERS)
T = 'YES'
OUTPUT = IDENT(S,T) 'UNDERSTOOD':S(NEW)
END

Counting Number of Information Items from Notes in Successive Groups of 15 Items

NEW &ANCHOR = 1
PATT = LEN(2) . AGENT '"', LEN(1) . NOTES

OUTPUT = "ENTER THE NAME OF THE FILE TO BE ANALYSED"
INSTRING = INPUT
INSTRING = TRIM(INSTRING)
INPUT('DATA', 7, INPUT) :S(OK)
OUTPUT = 'COULD NOT FIND FILE' :(NOVEL)
OK FPATT = LEN(6) . NAME REM . OTHER
INSTRING FPATT
OUTSTRING = NAME '22.DOC'
OUTPUT('OUTP', 8, OUTSTRING)
NEXT A = A + 1

STRINGY = "THE NUMBER OF Y'S IN GROUP " A " IS " Y
STRINGN = "THE NUMBER OF N'S IN GROUP " A " IS " N

OUTP = STRINGN
OUTP = STRINGY
OUTP =

OUTPUT = STRINGN
OUTPUT = STRINGY
OUTPUT =

OUTPUT = DIFFER(B,0) 'END' :F(NEXTG)

NOVEL OUTPUT = 'DO YOU WANT TO DO ANOTHER FILE? (YES/NO)'
UPPERS = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
LOWERS = 'abcdefghijklmnopqrstuvwxyz'
ANSWER = TRIM(INPUT)
S = REPLACE(ANSWER,LOWERS,UPPERS)
T = 'YES'
OUTPUT = IDENT(S,T) 'UNDERSTOOD' :S(NEW)

END
D) Graphs of the Dialogues not Given in Chapter 4
Parts of the Program

Overview

Diaries

Day Diary

Week-at-a-glance

Memo Pad
Parts of the Program

Overview

Introduction

Experiment Proper

Task 1

Task 2

Task 3

Results

Time

Session 5

Totals

1

1

16

6

8

13

10
Parts of the Program | Time | Session 6 | Totals

Overview | 8 | | 16

Instructions | 2 | 2 | 4 | 9

Text Presented | 7 | 3 | 4 | 12

Questions | 2 | 2 | 2 | 7

Pretest - Text Presented | 2 | 3 | 3 | 7

Pretest - General | 2 | 2 | 2 | 7

Pretest - Questions | 2 | 2 | 2 | 7

Other - Text Presented | 6 | 2 | 9

Other - General | 2 | 2 | 2 | 7

Other - Questions | 3 | 6 | 9

Results | 3 | 3 | 3 | 9

End | 1 | 1 | 12
Appendix II

Material from the Bibliographies Study

A) Summarised Descriptions of Subjects' Past Experience of Reference Systems

Subject A described two experiences:

Computerised. Semi-relational. Supported the analysis of numerical data, presenting fields as columns, sorting alphabetically, for entry into a book.

Computerised. Relational. Used to find addresses quickly.

Subject B described one experience:

No database, just actual papers kept. No organisation stated. Supported creation of reference lists for papers written. Supported making note of papers to read or obtain. Use it by: if he knows the reference, then write it straight in, if not go through his own papers, if not go through those of others that might have the reference in and then try to get hold of the paper itself. Too cumbersome to maintain a database separate from the reference lists he creates in papers.

Subject C described one experience:

Card-based. Ordered alphabetically by author. With name, initials, title, date, publisher, source. Supports doing literature reviews - for pulling out relevant items. Used by going through them all, pulling out relevant articles. Subject claimed to know roughly where the relevant ones would be.

Subject D described one experience:

Computerised. Unknown organisation. Supported finding articles either on a given topic or finding specific articles. Used by performing computer search by keywords or author(s) and then getting list of titles and location numbers, then getting the article by location number. The subject did not actually set up the database, and was just involved in getting information out of it.
Subject E described one experience, but mentioned some knowledge of another computerised database in a later stage:

Card-based. Organised loosely by topic, often jumbled together. Had author, date, title, source, read/not, summary. Supported finding articles on given topics. Used by flipping through them all, reading the summary to see what each is about. Supported generating literature reviews, because organised by author and date. Supported reminding subject of what had been read.

Subject F described two experiences:

Card-based. Organised by chronological number, both cards and actual papers. Also have special cards with number and subject. Supports finding articles on a topic or by author. Good because of easy placing of papers. Can spread cards on table and organise them. Used by using special cards that each refer to one topic or author and give number of cards about them. Can look for cards by author on a topic by finding numbers common to both special cards. The subject is not using this system any more as it took too much maintaining. It was difficult to deal with new topic areas.

None - just papers. Papers organised by topic and alphabetically by first letter of author. Also boxes of papers that are roughly known about. Supports finding desired articles. Used by just looking through the appropriate box or draw.

Subject G described one experience:

Card-based. Organised alphabetically by author. Has information about reference plus where it was cited and what papers it cited. Supports finding articles related to a given topic or paper. Supports building up a picture of an area. Used by using citations information to find related items.
Appendix II B

B) Summarised Descriptions of Subjects' Ideal Reference System

The main requests that each subject made are given here as well as explanatory comments (in bold) that are seen to be significant.

Subject A:

1) keywords
2) titles
3) authors
4) all authors in separate fields
5) categories of paper so can search within a category
6) search by keyword, title, author
7) keywords and key phrases for broad and narrow searches.

Subject B:

1) Have it reasonably accessible
2) Enter references of papers that look interesting. Type it in and it tells if already seen it. If not then write to library to get it.
3) Keep record of what not obtained yet and remind of ones not obtained.
   Straight-forward card index plus facilities.
4) Function for creating reference list, so can go through and indicate which are to go in.
5) Have it capable of printing out those specified in order
6) Print out in appropriate format for journal.
7) Would not use keyword system for searching

Subject C:

1) logically organised with name first,
2) select items automatically,
3) type name, access all such names and pick the desired one, get the whole reference
4) or get all references for an author.
5) involve in library system - acces to its info also,
6) delete a reference,
7) accessing all references before a given date - using date as search parameter.
Subject D:

Basically only imagine the same thing but with improvements.
1) Get abstracts of articles - so not have to get the actual article out
2) get printouts
3) capability to recognise relations between words and ask if these are wanted also, when doing keyword search
4) also be able to cut out those not really relevant.

Subject E:

1) show lots of cards on screen at once, so can sort and look at all.
2) less constrained way of putting info in - cf SuperFile
3) call up word processor
4) sophisticated search facilities for topics - enter rough description (not precise words) and get all ones in area
5) tag cards into groups so can group easily and pick out groups.

Subject F:

1) optically scan stuff in to get things in quickly.

Finds entering a fag
2) with character recognition - not nec whole article
3) author, keyword, summary, conclusion, space to enter ideas, keypoints
4) then assign a number to it and be able to find the article by the number
5) store all sorts of info about article’s contents, eg conclusions, methods used
6) able to retrieve specific article.
7) Combining search criteria

Problems in past of entering and keeping tidy.
8) windowing facility - for looking at several things at once
9) v. large screen
10) with good resolution
11) have icons for different windows with different topics in, or different search facilities
12) group items together in it and indicate that they form a group of some kind, then this info be stored with references. using icons in new ways to indicate new concepts, have this information temporary or permanent

Like spreading cards on a desk
13) have word processor open at same time.
Subject G:

1) find items via areas
   tried to do this with cards but too hard
2) keywords and access via these
3) work by name links. put one name in and get what they have written on their own,
   then with others, then what others have written, etc. so can scan that way
4) go in alphabetically.
5) move around where you are,
6) not have to jump out and in
This is annoying in other systems
7) go in via menus of subject, or quickly etc so can see what is available.
C) The Dialogues between the Subjects and the Experimenter
Represented in Terms of the Different Activities and the Duration of Each

The dialogues between the subjects and the experimenter were allocated into the five
different types of activity:

M = Meta-dialogue
C = Concepts, or descriptions of the HyperCard- facilities
B = Boundaries
P = Problems, or discussions of the requirements of the subjects
S = Solutions, or discussions of design of the stack

The dialogues are represented graphically as a series of these types of activity with the
number of lines of text that they consisted of, when transcribed, to the nearest whole
line. If it is not quite clear which type of dialogue is occurring at some point blocks to
represent both of the alternatives are given.
Subject A

M = Meta-Discourse, C = Description of HyperCard-
B = Boundaries, P = Requirements, S = Design Solns.

Subject C

M = Meta-Discourse, C = Description of HyperCard-
B = Boundaries, P = Requirements, S = Design Solns.

Lines from Transcripts

Appendix IIC
Subject B-1

MCIa-Dialogue, C=Description of HyperCard-
B=Boundaries, P=Requirements, S=Design Solns.

Subject B-2

Appendix II C
Subject F - 1


Subject F - 2
Subject F - 3

M = Meta-Dialogue, C = Description of HyperCard-
B = Boundaries, P = Requirements, S = Design Solns.

Lines from Transcripts
Subject G - 1

M = Meta-Dialogue, C = Description of HyperCard
B = Boundaries, P = Requirements, S = Design Solns.

Subject G - 2

Lines from Transcripts

Lines from Transcripts
D) The Relationships between the Needs and Specifications across the Different Stages, for the Different Subjects

The following pages give graphical representations of the dialogues observed in this study in terms of the different needs expressed and specifications made during the different stages of the study. Lines have been drawn to indicate where there appeared to be a source for a specification in an earlier expressed need or specification. Encircled items are those for which there was no evidence of a need or previous experience that led to that specification.

It can be seen from these diagrams that very many of the items specified in the final stage could be said to have their origin in comments made earlier about needs or problems with other systems.
Data analysis

Reformatting for book

Sort alphabetically by different fields

Sort by:
- Keywords
- Title
- Author

Categories of papers & search with category

Keyword & key phrases - for broad and narrow searches

Finding items quickly

Print out specific fields

Simple find using text items

Appendix II D
Create reference list for papers

Note items to be read and get hold of them

Reasonably accessible

Reference of papers seen and help get them and remind when not got them

Create reference list - go through indicating which + print out in order and in format

Separate authors fields for searching

Find cards using keywords - create a separate stack

Link to word-processor and do word search to check all there

In-tray of new cards automatically transferred when filled in

Keep integrity

Search by keyword, year and any field

Other fields including keywords

Print stack

Indicate card for references - separate stack

Print stack
Subject C

Pull out relevant articles - go through them all

Organised
Select items automatically

Access by author
Pick from list & get it all

Access by author
Or get all for an author

Search by entering text to search for

Back track to subject of author for search

Use date as search parameter

Delete

Involved in library

Description of Past Reference System(s)

Appendix II D
Subject D

Find article on topic or author

Find specific article

To get at actual article and read it

Get abstracts

Relations/meaning of keywords - to add or to ignore

Get printout

Index with logos about what is there

Different order and flip through

Want list - printout

Add fields?

Erase

Add
Finding article on topics - flip through
Show lots of cards on screen at once
Can sort them
Less constrained way of putting information in
Call up word processor
Not more than one per screen?
Link related cards using keywords
Simple search by rough description
Tag cards into groups and easily print out groups
Overview - list of those found

Subject E
Appendix II
(Superfile)
Finding related articles by co-author and citations & references

Ordered alphabetically

Find items by areas - keywords access

Name links - put in name and get authors

Go in alphabetically

Move around where you are not jump in and out

Go in via menu of subjects etc.

Information via authors use "Find" to find an author

Search for an author by noting co-authors

Fields - citations, references

Search for topics - keywords
Appendix III

Material from the Study Comparing Notations

A) The Description of Flowcharts and JSDs Used
Flowcharting

These sheets teach you how to use a notation known as Flowcharting. This is a way of structuring instructions by dividing them up into separate steps.

START and END Boxes

Any flowchart describing a set of instructions for a task must have one START Box and at least one END Box of the following form:

![START and END Boxes](image)

These are used to make it clear where the instructions begin and end. More than one END Box will be necessary if, for example, there are two possible outcomes of some event (see Decision Boxes) and they result in two possible ways of ending the process.

Process Boxes

These boxes are used for any process or activity that forms part of your instructions. The processes specified by each Process Box should be as simple as possible in order to make the best use of the flowchart structure. This is illustrated with two examples of possible Process Boxes plus (on the following page) a comparison of two flowchart structures where option A) is to be preferred to B).

Click the switch  Unscrew the bulb
Ordering Arrows and Following the Instructions

The different boxes are linked by arrows that indicate how whoever follows the instructions is to move through them. There is no strict rule for having the order go from top to bottom or left to right, as is shown in the following examples where the numbers indicate the order in which the boxes are to be executed. This example is only to show that it is the arrows that determine the order, not the physical relationships - it is not a recommended way of laying out a flowchart.
The main rules for following a flowchart are:
A) Look for the START Box and start there. There is nothing to execute here, so move from it immediately.
B) In order to move from a START or Process Box, follow the arrow coming out of it to the box it is pointing at.
C) If you move to a Process Box, execute it and then move from it.
D) If you move to an END Box, the instructions are finished so leave the flowchart.

Some instructions (for changing a light bulb) given in this notation are given followed by a description of how one would follow them.

Start with rule A) and find the START Box.
Use rule B) to move to box 1).
Use rule C) to execute "Turn light off".
Use rule B) to move to box 2).
Use rule C) to execute "Remove old bulb".
And so on until:
Use rule B) to move to END Box.
Use rule D) to leave the flowchart.
**Decision Boxes**

These boxes are used to indicate that a decision is required with the next step being determined by the outcome of this. They are of the following form:

![Decision Box Diagram]

Having Decision Boxes requires the use of two further rules for following a flowchart:

E) If you move to a Decision Box, decide on the currently true answer to the question and then move from it.

F) If you are moving from a Decision Box, follow the arrow labelled "Yes" if the answer to the question was 'yes' and the arrow labelled "No" if the answer was 'no'.

Decision Boxes can be used to make two courses of action possible according to the situation or to allow for repetition of a series of actions or a continuous process with some sort of stopping rule. The example above is of the former type. Two examples are given on the next page of the latter types. After each, an explanation is given.
The first one is a flowchart of the instructions "stir the mixture until it is smooth". Note the small circle after the "No" decision. This is needed purely because of the limitations of the package you will be using and can be thought of as a Process Box with no actual activity to execute, so you move straight on up to the "Stir a bit" box.

This second example shows how you could write a flowchart for the instructions "Peel and chop four carrots". The more processes there are to be repeated, the greater the advantage of using this approach. For example, you might want to make four cakes by repeating the same actions each time which could be very repetitive. The approach requires the use of an imaginary counter that enables you to keep track of how many times the process has been carried out. When it has been performed the required number of times, you move on.
A flowchart should never have a dangling arrow with no box at its head. Thus, for example, any Decision Box must have an outcome for both possible answers. Also, the only discontinuities can be START and END Boxes. Two illegal flowchart structures are shown below. The errors are highlighted.
Jackson Structured Diagrams

These sheets teach you how to use a notation known as Jackson Structured Diagrams. This is a way of structuring instructions by dividing them up into processes of different sorts arranged in a hierarchical manner.

Tasks and Subtasks

In this notation, tasks are organised into tasks and subtasks. This means that your overall task should be written in a box at the top with a row of boxes below it containing descriptions of the different subtasks that make it up. Thus, the link between one box and the ones joined to it by lines is that of "is defined as" or "consists of". This is illustrated below:

There can be any number of levels of tasks and subtasks like this and it is generally good practice when writing instructions in this notation to group tasks together in this way. For example, in the diagrams below, A) is better than B).

A)

B)
Process Boxes

Most of the boxes that are used in JSD's describe simple actions that are to be performed one after the other. The order in which they are to be read is from left to right. This is illustrated in the diagram below where the numbers indicate the order in which the boxes are executed.

This principle applies at all the levels of the hierarchy as shown:

Thus, in the example above, turning the light off is performed before removing the old bulb and, to turn the light off, you first walk to the switch and then click it. If the only structures in JSD's were Process Boxes, one would need only to go along the bottom row of boxes to follow a set of instructions. However, because there are other boxes that affect how process boxes are executed, more precise rules are needed for following JSD's.
Thus, to follow a set of simple instructions of this nature, you will always be doing one of four things:

A) If you are at a Process Box that has subtasks, you must move to the first subtask (i.e., that to the far left).
B) If you are at a Process Box that has no subtasks, carry out the instructions in the box.
C) If you have carried out the instructions of the Process Box you are at, move to the next one (one to the right) on the same level. This also applies if you have moved back up to a Process Box after having carried out all its subtasks.
D) If the Process Box you have just finished with is the last subtask of another box, return to this box.

To illustrate this, I will take you through the example given (repeated below), referring to the boxes by their numbers and the actions by the letters.

1) - A) (move to first subtask (1.1))
1.1) - A) (move to first subtask (1.1.1))
1.1.1) - B) (no subtasks so just carry out the action), C) (finished with this box so move to next one)
1.1.2) - B) (no subtasks so just carry out the action), D) (last subtask of 1.1 so move back up to 1.1)
1.1) - C) (finished with this box so move to next one)
1.2) - A) (move to first subtask (1.2.1))
1.2.1) - B) (no subtasks so just carry out the action), C) (finished with this box so move to next one)
And so on.
**Decision Boxes**

Should it be required to indicate that one of two operations should be performed according to a decision that must be made, the options are made subordinate to a Decision Box. An example is given:

Is the socket a screw-fitting?

- Yes: Use a screw-fitting bulb
- No: Use a bayonet bulb

The Decision Boxes add two further rules to the list of possible actions that determine how the instructions of a JSD are to be carried out:

E) If you arrive at a Decision Box, instead of carrying out the two boxes that derive from it in order, only one is carried out according to the result of the question contained in the box. In this way it is to be considered as a box with only one subtask so that once this (including any subtasks) is performed you use C) or D) to move on from the Decision Box.

F) If the question in the Decision Box results in you moving down a line that has nothing at the end of it (see below for an example), you just use C) or D) straight away to move on.

Is the light on?

- Yes: Turn it off
- No:
Repetition Boxes

If you want to indicate that some operation(s) will need to be carried out more than once, or have a process that is to be continued for a period of time, you should use one of these. The Repetition Boxes themselves contain "stopping rules" that indicate how to tell when the repetition should be stopped. Two examples are given:

- Until it is free
  - Turn the bulb in the socket

- 5 times
  - Turn the bulb once round in the socket

These boxes result in another rule about how to interpret instructions:

G)Whenever you meet a Repetition Box, whether after a move down from a higher level box or a move up from a lower one, examine the stopping rule in the box. If it holds, use rule C) or D) to move on from the Repetition Box. If it fails, use rule A) to move down to the first box below.
For a final example of how the rules work, consider the diagram above.

You start at box 1) and move down from it. This takes you to box 2), a Decision Box, and you examine the question inside. If the answer is "No", you have nowhere to go so return up to 1) and end. If the answer is "Yes", you move down to 3), a Repetition Box.

You examine the stopping rule. If the rule holds, you move back up to 2), which you ignore, and return to 1) and end. If the rule does not hold (e.g., the washing is not yet dry), move down to 4) which you execute. From this you move to 5), which you also execute. From this you return to 3). You examine the stopping rule. This may still not hold, in which case you repeat the last few steps. If it does hold, you move back up the hierarchy and end.
B) The Flowchart and JSD of the Recipe for Making Lasagne Used in Tasks One & Two
Get 6oz lasagna
Get 5oz cheese

Get half pint milk
Get 1oz flour

Get 1oz butter
Get 1 teaspoon mustard

Get 8oz mince
Get 1 onion

Get 4 tablespoons tomato puree
Get 1 pepper

Get 2oz mushrooms
Get some oil

Chop the onions
Chop the mushrooms
Chop the peppers

Fry onions and mushrooms in the oil a bit

Are the mushrooms and onions soft?
Yes
No

Add mince, puree and peppers
Melt the butter in a small saucepan
Add the flour
Fry the flour a bit

Is the flour like sand?

Yes

Is there any milk left?

Yes

Boil the sauce while stirring it a bit

Yes

Put half the mince sauce into the dish

Is the sauce thick?

Yes

Grate the cheese

No

Add half the cheese

Stir it all in

Heat a bit

Is cheese melted in?

Yes

Put the lasagna in the oven

No

Put half the mince sauce into the dish

Cover with pieces of lasagna

Put other half of the mince sauce over the lasagna

Cover with pieces of lasagna

Pour the cheese sauce over it all

Cover with remaining grated cheese

Cook it for 40 min. or until brown on top

Serve immediately
Get ingredients for lasagna

- 2 oz mushrooms
- 1 pepper
- 4 tablespoons tomato puree
- 1 onion
- 8 oz mince
- 1 teaspoon mustard
- 1 oz butter
- 1 oz flour
- 1 oz cheese
- 6 oz lasagne
- Half pint milk
- Some oil
Make Lasagna

Prepare cheese sauce

Adding the milk

Thickening the sauce

Adding flavourings

Grate the cheese

Add half the cheese

Stir it all in

Heat

Decide on a small amount of milk to go into a cup to start with

Until there is no more milk to add

Repeatedly adding small amounts of milk and stirring it in, adding a bit more each time

Is remaining milk less than the small amount?

Yes

Put all remaining milk in the cup

No

Put small amount of milk into the cup

Until evenly mixed in

Is the mixture lumpy?

Yes

Stir the milk into the contents of the pan

No

Increase the small amount that is to go into the cup

Until the lumps are gone

Stir it in

Is remaining mixture lumpy?

Yes

Beat the mixture hard

No

Put small amount of milk into the cup
Putting It together

Cooking lasagna

Put half the mince sauce into the dish

Cover with pieces of lasagna

Pour the cheese sauce over it all

Put other half of the mince sauce over the lasagna

Cover with remaining grated cheese

Serve immediately

Put the mince sauce into the dish

Cover with pieces of lasagna

Pour the cheese sauce over it all

Put other half of the mince sauce over the lasagna

Cover with remaining grated cheese

Cook it

Cook the lasagna in the oven for 40 min. or until brown on top

Cook it
C) The Diagram Segments Used in Task Three
4) Melt the butter in a small saucepan
1) Add the flour

2) Frying the flour

3) Fry the flour

5) Until the flour is like sand
2) Thickening the sauce

6) Until the sauce is thick

8) Boil the sauce, while stirring it

3) Thickening and adding flavourings

4) Adding flavourings

7) Grate the cheese

1) Add half the cheese

9) Add the mustard

5) Stir it all in
A3: Is remaining milk less than the small amount?

- Yes: Put all remaining milk in the cup
- No: Repeatedly adding small amounts of milk and stirring it in, adding a bit more each time

Until evenly mixed in

Stir the milk into the contents of the pan
4) START

8) Melt the butter in a small saucepan

3) Add the flour

5) Fry the flour a bit

2) Is the flour like sand?
   - Yes
   - No

6) Decide on a small amount of milk to go into a cup to start with

1) END
6) START

3) Boil the sauce, while stirring it a bit

2) Is the sauce thick?
   - Yes
   - No

9) Grate the cheese

4) Add half the cheese

1) Stir it all in

8) Add the mustard

5) END
1) START

8) Is remaining milk less than the small amount?
   Yes → 5) Put all remaining milk in the cup
   No → 7) Stir the milk into the contents of the pan a bit

5) Put all remaining milk in the cup

7) Stir the milk into the contents of the pan a bit

3) Is it mixed in?
   Yes → 4) END
   No → 2) Put small amount of milk into the cup

2) Put small amount of milk into the cup

4) END

6) END
D) The Cake Recipe Used in Task Four

Quick Chocolate Cake


(1) Ingredients

- 125g self-raising flour
- 1 teaspoon baking powder
- 125g caster sugar
- 125g soft margarine
- 2 large or 3 small eggs
- 25g cocoa powder
- 1 teaspoon instant coffee powder
- 3 tablespoons very hot water

For the icing

- 100g icing sugar
- 25g cocoa powder
- 50g butter
- 3 tablespoons water
- 75g sugar

(2) Set the oven at gas mark 4. Use two buttered or non-stick 18 or 20 cm diameter sandwich tins.

(3) Sift the flour and baking powder into a large mixing bowl, add sugar, margarine and eggs. Mix the cocoa and coffee powder to a smooth paste with water just off the boil and add to the bowl.

(4) Starting on the lowest speed, begin to mix the ingredients with an electric mixer, and as soon as they begin to blend increase the speed and beat at top speed for one minute only.

(5) Divide the mixture between the tins, spreading the mixture evenly and smoothing the top, and bake for 25-30 minutes, until the tops of the cakes are firm. Turn out on to wire trays and leave to cool.
(6) Sift the icing sugar and cocoa into a bowl.

(7) Put the remaining ingredients into a small saucepan and heat gently, stirring 'til the sugar is dissolved. Bring to the boil and immediately pour into the bowl, stirring to a smooth, thin cream.

(8) Leave to cool until the cake is ready to be iced; the mixture will thicken as it cools.

(9) When the cake has cooled, use half the mixture to sandwich the two cakes together and spread remaining icing over the top.
APPENDIX IV

MATERIAL FROM THE STUDY OF END-USER EVALUATION

A) Demonstration of the Diary of the Psion Organiser II Model LZ

The Organiser is turned on by pressing the ON key at the top left hand corner of the keyboard.

(Turn Organiser on.)

This will bring up a display of the Main Menu. Menus are the main way by which functions are selected in the Organiser.

There are two ways of selecting a function from a menu. By typing the first letter of the function or by using the cursor. In this demonstration, functions will be selected by means of the cursor.

The cursor is currently flashing over the "D" of DIARY. This means that diary is the selected program. The cursor can be moved around using the arrow keys on the keyboard.

(Move cursor around menu until it is over OFF.)

Once you are over the program you want, you press the EXE key at the bottom right of the keyboard.

(Press EXE.)

This has resulted in the function OFF being selected which is how the Organiser is turned off.

(Turn the Organiser back on.)

If you had selected one of the other programs of the Organiser, such as the Month, ...

(Select Month.)

you return to the Main Menu by pressing the CLEAR button, which is the same as the ON button.

(Press ON.)

Notice that at the Main Menu, you have the time at the top right hand corner of the screen (point to it) and a small symbol representing the Organiser at the top left (point to it). This shows one is at the Main Menu.

The diary (select diary) initially shows a display of the current week. This shows each day very crudely as four blocks of time periods. Thus, the first block is from
midnight to 6 am, the second from 6 am to midday, the third from midday to 6 pm and the last from 6 pm to midnight.

There is a cursor here also and one can move it to the day and block one wants to make an entry for or examine and press the EXE key.

(Move cursor to 21 July and last block and press EXE.)

One can see that the entry was "Jon's birthday party" from 8 pm to 11 pm.

On either side of the entry are free slots which enable one to see the size of the block of time allocated. If one wants to make an entry, one moves to the free slot, presses EXE, types in one's entry and presses EXE again.

(Move to free slot above, press EXE and type DENTIST. Press EXE.)

One can then set up the time for that entry as a block of time with up to 15 mins. accuracy. If one is going to the dentist at 10.30, one starts it at 10.30 and may want to allow half an hour for it - 10.30 to 11.00.

(Set up time for entry to 10.30 to 11.00.)

It then asks you if you want an alarm associated with this entry. Press Y for yes.

(Press y.)

One can then have a reminder up to one hour in advance of the time of the entry. We will give ourselves 20 mins. notice.

(Increase time to 20 mins.)

In order to get at the menu associated with a facility in the Organiser, you press the MODE key, next to the ON key.

(Press the MODE key.)

This will present a menu of functions that can be used with this program, running along the top or bottom of the screen. The menu for the diary is at the top of the screen.

The FIND (move cursor over it) function allows one to find an item based on a short piece of text contained in the item. Thus, one can type in JON to be taken to JON's birthday.

The GOTO function (move cursor over it) allows one to specify a particular date and one is taken to the diary entry for that date.

COPY, CUT & PASTE are used together (bring them all into view). With the CUT facility, the entry that is currently selected is cut out and placed in a separate
memory, known as a clipboard. The COPY facility also puts a copy of the entry into the clipboard but leaves the original one in place.

Once one has an entry in the clipboard, one can move to another day and select PASTE. The entry is then inserted into that day. There is no limit to how many times the contents of the clipboard can be pasted in.

The ALARM facility (move cursor over it) is for changing the alarm associated with an item to either on or off.

TIDY (move cursor over it) will delete all diary entries prior to the current date.

PRINT (move cursor over it) obtains a print-out of the diary, if one has the Organiser printer.

SAVE and RESTORE (move cursor over it) are to do with keeping records of entire diaries. With SAVE, one can save a file containing all the diary entries. RESTORE will then restore any such file into the diary.

XRESTORE (move cursor over it) works the same as RESTORE but works on files saved on a previous Organiser model.

SETUP (move cursor over it) allows one to specify the boundaries of the blocks of time used in the week-at-a-glance display.

(Return to the Main Menu.)

The Month program (select it) provides a basic calendar of the current month and other months. The cursor will be flashing over the current date but can be moved to any other date.

(Move the cursor.)

By pressing the EXE key, one is taken to the diary entry for that date.

If one moves outside the month, one is taken to the next month (do it). Similarly, one can move to parts of the month that are not previously visible (do it).

There is a menu associated with this (press MODE). This is much the same as the diary menu.

(Return to the Main Menu.)

The SAVE program (select it) allows one to save items consisting of a number of lines of text, like addresses into a database. One just types the item in and presses EXE.

(Type in entry: Joe Bloggs, 23 Back St, Hull. Press EXE.)
The FIND program (select it) allows one to retrieve items from this database by means of strings of text they contain. One types in the string and presses EXE. One is then given all such items.

(Type in: Joe, then press EXE until no more entries.)

(Return to the Main Menu.)

The NOTES program (Select it) allows one to keep lists of shorter, one line, text items.

(Type in: beans, eggs, milk, fish fingers.)

There is a menu associated with this program.

(Press MODE)

The FIND (move cursor to it) function works like the other FIND functions.

SAVE, LOAD and NEW (make all visible) allow one to set up a number of notepads. One selects NEW to create a new notepad and give it a name. One uses SAVE to save the current notepad after entering items. Having more than one notepad means that only one can be in use at any one time so the others are stored separately. If one wants to use a notepad other than that currently visible, one selects LOAD to obtain it. The menu item DIR, at the other end of the menu gives a list of all notepads stored. The menu item COPY, also at the other end of the menu allows one to copy notepads to datapaks.

HOME and END (make all visible) take one to the beginning and end of the current notepad.

CALC (move cursor to it) allows one to perform certain calculation on notepad entries containing figures.

SORT (move cursor to it) orders the notepad entries alphabetically and NUMBER numbers them.

PASSWORD (move cursor to it) provides password protection for a notepad.

PRINT (move cursor to it) obtains a printout of the current notepad.

DELETE (move cursor to it) can be used to get rid of any notepads and ZAP (move cursor to it) empties the current notepad. Both of these include checks to avoid unintentional data loss.

It is possible to have a particular notepad available directly from the Main Menu. There is a notepad called "Work" on which this will be demonstrated. One does this from the Main Menu, selecting the location in the menu one wants it to go and pressing the MODE key.
(Go the the Main Menu, move to the last place and press MODE.)

One types in the name of the notepad and indicates that it is a notes item.

(type in WORK and select Notes.)

The notepad's name then appears in the Main Menu and by selecting it one is taken directly to it.

(Select Work.)

This is the end of the demonstration.
B) List of Basic Diary Tasks that Psion Could not Support.

* Not handling symbols and other characters than text.

* Not allowing for time-flexible entries, but possibly with deadlines, etc..

* Not allowing for entries covering the whole day or like birthdays, etc..

* Not allowing for parallel entries.

* Limit to what can be stored with an item. esp cannot have extensive notes associated with day.

* Not being able to have it open all the time.

* Not allowing storage of other info or carrying of papers, etc..

* Not displaying whole day's items.

* Alarm limited to one hour before - no facility for having automatic reminders further in advance.

* Limited info in week display.

* Limited info in month display.

* Not storing task information, just appointments.

* Not distinguish different categories of item