

This item was submitted to Loughborough University as an MPhil thesis by the author and is made available in the Institutional Repository (<https://dspace.lboro.ac.uk/>) under the following Creative Commons Licence conditions.



For the full text of this licence, please go to:
<http://creativecommons.org/licenses/by-nc-nd/2.5/>

Pilkington Library

Author/Filing Title HUMPHREYS, C.A.

Accession/Copy No. 040152615

Vol. No. Class Mark

~~2nd copy~~

LOAN COPY

0401526151



BADMINTON PRESS
UNIT 1 BROOK ST
SYSTON
LEICESTER LE11 1GD
ENGLAND
TEL: 0118 260 2017
FAX: 0118 260 0039

**INVESTIGATING PROGRAMMING TOOLS AND
PROCESSES; DESIGNING VISUALLY POTENT
PROGRAMMING TOOLS**

by


C.A. Humphreys

A Master's Thesis

Submitted in partial fulfilment of the requirements
for the award of

Master of Philosophy of Loughborough University

October, 1997

 Loughborough University Library
Date Feb 98
Class
Acc No. 040152615

9/9161234

Investigating Programming Tools & Processes; Designing Visually Potent Programming Tools

Abstract

Thesis : Programmers do not yet possess the full complement of tools needed to fulfil the requirements of the programming task; particularly during code development and debugging. It is my contention that programming tools should exploit typographic effects further. Specifically, by: boosting the visibility of selected objects, simplifying information extraction tasks, and enhancing visual rapport with program text.

First year MacPascal student programmers rated MacPascal's programming tools, and commented on deficiencies and enhancements. A model was drawn correlating programming processes and the tools provided by a typical programming environment; with the aim of specifying functional gaps and/or mismatches. Speculation on filling these various requirements resulted in the design of 3 visually potent tools, namely: spotlighting, summary tables and layout aids.

Spotlighting. Most editor's search mechanisms only allow the viewer to locate ONE instance of the specified "search object" at a time, in sequence, using the cursor as locator. In contrast, the spotlighting tool would illuminate ALL instances of the specified "search object" within the text - using inverse video or colour - without reference to the cursor position. Thus each instance is visible in context, and the viewer can read and move around the text freely, according to his own strategy . The spotlighting tool was tested using paper experiments; student programmers' comments showed that it focused attention effectively, and helped them follow a spotlighted variable's trail through the code.

Summary tables present a program's declaration data from different perspectives. Specifying an attribute class causes a menu-list of its members to pop up. For example, listing the name of every integer variable; or listing local variable names alphabetically. Giving the programmer immediate access to accurate information in the required context.

Layout Aids. A questionnaire on debugging strategies showed that the layout of program text either aids or confounds program comprehension and the subsequent debugging efficiency of student programmers. Difficulty in understanding and debugging programs increases in proportion to the divergence from the reader's preferred layout style. Thus visual rapport is a significant factor in program comprehension and subsequent processes. Hence the need for a tool to produce (or reformat) code according to each individual's preferences.

Investigating Programming Tools & Processes; Designing Visually Potent Programming Tools

Chapter 1 Introduction, Aims of Research, & Thesis Summary	1
1.1 Core Concepts	2
1.2 Principal Aims of Research	2
1.3 Summary of Main Points Raised	3
1.4 Summary of Thesis Contents	5
Chapter 2 Literature Survey - Human Factors, Software Design & Programming	9
2.1 Human Factors & Human-Computer Interaction	9
2.1.1 Human Factors	9
2.1.2 Human Information Structures, Processing & Perception	11
2.1.3 User Interface Design & Task Models	12
2.1.4 Psychological Issues and Cognitive Engineering	14
2.1.5 User Characteristics and Differences	16
2.1.6 Visual Aspects of Cognition	17
2.1.7 Applying Typography to VDUs	18
2.2 Software Development & Programming	20
2.2.1 Programming Stages & Processes	20
2.2.2 Programming Knowledge	21
2.2.3 Programming Plans & Composition	21
2.2.4 Comprehension	23
2.2.5 Debugging	24
2.2.6 Errors	26
2.2.7 Programmers at Work	27
2.2.8 Programming Tools, User Aids & Task Assistants	29
2.2.9 Aspects of Tools	31
2.2.10 Aspects of Program Text	31
Chapter 3 Preliminary Data Collection - Observation & Analysis of Student Programmer's Reactions to MacPascal's Tools	34
3.1 Students' Problems Observed During Programming Practice Tutorials	34
3.1.1 Summary of Students' Problems	34
3.1.2 Discussion of Problems Observed	37
3.2 Design & Analysis of MacPascal Questionnaire	38
3.2.1 Dimensions of MacPascal Tool Calibration & Rating Scale Descriptions	39
3.2.2 Analysis of Quantitative Data	40
3.2.3 Interpreting the Data With Regard to Students' Comments	40
3.2.3.1 Interpreting the Data Graphically & Numerically	40
3.2.3.2 A Closer Look at Tool Frequency Data	43
§3.2.3.2 Data Table & Summary: Student Votes for Frequency of Use - in Order of Decreasing Mean Values	44
3.2.3.3 Summarizing the Mean Rating Scale Data	46
3.2.4 Consensus of MacPascal Tool Deficiencies or Necessary Enhancements	50
3.2.4.1 Summary of Problems and Deficiencies	50
3.2.4.2 Summary of Enhancements Suggested in Questionnaire	52
3.2.5 Attitudes Towards MacPascal's Tools	55
3.2.6 Consensus of Student Opinions on Comments and Program Design, & Prior Use of Computers & Programming Languages	56
3.3 Conclusions Resulting from Preliminary Data	61

Chapter 4 Relating Integrated Editing Tools & Cognitive Programming Tasks	65
4.1 Examining Editing Actions	65
4.1.1 Adding/Inserting Text	65
4.1.2 Deleting Text	65
4.1.3 Modifying Text	66
4.1.4 Moving Text	66
4.1.5 Moving The Cursor	66
4.2 Comparing the Editing Tools of MacPascal & Unix's Vi	67
4.2.1 Allocation of Mutative Editing Functions	67
4.2.2 Comparing How MacPascal & Vi Operate	68
4.2.3 Summary of Editing Tool Usage	69
4.3 Reprise of Programming Tasks From A Programmer's Viewpoint	70
4.3.1 Programming & Editing Shortcuts	70
4.3.2 Software Design in the Real World	71
4.3.3 Summary of Typical Design/Coding Strategy	72
4.3.4 Summary of Design Stages	73
4.3.5 Pre-execution Actions	74
Chapter 5 Discussion of Possible Tools	75
5.1 Programming Problems and Possible Tools	75
5.2 Choice of Tools For Further Discussion and Development	78
5.2.1 Layout Aids	78
5.2.2 Interpretation Aids - Summary Menus	79
5.2.3 Visibility Aids - Spotlighting	80
5.2.4 Moving Aids	83
5.2.5 Conclusions	84
Chapter 6 Design & Application of Proposed Tools	85
6.1 Conceptual Design & Tool Relevance	86
6.1.1 Layout Aids	86
6.1.1.1 Examples of Layout Variations	88
6.1.2 Spotighting	90
6.1.3 Summary Menus	95
6.1.4 Combining Spotighting & Summary Table Methods	98
6.1.5 Correlating Spotighting & Summary Tables With Programming Errors	98
6.2 Applying Spotighting & Summary Table Tools to the Signal Problem	99
6.2.1 Statement of Siddiqi's Signal Problem	99
6.2.2 Applying Spotighting to the Signal Problem	102
6.2.3 Applying the Summary Tool to the Signal Problem	105
6.3 Integrating Spotighting & Summary Tables Into A Supportive Environment	107
6.3.1 Discussion of Spotighting Implementation Issues	107
6.3.2 Discussion of Summary Table Implementation Issues	110
6.4 Conclusions	111
6.5 Discussion	112

Chapter 7 Further Data Collection & Experimental Evaluation of Tool Feasibility	113
7.1 Design & Analysis of Programming & Debugging Strategies Questionnaire	114
7.1.1 Analysis of the Questionnaire	115
7.1.3 Summary of Questionnaire Results	115
7.1.4 Amendments to Suggested Tools	120
7.1.5 Conclusions	121
7.2 Debugging Experiments	122
7.2.1 Design of Experiments & Experimental Method(s)	122
7.2.2 Student Debugging Strategies	123
7.2.2.1 Analysis of Students' Comments While Debugging	124
7.2.2.2 Reconstruction of Student Debugging Strategies	125
7.2.3 Results of Experiments - Interpretation & Evaluation	126
7.2.4 Comments on Debugging Experiments	130
7.2.5 Conclusions	130
7.3 Spotlighting Experiments	131
7.3.1 Definition of Hypotheses	131
7.3.2 Design of Experiments	133
7.3.2.1 Experimental Options for Testing Spotlighting	133
7.3.2.2 Experimental Tasks	134
7.3.3 Analysing Results Of Spotlighting Experiments	134
7.3.4 Discussion	143
7.3.5 Testing Spotlighting Using Paper Experiments	144
7.4 Results of Post-Spotlighting Questionnaire	144
7.4.1 Summary of Spotlighting Responses	144
7.4.2 Summary of Layout Responses	147
7.5 Discussion of Results	148
7.6 Conclusions	149
 Chapter 8 Summary of Findings & Future Work	 151
8.1 Summary of Main Findings and Conclusions	151
8.1.1 Questionnaire Design	151
8.1.2 Preliminary Data Collection	151
8.1.3 Spotlighting	152
8.1.4 Debugging Strategies Experiments	152
8.1.5 Spotlighting Experiments & Questionnaire Results	153
8.1.6 Summary Tables	154
8.1.7 Layout Aids	154
8.1.8 Final Comments	155
8.2 Future Work	155
8.2.1 Spotlighting	155
8.2.2 Summary Aids	156
8.2.3 Mental Simulation	156
8.2.4 Exploring Aspects of Layout & Possible Experiments	157
8.2.4.1 Cataloguing Layout Styles	158
8.2.4.2 Shadow Code Exploration & Experiments	159
8.2.4.3 Shadow Code Observations	159
8.2.4.4 Shadow Code Tasks	161
8.2.4.5 Example Experimental Method	162

Appendix Contents	Page
Appendix 2 Chapter 2 References	1
Appendix 3A	
- Comparison of Student Numbers & Percentages For Each Tool	20-25
Appendix 3B Additional Questionnaire Data	27
Appendix 7A Programming & Debugging Strategies Questionnaire Data	32
Appendix 7B Post-Spotlighting Questionnaire Data	61
Appendix 7C Debugging Strategies Experiment Sheets	70
Appendix 7D Spotlighting Debugging Task Experiment Sheets	80

Investigating Programming Tools & Processes; Designing Visually Potent Programming Tools

Chapter 1 Introduction, Aims of Research, & Thesis Summary

The Research: Why, what, and who for?

Working as a programmer for 3 years, developing and debugging real-time software on a daily basis, made me increasingly aware that the tools provided by typical programming environments were incomplete. As a consequence, the programmer is forced to expend time and effort on tasks that could be automated.

Simple, but effective programming tools were needed to make specific aspects of the programmer's task easier, faster, and more efficient. Interacting with (electronic) program text is a visual task, and it seemed appropriate to apply visual aids to it. However, such tools had not been provided, and they could not be patched into the editor to fill the void. Ease of use requires integrated tools; tools that are accessible from within the editor, not outside it.

The phrase "visual potency" expresses the need for tools to exploit typographic effects to benefit visual processing tasks that "feed" the various low-level cognitive processing tasks essential to the higher-level programming task. For example, extracting information from (program) text is much easier if it is shown in a format or representation suited to the specifics of the task.

This research focusses on the requirements of the (solo) programmer using a typical programming environment with screen editor and compiler. It is particularly important to aid the programmer in the debugging process, since this is the most time- and cost-consuming phase of software development (Yourdon & Constantine, 1979). Thus, the primary goal of this work was to design tools that support this task by reducing cognitive processing during the debug-edit-compile cycle.

Questionnaires provide quantitative, numerical evidence to support all claims whenever possible (see chapters 3 & 7). All assumptions have been checked, through discussion with others, to make sure that the views expressed here represent as accurate an account of the facts as possible.

The remainder of this chapter explains core (basic) concepts, principal aims of research, the main points raised, and a summary of the thesis contents.

1.1 Core Concepts

Program text has 2 principal (static) elements which interact:

- various program and procedural variables declared as specific data types; and
- the statements and program constructs that test, use, modify and manipulate each variable's value according to the programming plans deployed within that program.

However, control flow, the order in which individual statements are executed at run-time, depends on the range and value(s) of input data processed - this in turn determines which bugs will be revealed. Testing is geared towards revealing bugs, and debugging to exterminating them. However, although testing gives evidence of bugs that are present, it may not reveal all bugs. This is why debugging is such a difficult process, and why the programmer needs all the help he can get.

The visual appearance of the program text, regarding shape and structure, results from the way that the code is laid out on each page, with the continuity and flow of visual shape and pattern from one page to another. Although the various items expressed in the program text contribute to the working (and semantic meaning) of the program, namely: operations, data flow, control flow, state, and functions (Pennington 1987); the layout and visual presentation of the program text also has an effect on the programmer. Leventhal (1988), Baecker & Marcus (1986), and Molzberger (1983) all associate the comprehensibility of the program with its aesthetic appearance. Many studies show that indentation has a definitive effect on comprehension (Mynatt, 1990; Van Laar, 1989; Gilmore & Green 1984).

1.2 Principal Aims of Research

Namely, catering for the solo programmer with a typical program development system, incorporating a screen editor such as Vi, or an editing environment like MacPascal, and compiler. Designing tools that fit the programming task, and programmers' needs more closely than existing tools; filling the gap with new tools and/or "extrapolating" existing tools and concepts.

There are 3 main themes in my research :-

- Using typographic effects to focus visual attention and alleviate those visual processing tasks required to locate all instances of a specific word within a text (spotlighting). Putting that particular word in a spotlight of colour different from the surrounding text, to make it more visible, and instantly locatable.

- Reducing information processing burdens by providing essential information in alternative formats (summary tables/menus). For example, showing lists or tables of "declaration data" information, to enable the programmer to cross-reference between variable names, data types, and parent procedure(s), from the perspective most useful to the current inquiry task.
- Supporting individual aesthetic requirements in the visual presentation of program text (layout aids). Enhancing the programmer's visual rapport with the code as a means of maximizing code readability, comprehension and debugging accuracy.

The overall aim is to produce tools that are better suited to the needs of the programmer and the specifics of the task. In effect, increasing the programmer's ability, satisfaction and productivity in completing a task, by reducing the frustration and mental burdens being created by the inadequate, incomplete programming tools currently provided for the programming and debugging tasks.

The primary goal is to introduce the concepts of spotlighting and summary tables on a conscious level, and to define their essential characteristics. This was done using the editing environment as a discussion vehicle for the implementation of spotlighting and summary aids (see chapter 6); to show how useful spotlighting and summary aids could be. The secondary goal is to create a demand for these concepts to be implemented on other systems, with the further aim of raising an awareness of the power that spotlighting could bring to electronic text processing tasks in general.

1.3 Summary of Main Points Raised

The main points I wanted to raise with this research were :-

In General

- more extensive, but considered, application of typography to electronic text oriented tasks, especially programming;
- more consideration of the programmer's task beyond basic needs;
- correlating programming tasks with the tools provided by a typical programming environment to reveal missing tools or functions that new tools could fill;

Spotlighting

- the spotlighting concept itself - setting each instance of a given word in an inverse video or colour spotlight, so that it becomes more visible and easier to locate within the background text;
- using spotlighting to extend the search/find mechanisms;

- challenging the efficacy of the sequential access principle of existing search mechanisms, by proposing an alternative, complementary random access principle;
- to define the problems associated with debugging, and explain how spotlighting can help - especially with variable trail following;
- raising an awareness of the power that spotlighting could bring to electronic text processing tasks in general;

Summary Tables

- the summary tables concept itself - providing necessary information in alternative formats to save the programmer from wasting time and resources, and the cognitive effort required to do the (cross-referencing) task him/herself;
- using summary tables to avoid misinterpretation (or misperception) of the facts;
- using summary tables to make data typing and other declaration problems easier to detect and resolve;

Spotlighting & Summary Tables

- defining problems addressed by the implementation of spotlighting and summary tables individually, and when combined (see §6.1.5, the table of programming errors vs spotlighting and summary tables applicability);

Layout Aids

- showing the need for each programmer to work on code that is laid out according to his/her preferred style;
- being able reformat existing code into the programmer's preferred style (especially when a programmer has to modify unfamiliar code);
- maintaining (new or pre-existing) code in the programmer's preferred style during development or modification;

By Questionnaire Analysis

- defining specific programming errors/problems that need to be addressed;
- identifying the relationship between reading strategies, comprehension and debugging strategies;
- exposing the importance of **layout style and visual rapport** - making public its effects, positive and negative, on readability, comprehension and the accuracy and efficiency of debugging; and
- identifying the role of mental simulation in program development and debugging, and its importance.

1.4 Summary of Thesis Contents

Chapter 2 Literature Survey

The purpose of the Literature Survey was to provide a reprise of programming tasks from a cognitive viewpoint, as well as reviewing different aspects of the task; such as information processing aspects in terms of visual processing, and cognitive structures like programming plans. Thus it has 2 sections: human information processing, and software development and programming. The aim was to explore both sides of the programming task - the internal and external factors - and how they might interact.

For example, almost all information comes via visual processing, and all output via physical processes. The latter can be defined in terms of programming stages or project goals, and checked for accuracy by programming metrics. However, programmers vary, both in their range of skills and experience, and the type of language they choose to use.

Chapter 3 Preliminary Data Collection & Its Evaluation

The questionnaire of chapter 3 (set in February 1989) was an attempt to find out student's attitudes towards a typical environment. MacPascal was used to teach first year B.Sc. students the art of programming, so it (MacPascal) presented itself as an ideal candidate for investigation. It seemed reasonable to assume that any problems or inadequacies would show up faster when encountered by novice programmers.

The first half of the questionnaire drew out students' opinions on all MacPascal's tools on a 5-point rating scale for each of 5 dimensions. Namely, usefulness, frequency of use, ease of use, likeability and frequency of use of (an)other method in preference to that tool. The remaining questions investigated attitudes towards various programming issues. Such as the frequency of use of each code development methodology, and attitudes towards the use of comments within code.

I spent about 1-2 terms prior to setting the questionnaire attending the first year lectures, and helping out with the practicals/tutorials. Acting as a troubleshooter during tutorials gave me the ideal way of observing the students' behaviour as they went about designing, debugging and testing out their programs; so I was able to observe a variety of programming errors that the student programmers made "live".

Interacting with the students while observing, made gathering informal data on programming and text-editing in real life fairly easy. Some programming errors would disappear with experience, others, such as syntax errors are perennial, and happen regardless of experience. Gathering ratings, opinions and attitudes towards MacPascal's tools provided some surprises (see §3.2.4 & §3.3). The results of these

preliminary findings were used to focus in on the nature of the "missing" programming tools, and fed into the model of Chapter 4.

Chapter 4 Relating Integrated Editing Tools & Cognitive Programming Tasks

Chapter 4 attempts to correlate the range of tools provided by typical editors or programming environments, and the tasks/activities the programmer executes during development and debugging; to see if any missing tools came to light, or whether the omissions had a common factor. For example, the advent of WIMPs as applied to editing made the comparison of 2 different segments of code on screen possible (as with the Emacs and SunTools editors). Up till then this task was only possible with paper text (and origami exercises). I used some of the data from the questionnaire to inform my model, and for guidance in necessary tool functions/functionality.

The Literature Survey provided a reprise of programming across the spectrum. From an information processing and cognitive viewpoint, as well as reviewing different aspects of the task from a software development perspective. Relating information processing aspects in terms of visual processing, and cognitive structures like programming plans. This information was used as background knowledge in attempting to relate the physical activities and cognitive tasks of programming with the editing tools provided.

If the editing tools are completely compatible/congruent with the (cognitive) programming tasks, then there should be a one-to-one or many-to-one relationship between the 2 domains. However, if there is non-congruence at one or more points then this indicates areas of incompatibility and inappropriate or "missing" tools.

Chapter 5 Discussion of Possible Tools

Chapter 5 details the next stage - taking the findings of the previous chapters and using brainstorming to determine a design direction. Discussing all possible directions of research: defining the variety of tools and their relevant aspects, and giving reasons for their rejection or acceptance, and why I chose to develop the following tools, viz what aspects appeared innovative or important, and what area(s) of programming each tool was applicable to.

The main contenders all had a factor that gave some form of visual support or enhancement, that would increase visual rapport, and make visual (and subsequent cognitive) processing less arduous. The spotlighting, layout and summary tables tools emerged from this melée, as well as a list of other tools that should prove useful to the programming and debugging tasks. Most of the other tools considered were "checking" functions with the generic task of "checking ..." then reporting back the results. In the same way (modus operandi) that a compiler reports back error messages corresponding to "suspect" line numbers.

Chapter 6 Design & Application of Proposed Tools

Chapter 6 relates the development of the tool design and redefines the visual issues that are critical to the tools under conceptualisation. Giving an in depth discussion of the conceptual design of each tool and its relevance to the area of application, and its proposed implementation.

For example, spotlighting could be applied to other forms of electronic text, whereas summary tables and layout aids refer mainly to procedural programming languages, such as Pascal, which I have used for demonstration, since it is my preferred language. The interaction between spotlighting and summary tables is also discussed. A table (see §6.1.5) specifies the errors that spotlighting and summary tables are expected to be able to address. Whether any of these tools could be applied to logical languages is doubtful, although they may be of some use to object—oriented tasks. Providing there is sufficient text to work with, and some form of named or identifiable object to spotlight within a background text.

Chapter 7 Further Data Collection & Experimental Evaluation of Tool Feasibility

The subjects who tackled the questionnaire and debugging tasks (set in December 1990) were the finalists - from the same group who answered the questionnaire of chapter 3, but 2 years on in their experience. This chapter falls into 3 sections.

The first section summarizes a detailed questionnaire (see Q2A in Appendix 7) that was used to draw out aspects of students' debugging activities and their attitudes towards various tasks, tools and debugging techniques. Reading and comprehension strategies were also investigated with respect to debugging. Midway through the questionnaire, the students tackled 3 short debugging tasks (see §7.2). The remainder of the questionnaire asked specific questions about debugging strategies. This arrangement was used so that the debugging information required was fresh in the student's minds, and easier to recall.

The second section explains the aims of the debugging experiments and provides a summary of debugging strategies and techniques.

The last section describes various aspects of the spotlighting experiments and their hypotheses. The spotlighting tool was tested using paper-based experiments. A short post-experiment questionnaire (see Q2B in Appendix 7) found the range of subjective opinions about the spotlighting tool, and attempted to gain more details about why individual layout style differs.

Chapter 8 Summary of Findings & Future Work

The thesis ends with a summary of main findings and ideas for continuation of the work. I had hoped to be able to implement the spotlighting and summary table tools, but time ran out. Investigating the role of mental simulation was another area that was drawing attention. It deserves much closer scrutiny. However, I posed a few questions that could reveal its true role, or at least provoke further research.

I also planned to set a series of exploratory experiments to investigate different aspects of layout style, after cataloguing the gamut of stylistic variations. The crux of these layout experiments involved replacing program text characters with a ■. Even a cursory glance at some possible substitutions gave food for thought.

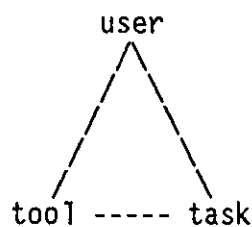
For example, blacking out all characters on a line up to the last alphanumeric (or punctuation) character. Comparing the effect of blacking out leading "indentation" spaces or not was quite dramatic. Losing the indentation characteristics changed the overall pattern of the program text, and made recalling the original text (and its former shape) more difficult than when the indentation cues were present to reflect the original shape of the text. This train of thought could provide a useful insight into the subject of layout style and its effects - both positive and negative.

Chapter 2 Literature Survey - Human Factors, Software Design & Programming

The aim of this research is to design new programming tools. This chapter covers all subjects regarded as contributing to the background knowledge of the work. There are 2 sections. The first deals with the multitude of human factors, including human information processing and the visual aspects of cognition that guide the design of such tools. The second section deals with the programming task itself, its structures, and its associated processes and products; ending with programming tools and user aids, and aspects of program text; since program text is the interaction medium and product of the programming task.

2.1 Human Factors & Human-Computer Interaction

The main prerequisite of a tool is that it is able to aid in achieving the goal either by making the task easier, more effort efficient, or faster and thus more time efficient. The diagram below illustrates unification of the user and the tool to accomplish the task (as originated by Eason 1984, 1988).



The user interface operates between user and tool; the task interface between tool and task; and the skill interface between the user and the task. Human factors covers each of these aspects, with human information processing (see §2.1.2) explaining the internal structures and mechanisms of the user's mental processes.

2.1.1 Human Factors

The central premise of human factors is simple - taking care of the the user's needs and designing systems that fit the user as closely and as comfortably as possible. To ergonomists the comfort of users is of prime importance - socially, physically and psychologically (Coats & Vlaeminke 1987).

The aspect of human factors has many facets, but the one which concerns HCI (human-computer interaction) most is usability (Shackel 1984a, 1984b, 1986), alternatively known as **Matching the user's requirements**. This is the most troublesome aspect of software design because the user is only able to tell you what he thinks he wants and/or needs to carry out any required task (Shackel 1984a, 1986; Brooke 1986; Damodoran 1983). The functions he actually needs may be very

different, or (perhaps worse) varying in a slight but non-trivial way from what he says he wants. In some cases the user may not be able to define his requirements at all in which case intensive (knowledge elicitation) interviews are required to try to establish the user's basic requirements regarding functions and task methods (both "now" and in the new system) (Gould 1987; Thomas 1984; Gardner, Mayfield & Maguire 1984).

Task analysis is used to find out how each task is performed, why, when, and in what order; and what specific information is needed for input and output. Allocation of function is used to define who should do each task - the human or the computer - regarding efficiency, accuracy, and ease of completing the task. Task match measures how well the system matches the requirements of the task and of the person who uses the system to perform that task.

It is important that potential users are educated on the benefits and drawbacks of computer systems, before the design really gets under way. This gives the users new perspectives on how the task may be accomplished. Once people see the range of possibilities, they are in a much better position to give informative, useful feedback, and to help the design along in constructive ways (Grudin 1991; Eason 1982a, 1982b, 1987; Damodoran 1983; Eason, Harker, Raven, Brailsford & Cross 1987).

According to Shneiderman (1987), "Paper mockups are useful to review alternative designs, but online prototype versions of the system create a more realistic review environment". Candy & Edmonds (1989), like Shneiderman and Eason, believe it is vital that user interface design proceeds in an evolutionary fashion, preferably using simulations or prototypes, so that users have a concrete model of what the system will be like and how it will operate (even if the functions are dummies). This enables them to reject designs which are just not suitable, or suggest ways of making the system easier to use or more flexible.

The primary ethos of user centred system design is to meet the human factor needs of the user and his task with a complementary computer system and set of tools (and a means of using them) via the user interface.

Thomas (1984) puts the case for a user centred, human factors approach to design very succinctly, as: "making sure that everything is covered by considering who will use it, what for, and in what context (both organisationally and physically)." Most importantly, he concludes that, "Making the design right in the first place is cheaper than changing it later."

2.1.2 Human Information Structures, Processing & Perception

Understanding the main human information structures and processes is a key issue in tool design, since perceptual processing determines how the tool is perceived.

Human memory is thought to have 3 types of memory structure: long term memory (LTM), short term memory (STM) and working memory (WM). In 1974, Fiegenbaum defined WM as a memory store that is more permanent than STM, but less permanent than LTM, and in which information from STM and LTM may be integrated into new structures.

In the context of programming and the generation of program code, generic programming plans are copied from LTM, and filled with the required details from STM, resulting in a free standing programming plan (Rist 1986) (see §2.2.4).

In 1957, Miller determined that STM could only hold 7 ± 2 chunks. However, these chunks could be quite substantial, since humans have the ability to group a collection of small data fragments into a more meaningful and larger chunk. Even so, Shneiderman (1986) thinks that 5 ± 2 chunks might be a more accurate value. STM uses a rehearsal loop in order to hold temporary information. According to Shneiderman (1980) "One of the by-products of the limitations of human STM is that there is great relief when information no longer needs to be retained. This produces a powerful desire to complete a task, in order to reduce memory load and gain relief." This factor affects all human activities, including programming and the use of tools.

However, LTM is permanent, and appears to have (almost) infinite capacity. LTM has 3 stages - encoding, storage and retrieval (Ormerod 1990). For material stored in LTM, recognition is far easier than recall. Recognition requires the matching of the given item with one already existing in LTM. Whereas recall enforces the extraction and recreation of the required item from LTM, which is much harder.

Lenorovitz, Phillips, Ardrey & Kloster (1984): "Perception deals with the process of getting information into the (human) system, as well as some initial level of recognition/classification/identification of that information. Cognition is concerned with the human information processing activities which users perform upon the information once it has been perceived."

There are 2 levels of cognitive processing: automatic and conscious. The automatic level uses subconscious processing which has a high capacity and operates in parallel, processing regular predictable information. In contrast, the conscious level is used to process new, unpredictable or unfamiliar information - anything that is out of the ordinary. According to Matlin (1989) the subconscious processor takes care of routine tasks, and only in unfamiliar environments and tasks is there need for higher level control of the processing by the versatile but slow sequential (conscious) processor.

According to Treisman (1986) human visual processing also has 2 levels; with pre-attentive vision operating in parallel, and focused attention in serial. Pre-attentive vision is used to locate the stimulus (ie. first sight impressions). Separate processes are used to describe colour, orientation, size, stereo distance etc. in the stimulus. Certain elements pop up out of the background as a result of perceptual grouping or differentiation. Perceptual grouping occurs due to colour and shape effects (Treisman 1982). Pre-attentive vision also detects texture boundaries - changes in the visual stimulus. Focused attention is used in order to recognise objects (using the attention spotlight). In search tasks this attention spotlight is used to distinguish between the background and the search object. This is an important factor regarding the searching of program text, whether on paper or VDU.

Shneiderman's (1987) list of human centred (neural) processes includes: learning, problem solving, decision making, attention and set, search and scanning, and time perception. There are many factors affecting perceptual motor performance, including: arousal and vigilance, fatigue, perceptual (mental) load, knowledge of results, monotony and boredom, and circadian rhythm. Hand-eye coordination is a prime example of perceptual motor performance relevant to the use of tools.

Most of the above factors are addressed by psychological, physical, environmental and organisational ergonomics (Coats & Vlaeminke 1987). Lessening the overall stress on the individual user by supporting him in the task, whilst retaining sufficient task elements to keep his attention without getting bored.

2.1.3 User Interface Design & Task Models

Formalisms and models exist at many levels of description. There are those which describe formal methods of software interface design; those which define the layered nature of the user interface in the form of taxonomies; and others which are a combination of formal methods and design principles.

Design Principles

Thimbleby (1985) described generative user engineering principles (GUEPs) as a means of guiding design with principles that are effective and easy to understand by designer and user alike. Many other design principles have been advanced: Hansen's (1971) design principles are the most well known; Baecker & Marcus (1990) have looked at them with regard to applying graphic design principles to the production (viz visual appearance) of C program text; and Gardiner & Christie (1987) from the cognitive psychologist's point of view. Maguire (1982) compared many guidelines for design, and drew attention to points of commonality and conflict. In cases of conflict, the designer must choose carefully which design principle has precedence or is more relevant to the design problem.

User Interface Design Models

Various layered models of interface design have been mooted: Foley & Van Dam's (1982) model has conceptual, semantic, syntactic & lexical layers; whereas Moran (1981), Clarke (1986), and Norman (1986) all propose triple layer, bipartite models; and Polson, Bovair & Kieras (1987) define a seven layer model. The main problem with user interface design is that it lies at many different levels - like the proverbial 'onion' paradigm originated by Sommerville (1988). Each layer contributes to, and supports the layer above and below it; with bi-directional communication between layers to maintain communication between the user and the system.

Moran's Command Language Grammar (CLG, 1981), can be seen as a layered model of interface design as above, or it can be compared with other grammars. For example, Reisner's BNF-like grammar (1981, 1984), Payne & Green's (1983) Task Action Grammar, Alty's path algebras (1984), Card, Moran & Newell's (1983) GOMS, Unit Task and Keystroke Model. All meet with various degrees of success in formalising interface design, but none are able to fulfil the requirements completely.

Types of Models - Definitions

Models held by users have been investigated by Young (1981, 1985), and Rich (1983), among others. There seems to be general agreement in the literature regarding nomenclature, as follows:

- designer's model - embodying the designer's mental model of system functions
- user's model - the model formed by the user through interacting with the system - defining the user's interpretation of how the system works
- user model - the designer's model of the user, in terms of skill level, task and information needs, and the order in which tasks are to be done.
- system image - the totality of what the user sees, accesses or operates in order to use the system, the VDUs, keyboards, manuals, online help, training - the overall presentation of the system. Thus the system image is an extension (a fuller description) of the user model, whereas the user's model is an interpretation of the system image.

Task models describe the task sequences at a variety of levels from conceptual goal statement to the operational level of procedural steps (Card et al, Moran, Reisner). Predictive task models are related to task models, but refine interaction into steps which can be quantified, to predict task performance (Card et al's 1983 Keystroke Model; Reisner 1981, 1984).

Carroll & Olson (1988) define mental models as embodying "knowledge of the components of a system, their interconnections, and the processes that change the components; knowledge that forms the basis for users being able to construct

reasonable actions; and explanations about why a set of actions is appropriate". He also distinguishes between *descriptive* and *prescriptive* model representations. The researcher holds descriptive models about "what the user does know", and the designer holds a prescriptive model "of what the user should know".

The term "conceptual model" usually refers to the user's model of the system, but it can also denote the designer's model of how the system functions. In an ideal world, after surmounting the learning curve, the user's model would be the same as the designer's model, since he would have a complete understanding of the system and the concepts driving it.

All these (task) models share one problem - they assume an ideal, error-free user who does not make mistakes, or become confused, or forget his place in the interaction following an interruption.

The aim of human-computer interaction is to match computer systems and human cognition, as exemplified by Runciman & Hammond (1986), or as modelled by Clarke (1986), and Norman (1986). In the first case a simple programming paradigm is used to define the user's cognitive processes. In the latter 2 models, attention is drawn to the actual stages a human goes through in converting human goals into operations on the machine's virtual objects, by applying physical and perceptual processes to the interface.

2.1.4 Psychological Issues and Cognitive Engineering

Card, Moran & Newell's (1983) work has contributed greatly to this field by increasing the awareness of subconscious cognitive processes, which is being extended by researchers such as Norman & Draper (1986), Gardiner & Christie (1987), Suchman (1987) and being applied in the field by human factor specialists such as Dillon and Sweeney (1987).

The psychological issues which affect HCI are being investigated and some are described below, and in the following sections. For example, regarding problem structuring, Dillon & Sweeney (1987) observed that: "the designer's thought processes seemed to chunk themselves primarily about consecutive and often discrete action points. That is, designers displayed a tendency to structure the problem into perceived manageable units and tackle these independently in a series of action cycles. These cycles were invariably initiated and preceded by a phase of decision making where, for example, the designer considered options and task parameters."

This can be explained in terms of Clarke's (1986) model, where achieving a goal involves forming, executing and evaluating an action sequence. This requires the transformation of a goal into an intention, which is mapped into a physical action sequence that is executed and then evaluated, to see if it produced the desired

outcome. If not, then another intention or different action sequence must be invoked and the process repeated until the desired outcome is reached.

Norman's (1986) model explicitly illustrates the gulf of execution and evaluation, and the psychological or physical activities needed to bridge them. If one of the steps is missing then that bridge/gulf cannot be crossed. In the article on direct manipulation, Hutchins, Hollan & Norman (1986) describe 2 aspects of direct manipulation tools - distance and engagement. "Distance relates the distance between one's thoughts and the physical requirements of the system under use." Put in more general terms, distance measures the fit of the physical device in relation to the psychological variables, such as how easy it is to achieve the task goals. Thus a good task fit implies a small distance, and a bad task fit implies a large distance, or gap. Engagement refers to "the feeling that one is directly manipulating the object of interest", or by Laurel's (1986) wider view of the system, as the level of rapport between the task and the interface.

One of the most difficult psychological issues to cope with is **user expectations**, since they vary from one individual to the next, by unknown quantities. However, according to Gaines & Shaw (1984), there is one expectation that is common among users of computer systems, namely - that the user will be able to do something useful with the system!

Expectations spring partly from what users interpret from the system image - even without using a system people have expectations about how it operates (Norman 1986, Gaines & Shaw 1984). Whether these expectations will prove to be valid or not is unanswerable until actual operation of the system. Some expectations come from using other systems and cross generalising (Scholtz & Wiedenbeck 1993; Card et al 1983; Shneiderman 1987; Suchman 1987). The remaining expectations spring from the user's model of the system. If his interpretation is wrong then the system will not operate as he expects in some (or all) aspects (Scholtz & Wiedenbeck 1993; Norman & Draper 1986; Gardiner & Christie 1987).

Context is a significant psychological aspect in the execution of any action. It depends on what a person knows about a particular task or activity, and how familiar and comfortable that person is in that specific situation (Matlin 1989, Suchman 1987, Civikly 1981). Changing the context around a task can change the human's response(s) to it. Suchman showed the importance of context and the way that different instructions lead to different responses when trying to operate a photocopier. She found that different instructions (or information supplied) affected the way that the conceptual model was built up. Civikly (1981) discussed conversations and the different types of knowledge that are used to establish a context, and the protocols that people use to enable communication to take place. Establishing a framework of communication requires a communication protocol, contextual (background)

knowledge, and a means of expressing and exchanging views. The same applies to human-computer interaction, as Suchman found out.

2.1.5 User Characteristics and Differences

Users vary in many different ways: in level of expertise, cognitive style and in personality factors. Heaton & Sinclair (1988) have multi-dimensional axes for user types, regarding computerized tasks, covering frequency of use, expertise (the degree of computer use), extent of task knowledge, training level, and knowledge of system functionality.

Most categories of user types regarding tools use the user's skill or familiarity with the tool as a base-line. Usually only 2 types of user are considered, novices and experts. However, Schneider's (1984) set of {parrot, novice, intermediate, expert, master} user classes, extends beyond the usual simplistic division of expert and novice, and eloquently describes the variation in approach, knowledge and comprehension of users at different levels. For example, a master is above an expert, because the master not only knows how the system works, and how to make it do what he wants using the appropriate tools (definition of expert), but can subvert the system using existing tools and strategies of his own, as well as being able to add other functions to the system to make it fulfil his needs. According to Petre (1990), expert users want tools that give them access to, and control of, the nuts and bolts of the system.

Rector, Newton & Marsden's (1985) view is that "Experts are fundamentally different from novices; not only do they know more, they know differently. They perceive their field in terms of rich inter-relationships, rather than isolated facts and can make use of far more information than can novices." This points out the difference in conceptual models as well as the more recognisable difference in experience. Also, "Exceptions and anomalies [in data or facts] appear to be particularly significant both as landmarks to experts and as difficulties for novices."

Shneiderman (1980) states that, "The pressure for closure means that users, especially novices may prefer multiple small operations to a single large operation. Not only can they monitor progress and ensure that all is going well but they can release the details of coping with the early portions of the task from short term memory."

Draper (1984) contradicts most novice-expert assumptions; he says that all users are specialists in some areas and novices in others - dependent on command and context. This means that each user has a different skill level for each tool that is encountered. Skill level has an obvious effect on how well the user can carry out the task.

Rector et al also state that "Cognitive style affects what forms of presentation and assistance they [ie. experts (and others)] find most helpful". So, personality and cognitive style can also influence skill levels. Dillon & Sweeney (1987) suggest that there may be as many as 19 different cognitive styles (each of which represents a "sliding scale" dimension) which can be combined and utilised in a way that varies from moment to moment, depending on the activity that the person is carrying out at the time.

2.1.6 Visual Aspects of Cognition

The visual senses are very important in HCI, because it is through the eyes that perception of the system state takes place, which in turn is interpreted and evaluated in terms of the user's goals (Norman 1986, Clarke 1986).

Marr (1982) and Treisman (1982, 1986) worked on the mechanisms of visual processing. Marr in terms of using the best representation to get a message across; and Treisman in terms of perceptual grouping processes (see §2.1.2).

Rogers (1986), Lodding (1983), Kindborg & Kollerbauer (1987) have all worked on different aspects of the use of icons, symbols and signs in visual representation. They conclude that the relationship between representation and form is critical, and that the underlying cognitive models and knowledge that they are associated with, must match in order to get the message across. Hence, the representation used and the way that it is interpreted and received by the viewer is of paramount importance. Using the wrong representation could result in the wrong message being received, causing the user to draw the wrong conclusions, and confusing him, or worse causing an incorrect modification to the conceptual model (Rogers 1986, Marr 1982, Lodding 1983, Kindborg & Kollerbauer 1987).

This explains why prototyping an interface with assessment by its prospective users is an important means of design. It allows bad interfaces, the ones with the wrong representation or means of executing a task to be thrown out. Perhaps more importantly, it allows the designers to discover why a particular representation or interface solution failed. Thus providing additional detail for the conceptual design, and driving it towards a more suitable design.

2.1.7 Applying Typography to VDUs

The human information processing side has been studied in 2 main ways. By Card, Moran & Newell (1983) who studied the psychological aspects of human-computer interaction. Whereas Treisman (1982), and Thompson (1985) dealt with visual perception and how human information processing affected perceptual tasks such as grouping and the focussing of attention in the search task.

Much work has gone into the study of how best to apply typography (and colour) to the VDU screen itself, in terms of legibility and readability, and the formation of guidelines for optimum usage of typography, spacing and information grouping. Such guidelines originate from Cakir, Hart & Stewart's (1980) detailed analysis of VDU characteristics and the use of colour and typographic effects on VDUs; which Hulme (1985), van Nes (1984 & 1986), Wright & Lickorish (1988), and van Laar (1989) have extended and applied to a variety of tasks on the VDU. For example, van Laar's programming tool used colour coding to reinforce the perceptual cue of indentation. Baecker & Marcus (1990) also considered the beneficial application of typography to making program text more readable, and outline specific ways of achieving this. One of their aims was to make program text "more useful in terms of such common programming tasks as scanning, navigating, manipulating, posing hypotheses and answering questions, debugging and maintaining code."

The main focus of human factors is to present the viewer with information that is easy to read and digest, in order for the viewer to proceed with the task smoothly, without needing to re-interpret the information or re-organize the current work plan. In short, giving guidelines that facilitate human information processing, and optimize visual processing by exemplifying good practices. These may include, for example: grouping information in an order that is comfortable for the reader, and appropriate to the task; and explaining those visual information processing and perceptual factors, like using more than 4 colours per screen, that increase processing.

Yourdon & Constantine (1979) define 3 factors that affect statement complexity:

- the amount of information that must be understood correctly;
- the accessibility of the information; and
- the structure of the information.

Furthermore, Green, Sime & Fitter (1981) emphasized the importance of applying typographic aids to programming notation by producing a set of 6 principles; elaborated upon by Winfield (1986) and applied more closely to the programming task itself. This idea of applying selective typographic aids in text processing/editing are directly supported by 4 of those 6 principles, reproduced below. Indeed, they demonstrate a strong case for applying spotlighting to the trail following task. The original numbering of the quotes has been retained, as in Winfield's book, but they

have been re-emphasized (using underlining) in tune with the important issues that are addressed in this thesis; as shown in the following (abbreviated) quotes:

- "1. 'Perceptual cues beat symbolic cues; symbolic cues beat no cues at all.' This means that if you want to transmit information, think initially about typographic cues or diagrams."
- "2. 'Short trails are better than long trails.' If meaning has to be deduced from text, think of how many hurdles the reader has to cross before the meaning is clear. The fewer the hurdles the better."
- "3. 'Following a trail should demand few mental operations.' Common mental operations in program comprehension include keeping track of changing names as data is transformed. Another is 'shopping' or accumulating a list of conditions which govern some aspect of a program's behaviour. Often a compromise has to be reached between a short but complex trail and a long but simple trail."
- "4. 'Only one mental finger should be necessary.' A mental finger is when we check off one thing against another. If there are strong perceptual aids such as flow charts, we do not need to use mental fingers to remember where we are ."

These principles express the ideas that define the spotlighting and summary table tools.

Combining the main themes of principles 3, 1, and 4, gives: trail following using typographic cues as location aids; defining spotlighting's main use.

Principle 2 relates to summary tables, where the user is presented with declaration data in the required, more consumable form; so that he does not have to waste time and effort (with the possibility of making a mistake or drawing the wrong conclusion), by processing the information himself. Giving the short route to accurate information, and avoiding information processing hurdles.

Chapters 5 and 6 discuss these tools, and their design in detail.

2.2 Software Development & Programming

The previous section covered the human factors aspects of the user-task-tool situation, this section will cover all aspects of the programming task relevant to the solo programmer, including tools and aspects of program text.

2.2.1 Programming Stages & Processes

Large systems require a more elaborate software design process than that considered for programming in the small, due to the number of people involved, and the explosion of documentation that must be considered and adhered to.

Schindler's (1982, p10) "waterfall diagram" of the Software Life Cycle has 7 stages:

System requirements, software requirements, architectural design, algorithm design, coding & debugging, testing & validation, operation & maintenance.

Schindler has explicit evaluation loops running backwards from each stage to the previous one. These loops are used to check for consistency within and across the design.

For small systems, especially those involving solo programmers, the programming process is more compact and flexible, and has less chance of serious inconsistencies. If they do occur, the scale of the problem is small, and can usually be solved fairly quickly, without too much hassle. Principally because there are no other team members to consult regarding keeping the software compatible. A solo programmer who can maintain consistency in his internal representation should be able to produce code that is consistent and compatible, since it derives from a single source, his own internal representation.

Gould (1975) defines the principal stages of the programming process as: formulate problem, generate plan, code, debug, and verify. Whereas Redmond & Gasen (1989) define the programming process to consist of the following sequential stages: problem specification, program specification, coding, and testing. However, there is an evaluation loop running back from each stage to any of the previous stages, which represents an evaluation point in the thought processes of the programmer.

Most programming models do not seem to pay much attention to these evaluation loops, although they are essential for verifying design. Schindler, and Redmond & Gasen are among the few who make them explicit. In contrast, Pennington & Grabowski (1990) consider the basic programming tasks to be understanding the problem, design, coding and maintenance (revision). All in all the programming process is the same although the naming of the stages has changed. However, Pennington & Grabowski note that, "programmers rarely complete one subtask before beginning the next", indeed, they repeatedly alternate among these subtasks.

2.2.2 Programming Knowledge

According to Soloway, Adelson & Ehrlich (1988), expert programmers have at least 2 types of knowledge that novices do not: "programming plans" and "rules of programming discourse" (rules specifying programming conventions for composing programs).

Lewis & Olson (1987) agree, saying that "Programming skill consists of a knowledge of the primitives and the rules of combination, together with a repertoire of higher-level plans for commonly used assemblies. Flexibility results from the fact that structures for which the higher-level plans are inappropriate, or must be modified, can be dealt with by descending to the level of the primitives and synthesizing novel forms."

Gilmore (1990) disagrees with the previous 2 views, stating that it is the novice's combination of fragile knowledge (Perkins & Martin 1986) and the lack of strategic knowledge, as well as knowing how and when to apply it, that really distinguishes novices from experts.

Fisher (1987) relates the interaction between task environment, programming process(es) and LTM (long term memory) and their components concisely. She describes how design errors creep in due to differences in knowledge or viewpoint between the customer and designer (mainly) regarding assumptions, the problem statement, and the perceived purpose of the software. Usually a requirements or specification document is used to define the system requirements or the problem statement. Guindon (1990) observed that the requirements presented to the software designer or programmer are usually incomplete or imprecise. The designer usually supplies the missing information either by inference or from personal knowledge of the application. This is confirmed by Visser (1987), and Visser & Hoc (1990)

2.2.3 Programming Plans & Composition

The programmer's mental model (viz Adelson & Soloway's Sketchy Model 1988) encapsulates the design objectives. An outline of coding in the programming plan scenario was defined by Brooks (1977), and elaborated by Shneiderman (1980), and Rist (1986). Each programming plan (or method, as Brooks calls them) is a sequence of code that achieves a task. For example, a "totalling" programming plan would read in a series of numbers and write out their total.

What happens internally, during plan composition, is that a generic programming plan from LTM (long term memory) is copied into WM (working memory), where appropriate details are added, and it is then output to external memory (added to existing program text). As a result of limited WM capacity Green, Bellamy & Parker (1987) observe that "Rather than build up the whole program internally and then output

it to the editor from start to finish, they [programmers] output fragments as they are completed, or incomplete fragments if working memory becomes overloaded." Their experimental data concerning their parsing-gnirap model supports the existence of plans when coding in Pascal (but not Basic).

Current opinions on programming plan structure are divided. Détienne & Soloway (1990), and Robertson & Yu (1990), consider that the plans have a flat structure consisting of a sequence of steps that are concatenated to achieve a goal. Whereas Rist (1990) assumes a more complex plan structure with 3 distinct mechanisms: concatenation, interleaving and hierarchy. Rist's approach seems more reasonable since it reflects the structure of the program text (and its meaning) with respect to hierarchies and nested procedures, as well as plans that are concatenated and thus independent, or are linked through function by being interleaved. For example, a totalling plan can stand on its own, or it can be interleaved with an averaging plan.

Wiedenbeck (1986a, 1986b) has shown that each plan contains at least one focal line, or beacon, that defines the plan. Green, Bellamy & Parker (1987) observe that for plans that have a major and minor beacon, the programmer is more likely to forget to include the minor beacon than the major one; since the major beacon characterizes the plan, and the minor one just sets the scene.

For example, in the sort plan, 2 values are swapped, but a temporary variable is needed to hold one of the values, prior to swapping. The programmer is more likely to forget to declare the temporary variable, than he is to miss out the series of statements that exchange the variable values. The same goes for the running total plan, where the major focal line is of the type "total := total + some_number;" and the minor beacon is the initialisation statement where "total := 0;".

Lewis & Olson (1987) note that combining plans is error prone, due to the required interleaving of plan elements.

Soloway, Bonar & Ehrlich (1983) describe 3 types of plan knowledge: strategic, tactical, and implementation knowledge. Strategic plans are language independent plans dealing with global strategies. Tactical plans are more detailed than strategic plans, but are still language independent; whereas implementation plans define how to turn a tactical plan into code for a specific programming language.

Brooks (1977) estimated that an experienced programmer must have tens or hundreds of thousands of methods (programming plans) in his head. Acquiring this amount of methods/plans takes a long time, due to the need for the programmer to be exposed to a sufficient variety of different problems, to increase the internal plan library. The faster this type of knowledge can be acquired, the faster a "novice" programmer becomes an "expert".

Davies' (1993) experimental data shows that expert programmers make more use of display-based skills and external memory sources than novices. When movement around an editing screen is restricted, experts' externalize more information to compensate for the restriction, and to reduce the corresponding memory load.

2.2.4 Comprehension

Boehm-Davis (1988) states that "software comprehension involves reconstructing the logic, structure and goals that were used to write a computer program". Shneiderman & Mayer (1979) proposed that this reconstruction process was accomplished through a bottom-up, hierarchical chunking process based on both syntactic and semantic knowledge; whereas Brooks (1983) argued that the process is driven from the top rather than from the bottom. It seems more likely that comprehension uses both processes, and is not limited to one or the other.

According to Baecker & Marcus (1990), program reading and understanding includes reading, skimming, searching, retrieving, memorizing, remembering, simulating, guessing, answering and problem solving. Each of these subtasks contributes to forming and maintaining a correct mental representation of the program.

Wiedenbeck (1986a) asserts that "using the beacon to recognize the program [or algorithm] is something like skimming a text. It gives a general, high level understanding of the program but is not sufficient for debugging or modification, which require deeper understanding." Indeed, Soloway, Adelson & Ehrlich (1988) used this feature to mislead their subjects into giving plan-like corrections for unplan-like code. For example, the variable name "max" was used to find the maximum value in a set of numbers in the plan-like code; but it actually returned the minimum value in the unplan-like code.

Pennington (1987) defines 5 types of program information available to programmers on reading program text: operations, data flow, control flow, state and functions. These are the basic ingredients that form the comprehension model. Van Dijk and Kintsch (1983) suggest that there are 2 distinct but cross referenced representations of text that are constructed during discourse comprehension.

Pennington's experimental evidence shows that there are also 2 types of model that programmers can form, in order to comprehend program text. One for the domain (the external "real life" setting where the system operates), and one for the application (the functioning of the program text in terms of program information). Pennington found that full program comprehension is only possible if a programmer holds both models and is able to cross reference them. In contrast, a programmer who only holds one model is unable to fully comprehend the program text, and is thus

unable to cross reference between them. Bergantz & Hassell (1991) found that Prolog programmers also use program (application) and domain models.

Littman, Pinto, Letovsky & Soloway (1986) investigated the program reading strategies of experienced programmers who studied 250 line Fortran database programs. They found that the "systematic [reading] strategy gives a strong mental model of static and causal knowledge," and the "as-needed strategy gives a weak mental model since it only contains static knowledge". This relates to Pennington's work, where the weak mental model of the static knowledge refers to the programmer's application model which is not cross referenced to the domain model. Another factor is that the as-needed strategy collects pieces of information but does not connect them to gain a fuller, higher-level (cross-referenced) model.

Letovsky's (1986) and Boehm-Davis' (1988) models of comprehension are similar, with Letovsky using inquiries and Boehm-Davis using hypotheses. Both models have a knowledge base, a mental model, and a process for building the mental model. Letovsky's model is built up by assimilating inquiries. An inquiry consists of 4 stages: question, conjecture, attempt to find an answer, and draw a conclusion. Boehm—Davis' synthesis process coordinates the generation of hypotheses and tests them based on an interaction of information from the knowledge base and from the current understanding of the program. In either case, information gathering proceeds by inquiring or hypothesizing, and depends on the comprehension strategy used and the programmer's motivation.

Robertson, Davis, Okabe & Fitz-Randolph (1990) analysed programmer's verbal comments during the reading of program text - viewed one line at a time. These comments defined 6 groups of programmer motivation: analyse (explain), assume (predict), question (query), answer (linked to question), function (of code), and strategy (of programmer - what next, or where to, or what information is needed).

Program comprehension is an essential programming task since it is a subtask of debugging, modification and learning (Shneiderman 1980).

2.2.5 Debugging

Wertz (1982) draws attention to the distinction between conceptual & teleological bugs. "Conceptual bugs manifest as a discrepancy between actual program behaviour and required program behaviour (as per specification); and teleological bugs are a discrepancy between actual program behaviour and program behaviour as intended by the program's author *independent* of required behaviour." Waddington & Henry (1990, p966) illustrate the relationship between teleological bugs & conceptual bugs, and their possible effect on expert debugging strategies. This diagram captures the variation in the nature of high-level errors very neatly.

Carver & Klahr (1986) distinguish 4 phases in the debugging process: program evaluation, bug identification, bug location, and bug correction. The production in their model draws on 4 sources of information: the correct solution, the program output, the code, and knowledge of the programming language.

Kessler & Anderson (1986) consider debugging to consist of several subskills. Including the ability to evaluate code correctly, to be able to locate errors by parsing the code and, matching it with the results obtained, and the ability to generate correct code to fix the bug. Debugging can occur in response to compilation errors or run-time errors, but it can also occur during coding, as an off-shoot of the "gnisrap" activity (Green, Bellamy & Parker 1987). Gray & Anderson (1987) describe the latter as a "change-episode"; where the programmer fixes the code as soon as a discrepancy is noticed, and verified as needing to be corrected.

The debugging activities/strategies activated by the programmer depend on his repertoire of debugging skills, the nature of the bug and/or the evidence that suggests that there is a bug, and the tools available to him. There are various debugging strategies: Shneiderman (1980) states that, "*Forward comprehension* is the ability to discover the output for a given input; and *backward comprehension* is the ability to discover the input necessary to produce the given output."

Rasmussen (1981) describes 2 major trouble shooting (debugging) strategies. In topographic search, a programmer uses clues in the output or tests of internal program states to narrow the possible location of a bug to a small part of the program. In symptomatic search, the programmer uses prior debugging knowledge and recalls a bug that has previously caused symptoms like the current ones."

Nanja & Cook (1987) also observed 2 debugging approaches. The comprehension approach is used to get a total understanding of the program and so place the error in context; whereas the isolation approach attempts to identify candidate bug location(s) by searching the output for clues, recalling similar bugs, and testing the program state.

Nanja & Cook's isolation approach covers both of Rasmussen's debugging strategies; and their debugging approaches also correlate to Littman, Pinto, Letovsky & Soloway's systematic and as-needed comprehension strategies respectively. Nanja & Cook also noticed that experts corrected multiple errors before verifying their correctness, while novices corrected and verified single errors.

Lukey's (1980) tentative debugging is based on debugging clues; whereas his other type of debugging makes use of all the different types and levels of descriptions produced during program understanding - another systematic or comprehension approach. The systematic or comprehension approach could be called a holistic approach - seeing how the code correlates to the task description and fixing any discrepancies either in design approach (specification into program plans) stage, or in

design translation (program plans into code) stage (such as missing declarations etc.). Using a top-down systematic hierarchical search - looking at the main program, then the sub-procedures in the order of calling.

Kessler & Anderson (1986) note that "debugging is not a taught skill - programmers have to learn for themselves. ... It became clear that debugging is a skill that does not immediately follow from the ability to write code". In the light of these comments it would seem prudent to ask why debugging is not taught as well as programming. It would certainly speed up debugging, and might make programming errors less frequent, and could even increase the overall productivity of programmers - especially since debugging takes up a fairly large proportion of the programming task.

As Comer (1981) states: "Error correction is more difficult and less important than error detection". The problem with coding/debugging is trying to differentiate between what is there in the program text, and what should be there, and checking that every item is in its correct position. It is mainly a question of using the right highlighting method to show the problem up - to make it visible to the naked eye. To differentiate the chameleon from its surroundings.

2.2.6 Errors

Offering tools that help the programmer to tidy up, and detect trivial errors using pre-compilation checking and tools that actively assist in preventing errors, or showing them up as soon as possible will surely increase programmer productivity and reduce debugging time and costs. Especially since cost per error escalates as time proceeds (Yourdon & Constantine, 1979). Brooks (1977) asserts that "errors or difficulties in programming are not random occurrences, produced by random occurrences in the programmer's environment. Instead, they are clearly linked to specific features or properties of programming languages."

There are 2 broad categories of errors: syntax and semantic errors. Syntax errors are most frequently made and corrected during the early debugging stages (using the compiler and/or syntax checker diagnostics). However, semantic errors are much more troublesome and time consuming to correct, due partly to their ephemeral effects, and partly to the fact that the programmer interprets what he thinks he sees in the code, rather than what he actually sees. This may be due to the programmer identifying a given section of code with a particular programming plan. Consequently, instead of actually reading the code, he assumes that it reflects (ie. executes) the plan as specified in his head, and thus ticks it off as Okay without reading it through thoroughly and finding it to be different from the intended plan. This is what Eisenstadt (1993) refers to as a WYSIPIG (What You See Is Probably Illusory

Guv'nor) error, where the programmer hallucinates keywords (or other program elements) within the code, that are actually non-existent.

On analysing the frequency of bugs in novice programs, Spohrer & Soloway (1986) made 2 significant findings. Namely, that:

- just a few bugs are made by lots of students learning to program; and
- most bugs do NOT arise because students have some misconceptions about some language constructs.

Youngs (1974) study of error rates in programming showed that experienced programmers initially make about the same number of errors as beginning programmers, but that they are able to find their errors faster. He classifies programmer errors into 4 categories: "syntax", "semantic", "logic", and "clerical".

Pattis's (1981) error categories are in terms of lexical, syntactic, intent, and beyond the horizon situations. Where the latter refers to error events occurring outside the range of expected behaviour, due to unexpected combinations of data, events or other factors.

Another possible means of categorizing errors is the reason for their occurrence from the programmer's viewpoint. Some of these errors could be avoided if the programmer was given the appropriate information instead of letting him guess incorrectly. It is usually more prudent (and less costly time-wise) to check and not make the error in the first place than it is to guess, and have to fix it later, when debugging costs rise geometrically.

2.2.7 Programmers at Work

Curtis (1988) defines several factors affecting individual programmer performance: intellectual aptitude, knowledge base, cognitive style, motivational structures, personality characteristics, and behavioural characteristics.

Working with other programmers and software designers is endlessly fascinating - not only because each person's experience differs, but also because of different working/designing methods and practices. Susan Lammers (1986) interviews of so-called "super-programmers" substantiates the anecdotally well-known "poles" of the software design continuum. On the one hand there are the "proper" programmers who put everything down on paper first, and then transfer it to the development system when it is "complete" on paper; and then there are the other programmers, the so-called "hackers" who design code more or less on-line, at the VDU screen itself. Green (1990a) refers to these programmer types as the "neats" and the "scruffies".

Molzberger's (1983) article describes a specific group of individual hackers as "trance programmers", since they appear to reach another level of consciousness, as they sit

in front of the VDU tapping away. Molzberger describes 2 types of reactions from his "trance programmers", in response to interruptions during these periods of intense concentration. One reaction is a temporary suspension of thoughts and activities while attending to the interrupt. Then returning to the previous activity and picking up where he left off with no ill effects. The other reaction is that any interrupt, no matter how short, causes the chain of thought up till then to be lost completely. So that the programmer has to go back to square 1, and start all over again. It seems obvious that this difference is due, at least in part, to the individual's inherent ability or inability to sustain a chain of thought using short term or working memory.

However, going back to the theme of variations in software design - most software design falls into a region between the 2 poles, where the programmer works out the central algorithms and data structures of the problem on paper (the software skeleton). This is then transferred to the computer and modified until it meets the requirements. Perhaps the difference between the software poles is merely a difference in interaction medium - paper and pen versus screen editing; but it may go deeper than that, and illustrate different types of programmer or programming ability.

Brooks (1977), Green (1990b), Rist (1990), and Visser (1990 - in the engineering field) support the opportunistic method of design - filling in details as the design problem and/or its domain of application becomes better defined and understood.

The complexity of the required software tends to influence the program development method chosen; the more complex the software, the more prevalent the paper designs; the simpler the software, the more prevalent on-line designing. For example, most experienced programmers wouldn't bother to write down the design for an averaging program, they would develop it directly on the VDU screen.

Rist (1990) describes 2 approaches to design like Ratcliff & Siddiqi (1985), where

- programs are designed forward from input and output and expanded if the designer can retrieve a known solution to the problem; (confirmed by D tienne 1991b) or
- designing backwards from the goals of the problem to create a solution - thus bottom up design, from an initial sketchy solution or focal idea that was expanded to define a complete solution.

According to Shneiderman (1980 p50), top-down design generates the most general levels first, followed by more detailed ones/analysis. It is also called "working backwards" or "reformulating the goal", (from general to specifics). Bottom-up software design generates low level code first, in an attempt to build up to the goal. This is also called "working forwards" or "reformulating the givens", (from specifics to the general), where the "givens" include the permissible statements of the language.

Rist (1990) found that programmers use either one or both of these design strategies, depending on task difficulty. Green, Bellamy & Parker (1987) support this opportunistic design method - where fragments of code may be generated in any order (in either whole or partial fragments), jumping from one plan to insert additional code for a second, interleaved, plan. The parsing part of the parsing-gnisrap cycle is used to remind the programmer of what was done and how they were doing it. One of the factors determining the frequency of memory refreshing activities is due to the role-expressiveness of the programming language.

As early as 1977, Brooks stated that complete top-down design depends on the programmer being very familiar with both problem and programming language, since top-down design is inappropriate for unfamiliar problems or languages. Thus an opportunistic approach to design will be inevitable whenever a designer comes to a knowledge or understanding gap, which he is unable to fill (Visser & Hoc 1990).

2.2.8 Programming Tools, User Aids & Task Assistants

The main aim of programming tools, user aids, and task assistants is to alleviate the (memory and work) load on the user, and to ensure that tasks are accomplished as efficiently as possible. There are various types, as below.

Teitelbaum and Reps (1981), Teitelman (1972), Waters (1982), Bourguignon (1984), and Goldenson & Wang (1991) took the same approach, in that they used programming cliches, such as the "while-do" or "repeat-until" loops, and presented them to the user to fill in. This of course is based on programming plans, which were originated by Brooks (1977). Teitelman and Waters also used knowledge based tools to actively assist them in the program design task.

Weiser & Lyle's (1986) slicing tool was meant to aid backward comprehension by printing out only those program statements relative to a chosen statement. In a previous experiment Weiser (1982a & 1982b) showed that programmers mentally construct slices when debugging. However, having a slicing tool available did not seem to benefit the users. One possible reason was that users preferred doing slicing mentally, and having the tool do it for them did not give the same effect.

Jerrams-Smith's (1985) SUSI (Smart User System Interface) is an intelligent interface to Unix, which corrects user's misconceptions about the system. The interface provides intelligent responses suited to the [individual] user, and modularity for ease of modification. SUSI incorporates an IKBS, (intelligent knowledge base system), and provides user modelling. It interprets user's actions and offers a personalised response which guides users to the easiest and most efficient method of carrying out their intention. SUSI trains the user by responding with computer aided

learning material when appropriate. The knowledge base is in the form of production rules, which are used to interpret the user's intention.

Young and Harris (1986) produced a viewdata structure editing tool to assist in the modification of viewdata frames; and to remind the user of interrupted tasks and other goals and plans which are pending. It also regroups actions "to be done" to alleviate memory load on the user and to gather all actions on one frame together, rather than having them scattered. Reitman Olson, Whitten & Gruenenfelder (1984) have produced a similar sort of system which deals with the editing of tree structures.

Card and Henderson's (1987) virtual-workspace interface facilitates task switching under a windowing environment, by the use of "rooms" (task windows). These "rooms" are allocated their own set of "engaged tools" - tools that have been invoked and are available for immediate use. Thus reducing the time and effort overheads associated with switching from one task to another.

O'Malley and Sharples (1986) have a system that enables the user to view and alter the organisational structure of text at different levels in an authoring environment. However, its main use is to keep track of all the various constraints under which the text is to be written, such as technical or layman's narrative style, pagination and format style, as well as reminding the author about connections in the narrative.

Monk's Personal Browser (1989) helps users move around a Hypertext network, by providing colour coded "footprint information". The user has a map of the network on-screen, which shows nodes already visited in one colour, and unvisited nodes in another. Monk distinguishes between rambling and orienteering. The latter is goal-directed movement geared towards a specific node, while the former has no particular destination in mind. He further suggests that the author of the Hypertext sets up a guided tour, defining which nodes to visit and in what order - giving his user tourists a uniform view of the data held by those nodes.

Catrambone & Carroll (1987), and Hewett (1988) use similar concepts. In the former, the functions of a word processing system are restricted to a manageable number using the Training Wheels systems, and in the latter, learning is achieved by guided exploration. In each case the user is restricted to a limited subset of system functions. This restriction is meant to let the user learn in a small, safe environment; so that confidence in using the system can grow smoothly, and without check, because destructive functions are inaccessible.

2.2.9 Aspects of Tools

The better the tool fits the task the more "invisible" its essential characteristics become. The very smoothness of tool use eliminates recognition of the combination of working components and the accompanying skill elements. The success of a tool is directly related to the immediacy which can be established between the task at hand and the tool's suitability for that task in relation to the goal at hand.

This is what Hutchins, Hollan & Norman (1986, p100) call semantic directness. Semantic directness defines "the relationship between the task one wishes to accomplish, and the ways the interface provides for accomplishing it." Engagement refers to "the feeling that one is directly manipulating the object of interest", or by Laurel's (1986) wider view of the system, as the level of rapport between the task and the (tool) interface.

It is well known that as a person develops a skill, and gradually moves from novice towards expert status, that that skill becomes increasingly automated, and the component stages blend together "seamlessly". As this happens the person becomes less able to talk about what is happening at each stage. This is because the skill is operating from a higher level, and is regarded as a "whole unit" (or cognitive process) instead of a series of component stages (Ormerod 1990, see §2.1.4).

Wastell (1990, p108) states that, "Much of information processing is non-conscious. Moreover, an inverse relationship between awareness and behaviour often prevails (eg. skilled performance) which severely limits subjective techniques. Behaviour as we have seen, is intrinsically ambiguous." He defines mental effort as the amount of *controlled processing* required by a task." This is a very important factor in the use of a tool, computerized or not.

Similarly, Kieras & Polson (1985) observe that from the user's point of view, the complexity of a device depends on the amount, content, and structure of knowledge required to operate that device successfully. Furthermore, that "for a new user, complexity is also determined by the difficulty in acquiring the new knowledge necessary for this purpose".

2.2.10 Aspects of Program Text

Leventhal (1988), like Molzberger (1983), links the aesthetic appearance of program text with its comprehensibility. Leventhal's view is that a well-presented text is easier to comprehend. Molzberger goes further, by saying that a particular combination of "good" aesthetic qualities go hand-in-hand with good quality, reliable programs. Although proving that such a program is correct formally is nigh impossible, such programs are not liable to breakdown during operation. The latter may be attributed to the code being produced under "trance" conditions, where code is

produced in a continuous stream, from start to finish, in one sitting. Thus there is perhaps less chance of confusion or conflicting requirements being worked in. A program having this type of "good" layout/presentation is almost guaranteed to be error free for most practical purposes. Since "good" aesthetic quality implies a "good", well-balanced program, and a "bad" aesthetic quality implies a "bad", unbalanced or ill-defined program, that is prone to many errors and breakdowns.

Molzberger's subjects recognize this "meta-style" of goodness or beauty, and inherently understand that a program that embodies it, is very efficient in operation and not susceptible to run-of-the-mill errors.

Gray & Anderson's (1987) change-episodes include stylistic changes to the code as well as code corrections. Riecken, Koenemann-Belliveau & Robertson (1991) found that the programmers in their experiment added vertical spacing between code chunks, as well as adding begin-end braces in several areas of the code on the first pass through the code (during comprehension). They preferred to adapt the visual structure of the program to their style of coding immediately, rather than leaving it until later. This is a very significant finding. Moreover, that indentation, comprehension, and the programmer's mental representation of the code are definitely linked, is illustrated by the following 3 quotes, and van Laar's (1989) experimental results and conclusions.

Mynatt (1990, p945) "Evidence that language-structure-based indenting aids comprehension (compared to no indenting or random indenting) would suggest that the hierarchy implied by the indenting corresponds in some way to the mental representation built by a programmer through comprehension."

Baecker & Marcus (1986 p51) "... the body of research they surveyed fails to provide clear experimental confirmation for what every programmer knows: a program's appearance drastically effects its comprehensibility and usability."

Holt, Boehm-Davis & Shultz (1987) found differences between the comprehension models built by professional and student programmers. "The mental models of the professionals were primarily affected by the difficulty of the program's assigned modifications, while the students were primarily affected by the structure and content of the programs."

Van Laar's programming tool used colour coding to reinforce the perceptual cue of indentation. Giving each level of indentation a different colour reinforced the program structure, and helped with program comprehension. In a series of experiments he found that this enhanced problem solving originating from questions of nestedness, and scope. He also found that if the indentation spacing is too large then the coherence of the code is lost because everything is seen in isolation and appears unconnected. Indentation is used as a comprehension aid, and provides mental/visible signposts within the code; it may be thought of as indicating program control signposts.

Budgen (1992) defines the dual nature of code as, "passive structure and dynamic behaviour, since software is represented by a program (with static qualities), which is then executed as a process (exhibiting dynamic behaviour)". This description catches the essential difference between the order of statements, and control flow - the order of their execution. This difference causes many problems for novice programmers.

Green's (1989) work on cognitive dimensions suggests that certain properties of notations (viz program text, and the tools applied to it) should be either avoided/minimized, or maximized. Hidden dependencies, viscosity, premature commitment, hard mental operations, diffuseness, and susceptibility to low level errors (viz discriminability and action slips), are all to be either avoided or minimized as far as possible. On the other hand, role-expressiveness, consistency, and adding (cognitively supportive or beneficial) perceptual cues to the structure, are encouraged since they make notations easier to comprehend and interact with. In this context, Van Laar's tool reinforced role-expressiveness (indentation) by additional perceptual cueing (colour coding), to support comprehension.

Furthermore, Green (1990a, p31) states that, "As programs get larger, programmers find it harder to locate the information they need". Obviously, what is needed are a means of locating the information quickly, or making it more accessible, flexible, or easier to manipulate or transform into the format required by the programmer.

Baecker & Marcus (1990) developed a visual compiler for the C language, called SEE, that achieves some of the latter goals, by applying graphic design principles to SEE's output. For example, SEE was used to reformat ordinary C program text so that comments "governing" chunks were extended to the lefthand side of the page, whilst leaving comments appearing within chunks as they were; and data declaration sections were aligned to specific tab stops to make the variable names and the data types easier to connect visually. Baecker & Marcus* also experimented with boldface and italicising, and found that both effects had to be used sparingly to be effective. Adding colour to differentiate between specific elements of the program text also improved clarity.

* They also included the various reading and searching aspects that underpin programming activities in the comprehension of program text :-

p12 "programming tasks: scanning, navigating, manipulating, posing hypotheses and answering questions, debugging, maintaining the code."

p261 "programming activities: sketching, writing, revising, documenting, familiarization, reading, reviewing, debugging."

p261 "program reading and understanding consists of a variety of tasks: reading, skimming, searching, returning [to the previous location], page turning, memorization, remembering, simulating, guessing, answering, problem solving."

Chapter 3 Preliminary Data Collection - Observation & Analysis of Student Programmer's Reactions to MacPascal's Tools & Environment

This chapter describes how first year Computer Science students approach the programming task, their problems, and how they respond to MacPascal's tools and programming environment. This particular group of students were the only ones who were studying (Mac)Pascal in depth. Thus, they were chosen for observation during programming practice tutorials; and later on, as questionnaire subjects regarding their attitudes towards MacPascal's tools and various aspects of programming.

I attended about 1½ terms of their programming lectures, so that I became familiar with both the students and the lecture material. I felt that this would make it easier for me to blend in with them, and that they would view me as an equal rather than as a "lecturer", when I attended their tutorials as an adviser. In this way I was able to follow what they were doing from my understanding of the lecture notes and the tutorial exercises they were expected to complete on the Apple Macintosh machines.

These exercises provided the student with program specifications in a mathematical type of notation based on VDM (Vienna Definition Method). Early lectures showed how to translate a specification in English into one in VDM. Tutorials gave practice in turning specifications in plain English or VDM into working MacPascal programs.

3.1 Students' Problems Observed During Programming Practice Tutorials

The tutorials were used to help the students, and to observe and define their problems as they tackled a range of programming tasks and different types of problems. The aim of the tutorials was to activate students' knowledge, and to build up actual experience of how to apply programming knowledge to different areas of application. So that they learnt how to tailor each programming solution to suit a specific task. Since it is the individual task requirements that determines the choice of variables and data types used to flesh out the appropriate programming plans/algorithms.

3.1.1 Summary of Students' Problems

Many types of programming problems came to light. However, only those that are directly related to programming, and turning a specification into a working program are listed. From designing on paper or screen, through code development and debugging using the facilities and tools provided by MacPascal, 20 problem variants were observed. They have been grouped into a more logical sequence, with 4 main problem headings - learning (due to novice status in a new skill), task-specific, general/common errors, and MacPascal-specific problems.

Learning (Novice) Problems:

- learning the differences between data types: viz integer and real, char and boolean; and the effect this has on what can or cannot be done with a variable of each type.

- learning how to manipulate variables of each type, and to select a data type suited to each variable required to perform the task.
- sub-optimal use of variables, data structures and control structures. In particular, knowing which combination of programming plans/algorithms, variables and data types is most appropriate to the problem at hand, and how to use them to best effect. For example, updating array values using either:
 - 2 arrays, one for the old values and one for the new values; or
 - just using one array holding both "current" and (previous) "old" values. With one simple variable (of the same data type as the array) to calculate and hold the new "current" value of the array element being updated. With the aim of cycling through the array elements one by one, recalculating the next new value (using the simple variable to hold the value), and writing it back into the array, until all array element values have been updated.
- retaining, or not eliminating redundant elements. There are 2 main causes:
 - using many variables to do the job of one;
 - having unused variables (and occasionally procedures and functions) when their use or functions have been superseded elsewhere.
- learning how to formulate assignment statements correctly, and the appropriate use and combination of operators and functions to get the required value.
- using round brackets (parentheses) to obtain the correct interpretation of complex mathematical formulae or boolean expressions.
- understanding the different types of statements, how they work and the effects they have: empty statements, assignment statements, procedure or function call statements, if-then and if-then-else statements, case statements, compound statements; for, while, and repeat-until loops. Knowing how to combine them successfully to achieve the desired effect(s).
- appropriate usage of iterative or recursive techniques. Knowing how to set up and use iterative and recursive algorithms. More importantly, knowing when a recursive technique is more appropriate than an iterative one, and vice versa.
- difficulty in appreciating and utilising the different effects achieved using procedures and functions. For example, procedures usually change the surrounding state - they cause some sort of action to be taken; whereas functions do not (usually) change the state - they just provide values (such as "trunc" or "abs") or report back as boolean flags (such as "odd").

Task-Specific Problems:

- difficulty in translating the lecturer's mathematical VDM program specification into code. Since the VDM specification has main operations followed by sub-operations. Whereas Pascal, being "declare before use", is the opposite. Also, not all things in the VDM specification define things to do - some are "status" type comments or define relationships between I/O (input/output) variables.
- setting up the required range of data structures, and using the variables correctly, to maximum effect, and eliminating redundant variables.
- initialisation and declaration of variables, viz appropriateness and correctness. Such as deciding whether a boolean variable should be initialised as true or false.
- ensuring correct sequencing of variable value or state changes (especially booleans) to achieve the desired effect (and avoid infinite or redundant loops).
- setting up and connecting the input/output infrastructure - making sure that the "correct" input/output variables are used to read in or write out the required data at appropriate junctures in the algorithm and the corresponding code.
- deciding how and/or where to partition subroutines. Sometimes this is a matter of style or personal preference, at other times it depends on the functionality of the subroutine itself. There is a general belief that procedures (or functions) should be 1 page long at most. However, splitting a complex algorithm arbitrarily to suit this limit, may cause bugs to be introduced unnecessarily. For example, leaving the top 3 lines of a nested loop and its conditions on the bottom of a page, with the rest of the loop body over-page, could lead to comprehension and memory strain. Whereas splitting code at chunk boundaries causes less comprehension problems all round (Riecken, Koenemann-Belliveau & Robertson 1991). Thus, (in my view) functionality of a module should take precedence over such an arbitrary limit.

General/Common Errors:

- syntax errors: usually missing or misplaced ';', 'begin', 'end', or mismatched '(', ')', '[', ']', '{', '}' symbols, or parameter list mismatches.
- semantic errors:
 - incorrect or non-initialisation of variables before or during use;
 - uncompleted variable name changes, such as changing 'i' to 'index', but not checking that all appropriate changes have been made;
 - using the wrong control strategy, like using a repeat loop instead of a while loop; or
 - not utilising the control structure features either to full effect, or to minimize resource usage. For example, initializing all the elements of an array to the same value is (usually) done most efficiently using a 'for' loop.

MacPascal-specific Problems:

- finding out how to set up and use data files, using MacPascal's non-standard input/output routines to gain access to these data files.
- failure to use shortcuts in operating the MacPascal environment. Such as invoking commands by control character or function keypresses, rather than by time-consuming "mousing" of windows and menus to select options.
- problems of screen reformatting after a bracketing error - sometimes the same editing operation has to be done several times before it is accepted. MacPascal can react badly when it detects a bracketing error, and it is difficult to correct a (complex) line when MacPascal keeps interrupting to tell you that there is a bracketing error there, when you are in the middle of trying to fix it. This may be due to an "eoln" (end of line or line feed) character being in the wrong place, in a complex expression that occupies 2 or more lines.
- coping with the effects of "invisible" control characters "hidden" on the MacPascal screen and finding out how to overcome and eliminate them. Usually such characters scramble the lines of screen text up, so that it is difficult to tell where the text really belongs, and where the rogue character might be. So you have to resort to deleting the original copy, line by line, in an effort to delete the spurious "invisible" control character. When the effect disappears, the missing text must be retyped - either from memory, notes, or the last printout. Such characters arise usually as a result of mis-typing, and unintended, simultaneous keypresses. This is an example of a problem that needs to be "visualized" in order to be resolved.

3.1.2 Discussion of Problems Observed

One problem not mentioned above is that some students did not appear to understand the difference between getting the program to run (no syntax errors) and getting it to do what is required (no semantic, logic or algorithmic errors). Of course, this is a consequence of their novice status as programmers; as understanding of programming methods and the language increase they will soon understand the difference.

Clearly, quite a lot of the above problems will disappear as the student programmer becomes more experienced, and more adept at program design and the efficient usage of data types and control structures. It was obvious that some students were much more experienced than others, and had fewer problems. They already knew how to use data structures and programming language to construct working programs.

Pascal is supposed to be a "standard", but each system variant has its own peculiarities. With MacPascal it is the use and setting up of input and output data files. MacPascal also uses an interpreter that stops at the first error it meets, so each individual error has to be fixed in turn. The code is then re-interpreted to find out whether it runs, or another error is found. When it runs, the code is free of

syntax errors, but may still contain semantic, logic, design or algorithmic errors which will have to be found and eliminated.

The problem of "invisible" control characters may only happen once in a blue moon, but when it does it is extremely difficult to get rid of. What is needed is a utility that makes invisible, non-printable screen/control characters visible, perhaps using reverse video blocks, or colour to counteract the invisibility problem, and so enable the rogue character to be seen and eliminated. Or better still, a more rigorous text input monitor, so that such disruptive characters can be filtered out, and not accepted as valid characters during text inputting.

A frequently heard remark was, "Wouldn't it be nice to have an editing tool that would enable you to go straight to a particular procedure so that you could modify it". Unfortunately this feature was not included in MacPascal at the time of the study, although LightSpeed Pascal and folding editors provide this option. Perhaps it should be more widely available, to satisfy this need.

3.2 Design & Analysis of MacPascal Questionnaire

The aim of the questionnaire was to find out how first year programmers regarded and responded to a typical programming environment, in this case MacPascal. As the students were relatively new to programming, and in the learning phase, they were expected to verbalize their views more openly than older students. Learning a new skill requires concentration of conscious thought and effort, like driving a car. But once the skill is learned, it submerges into subconscious processing, and becomes beyond the reach of verbal description, due to the automaticity effect (Ormerod 1991). This means that only part of the task is done consciously and under true user control, the rest is done subconsciously by automatic procedural processes. This is why most "experts" have difficulty explaining what to them are "routine tasks"; they can demonstrate the task, but not explain or verbalize it beyond the general outline.

MacPascal provides 17 separate programming tools/aids which can be called up via menus or in some cases by control codes (sequences of key presses) as follows :-

Cut: mark a section of text and remove it to the paste buffer;

Copy: mark a section of text and copy it to the paste buffer;

Paste: insert a copy of the paste buffer contents at the current cursor position;

Find: search for the next instance of the search string;

Replace: find the next instance of the search string, and exchange it for the new string (word or phrase) defined by the user;

Check: the syntax checker (stops and) reports the first syntax error it finds;

Reset: abandons execution of the current "running" program;

- Go: executes current program until next breakpoint or end of program is reached;
- Go-go: executes current program to end, ignoring all breakpoints;
- (single) Step: pointing hand - executes the "next" statement in the program;
- Step-step: pointing hand - executes the program statement by statement;
- Stops in: enables insertion of breakpoints within the program;
- Instant: enables "instant" execution of a chunk of Pascal code;
- Observe: shows value(s) of variables selected by the user;
- Clipboard: enables transference of a chunk of code from one program to another;
- Font control: enables the user to change the size/style of program text characters;
- Indent level control (under the preferences option):
 - enables the user to choose how many space characters to indent each level by.

There are 4 further features, activated by the MacPascal environment, that the user can respond to, but not control. Namely :-

- Layout style: how the text is laid out on the screen;
- Highlighting: by emboldening all reserved words;
- Bracketing errors: causes an erroneous statement (of 1 or more lines) to become highlighted using MacPascal's "outline" font; and
- Syntax error (bug) line: defines the nature of the (first) syntax error found during syntax checking.

3.2.1 Dimensions of MacPascal Tool Calibration & Rating Scale Descriptions

MacPascal's tools were tested on 5 dimensions: usefulness, frequency of use, ease of use, likeability and frequency of use of other (alternative) method or tool. Each dimension was associated with a 5 point rating scale, to gauge each individual's response to each dimension. The ratings ran from 1 to 5, with 1 representing the most negative, and 5 the most positive response, and 3 representing the average or middle of the road response, as below :-

Scale	1	2	3	4	5
Usefulness	useless	not very useful	Ok	useful	essential
Frequency	never	once or twice	sometimes	often	usually
Ease of Use	difficult	fairly difficult	Ok	fairly easy	easy
Likeability	disliked	mildly disliked	Ok	mildly liked	liked
Other Method	never	once or twice	sometimes	often	usually
Ease of					
Production	difficult	fairly difficult	Ok	fairly easy	easy

The ease of production rating scale was used to rate the ease of producing comment statements within program code (§3.2.6), for questions 28 & 29 in the questionnaire.

Students were asked to allocate a rating for each tool on each dimension, by marking a cross or tick in the appropriate rating scale "box". Comment spaces were also provided with instructions to make any comment that had a bearing on the tool/aid in question. A total of 66 students answered the questionnaire, and it took about 1 hour.

3.2.2 Analysis of Quantitative Data

Tool calibration data resulted in 5 sets of data for each tool. This showed how many students "voted" for each individual point on the rating scale for each tool, on each dimension; giving 5 sets of 5-point data for each tool. The data was then abstracted so that the results for usefulness, frequency of use, ease of use, likeability, and frequency of use of other (alternative) method or tool were completely separate. Calculating the mean rating scale value for all tools on each dimension made it easy to compare tools on and across each dimension.

The original data tables (see Appendix, Tables 3A - "Comparison of Student Numbers & Percentages For Each Tool") show student numbers and percentages for each rating scale value; as well as the rating scale mean, standard deviation and mean deviation values to illustrate the spread of the data across the rating scales for each tool on each dimension. The tables also show cumulative rating scale values in terms of both student numbers and percentages. Thus the tables provide a range of ways of viewing and interpreting the data. Appendix 3A also contains 2 sets of frequency data, one in tool order as per the other 4 dimensions, and one ordered according to decreasing mean rating scale value. With the most frequently used tool at the top, and the least frequently used tool at bottom. A reduced form of the latter table is used in §3.2.3.2 to illustrate the frequency of use of each tool.

3.2.3 Interpreting the Data With Regard to Students' Comments

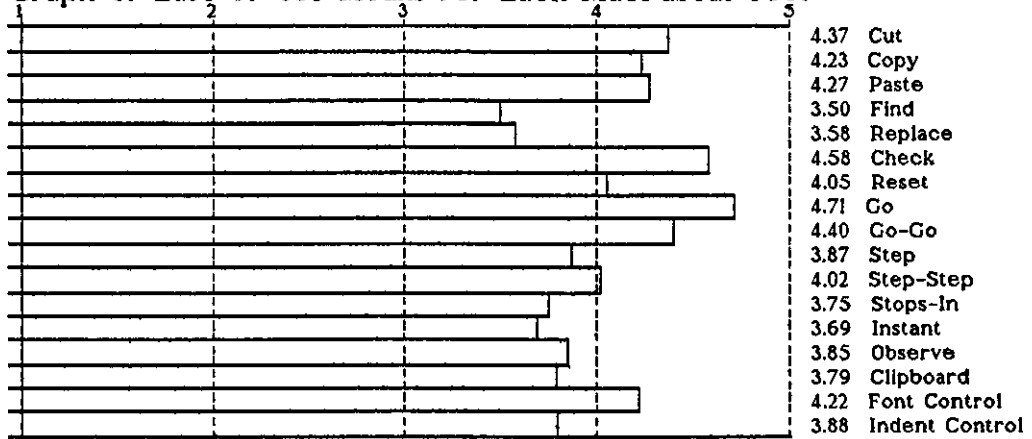
There are 3 main views: the graphs of the mean rating scale data, the frequency of use of each tool in terms of student votes, and summaries of the mean data. Student's comments helped put the different views into perspective.

3.2.3.1 Interpreting the Data Graphically & Numerically

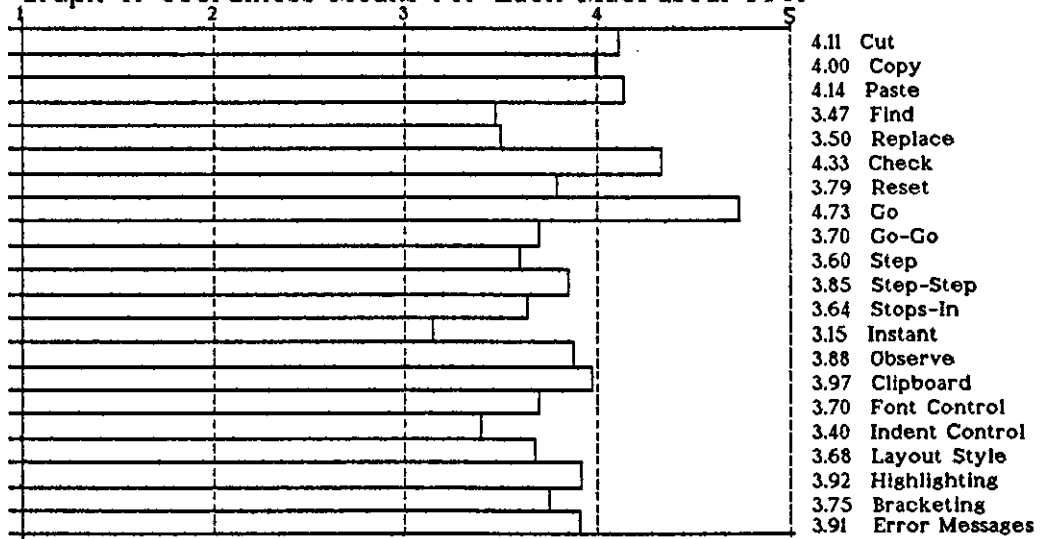
Graphing the mean rating scale value for each tool on each dimension made the results easy to visualise and compare. Bar graphs show the numeric mean value as well as the name of each tool, in tool order.

The Usefulness, Ease of Use, and Likeability graphs are remarkably similar, suggesting that they are interdependent. The mean rating scale value is larger than 3.0 for all tools on these 3 dimensions, reflecting the students positive attitudes. (The graphs appear on the next 2 pages with the Usefulness graph appearing on both pages for ease of comparison with the other dimensions. Text continues 3 pages on.)

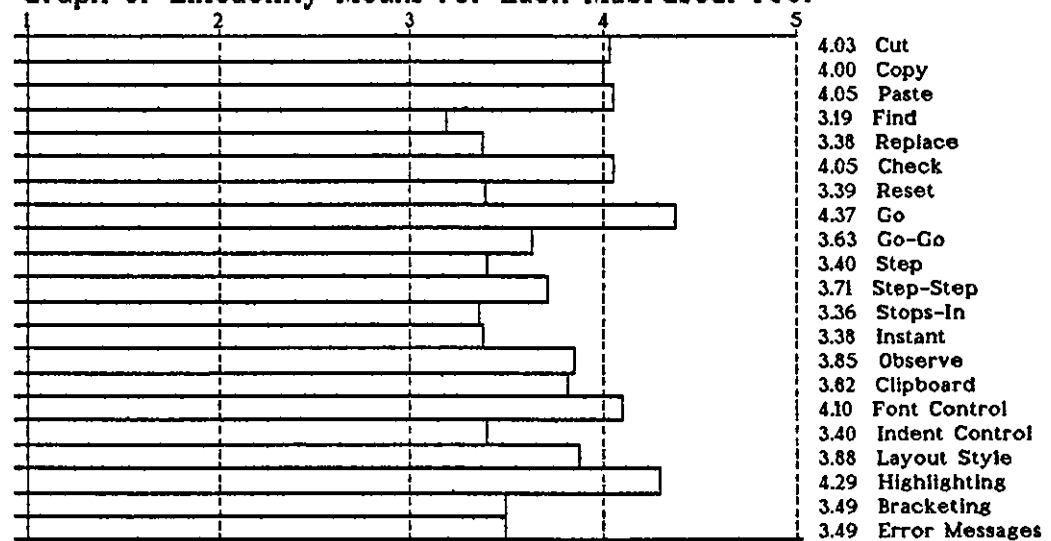
Graph of Ease of Use Means for Each MacPascal Tool



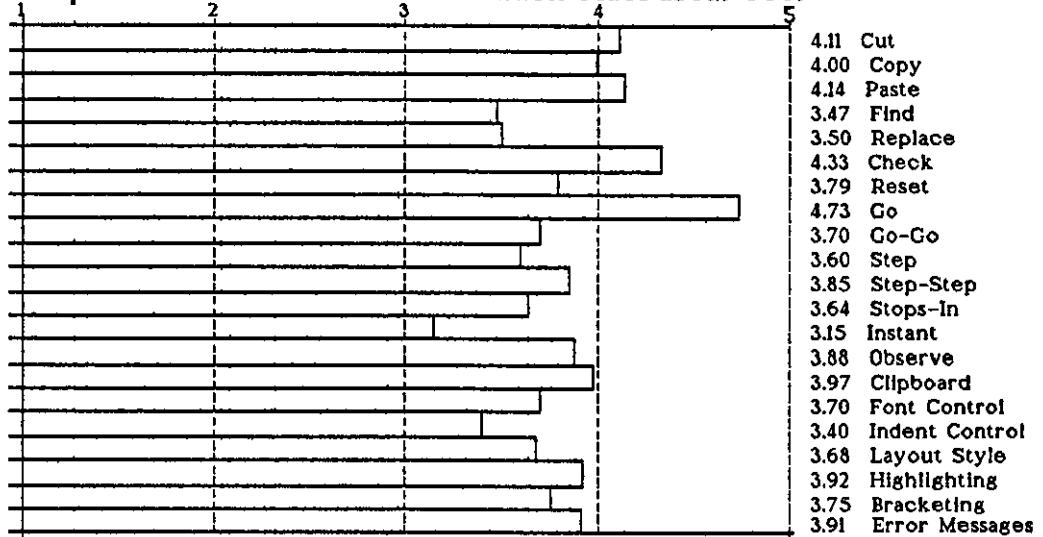
Graph of Usefulness Means for Each MacPascal Tool



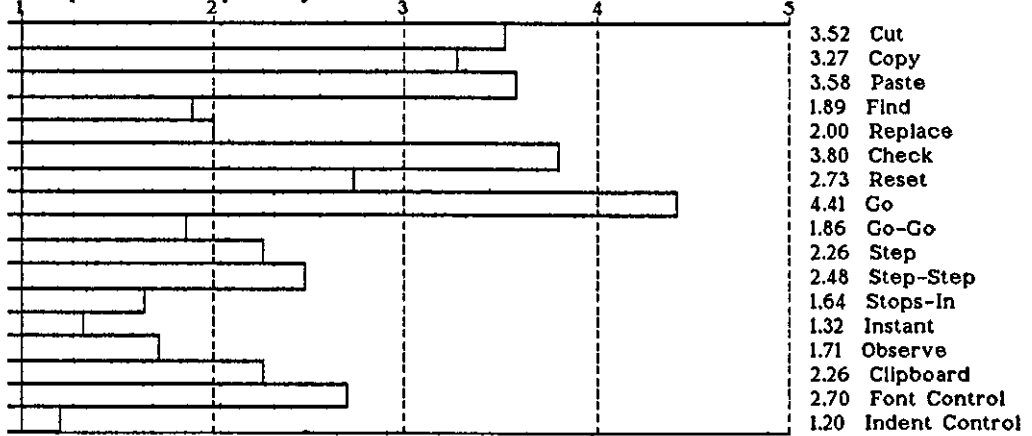
Graph of Likeability Means for Each MacPascal Tool



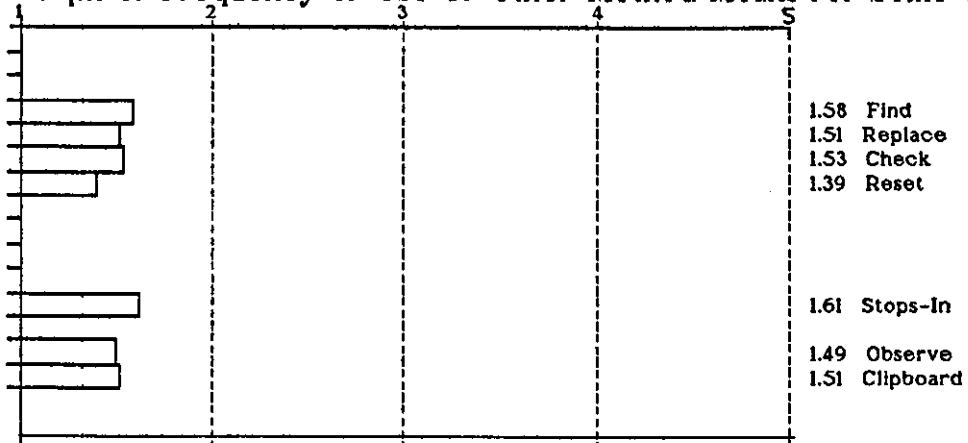
Graph of Usefulness Means for Each MacPascal Tool



Graph of Frequency of Use Means for Each MacPascal Tool



Graph of Frequency of Use of Other Method Means for Some Tools



But the Frequency graph shows a different story. The frequency mean for the "Reset" tool is higher than expected, probably due to the greater frequency of crashed systems or infinite loops in the students' (novice) programs. Students use the "Font Control" tool to reduce the current font size, so that they can get more text on the screen at one time, and avoid unnecessary windowing operations. That is resizing and/or moving the window to view the end of lines that go beyond the width of the window. It is very surprising that the "Find" and "Replace" tools have both got such low mean frequency scores. It would also seem that the debugging aids: "Observe", "Instant", "Stops-in", "Step", and "Step-Step"; have not been put to much use. Either these tools need to be made more user friendly, or they need to be explained or demonstrated, so that the students have a head start on understanding how they work, and how to use them. According to the students' comments, the usual debugging alternative (to most of the above debugging tools) is to insert "write" statements in the code at the appropriate points.

The Frequency of Use of Other Method graph (showing only those tools with alternatives) shows a range of very low mean values, indicating that alternative methods or tools are used infrequently, and according to the tables by few students — numerically 12 or less. Some students use visual inspection and manual editing in lieu of "Find" and "Replace" tools, while others use MacWrite's "Find" and "Replace" tools. Again, inserting "write" statements seems to be the students' preferred alternative to "Observe".

MacPascal performs syntax checks as it interprets the code, so some students rely on this instead of calling "Check" by menu. The alternative "Reset" method is achieved by rebooting - turning the machine off, then on again. "Stops-in" can be replaced by "write" statements and any instruction that causes the program to terminate (or pause for a specific interval, using a wait loop). "Clipboard" can be replaced either by manual typing, or copying a previous text and editing it down, or by using MacWrite's "Clipboard" instead.

3.2.3.2 A Closer Look at Tool Frequency Data

Some tools represent paired alternatives: like "Go" and "Go-Go", "Step" and "Step-Step"; so programmers are likely to develop a personal preference or programming/debugging style suited to one more than the other. The frequency data discussed here occurs on the following decreasing mean ordered table, with visual markers indicating the 1st and 2nd highest values for each tool. These latter values are drawn out separately in 2 paragraphs, defining the tools with highest and 2nd highest values in each frequency category, from Never to Usually.

(Frequency data appears on the next page, with the text continuing on the following page, discussing the results for each frequency category individually).

§3.2.3.2 Data Table & Summary:

Student Votes for Frequency of Use - in Order of Decreasing Mean Values

Name of Tool	Votes actual	Votes per Rating Scale	Mean Data	Swing Data	Majority
		1 2 3 4 5	mean stdv mndv	1+2 3 4+5	dif2
Go	66	3 1 7 #10 *45	4.41 1.04 0.81	4 7 55 51	
Check	66	6 4 11 #21 *24	3.80 1.25 1.00	10 11 45 35	
Paste	66	2 8 #15 *32	3.58 0.97 0.80	10 15 41 31	
Cut	66	2 7 #19 *31	3.52 0.93 0.77	9 19 38 29	
Copy	66	5 11 #18 *25	3.27 1.09 0.92	16 18 32 16	
Reset	66	9 #20 *23	2.73 1.12 0.91	29 23 14 -15	
Font Control	66	#16 13 *18 13	2.70 1.28 1.10	29 18 19 -10	
Step-Step	65	#17 13 *23 11	2.48 1.10 0.96	30 23 12 -18	
Step	66	#19 16 *28	2.26 0.99 0.85	35 28 3 -32	
Clipboard	65	*26 #14 11 10	2.26 1.29 1.12	40 11 14 -26	
Replace	66	*28 #19 11 7	2.00 1.07 0.85	47 11 8 -39	
Find	66	*30 #18 13 5	1.89 0.97 0.81	48 13 5 -43	
Go-go	66	*36 #14 7 7	1.86 1.15 0.94	50 7 9 -41	
Observe	65	*39 10 #12 4	1.71 0.97 0.85	49 12 4 -45	
Stops In	64	*39 #13 8 4	1.64 0.92 0.78	52 8 4 -48	
Instant	65	*52 #6 #6 1	1.32 0.70 0.52	58 6 1 -57	
Indent Cntrl	65	*55 #7 3 0	1.20 0.50 0.34	62 3 0 -62	

Where:

* indicates maximum (primary) student vote for each tool, and

indicates 2nd-maximum (secondary) student vote for each tool.

dif2 = positive - negative = (scale[4] + scale[5]) - (scale[1] + scale[2]).

"dif2" shows the direction of the "majority" vote - the difference between the positive and negative voting numbers. Scale[3] is the "neutral" or middle of the road vote.

Summary: Largest student votes for each tool, in decreasing order :-

never indent control 55, instant 52, observe 39, stops-in 39, go-go 36, find 30, replace 28, clipboard 26

once or twice none

sometimes step 28, step-step 23, reset 23, font control 18

often paste 32, cut 31, copy 25

usually go 45, check 24

3.2.3.2a Never Data

At first glance, it is obvious that a large proportion of students (52/65 or 83% for "Instant", and at least 36/66 or 54% otherwise) have not tried 5 of the more esoteric tools (see data range between "Go-Go" and "Indent Control").

It is much more surprising to know that 28-30 (45% of) students do not use the "Find" or "Replace" tools at all. According to some students, there is a fault in the search string recognition algorithm; it does not always find the next appropriate instance of the search string. An alternative explanation is that students may not be setting up the options correctly. Such as whether to search for whole or partial words. Or whether to disregard case differences between search string and candidate string matches as long as all alphanumerics are in the required order, or not.

3.2.3.2b Once or Twice Data

It would seem that on average between 10 and 20 of the students have tried each of the tools out at least once, just to see what they do. These numbers were expected to be less for this category and larger for the Sometimes category. But the students may not have had long enough to fully overcome the learning curve, or had the need to use the full range of tools. Of course, as time goes by, and programming tasks increase in complexity and variety, all tools would tend to be used more often.

3.2.3.2c Sometimes Data

It is encouraging to see that "Step" and "Step-Step" rate slightly over 33% of the sample size (23 and 28 students respectively) for this category. But the "Reset" value was expected to be much lower than 23; although students may be using this to recover from crashed programs or infinite loops.

3.2.3.2d Often Data

As expected "Cut", "Copy" and "Paste" have peak values in this category (31, 25 and 32 respectively), and "Check" is close behind with 21 votes. Programming always seems to involve a lot of text juggling, and some segments of code are easier to copy and modify rather than writing it all out from scratch. Also some students have skeleton layouts for pieces of text that are bound to recur, such as procedure headers and bodies, with declaration areas and begin-end loops ready for filling.

3.2.3.2e Usually Data

As expected "Go" and "Check" have the lead with 45 and 24 votes respectively. As some students remarked, "How can you run the program unless you use Go?".

3.2.3.3 Summarizing the Mean Rating Scale Data

Table(s) 3.2.3.3.A shows the rankings of tools for each dimension, in order of decreasing means. Usefulness and Likeability have 2 sets of rank values, for 17 and 21 tools, excluding and including MacPascal's 4 automatic features, respectively.

This data was then summarized, and the overall total mean (U+F+E+L) value for usefulness, frequency of use, ease of use, and likeability, for each tool calculated, and ranked on this value. The data is shown in 2 tables (Tables 3.2.3.3.B and 3.2.3.3.C) ranked in order of the decreasing total mean value. The first showing rankings for 17 tools. While the other shows the 17/21 tool ranking data for usefulness and likeability, with 17 rankings for ease of use and frequency, and the total mean ranking (for 17 tools only). The 4 automatic features are slightly separated from the 17 mean ordered tools, as they have no frequency or ease of use data, and thus cannot be interleaved with the other tools, but they show the rankings with regard to 21 tools overall, for comparison on usefulness and likeability. As the latter features are automatic, they were not ranked on frequency of use or ease of use. Considering the problems associated with bracketing errors and error messages, perhaps they should have been ranked on ease of use; but ease of use of response to the tool rather than of the tool itself. In terms of the problems associated with responding to tool messages or effects, and the actions taken in order to fix the error. The answers might well have been very revealing.

Table(s) 3.2.3.3.A shows the rankings of tools for each dimension, in order of decreasing means. The top 5 tools are "Go", "Check", "Paste", "Cut", and "Copy"; for usefulness, frequency, and overall rankings. They also occupy the top 6 rankings on ease of use and likeability. With "Go-Go" taking 3rd place on the ease of use dimension, and "Font Control" taking 2nd rank on likeability (3rd rank if you include "Highlighting" in the ranking of 21 tools); causing a minor alteration in ranking order. "Font Control" comes in 6th overall. With "Step-Step" coming in 7th overall, averaging 8th or 9th across the 4 main dimensions. Most other tool's ranks including "Go-Go" and "Font Control", vary from one dimension to another, with a range of between 3 and 8 ranks.

Table 3.2.3.3.B, The Summary of Means & Rankings table, shows that for 12 out of 17 (70% of) tools, the order of decreasing mean rating scale value for each individual tool on the 4 main dimensions is: ease of use, usefulness, likeability, and frequency. The range in mean values for each dimension was: usefulness 1.58, frequency of use 3.21, ease of use 1.21, likeability 1.18, and frequency of use of other method 0.22 (for 7 tools only).

Table 3.2.3.3.C. Interestingly, "Highlighting" ranks 2nd in likeability, and 7th in usefulness, when considering 21 tools rather than 17. Likewise, "Layout Style" ranks 8th in likeability, and 15th in usefulness, "Error Messages" 13th and 8th, and "Bracketing" 13th and 12th respectively. On the cumulative usefulness and likeability mean scale, "Highlighting" is well ahead of the others, coming in 3rd, compared to the other automatic tools (in the previous order) taking 10th, 12th and 14th place respectively. This means that "Highlighting" is the best liked of the automatic features. This points up how much the students appreciate typographic signalling, even if it only emboldens reserved words.

3.2.3.3.A: Means & Rankings for Each Tool, In Order of Decreasing Mean Value

Usefulness	Mean	Rank 17/21	Likeability	Mean	Rank 17/21
Go	4.73	1/1	Go	4.37	1/1
Check	4.33	2/2	*Highlighting	4.29	*/2
Paste	4.14	3/3	Font Control	4.10	2/3
Cut	4.11	4/4	Check	4.05	3/4
Copy	4.00	5/5	Paste	4.05	3/4
Clipboard	3.97	6/6	Cut	4.03	5/6
*Highlighting	3.92	*/7	Copy	4.00	6/7
*Error Messages	3.91	*/8	*Layout Style	3.88	*/8
Observe	3.88	7/9	Observe	3.85	7/9
Step-step	3.85	8/10	Clipboard	3.82	8/10
Reset	3.79	9/11	Step-step	3.71	9/11
*Bracketing	3.75	*/12	Go-go	3.63	10/12
Go-go	3.70	10/13	*Bracketing	3.49	*/13
Font Control	3.70	10/13	*Error Messages	3.49	*/13
*Layout Style	3.68	*/15	Step	3.40	11/15
Stops in	3.64	12/16	Indent Control	3.40	11/15
Step	3.60	13/17	Reset	3.39	13/17
Replace	3.50	14/18	Replace	3.38	14/18
Find	3.47	15/19	Instant	3.38	15/18
Indent Control	3.40	16/20	Stops in	3.36	16/20
Instant	3.15	17/21	Find	3.19	17/21

Frequency	Mean	Rank 17
Go	4.41	1
Check	3.80	2
Paste	3.58	3
Cut	3.52	4
Copy	3.27	5
Reset	2.73	-6
Font Control	2.70	-7
Step-step	2.48	-8
Step	2.26	-9
Clipboard	2.26	-9
Replace	2.00	-11
Find	1.89	-12
Go-go	1.86	-13
Observe	1.71	-14
Stops in	1.64	-15
Instant	1.32	-16
Indent Control	1.20	-17

Ease of Use	Mean	Rank 17
Go	4.71	1
Check	4.58	2
Go-go	4.40	3
Cut	4.37	4
Paste	4.27	5
Copy	4.23	6
Font Control	4.22	7
Reset	4.05	8
Step-step	4.02	9
Step	3.87	10
Observe	3.85	11
Indent Control	3.80	12
Clipboard	3.79	13
Stops in	3.75	14
Instant	3.69	15
Replace	3.58	16
Find	3.50	17

Other Method	Mean	Rank 7
Stops in	1.61	-1
Find	1.58	-2
Check	1.53	-3
Replace	1.51	-4
Clipboard	1.51	-4
Observe	1.49	-6
Reset	1.39	-7

NB. Negative rank indicates a mean rating scale value less than 3.00, thus representing a negative response attitude. Usefulness & Likeability have 2 rank values, for 17 and 21 tools respectively. * indicates MacPascal's automatic tools.

Table 3.2.3.3.B Summary

Means & Rankings in Decreasing Order for 17 MacPascal Tools per Dimension

	Usefulness		Frequency		Ease of Use		Likeability		U+F+E+L
	Mean	Rank	Mean	Rank	Mean	Rank	Mean	Rank	Mean
Go	4.73	1	4.41	1	4.71	1	4.37	1	18.22
Check	4.33	2	3.80	2	4.58	2	4.05	3	16.76
Paste	4.14	3	3.58	3	4.27	5	4.05	3	16.04
Cut	4.11	4	3.52	4	4.37	4	4.03	5	16.03
Copy	4.00	5	3.27	5	4.23	6	4.00	6	15.50
Font Control	3.70	10	2.70	-7	4.22	7	4.10	2	14.72
Step-step	3.85	8	2.48	-8	4.02	9	3.71	9	14.06
Reset	3.79	9	2.73	-6	4.05	8	3.39	13	13.96
Clipboard	3.97	6	2.26	-9	3.79	13	3.82	8	13.84
Go-go	3.70	10	1.86	-13	4.40	3	3.63	10	13.59
Observe	3.88	7	1.71	-14	3.85	11	3.85	7	13.29
Step	3.60	13	2.26	-9	3.87	10	3.40	11	13.13
Replace	3.50	14	2.00	-11	3.58	16	3.38	14	12.46
Stops in	3.64	12	1.64	-15	3.75	14	3.36	16	12.39
Find	3.47	15	1.89	-12	3.50	17	3.19	17	12.05
Indent Control	3.40	16	1.20	-17	3.80	12	3.40	11	11.80
Instant	3.15	17	1.32	-16	3.69	15	3.38	14	11.54

Table 3.2.3.3.C Summary

Means & Rankings in Decreasing Order for 17/21 Tools on Each Dimension

	Usefulness		Frequency		Ease of Use		Likeability		U+F+E+L
	Mean	Rank	Mean	Rank	Mean	Rank	Mean	Rank	Mean
Go	4.73	1/1	4.41	1	4.71	1	4.37	1/1	18.22
Check	4.33	2/2	3.80	2	4.58	2	4.05	3/4	16.76
Paste	4.14	3/3	3.58	3	4.27	5	4.05	3/4	16.04
Cut	4.11	4/4	3.52	4	4.37	4	4.03	5/6	16.03
Copy	4.00	5/5	3.27	5	4.23	6	4.00	6/7	15.50
Font Control	3.70	10/13	2.70	-7	4.22	7	4.10	2/3	14.72
Step-step	3.85	8/10	2.48	-8	4.02	9	3.71	9/11	14.06
Reset	3.79	9/11	2.73	-6	4.05	8	3.39	13/17	13.96
Clipboard	3.97	6/6	2.26	-9	3.79	13	3.82	8/10	13.84
Go-go	3.70	10/13	1.86	-13	4.40	3	3.63	10/12	13.59
Observe	3.88	7/9	1.71	-14	3.85	11	3.85	7/9	13.29
Step	3.60	13/17	2.26	-9	3.87	10	3.40	11/15	13.13
Replace	3.50	14/18	2.00	-11	3.58	16	3.38	14/18	12.46
Stops in	3.64	12/16	1.64	-15	3.75	14	3.36	16/20	12.39
Find	3.47	15/19	1.89	-12	3.50	17	3.19	17/21	12.05
Indent Control	3.40	16/20	1.20	-17	3.80	12	3.40	11/15	11.80
Instant	3.15	17/21	1.32	-16	3.69	15	3.38	14/18	11.54
*Highlighting	3.92	*/7					4.29	*/2	
*Layout Style	3.68	*/15					3.88	*/8	
*Error Messages	3.91	*/8					3.49	*/13	
*Bracketing	3.75	*/12					3.49	*/13	

Where:

* indicates MacPascal's automatic tools ranked on the 21 tool rankings, but excluded from the 17 tool rankings.

Negative rank indicates a mean rating scale value of less than 3.00

3.2.4 Consensus of MacPascal Tool Deficiencies or Necessary Enhancements

There are 2 sub-sections. The first concerns the students' problems, and their criticisms of the MacPascal tools and environment. The second discusses some of the enhancements suggested by the students.

3.2.4.1 Summary of Problems and Deficiencies

The following 2 tables show the number of students who voted for each particular issue, in relation to students' problems (Q26), and their complaints about MacPascal (Q24). The table heading defines the gist of each question, and the number of students expressing each opinion (frequency) appears at the righthand margin.

Q26. Students' Most Common or Time Consuming Problems/Difficulties :

<u>Problem</u>	<u>Frequency</u>
<u>syntax errors (unspecified)</u>	<u>13</u>
<u>misplaced/missing semicolons</u>	<u>8</u>
<u>typing/spelling errors</u>	<u>6</u>
<u>algorithm/semantic errors</u>	<u>6</u>
<u>file I/O errors</u>	<u>4</u>
<u>missing begin/end</u>	<u>3</u>
<u>variable/parameter declarations</u>	<u>3</u>
<u>bracketing errors</u>	<u>2</u>
<u>MacPascal construct misconceptions</u>	<u>2</u>
<u>switching/resizing (of windows) to see relevant code</u>	<u>2</u>
<u>lack of speed</u>	<u>2</u>
<u>problems in debugging due to layout</u>	<u>1</u>
<u>more comprehensive run-time error checking</u>	<u>1</u>
<u>listing all errors in one go</u>	<u>1</u>
<u>error elimination</u>	<u>1</u>
<u>finding out why it doesn't work</u>	<u>1</u>
<u>infinite loops</u>	<u>1</u>
<u>unwanted recursions</u>	<u>1</u>

Clearly most problems are caused by syntax errors in one form or another. Followed by a combination of algorithmic and semantic errors. Typing and spelling errors can contribute to either of the former, and bracketing errors usually arise from miscounting brackets during input typing and checking. File I/O, windowing problems and speed complaints result from a combination of the way MacPascal is implemented on the Macintosh system. Window resizing is particularly aggravated by the small screen dimensions (about 20cms diagonally) due to the combined Macintosh VDU screen and disc drive unit (the pre-1990 MacPlus look alike).

Finding out why it doesn't work is a problem that can only be solved by the programmer's ingenuity in assessing and identifying the error, and applying a suitable remedy.

<u>Q24. Complaints or Criticisms</u>	<u>Frequency</u>
<u>error clicking complaints</u>	<u>6</u>
<u>(lack of) speed complaints</u>	<u>6</u>
<u>students not using search</u>	<u>6</u>
<u>more informative error messages</u>	<u>5</u>
<u>noise criticisms</u>	<u>3</u>
<u>compiler requests</u>	<u>3</u>
<u>user layout control</u>	<u>3</u>
<u>decent hardcopy facility</u>	<u>3</u>
<u>reset complaints</u>	<u>3</u>
<u>bracketing complaints</u>	<u>3</u>
<u>window switching criticisms</u>	<u>3</u>
<u>multi-windowing/multiple application requests</u>	<u>3</u>
<u>window resizing criticisms</u>	<u>2</u>
<u>file handling</u>	<u>2</u>

Many students complained about error clicking - needing to move the mouse and click the error message (at the top of the screen) before they could go on to fix the bug itself. Many suggested hitting the "Return" key instead of manipulating the mouse unnecessarily. Again there were requests for better error messages, and the ability to send these to the printer, since the error message becomes irretrievable after it has been "clicked". The noise complaints mostly refer to the beep associated with the error message appearing. First of all, the sound itself "trumpets-aloud" the fact that you have an error, which is particularly offputting for a novice, and becomes an annoyance itself when heard 10 times in a row.

The requests for user control of layout, I sympathise with whole heartedly. Like them I find it difficult to interact with a piece of code that is not laid out according to my own preferences. The "Indent Control" can help lessen this feeling somewhat by altering indentation to the user's preferred spacing, but otherwise the layout style is unaffected. Speed and windowing complaints again are down to MacPascal and the Macintosh's screen size.

3.2.4.2 Summary of Enhancements Suggested in Questionnaire

The following is a distillation of the most interesting or necessary enhancements suggested by the students in the questionnaire. Mostly from questions 25, 24, and 26, respectively; but also from the comments given in response to different aspects of MacPascal's tools in previous questions. A Frequency table, like those in §3.2.4.1, can be found in Appendix 3B. It is not included here as it duplicates part of the previous tables, which have already been discussed above.

The enhancements have been grouped into 3 categories: MacPascal-specific, MacPascal-oriented but with wider implications, and general enhancements. The students have several specific enhancements in mind for MacPascal (regarding files, windowing, and speed) which are the responsibility of the MacPascal environment's designers. The more interesting enhancements suggested by the students have been elaborated further under the last 2 categories.

MacPascal-specific

- Compiler wanted rather than an interpreter (probably to get cumulative error messages, but see note "***" at top of next page).
- File handling requirements:
 - decent file handling (presumably referring to creation/deletion of files);
 - RAM (random access memory) files;
 - assigning of I/O files by the user; and
 - test file creation facility.
- Saving/editing of text window.
- Multi-windowing/multiple applications open and on-screen at the same time.
- Facility to speed up window switching
 - single keypress window switching; and
 - automatic switching and resizing on selection of a given window.
- Decent hardcopy facility - one that enables printing of all, or a selected portion, of the file rather than just the current window's worth.

Enhancements aimed at MacPascal, but with wider areas of application

- Error messages:
 - should be more accurate and informative, and less noisy (or better still noiseless);
 - the error message should be accessible/recallable until the error is fixed; that is, you should be able to remove the error message by a single keypress or mouse click and then be able to call the error message back up onto the screen, as needed;
 - you should be able to send cumulative error messages to file or printer rather than being restricted to finding and fixing one error at a time. However, the latter is a feature of MacPascal's implementation. But is certainly a valid point otherwise.

****Choosing whether to stop on the first error encountered or to accumulate error messages depends on the implementation chosen, rather than on whether the environment is based on an interpreter or a compiler (since either is possible).**

- Cut: one student suggested an inverse facility to delete all text, except for a prior selected block. This idea has innate appeal but whether it would be useful or even definable is questionable.

- Search and replace facilities. There is obviously a need for better search and replace functions, but the question is, in what way can they be improved and made more effective? What is the simplest way of making them more effective? Perhaps making the search string more visible would help. Or, alternatively, making the students more aware of the options available with the search mechanisms. Then checking that they know how to set the appropriate options would solve the problem of "Find" not visiting or locating all intended instances of the search string.

- Step-step: needs an infinite loop indicator. The problem with an infinite loop detector is in deciding at what numerical loop cycle maximum the loop should abort. Maybe there should be a tool that increments the loop count each cycle, and shows this value to the user - perhaps as an adjunct to the "Observe" or "Step-Step" tools. Then if the user decides that the loop count is too high, he/she should be able to break out of the program. An alternative would be to have an option whereby the user defines a maximum loop count value, and if any loop is executed in excess of this value, then the program stops. Giving the user the choice of exiting the program, or setting a new maximum loop value, and continuing with program execution.

- Colour. Now that colour screens are replacing green screens the use of colour is becoming less of a facile decorative feature; and its ability to provide instant differentiation between objects is beginning to be exploited to its full extent. For example, using colour coding to distinguish significant elements of program text; like having one colour for reserved words, one colour for variables, and another colour for procedure/function calls or declarations. To make it easy to identify each element's function in one glance, by colour, rather than needing to read each word individually (within a monochromatic and visually bland text) to determine what it represents (as proposed by Baecker & Marcus 1990, for the SEE visual compiler). Colour can be to accentuate cognitive markers or beacons within program text (Treisman 1982, 1984; Wiedenbeck 1986), or to differentiate levels of nesting within program text, as with Van Laar's (1989) colour coded support tool.

- User control of text formatting (individualizing layout style), giving the choice of:
 - choosing whether to highlight reserved words or not;
 - changing the layout of some sections of code; and
 - entering text in "free format" according to user's style.

Emboldening reserved words seems to be appreciated by most students. As shown by

the fact that the "Highlighting" feature came out well in the rankings. However, an option to turn it on or off would mollify those students who didn't like this feature. The other 2 enhancements suggest 3 possibilities. First, that students are asking for automatic layout in a default system style that is not open to customization by the user. But with the option to lay out sections of code in their own individual style, with some kind of marker(s), so that there is no chance of it getting reformatted automatically. A means of sending some kind of "hands off" marker to the formatting program to protect the relevant section(s) of text.

Second, that automatic layout is in the user's style - selectable from a matrix of choices for the disposition of each construct regarding indentation and the placement of reserved words. With the option to alter that style for a particular section as above (the "hands off" formatting feature). Say when the length of a group of individual lines is greater than the current level of indentation will allow, and the user decides to override the lefthand indentation margin temporarily, to avoid breaking each line.

Third, that layout is completely under user control - meaning the manual setting and maintenance of indentation levels. Perhaps with a little help from the system, such as Vi's auto-indent feature.

- Font control: the ability to choose a new "default" font for program text on subsequent sessions. Being able to choose a particular font to code in would help establish a visual rapport with the code, and add familiarity. Possibly helping to reduce the time taken to get "in tune" with the code at the beginning of an editing session.
- The ability to define and use libraries of user defined procedures and functions, and be able to interrogate them easily. This would add familiarity, and re-use existing code. So it might cut down on the development and debugging time used to reinvent the same or similar coded functions or functionality.
- Showing stack and heap pictures - providing a way of representing the "current memory free/used ratio" graphically. This would probably be most useful for recursive algorithms or other memory intensive algorithms.

General enhancements

- To widen the range of screen moving commands in terms of scrolling. To speed up the ballistic phase and cut out intermediate steps towards the goal. For example, moving by a full or half screen of text, to go to the start or end of a file or a named procedure (or one of its parts, such as the declaration area) directly, and so on.
- Providing a semicolon adding utility. Since missing or misplaced semicolons account for a good proportion of (Mac)Pascal errors, it is about time there was a facility to do this. After all the compiler knows where they should go, otherwise it

would not report them as missing. So why not put them in instead? At least in straightforward and simple cases. Leaving ambiguous situations to be resolved by the programmer directly. This could also be used to demonstrate the basic rules of syntax regarding semicolons to novices. As well as removing some of the (mental) burden and drudgery of fixing common syntactic errors.

- Provide more complete/helpful diagnostics. The better the diagnostics, the faster bugs can be eliminated; cutting down on debugging costs, by providing
 - (non syntax) semantic checker. For example, (a) checking that applied functions and variable types match - a pre-compiler checker; or (b) checking that real values were not assigned to integer variables unless a "trunc" or "round" function had been applied to the real part of the expression beforehand; or (c) checking that loop variables, values, and/or conditions are specified correctly. That is, using the right type of loop variable, in terms of data type and its inherent maximum and minimum values. For example, a boolean variable can only be true or false. Checking that loop interval values are in the right order, that upper bound is larger than lower bound.
 - more comprehensive run-time error checking and reporting back. Such as reporting on under- or over-flow of a variable's value; or being unable to evaluate a boolean expression, because one of its sub-parts is undefined, instead of bombing out without giving the user a clue as to what has caused the error.
- Reserved words help and syntax diagrams would be useful to novices and those who have transferred from a different language and need an on-line guide to get correct syntax.

3.2.5 Attitudes Towards MacPascal's Tools

Reading through the students' comments on the MacPascal tools, significant phrases (or paraphrases) kept cropping up. Defining specific aspects of the task-tool match that the students regard as important. Also indicating that the students were applying specific criteria when assessing the MacPascal environment and its tools in relation to the programming task. These phrases cover 4 areas of interest. The first group deals with aspects of direct engagement (Norman 1986) and task-tool efficiency. The second with error elimination and debugging, the third with aesthetics and comprehensibility of program text, and the fourth with areas of proposed user control.

- **Criteria for assessment of direct engagement and task-tool efficiency:** straightforwardness, speed, efficiency, saving effort, saving time, ease of use (and ease of operation), ease of text entry, ease of text modifiability, ease of re-ordering text, minimizing re-typing, and being able to use command or operational shortcuts (such as using control codes instead of pointing and selecting with a mouse).

- **Improving error elimination and debugging:** error reduction and/or error prevention at source, efficient error elimination (both syntactic and semantic), the need for good debugging aids, and informative and understandable error messages - with the option to print them out.
- **Criteria regarding aesthetics and comprehensibility of program text:** neatness, aesthetics of presentation, text presentation and layout style, code readability, and code comprehensibility.
- **Areas of proposed user control:** execution speed (especially step-step), text presentation and layout style, and windows (resizing and repositioning).

These criteria for judgement of tools and environment determine how tools are regarded, and on what basis the judgement is made. Not only are they judged on how well they do the job they are made for, but whether they collectively cover all aspects of the task. Thus they point up some important issues that need to be addressed.

For example, reducing or eliminating errors in the form of error prevention. By automatic solution of simple errors, such as missing semicolons, or inserting an "end" to match an unpaired "begin". To avoid complications, and the possibility of missing elements being put in the wrong place, an automatic helper could suggest where to put things. So that the programmer can choose either to confirm or abort the proposed action. Indeed, Teitelman's (1972) Programmer's Assistant was designed to do exactly this type of job in the Lisp environment.

The above list also has another function, it emphasizes and supports the human factors angle of the tool(s), viz straightforwardness, ease of use, and so on. Re-iterating the importance of making the tool fit the task, in a natural, comfortable way, and as closely as possible. To bridge the gap between the user and the tool.

I found it very significant that some students stated that the MacPascal style of layout made it difficult for them to debug their programs. I believe that each programmer develops a style of layout that heightens his or her rapport with the developing program, and this enhances understanding of the program, and thus the ability to debug it thoroughly.

3.2.6 Consensus of Student Opinions on Comments and Program Design, & Prior Use of Computers & Programming Languages

The last 3 questions in the questionnaire dealt with more general aspects of program design. Namely, students opinions on comments: firstly in their own programs, and secondly in other people's programs. This issue has relevance to comprehension of program text by the reader, especially regarding the updating and maintenance of

code. The last question (Q30) gathered information on which program design method the students used in real life. To make sure the design of the proposed programming tool(s) was based on a true assessment of students' actual program design behaviour, and not on assumptions. Question 27 provided data on students familiarity with computers and programming languages, defining who had or had not used a computer before the course began - this data is listed in Appendix 3B.

Q28 Students' Attitudes Towards Putting Comments in Their Own Programs

The following is a distillation of the students' answers to the question "Why do/don't you use comments, and if so, where?"

Positive Responses were:

- 9 students put comments at the start of the program and top of each procedure;
- 6 students sometimes comment, but not always;
- 6 students comment in order to explain complex parts or to jog their memory;
- 5 students always comment; and
- 1 student comments only if he thinks he might have to update the program later on.

Making a total of 27 students out of 66 who use comments to some extent.

However, there were also Negative Responses:

- 6 students are "too lazy" (the students' own words!) to comment their own programs;
- 4 students don't comment (but gave no reasons why not);
- 3 students don't comment because their programs are not for other people to read;
- 3 students thought their programs were not large enough to need them; and
- 1 student doesn't comment because he thinks they clutter up the window.

Making a total of 17 students out of 66 who don't comment for the above reasons.

However, according to the scoring table below, 12 never comment, 12 use comments once or twice, and 39 use comments on a more frequent basis, ranging from 16 who use them sometimes, to 8 who always use them.

<u>Rating Scale</u>	<u>1</u>	<u>2</u>	<u>3</u>	<u>4</u>	<u>5</u>	<u>mean</u>
<u>Usefulness</u>	2	3	17	22	12	3.14
<u>Frequency</u>	12	12	16	15	8	2.79
<u>Ease of Production</u>	1	1	15	10	27	3.38
<u>Likeability</u>	2	1	21	11	17	2.97

Looking at the range of student votes per rating scale for the other dimensions, most students think that comments are useful to some degree, and they are not regarded as being difficult to produce. They score slightly less for likeability, but again the majority scores range between Ok and liked.

Thus comments are being under used for several reasons, principally :-

- 1) because students don't bother to spend the extra time or effort producing them; or
- 2) at the moment they are only producing code basically for themselves, in order to practise programming and learn how to do it. So they don't see the need for comments that no-one else except their tutor will see; also
- 3) the programs being produced are fairly short and well understood, and the algorithms and underlying structure of the code make it relatively easy to see what is going on.

Of course, (Mac)Pascal is regarded as an easy to read, fairly easy to comprehend programming language. From the start student programmers are impressed with the idea of naming variables to reflect what they are used for, and what the data they carry represents functionally; such as "maxtemp" or "timecount" and so on. Using self-explanatory variable names may reduce the need for comments overall, but does not eliminate the need for comments altogether.

What worries me is that students may get into the habit of not commenting their code and will never learn the discipline of inserting comments as they go. Perhaps they will see the error of their ways when they come face to face with a 20 page program with few comments in it! Personally I find that comments often show me the variation between the intended action of the code and the actual code, and thus provide useful debugging help. Also if it is difficult to form an appropriate comment phrase, then there may be something wrong with the intended action to be coded. On the other hand some things are easier to code than they are to describe!

Q29 Students' Attitudes Towards Comments in Other People's Programs

The following is a distillation of students' answers to the question "What do you think of other people's comments?"

Positive Responses were:

- 3 students think that longer or more complex programs should have comments;
- 1 student thinks comments are always useful;
- 1 student thinks comments are useful in showing structure;
- 1 student thinks comments make a program more readable; and
- 1 student thinks they make understanding the algorithm easier.

Making a total of 7 students who think that comments are useful and aid understanding of other people's programs.

However, there are also Negative Views:

- 2 students consider other people's comments to be too "cryptic";
- 1 student finds that few people put comments in their programs;
- 1 student thinks that comments disrupt readability;
- 1 student thinks that other people's programs are harder to follow;
- 1 student would prefer a separate explanation; and
- 1 student doesn't look at other people's programs!

The table below shows the division of opinion about how useful comments are according to dimensions a), b) and c). Looking at the data I now realise that b) may have been interpreted ambiguously - some of the "No" votes may be disagreeing that comments are only useful in long programs; and may be indicating that comments are useful regardless of length of code. However, dimensions a) and c) are more clear cut. Showing that only 44 % of students feel that comments are useful in general. I think this bodes ill for them professionally in the future, or for the programmers who have to maintain their programs.

<u>Dimension of Usefulness</u>	<u>No</u>	<u>Yes</u>
a) Generally	36	30
b) Depends upon length of code (useful only in long programs)	58	8
c) Depends upon the complexity of the code	37	29

Q30 Actual Distribution of Program Design Methods Being Used

Student programmers (of my generation 1980-83) were always being exhorted to use Step-wise Refinement as a design method. Where the emphasis is on the importance of working out the entire design algorithm on paper before inputting program text to the computer, and entering the iterative "edit, then attempt compilation" phase. The more recent programming lectures (attended prior to observation) continued this traditional emphasis. The following question finds out if they do, and to what extent. Each choice [except e)] was accompanied by "usually" and "sometimes" tick boxes.

What is your natural program development/coding methodology? Do you :

- work out the entire algorithm then translate it into code
- work out most of the entire algorithm then translate it into code, and fill out the missing parts as you go along
- work out a partial algorithm and then continue as for b)
- use direct terminal composition - where you express the algorithm in code (on the VDU screen) directly without working it out on paper first
- use method a) b) c) or d) depending on task complexity - if so define the conditions where each method(s) is used eg. depending on task difficulty, or length of code required
- use other method(s) - please give details.

<u>Design Method Used</u>	<u>Student Numbers</u>				<u>Student Percentages</u>			
	<u>N</u>	<u>S</u>	<u>U</u>	<u>S+U</u>	<u>N</u>	<u>S</u>	<u>U</u>	<u>S+U</u>
a) entire algorithm	28	20	18	38	42.4	30.3	27.3	57.6
b) most of algorithm	21	21	24	45	31.8	31.8	36.4	68.2
c) partial algorithm	37	14	15	29	56.1	21.2	22.7	43.9
d) direct terminal composition	42	20	4	24	63.6	30.3	6.1	36.4
f) other methods	58	5	3	8	87.9	7.6	4.5	12.1
		80	64			121.2	97.0	

Where N = never, S = sometimes, U = usually.

Looking at the data, it seems obvious that each student uses one main method and at least one secondary one. Or alternately, two principal methods which are used depending on the circumstances, such as task complexity. Ranking the design methods according to student numbers, and in order of decreasing frequency of use, gives b) a) c) d) with f) last.

The only real difference between methods a) b) c) and d), is the proportion of time spent designing on paper or on VDU. Does it really matter which medium is used most? Or is it just an outdated economic consideration that assumes that 100% paper designing is better, or at least cheaper than designing on-screen. Of course with an undergraduate population of about 90 students on a programming course, and about 20—30 Macintoshes or terminals linked to the mainframe, there is going to be a scarcity of resources if everyone decides to design from scratch on-screen.

I know from experience that it is easier to start the design on paper, because you are not restricted physically in the same way. With an editor you have to use edit commands to do anything and the screen size is limited, and usually makes it difficult to compare different sections of code. Although using multiple windows (as with the EMACS or SunTools editor) can lessen this effect. But the real advantage of paper is that you can switch between sketch and write mode instantly, and go from one location to another (either within a page, or document, or even from document to document) freely, without issuing a stream of commands that break your train of thought. This is what Green (1989) calls role-expressiveness of the interaction medium. With pen and paper role-expressiveness is high, but with an editor it is low.

However, for a 10-20 page program, after the basic design is sketched out on paper and has gone through a couple of revisions to smooth the joins between algorithms and different modules; it becomes increasingly irksome to copy out all previous text in order to add or change details. Not only does the paper get dog-eared and so full of writing that it is difficult to add further details, but it begins to become illegible. This leads to the misplacement of lines of code, so that the sequencing of events or data modifications becomes incorrect and muddled. Also, transcription errors become more likely as the copying process reiterates from one full sheet of paper to a clean sheet. Thus, it is much quicker, simpler, and less error prone to get the text onto the computer, so that the printer gives a fresh, legible copy for the programmer to amend.

Q27 Students' Prior Use of Computers & Programming Languages

Out in the real world, prices of personal and business computers continue to drop, and sales increase, making it relatively cheap to supply each employee who needs one with a computer. With the advent of the Sinclair Spectrum, Amstrad and the IBM machines in the early 1980s, came the concept of the Personal Computer for "ordinary" people. And it is no longer uncommon for a family to possess a computer, whether for wordprocessing, desktop publishing or programming.

According to data supplied by the students regarding previous experience of computers and programming environments (see Appendix 3B); 41 students had prior working knowledge of computers, and 28 students were familiar with at least one programming language or environment. Only 5 students said that they had not used a computer before starting the undergraduate course; 9 had used one form of Basic or another; 18 had already used at least one version of Pascal (but not MacPascal); 5 had used different versions of C, and 3 had used different versions of Cobol; 6 list VML (Virtual Machine Language) only, and 14 skipped the question. Thus it is probable that at most 25 students (including the 5 students mentioned at the start of paragraph) had not used a computer before the course began. So 25 out of 66 (37%) students were not computer literate, compared to 63% who were.

3.3 Conclusions Resulting from Preliminary Data

Quantitatively speaking, the Frequency of Tool Use data was most interesting. It showed that many of MacPascal's inbuilt tools were under-utilised by the students. This may have been due to the students' lack of confidence or reluctance in trying new tools out, being novices; or simply due to laziness, in avoiding the learning curve for each additional tool; or because they hadn't done enough programming to bulk up the frequency mean values to their "true" values (analogous to the law of averages).

That 25 (37% of) students had not used computers before starting the course might explain the low use of debugging tools, and the difficulties encountered with the search mechanism. After all, it is difficult for novices to appreciate the difference in effect when a search task is case sensitive, rather than case insensitive. In everyday life, case sensitivity is ignored by and large, since the message matters more than the presentation case-wise.

Thinking back, I don't actually remember anyone telling me how to use an editor. The most help current students had was probably a brief description or demonstration of how to use the mouse to pull down menus, and to select an option. With no further help, unless they were given a (written) "step by step" example editing session to follow and try out. I don't know that any programming class ever discusses the tools provided by an environment. Learning to use an editor is a question of trying things out, and seeing what happens. Building an internal model of editing operations as you go along, and watching how other people use the tools, or asking "how do I do ...?".

Recapping, the ranking of MacPascal tools regarding Frequency of Use are :-

Go, Check, Paste, Cut, Copy, Reset, Font Control, Step-step, Step, Clipboard, Replace, Find, Go-go, Observe, Stops in, Instant, and Indent Control.

Whereas the overall rankings are:-

Go, Check, Paste, Cut, Copy, Font Control, Step-step, Reset, Clipboard, Go-go, Observe, Step, Replace, Stops in, Find, Indent Control, and Instant.

That programming involves a lot of text juggling is borne out by the positioning of "Paste", "Cut", "and Copy", within the top 5 tools. It is obvious that the "Step"ing tools are the most popular for debugging. The others are way behind the field. As stated earlier, the students' prefer to insert "write" statements in the code, rather than using the other debugging tools provided. Inserting "write" statements is a well known and probably ingrained programmer behaviour, which aids debugging independent of the tools provided by the programming environment. Perhaps more importantly, it is simple, effective and completely under the programmer's control, and is applicable across ALL systems. Thus providing debugging information without having to learn a variety of other debugging tools and their quirks.

Finding out that 28-30 (45% of) students do not use the "Find" and "Replace" tools at all was unexpected. Either there is a fault with MacPascal's search string recognition algorithm that needs to be fixed; or the students need to be shown how to set up the options correctly. Using visual inspection as an alternative is slow and prone to error.

Although invisible character problems were seen on a MacPascal system, they affect other programming systems as well - including Vi. It is about time that the creators of editors and editing environments, either developed filters so that invisible control characters cannot be inputted into program text, or a means of showing their location and eliminating them was provided.

The ranking data showed that emboldening reserved words seem to be appreciated by most students. Colour could also play a useful part in tasks requiring visual distinction or differentiation within an otherwise bland monochromatic program text. To show the location of each instance of the search string within the text, for example.

Particularly, if the user's goal is to follow a variable's trail - by finding all instances of that particular variable name.

Increasing the Macintosh's screen size would help a lot - it is just too small. In fact the reason the students use the font control tool so much is to reduce each character's height and width. So that they can squeeze as many characters onto the screen as possible and avoid unnecessary window movement and resizing. A problem that could be reduced significantly, by increasing the screen size, and the number of characters that it can hold.

The main difference between MacPascal and most other procedural programming environments, is the fact that the programmer is forced to fix errors one at a time, instead of being able to generate a list of all errors, in one go; so that he/she can consult the list whilst modifying the program text in the editor. The latter is much faster because you don't have to keep popping in and out of the editor to check whether you fixed the last error and can now go on to the next. Thus wasting less time and mental energy per error fix. Nanja & Cook's (1987) data shows that experienced programmers tend to fix several errors at once, in bursts, rather than one

error at a time. So a "fix one error at a time" based environment is contra-indicated for experienced programmers. Although it might be more helpful to first time programmers since it concentrates their attention on fixing one error at a time, instead of overwhelming them with a long list of errors (Shneiderman 1980).

Looking through the list of suggested enhancements, there is still a need for a wide variety of programming tools, as well as modification to existing tools. There is also a need for more debugging and checking tools. Syntax errors are most frequent according to §3.2.4.1, but are usually easier to fix than semantic errors. A syntax checking and assisting tool, incorporating a semicolon adding feature, would reduce syntax errors substantially. Particularly, if it fixed simple syntax errors like missing semicolons, and suggested solutions to more complex problems. Such as the placement of a missing "end", but left the final decision to the user.

The need for more accurate and informative error messages cropped up time and again, and MacPascal is not the only culprit. Most programming system messages are uninformative and incomprehensible to some degree. Perhaps in the past the shortage of memory dictated the need for short, terse messages. But this is no longer the case, and the programmer's time should not be wasted in error decryption.

The observation data showed 4 types of problems: learning-related, task-specific, general, and MacPascal-specific. Time and experience usually counteract most learning and task-specific problems. Although some problems, like infinite loops and the correct sequencing of variable value changes, crop up every so often. Especially when there are a lot of variables to keep track of and to control.

One suggested enhancement was for an infinite loop indicator. Either reporting the loop count at each cycle, or pausing when a user-defined maximum was reached. In either case giving the user some control over the situation. An alternative would be to provide some kind of loop or condition checker. To check that loop variables, values, and/or conditions are specified correctly. Such as using the right type of variable, and checking that the loop interval values are in the right order. Thus preventing infinite or redundant loops from occurring.

The data on students' attitudes towards comments is rather worrying. Out in the real world, the majority of programming work is maintenance work. Where getting in tune with an existing program, or body of software, is essential in order to be able to debug and/or modify it. Comments provide one way of gaining an insight into the way the code was intended to work by its author. Thus passing vital information to the reader who needs it most. If the comments aren't there the comprehension task becomes that much harder, especially if the person reading the code is (initially) unfamiliar with the role the software plays, viz its operational function and the consequences of each action in the real life domain.

The program development data shows that the most prevalent design method is to lay down the broad structure of the program elements. Then to add any missing parts or details, and to gradually fine tune the program, during the edit-compile-debug or debug-edit-compile cycle (as Pennington & Grabowski 1990 found, most programming activities are interleaved, including editing and debugging). This is why it is important to provide tools to help get the details correct at an early stage. Instead of letting them get through to later stages when they become increasingly expensive to extract both time- and damage-wise (Yourdon & Constantine 1979).

As to the question of program text layout - MacPascal emboldens reserved words, and blocks out program text at 2 spaces per level of indentation (by default), for each controlling construct. Keeping the indentation of "child" blocks or compound statements at the same level, starting at the same column on the screen.

Automatic layout relieves the programmer of the tedious job of checking that indentation is correct, and the placing of reserved words, thus speeding up text entry. And, if a loop is added that encircles a section of code, the layout changes automatically. But automatic layout at present has a fixed format and there is only one "style" of layout for each construct. The system is inflexible and will not allow the programmer any freedom of choice in the individual disposition of construct elements.

In contrast, with free format editors such as Vi, the programmer can lay out the text in any way at all. However, adding or deleting a loop means that all indenting and reformatting of program text must be altered "by hand" - physically adding or deleting spaces and repositioning the affected text. A time consuming task that uses up time that could be better spent on development and debugging.

The students' comments suggest a need for some kind of "hands off" marker, that would allow them to write code in their own style - if only for specific sections of text - as well as having the facility of automatic formatting of program text. At the moment the options regarding text layout are either fully automatic, or fully manual (do-it-yourself), formatting of text. There are no dual-mode layout tools, as yet. But perhaps there should be. The data certainly shows a need for them.

Chapter 4 Relating Integrated Editing Tools & Cognitive Programming Tasks

One aim of my research is to try to provide some kind of model that relates software tools and their usage against the cognitive operations that the solo programmer-designer uses to drive code production.

4.1 Examining Editing Actions

According to Allen (1982) there are 4 basic text editing actions: insert, delete, modify, and move. Similarly, Douglas & Moran (1983) define 2 text editor operator classes: mutative - changing the text; and locative - moving the cursor. Galambos, Wikler & Black (1983) define 2 types of moving operation: ballistic - moving to the general area of interest; and precise - moving to an exact (x,y) location on the screen.

So, the editor must provide a means of adding, deleting, modifying, or moving text; as well as relocating the cursor position. For example, by using a mouse, and/or keyboard based and screen scrolling function keys (for ballistic movement); or using command mode search functions to move to specific locations (precise movement). In either case the cursor position focuses visual attention.

Most of these actions are specified by the user and are carried out by the tools automatically. However, the user can also choose to alter (add or delete) text directly by manual input through the keyboard. The following categories show the range of variations in tools and utilities for each of these 5 activities :-

4.1.1 Adding/Inserting Text

- inserting or appending lines of text manually character by character;
- importing text from other files - either whole or partial files; or
- using a buffer to add partial text from the current file being edited.

4.1.2 Deleting Text

- using "grammar unit" deletion chunks^{*}; such as delete (from current cursor position to beginning or end of next) word, line, sentence, paragraph, to end of file etc.;
- using backspace and any other keyboard functions to delete text; or
- using a buffer to delete text from the current file being edited.

^{*} This is usually done using a specific symbol to define each "grammar unit". For example, "w" for word, ")" for sentence, "}" for paragraph, and so on.

4.1.3 Modifying Text

According to Gray & Anderson (1987), a change-episode has 3 stages: the noticing event, the decision to change the code, and the "the fix" (modifying the code).

Modifying text can be achieved by combining the other 3 text editing operations, or by using specific modification or change commands.

- small scale text modifications - using the search & replace mechanisms on a word or phrase. These are usually used to make variable, procedure, or function names more meaningful. One problem involves determining the "depth" or number of replacements to be executed. Also determining whether case sensitivity matters. And, deciding (either beforehand, or on the fly) which strings are, or are not, to be replaced; or
- large scale text modification - using "grammar unit" change chunks; such as change word, line, sentence, paragraph, to end of file etc.

4.1.4 Moving Text

Text movements can range anywhere from small (a couple of lines) to large (60 lines or more), with movements based on line numbers or screen coordinates. The latter case is usually oriented towards marking text for copying or cutting using a mouse or other pointing device. Using a pointer to specify screen coordinates, such as (x1,y1) and (x2,y2), to mark the start and end positions for copying or cutting means that partial lines can be included as well as whole lines.

- small scale movements - moving one or more lines of code up or down within a loop or procedure body to occupy a more relevant position; or
- large scale movements - altering the order of procedures or subprocedures, either by shuffling the relative positions of 2 or more procedures, or to combine (nest) 2 procedures, or to extract a child procedure from its parent (to make them independent).

Text copying and cutting can be achieved by yanking 1 or more lines of text into a general purpose buffer; or specifically defining which line numbers or sections of contiguous marked text are to be copied, or cut and pasted. In copying the original text remains where it was, and a copy of it goes into the buffer; but cutting removes the text from its original position, and puts it into the buffer.

4.1.5 Moving The Cursor

- moving around the text by changing the cursor position - using moving commands/mechanisms: mouse movements, cursor control keys; operation of file, page, line or screen related control keys (such as go to start or end of file, up or down ½ page or screen, go to line 200 etc.); or
- using find/search mechanisms to enable movement to an area of text which contains the search string; or to confirm or refute the hypothesis regarding the existence of such a search string (eg. if search string "xxx" cannot be found then variable "xxx" does not

exist). A critical aspect involves determining the position from which the search is to commence ie. at the top of the file, or at the current position, and whether the search is to be case sensitive or not; or

- using search and moving commands in combination - either to view a section of code in the current window or outside it. Text reviewing can be used to check that the previous operation (eg. text copying or deletion) had the desired result, if not then another cycle of activities will have to be set into motion to rectify this.

4.2 Comparing the Editing Tools of MacPascal & Unix's Vi

The editing tools of MacPascal and those associated with Unix's Vi editor will be compared with regard to the mutative editing actions. To see what variations exist, and what effect, if any, they have on the programmer and the programming task.

Unix's Vi was built by programmers for programmers. So it should exemplify a good, possibly complete, set of programming tools. Although it takes most people some time to get used to, mainly because there are no visible hints, in the form of menu labels. Vi has a terse user interface^{*}, and depends mainly on the programmer being able to remember its commands, and how to combine them^{**}.

MacPascal, on the other hand, has a graphical WIMP interface, which novices find relatively easy to learn. Menu labels at the top of the screen cue the programmer as to where to look for each tool.

4.2.1 Allocation of Mutative Editing Functions

MacPascal's tools are allocated to the editing functions as follows :-

Insert: paste, clipboard, and keyboard text entry;

Delete: cut, replace (using a "null-string" argument as replacement), and keyboard deletion functions (such as backspace);

Modify: cut, paste, replace, clipboard, and keyboard text entry and deletion functions;

Moving text: copy or cut, and paste; in addition to using cursor moving commands; and

^{*} However, a help function is available. Pressing the "?" key prompts the system to present a very brief description of each command and what it does.

^{**} Several interfaces have been built to shield the user from Unix's brusqueness, and to make commands (and their parameters) easier to learn and combine: namely, SUSI (Jerrams-Smith 1985), Unicon (Gittins, Winder & Bez 1984), and Omni (Arthur & Comer 1987).

Vi's **mutative** commands are: change, copy, move, and delete. These commands are a little more complex than MacPascal's, and are based on 3 text views: line numbers; "grammar units" (using symbols representing word, line, sentence, paragraph, to end of file) relative to the cursor position; and marked text. Text that is referenced by line numbers or has been marked can be moved, copied, or deleted; whereas "grammar unit" text can only be changed or deleted. For example, typing "dd" followed by "c3)" deletes the current line, and then changes the following 3 sentences.

Thus, of all 4 commands, only delete works on all 3 text views. Vi's main editing functions, based on line numbers, are invoked using the command mode ":" prompt. However, the scope of the commands based on line numbers is increased by the use of the current line symbol "." that modifies text relative to the current line. For example, ":d,.,+5" will delete the current line, and the following 5 lines.

4.2.2 Comparing How MacPascal & Vi Operate

MacPascal and Vi cover all the basic editing functions, but they are not identical regarding flexibility, or power and subtlety in the range of actions or effects that can be achieved. Basic cursor movements are similar, depending on keyboard based cursor or screen scrolling functions (plus mouse movement for MacPascal), or by locative (search) mechanisms.

Text Entry & Moving Mode

Text entry in MacPascal and Vi means typing in program text, character by character, using backspace to delete the odd unwanted character as it occurs. In MacPascal, control remains in text entry or moving mode all the time; except when one of the tools is invoked using the mouse or control codes. After using a tool, control returns to text entry or moving mode. However, with Vi, only the cursor controls and screen scrolling functions are immediately available. Insertion (or overwrite) mode has to be invoked (and subsequently exited) using an appropriate key press^{*}. Vi's insert mode adds characters to existing text at the cursor position. Overwrite mode deletes one old character for each new character that is entered.

Cut & Copy

In MacPascal, areas to be cut or copied are defined by marking the start and end locations by mouse or cursor movements^{**}, and then activating the cutting or copying operation. With Vi, cut or copy can be achieved using line numbers or grammar unit (word, line, sentence, or paragraph) symbols to define the relevant section of text with respect to the cursor's current position.

^{*}. Using the "Escape" code followed by: "i" or "I" for inserting; or "a" or "A" for appending text. To exit insert or append mode "Control-Z" has to be activated.

^{**} All text within these bounds is put in inverse video, so that the programmer has clear evidence that the text has been marked, and where the start and end markers are.

There are other variations in the operation of the copying or pasting buffer. In MacPascal (as with most editing buffers), the previous contents are erased and replaced by the new contents. But with Vi, the programmer can choose to append the new section of text onto the existing buffer contents instead of erasing them. This enables previously separate sections of text to be combined into a larger single contiguous body of text. For example, accumulating an index from separate section headings.

Locative Commands/Tools

Vi and MacPascal both provide find/search mechanisms. The location of the search string is indicated by the cursor flashing on the first character of the search string within the text. Vi uses `/search string/` for forward search, and `?search string?` for backward search. In contrast, MacPascal only provides forward search; although it has options for case sensitivity, and whether to search for whole or partial words. The latter options are absent from both VAX and XENIX versions of Vi.

4.2.3 Summary of Editing Tool Usage

In general, the add, delete, modify, and move text commands/tools are used for text composition and modification. While the cursor moving and search mechanisms are used while reading or scanning text for the various purposes of comprehension and debugging (Davies 1989; Robertson, Davis, Okabe & Fitz-Randolph 1990). Search commands can be used to follow a variable trail through the code to gather information on its function and use in the code.

Vi's search and replace commands are more comprehensive than most, since they include symbols that define the start and end of lines. This makes them much more useful as it allows text to be added or deleted from the start or ends of lines which is impossible otherwise. Vi's wildcard attributes also extend the flexibility of the search and replace tools, by matching one (or more) characters that prefix or postfix the actual/specific characters defined in the search string.

As well as providing forward or backward search, Vi also has a confirm option which can be applied to the find/replace command. This allows the programmer to check that only relevant strings are swapped. One very useful feature of Vi is the "undo" command that restores the text state as it was before the last command was carried out. This helps to remedy many "action slips" (defined by Green 1989), where the programmer specifies and executes the wrong command instead of the intended one.

There are no specific debugging tools available within Vi itself. Debugging tools are generally geared towards the analysis of test data (and setting breakpoints) rather than assisting with conceptual debugging. However, MacPascal's Step, Step-Step, and Observe tools are geared to support/automate hand simulation. Step and Step-Step

execute the code line by line at the user's required speed; whereas Observe reports the values of variables that the user has selected for investigation.

4.3 Reprise of Programming Tasks From A Programmer's Viewpoint

§2.2 covered the programming task mainly from a process viewpoint. In this section it will be covered from the programmer's viewpoint as a practitioner. The former dealt with what happens, the latter will attempt to explain the why and/or how of real life software design.

4.3.1 Programming & Editing Shortcuts

Most programmers (including the author) have a penchant for taking a chunk of existing code, moving it to a new location, and then editing it into the required form. Either wholesale, using an entire program as the base material to produce a variation on the original source code (Détienne 1990); or on a smaller scale, say a variation on a existing procedure or block. In the either case, the search/replace mechanism can be used to replace old variable names with new variable names appropriate to the purpose of the new segment. The copy function can also be used to reproduce partial templates or programming plans. For example, to duplicate a skeleton case selector block (compound statement) "n" times, to reduce the time needed to produce "n" different individual templates having common features. Each individual template can then be filled with its own specific details.

There are several other short cuts that the programmer can use :-

1. Importing an external file holding ready prepared procedure and function outline templates; or any other significant code skeletons that speed up code production.
2. Instigating the pairing of "reserved word bracketing" templates. For example, always on producing "begin-end" loops, write the "begin" and "end" on separate lines (with the first character of each word at the same column on screen) and then fill in the middle. This ensures that there is an "end" for each "begin" and that they have the same indentation. The same theory applies to the positioning of other pairs or multiples, such as, "repeat-until" loops, "if-then", and "if-then-else" statements. This method has the added benefit of reminding the programmer of empty parts that are to be filled in later, and ensures that component parts have the same indentation.
3. Using auto-indent inside the Vi editor to bring the (first character of the) current line to the previous line's last level of indentation, to ensure that indenting occurs at the required position. Thus making the pairing process much easier and faster.
4. Automating as many repetitive actions as possible using macros or shell scripts. For example, setting up macros to compile code and then link all required libraries together to produce executable code, and, if necessary, to put new executable code in a specific

directory (Berlin 1993). The aim is to avoid unnecessary typing, to eliminate time wasting typos, and to maintain consistency. A correct macro will run properly each time; whereas a hand inputted one - typed in anew each time - is prone to errors.

NB. Since MacPascal formats text automatically, the alignment of columns of code is done without further effort from the programmer - thus saving time, although the pairing tactic is still useful.

4.3.2 Software Design in the Real World

Each software project derives from the task or system requirements. These are usually expressed in the specification document. Written in good old ambiguous English. Varying anywhere between a very vague idea of what is needed, to very specific details - such as descriptions or mock-ups of screen layouts to be used in the end product.

However, as Visser 1987, Détienne (1990), Visser & Hoc (1990), Green (1990a), and Guindon (1990) note, the specification is rarely complete, and much detailed information has to be unearthed, clarified and interpreted by the designer in order to understand the true extent of the required system, and its idiosyncracies.

Understanding the Problem - Comprehension

As stated in Chapter 1, I spent 3 years programming commercially, designing and developing Process Control (Pascal) software. Each new project started with a brief (usually verbal) description of the proposed system from the Project Engineer; followed by a discussion of what was wanted, and how it was expected to work in very general terms. After receiving a "system" specification, the first thing to do was to read it thoroughly. Making notes as to the main points of the system and grouping items together generically. With some specifications the programmer has to hunt through the documents, deciding which pieces of information need to be grouped together.

The primary task is understanding all the parts of the system, how they connect, and how they are expected to operate individually and then as part of the integrated system. This constitutes what Pennington (1987) refers to as the domain model. The next task is to design the software to fit the task. Understanding the problem (via the domain model) and designing a software solution (building a program or application model) progress until convergence - until the design solution covers the design problem to the required depth. Understanding the main function(s) of the system in terms of both models, and being able to cross-reference between them is crucial to the design. Any faults in either of the models, or in the cross referencing between them, means that the programmer will not be able to translate the problem into the appropriate design terms, or vice versa.

Programming Practice - Coding

Lammers (1986) has found 2 distinct poles on the programming in practice continuum. Those programmers who write out paper designs first (whether in flow chart, Jackson, Yourdon, Warnier form) and who then translate the design into code on paper or screen; and those programmers who evolve the design of the software and the corresponding code simultaneously at the terminal. Green 1990a refers to these as the "neat" and "scruffy" ends of the programming poles. However, most software design falls into a region between the 2 poles. Where the programmer works out the central algorithms and data structures of the problem on paper (the software skeleton), which is then transferred to the computer and modified until it fulfils the requirements. The data of chapter 3, on the design development method chosen by student programmers, confirms that software usually follows an opportunistic and iterative design method.

One of the main reasons I prefer the latter method is that, after a while spent designing on paper, I find myself continually copying out whole chunks of code, just to insert a few more lines of code in the right places. And, after the second set of rewrites I get very frustrated, because with every hand copying operation there is the chance of losing or inadvertently modifying code and introducing bugs. So that the code will not work as intended. It is much easier, to my mind, to get a printout of existing code and hand edit it, then transfer it into the computer via screen editing, and then to print a new copy, than it is to keep producing hand copied versions, which tend towards illegibility after a while (due to hasty copying etc.).

4.3.3 Summary of Typical Design/Coding Strategy

For programs of less than 20 pages, or 1200 lines, of code; the iterative software design method runs as follows (as observed with student programmers of chapter 3, and consistent with personal experience):-

1. Mainly mental operations: reading the task description or specification thoroughly, making written notes en route, to get the main thrust of the requirements. Any ambiguous or conflicting requirements should be noted and resolved by consulting an authority on those particular details. Guindon (1990) confirms these essential fact finding phases of the software design task.
2. Externalising the internal representation using pen and paper: writing down the central and sub-algorithms, and all associated (obvious links) in English, semi or full Pascal, pseudo-code or some other notation. Each component of the program, module or procedure, is designed, in turn, by continuously decomposing and elaborating design goals, whilst keeping in mind any interactions with existing (or proposed) modules or procedures (Green, Bellamy & Parker 1987).

3. Editing at terminal: transferring the central and sub-plans into program form and filling in any missing programming plan or language details as needed.
4. Design verification and evaluation: checking for congruence between the specification, the mental representation of the code, and the code itself; with the aim of resolving any inconsistencies. Mental simulation is used initially (during the parsing-gnirap cycle of composition) to check that all design elements of each individual programming plan are in place and that they work as intended. See §4.3.5 for a full breakdown of the checks. As code grows, due to the addition of all the implementation details needed to flesh out each tactical plan, so must the corresponding mental simulations required to check it. However, due to working memory limitations, these simulations may only cover the "essential" parts of a plan rather than the whole plan (Guindon 1990).
5. The next step is to test thoroughly. Firstly with expected data - if this passes without a hitch, then invalid data is run through. To verify that adequate and appropriate error response or prevention is in place and working as expected.
6. Debugging: any design faults or bugs found in previous stages are eliminated.

In more general terms, these stages can be defined as: problem understanding; design composition; coding; design verification by mental simulation during composition; thorough testing; and debugging.

4.3.4 Summary of Design Stages

Thus, on one level we have : design specification —> translation —> code

On another level we must verify the correctness and veracity of the translation

On yet another level we must verify the design's correctness and suitability - this means that the software and its outputs must conform to both the design specification and the user's requirements. If there is a conflict between design specification and user requirements, then the latter should take precedence, since the design specification is supposed to be a distillation of user requirements. If there is a discrepancy, then the design specification has failed to capture the user's real requirements fully, or sufficiently accurately. Meeting the user's needs and achieving user satisfaction is more important than blind and unswerving conformance to the design specification.

4.3.5 Pre-execution Actions

After the program runs through the compiler without errors, it is usually a good idea to make a final run through of the following checks before starting testing. Many checks are made during composition to make sure that all the elements of each plan are present, but it is best to double check and eliminate obvious bugs before testing.

- variable declaration checking - checking that each variable has been assigned to an appropriate data type and data range;
- variable scope checking - making sure that local and global variables are used correctly, and within the bounds associated with the original data types;
- checking for correct initialisation and re-initialisation of variable values;
- checking that all variables are kept within valid ranges according to the specifications;
- checking the parameters of procedure and function calls against their declared (formal) parameters to see that they match - type-wise and activation-wise;
- checking that "var" and non-"var" parameters have been assigned correctly within each procedure call and appropriate to the procedure's parameter declaration;
- checking that values are passed back through "var" parameters to the appropriate variable; and that the correct non-"var" variable has been passed to the procedure call; and
- checking that the order of variables in the parameter list match up properly. For example, should order be `calc_range(min, max)` or `calc_range(max, min)`?

Chapter 5 Discussion of Possible Tools

Looking through the list of enhancements and programming problems in §3.2 (and the checks of §4.3.5), checking aids seem to be in greatest demand. Mostly to reduce syntax and semantic errors at the coding stage, and runtime errors later on.

5.1 Programming Problems and Possible Tools

The following list is a distillation of those problems that need to be addressed:

- invisible control characters;
- incorrect or non-initialisation of variables before use;
- incorrect sequencing of variable value changes;
- infinite or redundant loops, due to incorrect initiating and/or terminating conditions;
- eliminating unused variables, procedures or functions;
- undeclared variables;
- incorrectly specified procedure calls arising from defective parameter lists (too few or too many variables, or mis-ordering of variables in parameter lists);
- applying the wrong or inappropriate function to a variable;
- type mismatch between left and right sides of an assignment statement;
- placement of brackets to get correct interpretation of an expression; and
- missing or mismatched bracket problems, due to one of (,), [], { or } going astray.

As stated in Chapter 3, the invisible control characters are a problem because you can't see them. If you could make them visible using something like inverse video they wouldn't be such a problem, because you could eliminate them straight away.

It is an accepted fact that most programmers debug their software by applying pen (and mind) to paper text (Eisenstadt 1993). One particular debugging technique is trail following. This requires each occurrence of one (or more) specific variable(s) "names" to be highlighted through either a specific section of text, or throughout the entire text; so that its trail is easy to follow. A forward (top-bottom) search is usually taken to mark each occurrence of the suspect variable. Using a forward search also helps the programmer to gather control flow and data flow information pertaining to that variable. Whereas back tracking helps the programmer to take note of the "most recent" variable values that had an effect on the suspect variable's value or state. Thus forward search establishes locations, and backward search establishes prior contacts with other variables.

An editor's search mechanisms can also be used to follow the trail of a variable. One aspect of the search mechanism is to "find" the required string, to locate it within the text. Another is to be able to "see" it within that text, to be able to distinguish it from the surrounding text, yet to be able to see it in context. The former is a location task; while the latter aspect concerns visibility (and discriminability Green 1989). Thus the search mechanisms bind identity, spatial location, and context, using a visual marker.

For most search mechanisms this visual marker is unique - viz the cursor position - and only applies to one instance of the search string at a time. The cursor blinks on the first character of the search string, but the rest of the screen remains visually "bland" and unchanged. Once the cursor moves to the next search string instance, you lose the visually indicated position of the previous one. This is the main reason why programmers have to resort to marking up a variable's trail manually, using visual inspection on a printout. A laborious, tiring and tedious job where it is easy to miss one or more instance's of the variable the first time round; and all because there is no such facility provided by the programming environment.

One of the most common syntax problems that most programmers have is the undeclared variables problem. The main problem with this is that you can't get a list of the undeclared variables, and it is difficult to remember them unless you write them down. What you are really looking for is a list of them so that you can refer to them easily without having to write it down yourself. Thus the main problems with undeclared variables are:

- getting a list in the first place (difficult using visual scanning alone, and even compiler messages do not list the required information in its most useful format); and
- seeing that all undeclared elements are accounted for - either they get declared, or they get reassigned (misspellings/typos), or they get scrubbed (redundant variables).

Providing an easy means of transferring an undeclared variables list to the appropriate declaration area (either in part or in whole) would probably speed the whole process up. So, how about a tool which generated an undeclared list for each procedure, that could then be transferred into the appropriate declaration area (at the user's request), to ensure that all undeclared items get transferred, and have a good chance of being assigned.

One of the enhancements suggested in §3.2 was user control of text formatting. This presents the problem of combining automatic text layout and catering for a range of different layout styles, so that each user gets fast layout of text in a familiar style. Defining better screen moving commands and search facilities, and the use of colour on screen, are all good ideas, but they need further thought.

My "first-thought" solutions to these problems are as follows:

Layout style determiner/generator:

Display different versions of each construct's layout, and correlate each style to an option number. The user selects a layout style for each construct in the programming language, resulting in a matrix of selected options, that define the preferred layout style for that language. For code entry: either code is set in directly according to the user's manual layout (as the user enters text, as with MacPascal); or templates are selected and filled in on screen.

Autocompletion & Templates:

To provide a terminating symbol to match the current symbol eg. to provide a ")" for each "(" generated. This tool could also produce an "until" clause at the same level of indentation, every time a "repeat" was typed in; or a "repeat-until" template could be called up on a window, filled in, and then transferred into its final position in the program text.

Visibility Aids: To provide a means of highlighting all instances of a given word, phrase, or symbol (or illegal control codes).

Procedure & function declaration checker:

To show the names of all built-in or user defined procedures and functions via a menu window; selecting a given name would cause the associated parameter list to be shown in a child window. Access to the names and parameter lists of the built-in procedures and functions would act as an on-line reference for users, so they wouldn't have to waste time going through the manuals (or re-inventing the required code from scratch) - providing there was a brief description of the effects of each procedure or function.

"Variables" declaration checker:

A menu system could list "variables" either in order of declaration precedence or alphabetically, with an option on global or local "variables". "Variables" could be partitioned into data types: real, integer, char, boolean, and array declarations. That way it might be possible to associate another file with each partition of "variables" which could indicate the functions or operators that can be applied; thus detecting "illegal" operators.

"Variables" could include "const", "type" and "var" declarations, or each different type of declaration could be considered separately.

An "undeclared variables" list could be produced as well. This would contain all "words" that are not already declared either locally or globally within the scope of the parent procedure or function, or as a reserved word, or the name of a procedure or function.

"Statement" checker:

To check that applied functions and variable types match; a pre-compiler checker.

For example, checking that real values were not assigned to integers unless a "trunc" or "round" function had been applied to the real part of the expression beforehand.

Also to check that the left and righthand sides of the statement were data type compatible. So that a real value was not assigned to an integer variable for example.

Loop checker:

To check that loop variables, values, and conditions are specified correctly - using the right type of variable, and checking that the loop interval values are in the right order. For example, a "for" loop can use an integer or "char" typed variable, but not a real.

5.2 Choice of Tools For Further Discussion and Development

Current software development systems provide little in the way of decision and perception aids. It seems that there are 2 types of decision aids:

- those that help you to pick out or emphasize certain facts; and
- those that present facts from a variety of perspectives, thus "simplifying" the interpretation task.

Applying inverse video to the search mechanism would certainly act as a visualisation aid, and would satisfy the former criteria. Whilst displaying the declaration data as a set of alternative summary menus - and thus acting as an interpretation aid - would facilitate comparison and interrogation of declaration data.

Reading through the previous section, the tools that need developing fall into 5 categories:

- layout aids;
- interpretation aids;
- visibility aids;
- moving aids; and
- checking aids

Combining visibility and moving aids should produce enhanced browsing and user control of visiting areas. This should help with the various reading, reviewing, scanning and navigating tasks described by Baecker & Marcus (1990) as being involved in the program understanding task.

5.2.1 Layout Aids

The students' comments from the questionnaire showed varying views on automatic layout of code. From those who didn't mind, and were happy to give up the task of laying out text; to those who found it difficult to relate to the code because the layout style was different from their own. Some of the latter group stated quite plainly that the difference in format reduced the code's readability and comprehensibility, and made debugging more difficult.

A programming colleague of mine once took it upon himself to reconfigure an entire program of my code in his own style. His excuse was that some modifications had been required, and he had done them in my absence. When I read the code to find out what alterations he had done, I found it incredibly difficult to re-establish my usual rapport with the code. My mental representation of the code was not congruent/connected to the new shape and style of layout. I could no longer identify with the code, and lost my bearings easily, even though the program was one that I was very familiar with prior to its "cosmetic surgery".

I have asked other programmer's of their experiences with code in layout styles different to their own, and the majority find it difficult to understand and work with such code if the style is significantly different. Most programmers can cope with minor differences, such as having indentation set at 4 spaces per indent level instead of 2.

The interesting question is "Why is layout so important to designers?" Perhaps because:

- it maximizes readability and comprehensibility for each user;
- it maximizes rapport and personal identification with the code;
- it enables placement and (enhanced) memorability of cognitive markers: and provides a visual pattern, a route for navigation along the paths of software control.

Wiedenbeck (1986a) calls these cognitive markers, beacons. There have been some attempts to emphasize indentation, such as van Laar's (1989) colour coded support tool. But on the whole the differences between individual layout styles has been ignored.

The reason for designing a layout aid is to enable each user to maintain his own layout style, without having standards imposed from above. Where each user selects his own personal configuration style for each of the constructs from a (pre-defined) matrix of layout styles. Notice that this is a reversal of the pretty print standardisation ideology!

By using the layout matrix it should be possible to reinterpret and reformat other people's code into your own personal style, which would make comprehension of unfamiliar code much easier and more straightforward (due to enhanced readability as a result of stylistic rapport), more efficient, less time consuming, and hopefully less prone to errors. This should be particularly useful to maintenance programmers, where intimate knowledge of the code is essential, so that the required modifications can be made without introducing inadvertent bugs. Of course, code could be reformatted back to its previous style when the modifications had been completed, and all testing had proved it free of errors.

5.2.2 Interpretation Aids - Summary Menus

Current systems provide minimal direct comparison of any significance. Typical examples are Suntools and Smalltalk, where the programmer can select copies of one or more (different) sections of code to be put in separate windows. So that they can be compared with each other, or the current window's contents. None of the systems I have seen offer any kind of summary/comparison data in a format which is immediately useful to the programmer. It is always up to the programmer to locate the required data, manipulate and then interpret it into the desired information, in

order to finalise decisions. For example, to find out whether a variable that has been assigned a value, is a local or parametric variable; you need to check the associated declaration area. What you find there determines your next course of action. If it is a local variable or formal/value parameter variable then it (usually) doesn't matter. But if it is an actual/variable parameter variable then the assigned value will be written to the address held by the variable named in the procedure call.

So, the idea behind this tool is to provide alternative means for showing declaration data so that the correct "perspective view" will enable the programmer to gain the required information in the easiest, simplest, most "consumable" form. Its purpose is to reduce the time and effort, and the distraction of switching between declaration areas to confirm format and/or content of procedure/function parameter lists; or checking various declaration lists to ensure compatibility with declared data structure characteristics and/or applied functions or operators. It should also be possible to create a menu for each procedure/function that lists all unknown or undeclared words. This should help considerably with the undeclared variables problem. By being able to compare the undeclared words and declared variable lists, you should be able to sort out which undeclared words are simple misspellings, and those that need to be assigned to a data type.

The other principle problem is getting procedure/function call parameter lists specified correctly. Having a menu that shows the original parameter list declaration for the chosen procedure/function should practically eliminate this problem.

Main menus needed:

- user procedure/function name list, with parameter declaration list as child for each;
- pre-defined procedure/function name list, with parameter declaration list as child for each;
- variable declaration list for each procedure/function (parameters + local + undeclared list);
- alphabetic variable declaration list irrespective of data type class (with pointers to "parent" procedure/function, and data type class);
- alphabetically sorted data type class variable declaration list (with pointers to "parent" procedure/function).

5.2.3 Visibility Aids - Spotlighting

The purpose of a visibility aid is to apply visual distinction to important information, so that it stands out from the background text, but remains in context. Typography uses various effects to achieve this visual distinction: blocking and spacing; using boxes of different sizes or with different borders; using italics, boldening, inverse video and

colour; or by changing the size of the font - as in newspapers. At present most programming is done on monochrome VDU screens, although WIMP (Windows, Icons, Menus, Pointer) systems and colour screens are gaining ground fast. Even so, few systems have been designed to make use of any typographic effects in the programming environment - where they would do most good. MacPascal uses emboldening to distinguish reserved words, but that is as far as it goes.

It seems strange that even such simple, but powerful effects as inverse video have not been put to better use in an editing context. Although, Unix's vi editor does use a blinking cursor to indicate the position of a search string when "finding".

The primary idea is to extend the search mechanism by making all instances of the search string stand out, using a strong visual emphasis, such as inverse video or colour, to provide an attention spotlight. This spotlighting tool should make following a variable's trail much easier, and less wearing on the visual perception and cognitive processing front; since it basically eliminates the "find location" side of the task.

At present the only means of finding all locations of a given variable is:

- (a) by using a feature like Unix's "grep" to show all such lines; or
- (b) by using a find utility to jump from one occurrence to the next.

The problem with solution (a) is that you only get those lines which include the variable - so they are on their own, isolated and out of context. The problem with (b) is that you can usually only travel in one direction (top to bottom) and there is no way of marking each individual occurrence of that variable, hence you cannot easily locate the positions of the last and next locations, only the current position is obvious.

With variable trail following, you are following the data flow of that variable, and examining each operation that affects that variable, in order to detect and eliminate a bug associated with that variable. With normal text editing you have to rely on being able to spot all occurrences of a given identifier, to remember their locations and the pattern of the data flow resulting from each combination of successive individual relationships. If you could immediately spot each occurrence this would reduce the amount of short term memory and attention needed to pick out each occurrence every time you needed to look at it.

Spotlighting could be implemented in inverse video, colour, emboldening, italicising, underlining or even by change of font size. If colour or inverse video were chosen, you could spotlight 2 or more different search strings (multi-spotlighting), and see where they interacted. For such an apparently simple and obvious innovation - applying the highlighter pens concept to electronic text - the effects could be extremely powerful and useful in many different ways. I am particularly interested in applying it to the programming context, so that it can be of use in various debugging tasks. But it could be very useful with other forms of electronic text - especially large documents, perhaps even system specifications! I well remember having to wade

through a hardcopy specification of 100+ pages, trying to find all the alarm conditions associated with the system, since they were scattered throughout the document.

In a programming environment, applying the spotlighting tool to an entity (a search string, or phrase, or perhaps even an expression) would:

- enable easy tracking of an entity throughout the code;
- make visible all statements involving the same entity;
- increase the probability of incorrect sequencing of initializing or terminating statements involving that entity being seen, and corrected;
- enable cycling through a sequence of undeclared variable names (spotlighting each one in turn) in order to spot usage characteristics within the code, and hence deduce prospective data structure; and
- with a colour screen, multi-spotlighting could prove very useful in pointing up the relationships between different spotlighted entities.

Applications:

- for programming: primarily emphasizing the location of a given identifier. Multi spotlighting could be done using either a different colour for each identifier or by lighting a sequence of identifiers in turn (one at a time);
- for spreadsheets: to emphasize all equations that used a particular cell value;
- for electronic documents in general - providing the electronic equivalent of highlighter pens, with the advantage of being able to produce a hardcopy at any time, that shows the spotlighted words in context.

The primary idea is to apply spotlighting to a search string using the familiar search mechanism technique. But this idea can be extended to more esoteric problems. Like the invisible control characters, that gave me the idea in the first place. It could also be applied to the missing/mismatched bracket symbols problem, by setting inverse video on, on encountering a lefthand bracket, and then turning inverse video off, on encountering the next righthand bracket. This would be invaluable for catching those missing comment brackets, "}", which cause the compiler to ignore all code or text appearing between successive "{" and "}" symbols. It may also help keep the number of parentheses, "(" and ")", equal in complex expressions, and assignment statements.

Even this only points up the versatility of one chosen type of visibility aid. There are certainly more, this is only one instance of it - the tip of the iceberg. Harnessing different typographic aspects to tools on VDU screens under user control will surely provide many more powerful tools. Windowing systems are one aspect of this. Where each window is a bordered typographic box containing text, that users have been conditioned to think of as the electronic equivalent of pieces of paper. Where icons represent actions or objects, such as the filing cabinet or the rubbish bin.

5.2.4 Moving Aids

Robertson, Davis, Okabe & Fitz-Randolph (1990) have shown evidence of the programmer's need for easy forwards and backwards movement through program text, during the program reading/comprehension phase, which are attributed to 6 categories of programmer motivation - assume, question, answer, analyse, function and strategy. There are 4 basic movements forward-forward (thus continuing with forward motion), backward-backward, and the switch movements (when changing direction) either forward-backward or backward-forward. This means that design must be geared to extend the variety of existing moving commands to fit in better with programming habits and task characteristics.

The 2 principal reasons/motivations for movement were:-

- forward-forward - function & assume;
- backward-backward - question & strategy;
- forward-backward - strategy & function; and
- backward-forward - function & assume.

Davies (1989) has identified several aspects of programmer behaviour - most significantly the frequency with which inter- or intra-plan jumps are made. These results are corroborated by Green, Bellamy & Parker's* research and indicates that moving aids should extend the variety of existing moving commands to fit in better with known programming habits/task characteristics. In short, to enable:

- free movement, forwards or backwards, between points that are significant to the programmer without arbitrary restriction; and
- immediate transit between points of interest to enable relevant code review and/or tuning of software in light of current design decisions (ie. to ensure compatibility).

For example:

- to move relative to the screen (visual page) size eg. up or down one screen or more;
- implementation of markers to enable flipping between alternate sections of code;
- to go to definite areas within a procedure boundary such as the declaration area, first/last line of code, line N of a procedure/function; and
- to go directly to the start or end of a file, or to line N of file (assuming definition of line numbers is available on user's request).

Also, being able to move between spotlighted entities would be very useful. The role of moving aids is essential to the programming task as it supports the program comprehension tasks defined by Robertson et al. Thus freedom to move around the code is paramount.

* Green, Bellamy & Parker (1987) found that Pascal programmers tend to traverse their programs more often than Basic or Prolog programmers.

5.2.5 Conclusions

There are still not enough checking aids provided by the compiler, but I am more interested in developing tools that aid decision making. As it is more useful to make relevant information more visible, and to provide alternate interpretations of information, in order to eliminate errors at source, and to reduce the overall perceptual and cognitive effort needed to extract relevant information. After all, picking out the relevant information is the first step in decision making; particularly when it comes to debugging.

Most errors arise through a combination of interrupted tasks, carelessness, inattentiveness and forgetfulness. The earlier they are eliminated the better, since the cost in terms of time and effort increase geometrically the later the bug is discovered.

According to Baecker & Marcus (1990) "*Program visualisation* is the use of graphics (including typography, graphic design, ..., and interactive computer graphics) ... to facilitate the development, understanding, and effective use of computer programs by people." Current program visualisation is inadequate at present, and needs to be extended further and more fully, to get the best effect. Beginning with the more extensive, but considered, application of typography and visual enhancement to computer-aided text processing tasks, in a decision aiding capacity. These tools should speed up error correction and detection during the code development stage, where bug costs are lowest.

Chapter 6 Design & Application of Proposed Tools

This chapter is derived from a paper (Humphreys 1992) presented to the Psychology of Programming (Special) Interest Group Workshop (PPIG-4) held at Loughborough University from 2-4/1/92. It contains a discussion of the conceptual design of each tool, its relevance to the area of application - and examples of what the tools would produce viz the visual effects (on the program text or user interface), and how it could be used; including outlines of proposed implementation or implementation issues.

An application was made to LightSpeed Pascal and other software providers, for access to the source code, outlining the nature and applicability of the tools and the necessity for full integration with an existing environment. However, no software was forthcoming, so a working, fully integrated implementation was out of the question. The alternative was to produce paper designs, and demonstration art-work to simulate how the tools would look, and how they could be used. This helped to focus conceptual design on the primary features of each tool, and the essential characteristics that would make them useful in the programming tasks.

Examples of layout variations follow the initial section on layout aids. The remainder of the chapter is devoted to the spotlighting and summary aids.

The spotlighting and summary table tools have been applied to Siddiqi's Signal problem, as a means of showing their essential characteristics and features; and how they can be used. A complete, commented program listing of the solution to Siddiqi's Signal problem, plus the full statement of that problem appears in §6.2 (near the beginning). This particular problem was chosen because it has a short, simple solution that only required a few variables, but is long enough for the purposes of effective demonstration of both tools.

Note that:

All example program listings are in Pascal, but all tools should be applicable to other procedural languages as well.

The terms summary tables and summary menus are used synonymously.

For brevity, all references to procedures will be regarded as referring equally to functions, on issues such as parameter calls, and the scoping of variables etc. Any specific details regarding the data type of functions will be discussed separately.

6.1 Conceptual Design & Tool Relevance

The central concepts behind the tools are:

- Using typographic effects to (attract and) focus visual attention and to support those programming tasks dependent on visual processing (spotlighting).
- Reducing information processing burdens by providing essential information in alternative formats (summary tables/menus).
- Supporting individual aesthetic requirements and promoting visual rapport between the programmer and the program code being developed or debugged (layout aids).

6.1.1 Layout Aids

The idea of this tool is to enable any program to be laid out according to the programmer's own particular preferences. Thus enhancing the visual and mental rapport with that program, and making it easier to understand.

The main layout factors are indentation and the relative disposition of the programming constructs. The indentation has quite a drastic effect on the code. If you indent by 1 space each time (for each new block level) then of course your code is going to remain much closer to the lefthand edge, whereas, if you indent by 4 spaces each level, then you're soon going to reach the righthand edge of the page, and you are going to run over lines much more quickly. Most programmers seem to prefer 2 spaces per indentation level.

The layout aid is intended to format program text as you type it in or modify it, or alternately to reformat someone else's code that you have to work on. Observations and the results of the questionnaire (in chapter 3) show that people have their own different approaches to the layout of the code. Having a particular piece of program text in your own layout provides the mental rapport which you really need to get a grip on the code itself. In effect it is your own personal interface to it. If you have to work on a program text with a different layout style to that which you're used to; then it is that much more difficult to get into the feel and flow of the program - it seems to interfere with the comprehension (program understanding) process.

Before the tool can be used for the first time, it will have to be set up to handle all the different variables that can be applied to program text. Such as the indentation, and a matrix of choices to define how each of the different program constructs are to be laid out (see below).

The basic procedure will be to go through a series of layout options, selecting the preferred layout for each individual construct. Resulting in a matrix of choices defining the preferred layout for each construct in the programming language. These definitions can then be used to lay code out in your own style as you enter or edit it.

So that it is immediately acceptable to you, and more amenable for your mind to get a grip on it. Once the matrix of choices has been selected it can be kept for future reference by the layout tool, and the selection process need only occur once.

There is a "usual/normal" spatial disposition for each construct's sub-components - the preferred layout - but there may also be exceptions under which a different spatial disposition is used. For example, when long conditions or expressions cause a statement to exceed one line in length. Thus the matrix of layout choices must also cater for those situations. The combination of these choices determines the shape of the code that will be produced. The indentation controls the lateral shift of the visual shape, and the disposition of constructs affects the width and length taken up by each successive construct.

Gray & Anderson (1987) found that programmers make stylistic changes to code as well as code corrections during change-episodes. This shows the importance of people being able to work on code in their own style and the importance of being able to convert from one layout style to another. The reason behind the layout aid is to promote transformations between layout styles. For example, you could have a piece of code in your own style, then transform it into the house style and then at some later date, someone who wants to modify that program, or just check up on it for maintenance purposes can format it to his or her own style with no detrimental effects to the code at all, and of course making it that much easier for the individual programmer to work on it.

There are some aspects of layout style that may be more difficult to automate:

- The use (or not) of headers is one difference. Some people produce a large comment label, about 1 inch high announcing the name of each procedure, as below. Individual aesthetics vary as to the choice of characters used to pad the label heading, and how far it extends across the page (halfway or less, or right across). But basically it is just a means of making it clear where each new procedure begins.

```
{ -----  
* procedure average  
----- }
```

- There are also individual differences concerning the placing of comments: where, when and why (Baecker & Marcus 1990; Riecken, Koenemann-Belliveau & Robertson 1991). Some people only comment particularly difficult pieces of code - they regard the rest as self-explanatory. Whereas others comment strategically: at the head of procedures and at the start of main blocks. Some carry this further, by commenting at each significant stage of the program (Gellenbeck & Cook 1991b). Not merely to say

what the code does but why, in terms of the programming plan or the effect of the code when it is in actual operation - such as, "roll-over counter value when count exceeds maximum value" or "send signal to Sample probe to take an oil sample".

- A further variation concerns the spacings between paragraphs of program text. This seems to support visual and cognitive/conceptual blocking or structuring. It is also used to delineate between different programming plans, or a significant juncture in a single or combined plan (Riecken, Koenemann-Belliveau & Robertson 1991). It may also allow a mental pause for breath, as well as acting as a cognitive marker. All these little personal preferences have an effect on how easy the code is for the programmer to get into, and grasp hold of the meaning of the code and to actually get on with developing, modifying, and debugging it.

6.1.1.1 Examples of Layout Variations

The following examples show (some of) the variations in layout for the if-then statement, if-then-else statement, if-then with compound statement, loop with compound statement, and the placement of logical operators in multi-line conditions.

- Alternate layouts for each construct of the if-then statement; where "<cond>" stands for condition, and "... " stands for the actual statement to be executed. If the IF condition and the THEN statement are both short, the first variation may be used, but the succeeding 3 variations are more common.

```
if <cond> then ...;
```

```
if <cond>      if <cond>      if <cond> then
then ...;      then ...;      ...;
```

- With an if-then-else statement, some people like to make that into 3 sections, the IF with its condition, the THEN with its statement, and the ELSE with its statement. Other people, if its only a short condition and a short THEN statement, like to put the IF condition and the THEN statement on the same line, and then have the ELSE statement below it.

```
if <cond> then  if <cond> then  if <cond>  if <cond>  if <cond>
...            ...            then ...    then ...    then ...
else           else ...;      else ...;  else ...;  else ...;
...;
```

- Another variation occurs for the **if-then statement with a begin-end loop**. Some people have a tendency to put the THEN on the end of the line following the IF condition, whereas other people put the THEN on its own line just above the BEGIN loop. Where "<cond>" stands for condition, and "...;" stands for 1 or more statements.

if <cond> then begin ...; end;	if <cond> then begin ...; end;	if <cond> then begin ...; end;	if <cond> then begin ...; end;	if <cond> then begin ...; end;
--	--	--	--	---

- **Loop with begin-end statement.** Layout acts as a graphical representation of the way the programmer sees the code control-wise. For example, some people consider the begin-end loop to be at the same level as the "calling" loop, whereas others consider the begin-end loop to be at the same level as the internal statements it contains, and yet others consider the begin-end loop to be 1 level further in, and the statements it contains to be 1 level further in than that; as shown below. This gives at least 4 possible layout variations :-

"loop" begin ...; end;	"loop" begin ...; end;	"loop" begin ...; end;	"loop" begin ...; end;
---------------------------------	---------------------------------	---------------------------------	---------------------------------

- The only other major thing with layout aids - again with conditions, is the arrangement of **complex conditions and the placement of logical operators**, occupying 2 or more lines. Some people like to put the AND or OR logical operators on the very end of the line, to make you realise that there is another part of the condition coming below that. Other people like to put the AND or the OR at the beginning of the line (this arrangement seems more apt somehow). Compare the following 2 examples, where "(.....)" stands for a complex condition:

```
((.....) OR (.....)) AND
  (.....)

((.....) OR (.....))
AND (.....)
```

These examples are only the tip of the iceberg - collecting and sorting a taxonomy of layout variations will be quite a task, in order to cover the basic set of stylistic variations. As a result of the latter consideration, I have decided to develop the other 2 tools, especially spotlighting in greater depth, due to the more novel aspects.

6.1.2 Spotlighting

If you observe a group of programmers for any length of time, then sooner or later at least one of them is going to pick up a pen and start marking up program text in order to follow a variable's trail. There are many different manual methods for doing this - using a pen to put a dash beside the appropriate lines or underlining or circling the occurrence of each variable, but fluorescent pen is one of the most useful for this, and of course it has the direct correlation on the VDU of inverse video or colour.

The reason why people use manual spotlighting (marking the trail of a specific variable), how it helps, and why they find it useful as a debugging technique probably depends upon the following features:

- the simple and straightforward marking method, gives direct visual evidence of the variable's trail;
- making the chosen variable's trail visible "in context" distinguishes those statements that are central or critical, and those that set up conditions for the main event; and
- the direct visual clues in context may facilitate parallel processing; because merely by looking at a spotlighted listing, you can usually tell where the main action is, and thus guesstimate the most fruitful location to start error prospecting from.

But manual spotlighting is slow and prone to error. The idea of spotlighting is to automate this process, and to extrapolate the search mechanisms accordingly.

The main problem with current search mechanisms regarding trail following is that the cursor position has 2 functions: focussing visual attention on the current instance of the search string; and indicating the position where mutative editing operations are enabled. But trail following requires that these functions be separable; the spotlighting concept achieves this by making all instances of the selected word situated within the current screen window visible at one glance*.

There is a fundamental importance about being able to see multiple occurrences of a variable **simultaneously** within the code, that is **more** informationally or contextually important than being able to jump between them using the search mechanisms.

The priority with trail following is seeing the trail and investigating it.

One of spotlighting's main features is that it facilitates random access to any location involving the spotlighted variable, and thus to any statement that refers to, or alter its value, **wherever** it occurs within the program text - making trail following easy.

So, the programmer can quickly pick out the best place to start looking for the bug, and individual statements in the sequencing can be checked in any order. So that the

* However, all moving, screen scrolling, and editing aspects of the cursor remain unaffected; the main difference is that the text is no longer visually bland.

programmer is not forced into a top-bottom search method which might not be suitable for the particular task at hand. Plus, you can go forwards and backwards, as long and often as you like, because you've got the indication of where to look which is most helpful, and it remains permanent for as long as required. Usually backwards jumps are a hit and miss affair because you have to estimate whether you went far enough back to reach the target destination. This means reading the code to find out how close you are. The spotlights will provide an easy target to home in on, thus avoiding the usual waste of time and mental effort on bland text.

Thus, spotlighting is a great orientation aid. Spotlighting can also be applied strategically, since the spotlights can be used as boundary markers, so that you can investigate the intermediate statements without losing your place. For example, say you spotlight variable "x" because there is a discrepancy in its expected value, and the assignment statement, "x := x + y;" has been brought to your attention by the spotlights. Then you can use the location of that spotlighted line, and the previous spotlighted line as boundaries; and proceed to check all the assignments to variable "y" within that section. Because it might be the "y" value that is contributing the erroneous value instead of the "x" value by itself.

One of the most important factors that designers of editing environments fail to take into consideration is that programmers keep a mental map of instance locations. If you don't have to remember precisely where everything is, then of course you won't need that memory being expended on that task - or at least not as much. Since spotlighting is supporting and visually reinforcing your own mental map.

Of course automatic spotlighting also avoids the "missed one" phenomenon, which you get with the manual method. Where you miss one or more instances of the chosen variable, either because there are too few or too many instances of the chosen variable (so you get mentally overloaded).

There are several different typographic effects that could be used: italicising, underlining, emboldening, and inverse video, of course. See next page for a comparison of these different effects. Even with a quick glance at the page, your attention is drawn immediately towards the inverse videoed text. None of the others grab your attention as quickly, simply because their visual effect is more subtle. An italic word isn't very easy to spot on VDUs, similarly for underlining. Emboldening does appear a little more distinct on systems like MacPascal which actually uses it for reserved words. But again that simply isn't strong enough for debugging purposes. Thus I regard inverse video as the outright winner in the spotlighting contest.

Comparing Different Typographic Effects on the Visibility of the Signal Varial

```
program survey(input, output);
var
  time, vehicles, wait, maxwait : integer;

begin
  wait := 0;
  vehicles := 0;
  read(signal);
  repeat
    if signal = 2 then
      begin
        time := time + 1;
        if wait > maxwait
          then maxwait := wait;
        wait := wait + 1;
      end;
    if signal = 1 then
      begin
        vehicles := vehicles + 1;
      end;
  until signal = 0;
  writeln('Time-span-', time, 'secs');
  writeln('Vehicle-count-', vehicles);
  writeln('Max-wait-', maxwait, 'secs');
end.
```

Italicising

```
program survey(input, output);
var
  time, vehicles, wait, maxwait : integer;

begin
  wait := 0;
  vehicles := 0;
  read(signal);
  repeat
    if signal = 2 then
      begin
        time := time + 1;
        if wait > maxwait
          then maxwait := wait;
        wait := wait + 1;
      end;
    if signal = 1 then
      begin
        vehicles := vehicles + 1;
      end;
  until signal = 0;
  writeln('Time-span-', time, 'secs');
  writeln('Vehicle-count-', vehicles);
  writeln('Max-wait-', maxwait, 'secs');
end.
```

Emboldening

```
program survey(input, output);
var
  time, vehicles, wait, maxwait : integer;

begin
  wait := 0;
  vehicles := 0;
  read(signal);
  repeat
    if signal = 2 then
      begin
        time := time + 1;
        if wait > maxwait
          then maxwait := wait;
        wait := wait + 1;
      end;
    if signal = 1 then
      begin
        vehicles := vehicles + 1;
      end;
  until signal = 0;
  writeln('Time-span-', time, 'secs');
  writeln('Vehicle-count-', vehicles);
  writeln('Max-wait-', maxwait, 'secs');
end.
```

Underlining

```
program survey(input, output);
var
  time, vehicles, wait, maxwait : integer;

begin
  wait := 0;
  vehicles := 0;
  read(signal);
  repeat
    if signal = 2 then
      begin
        time := time + 1;
        if wait > maxwait
          then maxwait := wait;
        wait := wait + 1;
      end;
    if signal = 1 then
      begin
        vehicles := vehicles + 1;
      end;
  until signal = 0;
  writeln('Time-span-', time, 'secs');
  writeln('Vehicle-count-', vehicles);
  writeln('Max-wait-', maxwait, 'secs');
end.
```

Inverse Video

Spotlighting a variable's name throughout a program text can show up many kinds of errors. For example, an undeclared error can be detected by the fact that a spotlight of the named variable does not appear in the declaration area, although it does appear elsewhere in the text. Redundant variables have the reverse effect, their spotlights appear only in the declaration area, and not in the program text since they are (usually) unused otherwise. Of course the most useful types of errors shown up by spotlighting are those that can be detected by following a variable's trail, such as: missing variable initialisation statements, or incorrect initialisation or modification of a variable's value; or sequencing errors.

For the missing variable initialisation error, all you have to do is to spotlight the required variable; then the first time you find a spotlight, will be when that variable's value is being tested, or fed into a different variable's assignment equation, or even to modify its own value. This means that an initialisation statement is needed somewhere above that first spotlight. A similar theory applies to the modification error. For example, say you have a simple calculating program that prints the value of a variable called "x". On testing, the program prints the "x" value as 8 instead of 10. Then spotlighting each occurrence of "x" might lead you to an equation where "x := x - 1;". So you might decide that the error is due to decrementing 1 instead of incrementing 1, so you need to change the "-" to a "+" sign.

Alternately, you might have used the wrong variable name in an assignment statement, or one of the constant factors might be wrong. Whatever the case, spotlighting helps to draw your attention to those statements in the code that involve the spotlighted variable in some way. It gives a clear indication of where the spotlighted variable's value is changing or being checked, so that you can check these easily. And, once you've got the names of any other "suspicious" variables you can spotlight them as well. For example, you might have a statement where "x := x + y;", this might lead you to believe that there is a problem with the "y" value. Perhaps the "y" value is being changed a few lines above the point where its value is being added to the "x" value. Careful study of the code and comparing it with your mental model of the way the code should work, might suggest that the "x := x + y;" statement should be moved upwards, so that it is executed just before the "y" value is changed. This is an example of a sequencing error, due to the dependency of the "x" variable's value on the value of variable "y".

Sequencing errors of this type usually require multiple spotlighting - in the latter case, both the "x" and "y" variables could have been spotlighted, in order to make the dependency problem easier to see. The signal problem gives some examples of sequencing errors (see figures indicated in §6.2.1).

One very direct transference from manual to electronic spotlighting would be the use of a different colour to spotlight each individual variable's trail. That way you could tell at a glance where 2 variable trails intersected by the clashing of the colours within the same line or statement.

If you arrange to spotlight anything that appears between contiguous "{" and "}" symbols, disregarding any surplus "{" symbols, then you should only spotlight "comment text", but if a "}" symbol has gone missing, then it will be obvious where, since all intervening comment and program text will be put into inverse video. This should counter the missing comment bracket problem, which causes the compiler to ignore all code or text appearing between successive "{" and "}" symbols.

The spotlight effect could also be used as a memory jogger, to guard against uncompleted variable name changes. Like using the search and replace tool to change "i" to "index", but without checking that all appropriate changes have been made. So you could spotlight "i" and "index" either sequentially or in combination, to enable checking that all previous instances of "i" have been removed, and replaced by "index" instead. This would be particularly useful where the scope of a variable extends across a large section of text, with a "blank area" in the middle. For example, where a variable is spread across 3 screen "pages", occurring on the first and third pages, but not on the second page. The wider the gap - the more useful the reminding effect.

The latter implies that it would also be useful to know how many instances of the given spotlighted word there were all together, and to know which "position" the "current" spotlight holds. The use of such a "current instance/total instance count" indicator could be used as a strategic (planning-wise) or pure orientation aid. For example, "2/5" would mean that the current spotlight* is focussed on the 2nd instance, and that there are 5 instances altogether - this would be particularly useful if the instances of the selected item were widely scattered amongst the code, particularly for global variables. An alternate use of this spotlighting count ratio would be to detect redundant variables - indicated by a spotlighting count ratio of "1/1".

Spotlighting also implies a need for additional "movement" commands, such as, go to 6th spotlight, or go forwards (or backwards) 4 spotlights, and so on. Thus jumping the cursor position (the primary focus of visual attention) from one spotlight to another without having to visit all intermediate spotlights (viz instances of the search string, as is necessary with current search mechanisms). This would fit in much better with the characteristics of the trail following task, since the programmer is free to move to the most significant spotlight in one jump, rather than several small jumps.

* The current spotlight can be indicated by positioning the cursor so that it flashes on the first character of the spotlighted word (as is used by current search mechanisms).

6.1.3 Summary Menus

The aim of summary menus is to provide alternative views of all or selected data declarations in the program. Making the information more consumable and instantly accessible to the viewer. The simplest transform of the declaration data would be to copy it directly into the confines of a sub-window, so that the programmer could scroll down to find the required information. However, there are more useful, alternative formats for this information.

A variable has 5 attributes: a name (its identifier); a data type; a value range; a parent procedure (the name of the procedure where the variable is declared); and type/scope of the variable - local or global variable, or "var" or non-"var" parameter variable. Thus the main indices for variable summary menus are name, data type, and parent procedure. Consider alphabetizing variable names, summary menus could:

- list variables* alphabetically within one specific procedure (alphabetical by procedure);
- list variables* alphabetically within each individual procedure in the program, following each procedure name with an alphabetical list of its variables (cumulative alphabetical by procedure); or
- list variables** alphabetically within the entire program (cumulative alphabetical).

Instead of listing variable names alphabetically, they could be listed alphabetically under each data type either belonging to a specific procedure (alphabetical by data type by procedure), or listed alphabetically for a specific data type across the entire program (cumulative alphabetical by data type). The depth or range of information seen by the viewer depends on his requirements. The aim is to provide a means of showing the same information from different perspectives and to enable variable declaration and variable usage errors to be detected, both faster and more easily - requiring less effort from the viewer when using the summary tables.

The main purpose behind the summary tables, is to answer questions like 'What data type is variable "x", and where is it declared?' If you call up the summary table submenu associated with variable "x" on the cumulative alphabetical menu, the child menu will list its data type, and the name of its parent procedure. So you'll be able to find out quite easily how and where it's been used, from an on the spot reference. Instead of losing your train of thought searching the declaration areas, you can just call up the information, find out the answer and then carry on with what you were doing.

* with each variable name having a pop-up submenu to show its data type.

** with each variable name having a pop-up submenu to show its data type for each individual parent procedure name.

Some people would argue that a summary menu tool is not needed, because this information is derived from the declaration area in the code, which the programmer could access and read for him/herself. This is indeed true, but they provide the equivalent of library index and cross reference cards. So that the programmer avoids expending time and energy scrolling back to the declaration area, and searching through for a particular variable declaration. It's much simpler just to call up the chosen variable name on a summary menu, and to be told what data type it is. This also excludes errors in thinking the variable is one data type and then finding out much later (when debugging perhaps) that it is something else.

Calling up the required variable name on a cumulative summary menu, would show what different attributes it has under different procedure names. For example to find out which other procedures use the same variable name, and if they are declared as the same or different data types. Perhaps in one procedure it is declared as an integer, and in another it is declared as a real data type. That difference might indicate that there is an error somewhere; and that both should be declared as the same type, either both as reals or both as integers, and of course changing the data type will affect the actions of any operators applied to them.

Alternately, you can check the variable's data type to make sure that you're using the right functions and operators to manipulate it, and also to check that if you're modifying a value and passing it to another variable, that it is assigned to a variable of the correct or compatible data type. For example if you create or modify a real value on one side of an assignment statement, and it is assigned to an integer variable on the other side, then you are going to lose value across the operation, because a real value will be truncated to an integer value (thus losing the fractional value).

Summary tables provide other possibilities for checking. For example, to resolve such problems as 'I need a new timing variable, has the name "timer" already been declared/used in this procedure?'. If no "timer" variable appears on the alphabetical listing then that variable name can be declared and used, without further qualms.

The summary menus could show 2 additional values for each variable: the declaration and usage counters. A double declared variable will give a declaration count of 2, and a usage count of 0 or more. An undeclared variable will give a declaration count of 0, and a usage count of 1 or more. A redundant variable will give a declaration count of 1, and a usage count of 0-1.

Summary tables are meant to collect and show all the different variable and procedure names that the user has generated throughout the program. Comparing any word that is not already declared or is a reserved word or pre- or user- defined procedure, means that it is automatically pin-pointed as an undeclared word. If the summary tool is made to collect all anomalous words into one particular menu labelled

"undeclared", then the viewer will be able to see at a glance, all the different words that are undeclared throughout the text.

All these suggestions are ways of making the actual programming and debugging tasks easier for the programmer, because there is such a lot to remember, and as memory load increases, so does the probability of mistakes.

One of the things you most frequently forget about variables is what data type* a particular variable is, and also, what kind of variable - local or global, or whether it is a formal (non-"var" variable) or actual parameter variable ("var" variable). This is important because you have to take care to modify the right (kind of) variable for the right reason. For example, formal parameter values can be altered "carte blanche" within the scope of the parent procedure, since the original "value" (in the procedure call) is copied to a new variable on entry to the called procedure - thus no modifications will affect the original value/variable passed in. But with actual parameter variables, the corresponding variable's address is passed in and is used to store all modifications to that variable's value.

The other principle feature of summary tables, apart from working on the user defined variables, is for work on user defined procedures (and functions) and their parameter lists. For example, quite a lot of programming errors are to do with the content and format of procedure calls, viz the number and ordering of parameter variables.

Summary tables should give the programmer the option to call up the procedure name and then be able to call up its parameter list as it was declared originally. This would define the order and data type of each parameter, and whether it was an actual or formal parameter. Making it easy to assign the right variables into the right position in the procedure call's parameter list. This should eliminate the problem of adding or omitting a parameter, or getting the order of the parameters muddled.

The foregoing applies to the user's own procedures, but it is perhaps even more useful for unfamiliar predefined procedures/functions, when you need to find out the parameter list. Instead of needing to look at the manual, it's much simpler to call up the procedure's name on the menu, find out it's parameter list and then just fill the procedure call's variables in to correspond, rather than having to go through the aggravation of getting the manuals out and trying to find out more about the required procedure. Also it should be error free, because you will have all of the procedural parameter list there, and, perhaps with a predefined procedure, there may even be some additional information on the actual use of the procedure. This all goes to making life easier for the programmer.

* Pascal has "simple" data types: integer, real, char, boolean, text; and "user-definable" ones: array, string, file, pointer, and records. So there would be at least 10 varieties of data type menus.

6.1.4 Combining Spotlighting & Summary Table Methods

Summary menus would detect undeclared variables by compiling a list of all the variables (and which procedures they belong to). Then any variable name or any word which does not occur in the declared variables list, or is not a reserved word or reserved procedure/function name is obviously undeclared, and can be added to the undeclared list. The programmer could compare the summary menu's undeclared list with the declared list(s) to pick out any discrepancies in spelling eg. declaring "time", and misspelling it as "ttime", "yime" etc. in the text.

Alternately, knowing which variable names or words are undeclared (from the summary menus) makes it easy to choose which to spotlight. Using the undeclared list to select which name to spotlight, either spotlighting each name in turn, or several (or even all) at once, would give a relatively fast and easy means of name selection. Thus avoiding the need for each name to be input manually, and so eliminating typos and spelling mistakes. The programmer would then be free to go to the (spotlighted) program text to see how each name is used, so that the appropriate action could be taken, either to declare a name, or correct the name if it is misspelt, or to erase it if it is redundant. The spotlights also act as a scoping guide as to where (within which "parent" procedure boundary) each new variable should be declared.

6.1.5 Correlating Spotlighting & Summary Tables With Programming Errors

This table is just a brief run down of all the different programming errors that spotlighting and the summary tables should help resolve. Some errors may be better or more easily solved with the spotlighting tool than the summary tables and vice versa. Whereas other errors, such as undeclared variable errors, utilise the features of both tools for the best effect. For example, the summary tables can provide a list of undeclared words; and spotlighting each individual undeclared word would show where it was used, so that the programmer could decide what to do.

Key

Summ means the declaration data summary menu listing tool;

Spot means the Spotlighting (inverse video or colour) tool; and

"y" indicates that Yes, that tool should provide assistance with that particular programming problem (and "n" for No).

Common Errors During Coding	Spot Summ	
undeclared variables	y	y
redundant variables	y	y
double declared variables	y	y
misspelt variable names	y	y
infinite loops	y	n
redundant loops	y	n
missing or non-initialisation of a variable before use	y	n
inappropriate initialisation or modification of variable values	y	n
incorrect sequencing of "dependent" variable assignments	y	n
missing/mis-matched comment brackets	y	n
incompatible format & content of procedure parameter lists	n	y
inappropriate passing/return of variable values via proc calls	y	y

6.2 Applying Spotlighting & Summary Table Tools to the Signal Problem

The spotlighting and summary table tools have been applied to Siddiqi's signal problem, as a means of showing how they can be used.

6.2.1 Statement of Siddiqi's Signal Problem

The following page contains a complete, commented program listing of the solution to this problem, plus the full statement of that problem. Basically, it is a question of repeatedly reading the value of the signal, and incrementing, modifying or resetting different counting variables.

There are 3 choices of "signal" value: 0 to indicate the end of the survey period; 1 to indicate that another vehicle has passed the detector; and 2 to indicate that another second has passed. The problem is to determine the length of the survey period, the number of vehicles passed, and the maximum waiting period; and then to print each of these values.

The complete commented Pascal programming solution caters for each aspect of the problem statement. Including all the initialisation statements, the repeat loop to read each "signal" value as it comes in, and to increment the appropriate counters.

Reading a "vehicle" signal means incrementing the vehicle counter and resetting the current waiting period counter, "wait". Calculating the longest waiting period means maintaining a variable, "maxwait", for the current maximum waiting period, as well as the one to tot up the current waiting period. Each second the "time" and "wait" counters are incremented. If the current "maxwait" value is less than the "wait" value, then the "maxwait" value is updated to have the same value as "wait".

Statement of Siddiqi's Signal Problem & A Complete Code Solution

Siddiqi's (1985) signal problem (Program designer behaviour, People & Computers 1, p377) stated as follows :

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

- (a) the length of the survey period;
- (b) the number of vehicles recorded;
- (c) the length of the longest waiting period without a vehicle.

The program text below shows a complete, commented solution to Siddiqi's signal problem - this can be used for reference and comparison of the subsequent partial solutions, and the variety of errors that spotlighting emphasizes in each case.

```
program survey(input, output);
var
  signal : 0..2;
  { 0 indicates end of survey period,
    1 indicates another vehicle has passed the detector,
    2 indicates another second has passed. }

  time, { length of survey period in seconds }
  vehicles, { no. of vehicles detected so far }
  wait, { time in seconds since last vehicle was detected }
  maxwait : integer; { maximum waiting period so far }

begin { initialise }
  time := 0;
  vehicles := 0;
  wait := 0;
  maxwait := 0;
  repeat { read and process signals until end of survey period }
  read(signal);
  if signal = 2 then { another second has passed, so increment time counters }
  begin
    time := time + 1;
    wait := wait + 1;
    if wait > maxwait { adjust maxwait to new maximum wait value }
    then maxwait := wait;
  end;
  if signal = 1 then
  { a vehicle has passed, so reset wait counter, and increment vehicle count }
  begin
    wait := 0;
    vehicles := vehicles + 1;
  end;
  until signal = 0; { end of survey period }
  { Print out required data }
  writeln('Length of survey period is ', time, 'secs');
  writeln('No. of vehicles recorded is ', vehicles);
  writeln('Longest waiting period is ', maxwait, 'secs');
end.
```

The Effect of Spotlighting Different Variables

```
program survey(input, output);
var
  time, vehicles, wait, maxwait : integer;

begin
  wait := 0;
  vehicles := 0;
  read(signal);
  repeat
    if signal = 2 then
      begin
        time := time + 1;
        if wait > maxwait
          then maxwait := wait;
        wait := wait + 1;
      end;
    if signal = 1 then
      begin
        vehicles := vehicles + 1;
      end;
  until signal = 0;
  writeln('Time-span=', time, 'secs');
  writeln('Vehicle-count=', vehicles);
  writeln('Max-wait=', maxwait, 'secs');
end.
```

Fig 1 Plain Text

```
program survey(input, output);
var
  time, vehicles, wait, maxwait : integer;

begin
  wait := 0;
  vehicles := 0;
  read(signal);
  repeat
    if signal = 2 then
      begin
        time := time + 1;
        if wait > maxwait
          then maxwait := wait;
        wait := wait + 1;
      end;
    if signal = 1 then
      begin
        vehicles := vehicles + 1;
      end;
  until signal = 0;
  writeln('Time-span=', time, 'secs');
  writeln('Vehicle-count=', vehicles);
  writeln('Max-wait=', maxwait, 'secs');
end.
```

Fig 2 Signal Variable Only

```
program survey(input, output);
var
  time, vehicles, wait, maxwait : integer;

begin
  wait := 0;
  vehicles := 0;
  read(signal);
  repeat
    if signal = 2 then
      begin
        time := time + 1;
        if wait > maxwait
          then maxwait := wait;
        wait := wait + 1;
      end;
    if signal = 1 then
      begin
        vehicles := vehicles + 1;
      end;
  until signal = 0;
  writeln('Time-span=', time, 'secs');
  writeln('Vehicle-count=', vehicles);
  writeln('Max-wait=', maxwait, 'secs');
end.
```

Fig 3 Time & Vehicles Variables

```
program survey(input, output);
var
  time, vehicles, wait, maxwait : integer;

begin
  wait := 0;
  vehicles := 0;
  read(signal);
  repeat
    if signal = 2 then
      begin
        time := time + 1;
        if wait > maxwait
          then maxwait := wait;
        wait := wait + 1;
      end;
    if signal = 1 then
      begin
        vehicles := vehicles + 1;
      end;
  until signal = 0;
  writeln('Time-span=', time, 'secs');
  writeln('Vehicle-count=', vehicles);
  writeln('Max-wait=', maxwait, 'secs');
end.
```

Fig 4 Signal, Time & Vehicles Variables

6.2.2 Applying Spotlighting to the Signal Problem

The examples show a combination of errors. I shall only point out those that spotlighting helps to pin-point with respect to the current variable(s) being spotlighted.

Fig. 1 shows plain program text - a partial solution to Siddiqi's signal problem - as it would appear on most VDU screens. The only noticeable feature of the text is its shape. Nothing else stands out at first glance.

The inverse video blocks of Fig. 2, however, immediately draw the eye towards them - clearly showing the "signal" variable in context. It's easy to see that there is no spotlight in the declaration area, so you need to declare the "signal" variable. And looking down you see that the "read(signal);" statement appears above the "repeat" loop, which is the wrong place. It needs to be moved down a line, so that the "signal" value is read each time you go round the loop. All the rest are correct - the "if" statements increment the right counters, and the condition at the end of the repeat-until loop is correct.

In Fig. 3 the "time" and "vehicles" variables are spotlighted together, to see if there are any dependency effects. Dependency effects of course being conveyed by the fact that if you are using a colour system (lucky you) and you are using different colours for different variables, then you will have a clash of 2 colours within the same line (or same statement if the statement spans one or more lines). On a green screen, you will only have one colour - inverse video - but it should still prove effective. So that's the simple way of checking for dependency. If 2 variables are independent (or not directly dependent) and they are spotlighted together then their spotlights will not generally occur within the same statement. In this example the "time" and "vehicles" variables do not appear within the same statement, at any point. So they are clearly independent of each other. However, in other cases, an intermediate variable could cause an indirect dependence.

Another feature of inverse video that aids distinguishing between the 2 variables is the different lengths of the inverse video blocks - "time" is a 4-letter word whereas "vehicles" has 8. This second effect is purely coincidental in this case, but it will obviously provide some benefit as a means of distinguishing between different letter-length variables on a green screen system. Ideally, a colour system would provide the means for using a different colour for each variable used in multi-spotlighting.

In Fig. 4 the "signal" variable is spotlighted in addition to the "time" and "vehicles" variables. This is just a further demonstration of the dependence/independence spotlighting effect. If you disregard the spotlights for the "signal" variable temporarily, then you can see that the "time" and "vehicles" counters are completely independent of each other, whereas they both rely to some extent on the "signal"

variable (as expected). So spotlighting is helping to display sequencing dependencies between the 3 variables.

The next figures (Figs 5 & 6) show a slightly different semi-developed version of the solution to Siddiqi's signal problem. In this case some comment text has been added, as well as proceduralising the segment of the program that increments each of the timer variables, and updates the "maxwait" counter. This sub-procedure that deals with the timer variables is then called from within the main program loop. This is just to show the effect of having spotlighting working on scoped text rather than flat text, which is what has already been demonstrated, by simply working on a name basis. In this case you could apply spotlighting to a scoped version of a name, and tie it in with the summary tables for selection of the right variable name under whichever procedure you wanted to investigate. So you could make sure that it was only the global version of the "wait" variable that you wanted spotlighted, as in fig 5, and not the local variable of the same name.

Fig 5 shows the effect of spotlighting, when the global (main program) variable "wait" is selected - the declaration, initialisation, re-initialisation and procedure call statements involving "wait" have all become highly visible. However, the "wait" variable statements in the sub-procedure remain camouflaged, because they are associated with the local "wait" variable belonging to procedure "inc_timers", which is not the same as the global (main program) variable of the same name. If the procedural parameter list for "inc_timers" had not included the "wait" variable, then the references would have referred to the global variable (in this particular case) and spotlighting would have emphasized these instances of the "wait" variable as well.

There are 2 possible ways in which the user could specify the word that is to be selected for spotlighting (in the code editing context): either the word is inputted manually, as per the usual mechanism for the search command, and spotlighting works on flat text; or the user could choose the relevant word from one of a series of menus showing lists of variable names (see Figs 9-10) so that spotlighting works on scoped text. For example, selecting the global "wait" variable from the Fig 10 menu would produce the spotlighted text of Fig 5. In the latter case, the scope of the chosen variable restricts the areas where spotlights could appear.

Combining the functionality of such menus with the spotlighting facility should enable powerful operations to be performed. For example, enabling the individual spotlighting of each undeclared variable that appears on an undeclared variable menu - thus enabling checking and correcting of such errors. However, the programmer may choose to see all undeclared variables in the text spotlighted at once, or alternatively to cycle through the undeclared variable list, spotlighting each one in turn, either singly or cumulatively.

```

program survey(input, output);
var
  time, vehicles, wait, maxwait : integer;

procedure inc-timers
  (var time, wait, maxwait : integer);
begin
  time := time + 1;
  wait := wait + 1;
  if wait > maxwait
    then maxwait := wait;
end;

begin { main program }
  wait := 0;
  vehicles := 0;
  read(signal);
  repeat { process signal }
    if signal = 2 then
      inc-timers(time, wait, maxwait);
    if signal = 1 then
      begin
        wait := 0;
        vehicles := vehicles + 1;
      end;
  until signal = 0;
  writeln('Time-span=', time, 'secs');
  writeln('Vehicle-count=', vehicles);
  writeln('Max-wait=', maxwait, 'secs');
end.

```

Fig 5 : Wait (global variable only)

```

program survey(input, output);
var
  time, vehicles, wait, maxwait : integer;

procedure inc-timers
  (var time, wait, maxwait : integer);
begin
  time := time + 1;
  wait := wait + 1;
  if wait > maxwait
    then maxwait := wait;
end;

begin { main program
  wait := 0;
  vehicles := 0;
  read(signal);
  repeat { process signal }
    if signal = 2 then
      inc-timers(time, wait, maxwait);
    if signal = 1 then
      begin
        wait := 0;
        vehicles := vehicles + 1;
      end;
  until signal = 0;
  writeln('Time-span=', time, 'secs');
  writeln('Vehicle-count=', vehicles);
  writeln('Max-wait=', maxwait, 'secs');
end.

```

Fig 6 : Matching Comment Brackets

Examples of Summary Menus

Fig. 7

```

Program Survey
time
vehicles
wait
maxwait
Undeclared
signal

```

Fig. 8

```

Component List
program survey
procedure inc_timers

```

Fig. 9

```

inc_timers
time
wait
maxwait
Undeclared

```

Fig. 10

```

Global
time
vehicles
wait
maxwait
inc_timers
time
wait
maxwait

```

Fig. 11

```

Alphabetical
maxwait 2
time 2
vehicles
wait 2

```

Fig. 12

```

Undeclared
signal

```

Fig. 13

```

Component List
program survey
procedure inc_timers (var time, wait, maxwait : integer)

```

Fig. 14

```

procedure inc_timers
(var time, wait, maxwait : integer)

```

Fig. 6 demonstrates the missing comment bracket problem, which can be a major problem. If you ask someone else to look, they see it immediately, but you can't see it yourself until it's pointed out to you. So for this problem, spotlighting (inverse video) is activated whenever it finds an open comment bracket, "{", and deactivated when it finds the next closing comment bracket, "}". In the example shown, there are 5 or 6 lines of code that become commented out because a "}" curly bracket has been missed off the end of the first comment. This gives a clear indication that you should do something about it, unless you intended this to happen. For example, to test 2 or more alternative programming plans that achieve the same/similar effect, to find out which is better in terms of effect or efficiency.

The brevity of the examples give an indication of the interpretational power afforded by spotlighting. However, it must be remembered that in longer texts, this power should increase as the (potential) number of selected item instances increases. If the selected item has a low density (few instances within a large chunk of text), then it becomes increasingly easy, especially with unassisted visual scanning, to overlook some instances. The same is true for high densities, where the same effect occurs due to information overload and confusion between successive statements (Card et. al. 1983).

6.2.3 Applying the Summary Tool to the Signal Problem

Figs 7-14 are examples of the summary menus that could be produced through the use of an interrogative program, that "reads" a file of program text and creates lists of entities significant to the programmer, the most useful of which relate to variables, of course.

Fig 7 shows the declaration ordered variable list of "program survey" with its undeclared list corresponding to (an interrogation of) the program text of Fig 1.

In contrast, Figs 8-14 show the summary table lists that could be produced after interpreting the structure produced through interrogation of the program text of Fig 5. Fig 8 shows the component list - the full list of (user-defined) procedures and functions, including the program name. Selecting a name shown on the component list would cause the associated child lists to become available - either in declaration or alphabetical order; with or without the associated undeclared variable lists. The examples demonstrate some of the possibilities.

Fig 9 shows the declaration ordered variable list of "procedure inc_timers" corresponding to Fig 8. Note that the lower portion of both Figs 7 & 9 is devoted to undeclared variables.

In contrast, Fig 10 defines the list of declared variables that are accessible and can be used, in terms of global and local variables, when seen from within procedure

"inc_timers". This could be really useful for nested procedures, when deciding which variables need to be declared between parent and child procedures; and to decide what type of variables go into the child procedure. Whether as parameter variables (either formal or actual), or as local variables.

Fig 11 shows the cumulative alphabetical list of variables declared throughout the program. Notice that each variable is associated with a number, if it is declared more than once. Selecting any individual variable name would cause a list of its "parental" procedure names (denoting declaration origin), to pop up, with or without an accompanying definition of the variable's data type (depending on the viewer's requirements). For example, with the "wait" variable, the program "survey" is one parent and the procedure "inc_timers" is another.

Fig 12 shows the (cumulative) undeclared variable list, which for this program is very short. Fig 12 would result from interrogating either version of the program text (either Fig 1 or 5) in this *particular* case, but in general the contents of the undeclared list would change from one version of text to another.

Combining the spotlighting and summary tools for the undeclared variables problem should be very useful. The summary tables can tot up which variables are undeclared (as shown by Fig 7 or 12) and then the programmer can choose to spotlight all of them, either all at once, or individually, and then go and look at them. That should help to get rid of all the undeclared variable errors - rather than waiting for the compiler to define them later on. There are 2 ways of showing the entire declared and undeclared lists - either list all declarations and allocate them as given (Fig 10), and list all undeclared items separately in a "floating" cumulative list (Fig 12); or list everything in association with its parent list, noting declared items first and undeclared items second (Fig 9), so that it is easy to tell where each item appeared, and hence to allocate it. Having a choice of view should alleviate the degree of memory load and cognitive processing, and make the undeclared problem easier and faster to resolve.

Each of the lists shown in Figs 8-10 provides a means of associating the spotlighting concept with a selection mechanism, that is fine-grained enough to uniquely identify the spotlighting target. Thus, tracing a given procedure name/call through the text, using spotlighting, could be achieved by selecting the required name on the component list of procedure/function names (Fig 8). Variables could be traced by invoking the required child list (Fig 9), from the component list (Fig 8), and selecting the required variable name (either from a declaration or alphabetically ordered list).

Alternatively, variables could be listed alphabetically from the entire text (with dangling "parentage" references, see Fig 11), so that when a variable name is chosen, the required parent procedure could be selected from the accompanying pop-up menu.

The final declaration problem, dealing with incompatibility in format and content of procedure or function parameter lists, could be resolved by using the component list of

procedure and function names. When one name was chosen this would call up a child menu - listing the original declaration of the parameter list. For example, Fig 13, shows the parameter list for "inc_timers" as a child menu attached to the parent menu at a position showing the parent procedure's name. Whereas, Fig 14 shows the named procedure with its parameter list as a single menu. The latter menu seems clearer in its meaning, since it is a more direct reference to the procedural declaration - either view (Fig.s 13 or 14) should provide all the significant clues (schema details) needed by the programmer to "prime" the procedure call with all the appropriate elements, and in the correct positions.

The component list of procedure/function names (shown alone in Fig 8, or as a parent with a derivative child menu in Fig 13), could be ordered either alphabetically, or in "declaration" order. Depending on what the user finds most comfortable to work with. This list could also be used to call up a series of "child" variable lists (for each individual procedure): declared variables only, undeclared variables only, or, declared and undeclared variables together; with an option to restrict each such list to the named parent procedure, or to view each list from the current perspective in terms of "accessible" global and local variables.

6.3 Integrating Spotlighting & Summary Tables Into A Supportive Environment

This section gives an outline sketch of implementation issues in a hypothetical Pascal environment, and the basic infrastructure underlying the spotlighting and summary aids. Note that all references to procedures are equally applicable to functions, unless stated to the contrary. Also, that the word "global" has 2 meanings: meaning either global with respect to the current context, or belonging to the main program.

6.3.1 Discussion of Spotlighting Implementation Issues

Now, the purpose of spotlighting, in the programming context, is to make errors easier to find. However, the next task after identifying and locating an error is to correct it - thus, spotlighting must not interfere with this. This means that all the usual (mutative and locative) editing commands and tasks must be able to take place on the spotlighted text as easily as on plain text - that is, with no detrimental or knock-on effects, either in terms of functionality or (visual) screen effects. One possibility is to take note of the "earliest" line number when editing, and the "latest" line when editing finishes, and to re-spotlight all intermediate code.

The first task of implementation is to define how to select a variable or whatever items are selectable. Selected items could include variables, procedure or function names, types, constants - in effect any distinguishable textual element included in the edit-document, whether source code or not. Thus the primary task is to define what is

selectable, and then to produce mechanisms to facilitate selection. Also, to decide whether to allow selection of partial words, or to restrict selection to whole words only. Consideration must also be paid to the use of scoping context in the selection.

For example, to match comment brackets, the program text is regarded as flat text, since missing (closing) comment brackets over-throw what appears to be scoped text (as in Fig 6 of §6.2). But, for a scoped variable eg. the global version of the "wait" variable (Fig 5 of §6.2), attention must be paid to spotlighting only those items which meet the scope requirements - this may or may not include those (potentially significant or irrelevant) items which occur within comment statements. A secondary problem is how to deal with items appearing within comments, and whether comments should be disregarded or included in the definition of searchable program text.

With multiple spotlighting there will be an accumulation of spotlights on a series of selected items - each individual item having its own spotlight pattern, which will contribute to the overall pattern of spotlighted items. On a colour system it is feasible to have different colours, with each selected item having its own colour code. This should make it easier to distinguish each spotlighted item's individual contributions to the larger pattern.

On a monochrome system, either all spotlights are the same (all in inverse video), or an alternative means of typographic differentiation could be used - such as italicising, boldening, underlining, or perhaps capitalising. However, the latter approaches may dissipate the visual power of multiple spotlighting due to the inherent imbalance of visual power (visibility) provided by the various typographic effects (as is evident from the visual comparison of typographic effects shown in §6.1). This is because italicising is more subtle and less obvious than inverse video - whereas colour coding works on the same (cognitive) level and recognition mechanism.

There would also have to be a mechanism for removing spotlighting - either wholesale (removing spotlights from all items) or selectively (removing spotlights from one or more already spotlighted items by individual selection). This suggests that a list (or menu) of spotlighted items should be available so that the viewer has an up to date summary of those items which are already spotlighted, and thus of the state of the text spotlight-wise. The current spotlighted item could be identified on the menu list, by reversing the background or bordering its name, as is done on MacPascal and other WIMP systems, to show the current selected item within a list of other items already spotlighted. Selective removal would be useful for multiple spotlights when checking dependencies between multiple variables, so that for each new variable selected, an "old" one is deselected. For example, checking a set of 4 variables {a,b,c,d} would give 6 sets of paired spotlights - {a+b, a+c, a+d, b+c, b+d, c+d}; or 4 sets of triple spotlights - {a+b+c, b+c+d, a+c+d, a+b+d}. An electronic note pad would be useful, for this type of activity, for checking progress and "ticking off" jobs

as they are completed, and to remind the programmer of the next or current task - and to gather evidence, data or sub-plans. It would also be useful to be able to append selected chunks of code onto a series of data buffers, rather than being restricted to a single pasteboard or data buffer.

It would be useful to provide a spotlighting instance counter to show how many instances of the item had been spotlighted. A "current instance / total instance count" would be even more helpful for orientation purposes. For example, "3/8" would mean that the cursor position is focussed on the 3rd spotlighted instance of the item, and that there are 8 spotlights altogether - this would be particularly useful if the instances of the selected item were widely scattered amongst the code, particularly for global variables. You could really only have the current instance/total instance counter for one variable at a time, since the cursor position that indicates the current spotlight is a unique feature. Applying it to multi-spotlighting is not feasible.

Davies' (1989) results indicate that moving aids should extend the variety of existing moving commands to fit in better with programming habits and task characteristics. Embedding spotlight codes in program text implies additional facilities, such as movement from one spotlight to another - this would be particularly useful if the distance between spotlights exceeds the screen. For example, the ability to move to the nth spotlight below or above the current one, or to the "nth instance" spotlight. The programmer's goal is to get to the current point of interest in one jump, rather than in a series of distracting screen-hopping jumps. What Monk (1989) would refer to as orienteering, rather than rambling - getting from A to Z in 1, not 26, jumps. In essence, this means giving the user more control over which areas of text are visited, by supplying appropriate moving commands that are linked to visually emphasized points of significance.

One facility that must be available, is the ability to produce a printed version of the spotlighted file, for more intensive off-line study. This is especially necessary for lengthy texts, or when the spotlights are widely scattered, and it is not practicable to jump around the screen text without gaining a better appreciation of the context associated with each spotlight from a wider perspective.

Infrastructure

The environment must support 2 different aspects of spotlighting - a hierarchical, scoped version and a "normal" flat text version. The former for detecting scope errors and the latter for detecting missing bracket type errors. For example, using the flat text version to enable selection and detection of "partial" items and symbols, like those spurious control characters, or matching symbols such as "(" and ")", or "[" and "]", or "{" and "}". Scoping requires that the code be structured, either fully, like a parse tree with accompanying symbol tables, or semi-structured, like hanging each block of procedural text off a "procedure" node, so that the main program consists of

a network of hierarchical nodes and their associated text. Thus, in the latter 2 structures, scoping would be achieved by going (top-down) through the successive child nodes (any nodes above the "start" node being outside the scoping bounds).

6.3.2 Discussion of Summary Table Implementation Issues

Assuming that the base environment contains all the familiar functional elements associated with existing editing environments, including pop-up or pull-down menus, and a means of selecting a section of text by indicating start and stop points (either using line numbers, or cursor and mouse movements, as with MacPascal). Then the hypothetical element is an interrogative program that goes through the current version of the source code, producing a set of summary tables or (linked) lists, defining the contents of each menu. After running the interrogative program, there would be lists defining the declared and undeclared variables for each individual procedure and function, as well as for the parent program.

Instead of just showing the procedure and its parameter list on screen, it might be useful to have an option to cause either the full procedure declaration (including parameter variable names and data types), and/or a procedure list of variable names (without data types) to be inserted on or above/below the line occupied by the current cursor position, followed by an "empty" procedure call that defines the number of parameters by the number of commas. Plus empty lines above and below, to remind the programmer to move the procedure/function call if it is in the wrong position in the statement sequence. For example, "function maxval(val1, val2 : real) : real;" could be followed by either "maxval(val1, val2 : real) : real;" or "maxval(,);"

If global variables were only allowed in the main program loop, and thereafter defined in terms of procedure calls only, then all items not declared in a procedure could be accounted for either as undeclared local variables or local parameter variables. Otherwise the global list might account for some of them.

A facility to transfer the contents of the undeclared variable list into the appropriate declaration area (selectively or wholesale at the user's request), should speed up declaration of the previously undeclared items.

Infrastructure

It seems fairly clear that if you are going to have the declaration data expressed as menu lists which you can go up and down on, then the underlying representational structure must be tree based with (streams of) forward and backward pointers for each variety of menus that are going to be produced. The main variety of menus for variables are related to variable names (in declaration or alphabetical order), data types, and parent procedure names (in declaration or alphabetical order). Each paired

stream of forward and backward pointers will create a linked list, making it easy go up and down the tree structure corresponding to the list of items on the menu screen.

The links are traversed in order to access the next variable name in the tree structure and to add it to the menu list. For example, scrolling down a declaration ordered menu represents a top-down traversal of a tree structure. Thus, as the viewer scrolls down the menu, the corresponding forward pointers are being traversed in order to provide names to be added to the bottom of the menu. New names will continue to be added until the last pointer in the link is traversed.

Paired streams of pointers will be needed for each type of cumulative (global) list as well as for each list relative to an individual procedure block, in terms of alphabetical or declaration ordered listing, each individual data type, and so on.

The tree structure could also include "types" and "consts". This would help to tie variables up with the user-defined data types. For example, a data type list could be produced, and if by the end of the program traversal, there were no variables assigned or corresponding to that entry, then this would indicate a redundant data type.

6.4 Conclusions

Applying Green's (1989) cognitive dimensions to the tools :-

Spotlighting:

- adds perceptual cues to the structure to aid/simplify trail following.
- boosts visibility and discriminability.
- reveals hidden/explicit dependencies: if A, B, and C are variables, and there is an error associated with the value given by an assignment statement, "A := B + C;" then you need to spotlight both B, and C, to find out which variable is responsible.
- counters viscosity, diffuseness, and hard mental operations by its suitability for the trail following task (its role expressiveness); by providing an easy to trace path, so that the viewer can see where each variable is, and how it is used.

Summary tables:

- reveal hidden/explicit dependencies in procedure call parameter lists, viz the assignment of "var" and non-"var" parameter variables.
- avoid premature commitment eg. declaring the wrong variable name (one that is already declared), or assigning values between incompatible data types.
- avoid hard mental operations and diffuseness by showing all required attributes on demand, according to the viewer's requirements.
- support consistency - the required information format is defined by the viewer.
- minimize (and helps resolve) action slips, such as misspellings and typos by listing undeclared variables and enabling comparison with declared variables.

According to Curtis's (1988a) restatement of Fitter & Green's (1979) conclusions about notational design, both tools are: relevant to their individual tasks, they provide revelation and redundant recoding, and their effects are revisable.

Davies's (1993) research on expert programmers showed that, "Experts develop display-based skills in order to reduce load on working memory". Both spotlighting and summary tables should help to reduce working memory further.

6.5 Discussion

The usage of highlighter pens on paper text has a long history, ranging across all varieties of paper based work. In each case, they provide a means of picking out the important words or data from the surrounding background text. It is time the electronic viewer had an equivalent tool to apply to electronic text.

Since spotlighting is intended mainly as a feature to be used in computer-aided text editing, it must be implemented in such a way that it does not interfere with the usual editing commands. Also, there must be an option to list spotlighted text to the printer, so that the text can be studied at greater length.

The only existing software facilities that have similar actions or results to summary tables are cross indexing programs, such as AnsibleIndex, where you feed in the main text containing a selection of words or phrases that are marked for indexing, and the program returns a list of page numbers where each marked word or phrase occurs. Spotlighting's nearest relations are spelling checkers, such as Locospell, which use inverse video to pick out the offending word, so that the viewer can identify and locate it, and in so doing begin to absorb its context by reading the text in close proximity, as a means of determining what word was intended, before correcting the error.

Spotlighting is more generally applicable, for example to electronic books (as an online interactive index), electronic spreadsheets (for checking variable entries in formulae), and many other environments where it is useful to find out the location of specific items. Summary tables may be relevant to stock control and inventory or other cross referencing cataloguing systems. Layout aids are probably only relevant to the programming environment.

Chapter 7 Further Data Collection & Experimental Evaluation of Tool Feasibility

December 1990: the finalists were chosen as subjects to provide data on debugging.

They were the same group of students who provided the data for chapter 3, but 2 years on in their experience.

§7.1 deals with the main questionnaire results

§7.2 deals with the debugging tasks

§7.3 deals with the spotlighted debugging tasks

§7.4 deals with the post spotlighting questionnaire

Data Collection - Questionnaire & Debugging Tasks

Data collection had 2 aspects - a questionnaire with 40 questions (see Appendix Q7A), and a series of 3 debugging tasks (see Appendix 7C); with the debugging tasks to be done mid-questionnaire, after answering question 15 and before going on to question 16. It was expected to take about 1½-2hrs to complete.

The purpose of this questionnaire and the debugging tasks was to gather as much open ended data as possible about debugging strategies and each student programmer's approach to programming. From developing code from a task description themselves, as well as the actual comprehension strategies they applied to code they had not written themselves.

The questions that prompted these experiments were a) what sort of information do programmers need to be able to successfully implement their debugging strategies, and b) how do they use this information. To differentiate between the successful and unsuccessful strategies behind the interpretation and use of such data.

The reason for doing the debugging tasks part-way through the questionnaire was to take full advantage of this fresh debugging experience. So that their answers about debugging strategies would be easier to bring to mind, and hopefully, in greater detail than usual.

The questionnaire questions were ordered specifically to lead the students through all aspects of software development and the debugging process. So they could consider the methods they use, what effects they have and what sort of errors they have to deal with.

In effect I used a double pronged strategy. Balancing the "basic" question I wanted answered with an array of "expected" answers; appended by an open ended "Give reasons for this choice" type of question, to elicit the rationale behind the chosen answer.

The objective of the debugging experiment was to find out what methods the students used to develop and debug code, and to discover the reasons why they do or do not use these methods. Ample space was provided to give these reasons and to make any comments they thought were relevant.

Students expressed their reactions to the debugging tasks and gave a "difficulty rating" for each task on the comment sheets provided (see last page of Appendix 7C).

Experimental Evaluation of the Spotlighting Tool & Subjective Questionnaire

About a week later, the students evaluated the spotlighting tool by debugging 4 (incomplete) Pascal programs; with 2 tasks on plain (non-spotlighted) code, and 2 tasks set on text that had been individually spotlighted for each of the 4 principal variables. Thus the task on spotlighted code had 4 sets of code per task, with a different (single) variable spotlighted on each.

Students answered a short questionnaire (of 15 questions, see Appendix Q7B) after finishing all 4 debugging tasks. The questions were primed to get the students reactions to different aspects of the spotlighting concept after trying it out for themselves in the experiments. Question 37 (from the questionnaire, Q7A, of §7.1) was restated, to find out whether the students had changed their opinions in any way. The remaining questions were all new, 7 concerning spotlighting, and 7 concerning the layout style used in the spotlighting experiments. The latter asked about aspects of layout style that might have had an effect on the experiments, and gave additional data relating to the proposed layout tool.

Students expressed their reactions to the experimental debugging tasks and gave a "difficulty rating" for each task on the comment sheets provided (see last page of Appendix 7D).

7.1 Design & Analysis of Programming & Debugging Strategies Questionnaire

The questionnaire had 2 types of questions.

There were "open" questions which were intended to find out each student's views/opinions about debugging, and the variety of methods and strategies that they use to detect and resolve different types of bugs.

There were "multiple choice" questions where the expected answers were shown in square brackets. If the students had an alternative answer to those presented, I asked them to write down their answer - with a brief explanation (if necessary), so that I would be able to interpret their answer correctly.

The "open" and "multiple choice" form of questions both appeared at the beginning and end of the experiment, to get the student's views on different aspects of debugging, error types etc., before and after attempting the debugging tasks.

The variation in question types was intended to draw out students' opinions as fully as possible, by making the questions as explicit as possible.

Some of the questions were multi-part, having parts i, ii, and iii, to explore different aspects of the same topic.

7.1.1 Analysis of the Questionnaire

Analysing the results of the questionnaire proved more difficult than expected, due to the complexity of the data, and the lack of simplifying methods applicable. Appendix Q7A shows the raw data and its corresponding data tables to give an idea of the variation in answers and attitudes. Plus a summarizing comment giving the gist of the responses.

7.1.3 Summary of Questionnaire Results

Brief summary of important points brought out by the questionnaire :-

Experience & Programming Ability - Q1-3

Of the 8 students tested, 6 regard themselves as having average (or better) programming ability with Pascal; so they represent a fairly "typical" subset of students ability-wise.

Program Development and Coding Methodology - Q4

The main development strategy is to work out most of the algorithm, translate it into code, and fill in the missing parts as development continues. As task complexity and code length increases, top-down modularisation techniques become more prevalent.

Development & Debugging Attitudes - Q6, Q8, Q10, Q13 & Q29

Q8. Most (7/8) students check their code before development and (6/8) after development.

The results of Q10 & Q13, show that the students' priority regarding errors is: error prevention, error elimination, and checking for task correctness after syntactic elimination.

Use of Debugging Techniques & Tools - Q9, Q11, Q12 & Q22

Q9. During code development, quick or slow read throughs, and mental simulation are the main single debugging strategies, scoring 4, 5, and 5 respectively. With 7 (out of 7) students reading through code either quickly or slowly, with 5 (out of 7) of them using mental simulation as well.

After development the main debugging strategy is reading through rather than using mental or hand simulation - given by 5 quick and 4 slow readings; 2 mental and 3 hand simulations. Totalling 7 students reading through, with only 3 of them using mental or hand simulation.

The predominant debugging strategy after compiler defines errors is a slow read through with mental and/or hand simulation (for 4 out of 8 students). There are 7 students reading through, with 5 doing both mental and hand simulation as well, while the other 2 choose to do either mental or hand simulation.

This indicates a higher degree of problem solving during actual development and in response to compiler defined errors.

Q11. Students' rating of 6 debugging strategies, by both frequency of use and preference, in descending order is: inserting write statements, hand simulation of code, mental simulation of code, tracing variable values by hand/eye, tracing variable values by debugger, tracing variable values by search mechanisms.

Q22. The top 5 debugging techniques are: inserting write statements wins with 5 votes; hand simulation is next with 3 votes; while mental simulation, and comparing the intended code to the actual code, tie with 2 votes each; and tracing variable values by search mechanisms comes in last.

Summary

It seems that using write statements is the most frequently used and best liked debugging method. No other tool(s) seem able to match it for flexibility. Debugging tools are hardly ever used unless absolutely necessary (Q12), and the same goes for the search mechanisms (Q11 & Q22).

Defining The Nature of Errors, Their Frequency & Troublesomeness - Q14-15

Q14. The top 3 most common semantic, logic and/or algorithmic errors that students check for are: faulty procedure/function calls (6 votes), variable faults (5 votes), and faulty conditional statements (3 votes).

Q15. Top 12 Most Frequent Errors

- 1 Missing or extra bracket in an expression
- 2 Undeclared variables, types or constants
- 3 Misspelt names (eg. variables, types, constants, reserved words)
- 4 Incorrect placing of brackets in an expression
- 5 Using round brackets, (), instead of square brackets, []
- 6 Inappropriate data typing of variables (eg. using real instead of integer)
- 7 Redundant declarations of variables, types or constants
- 7 Inappropriate placing of "end" statements
- 9 Incorrect modification of variable values
- 10 Incorrect initialisation or termination of variable values
- 11 Inappropriate choice of loop variable
- 12 Missing "else" statement(s) to complement an "if" statement

Q15. Top 12 Most Troublesome/Time Consuming Ordering

- 1 Infinite loop(s)
- 2 Incorrect content of procedural parameter list call
- 3 Inappropriate declaration of a procedural parameter list
- 4 Incorrect placing of brackets in an expression
- 5 Inappropriate declaration of a procedural parameter list
- 6 Run-time errors (divide by zero, under- or over-flow of values)
- 7 Inappropriate placing of "end" statements
- 8 Incorrect modification of variable values
- 9 Incorrect sequencing of variable value assignments
- 10 Incorrect sequencing of control structures
- 11 Incorrect sequencing of procedure or function calls
- 12 Incorrect choice of loop variable value ranges (eg. in "for" statement)

There are very few errors that appear on both lists, except the 4th, 7th, 8th and 9th entries which occupy (almost) the same rank on both lists. Most other errors appear on one list or the other, which seems significant.

Investigating Reading Strategies - Q16-19

These questions relate to the research on comprehension and reading by Pennington (1987), Nanja & Cook (1987), Gugerty & Olson (1986), and Holt, Boehm-Davis & Shultz (1987). The results support their main findings discussed in Chapter 2 :-

For example, Pennington (1987) has found that programmers who are able to cross reference the application and domain models are able to create a much richer task model. This enables full(er) understanding of the task, thus making debugging much more accurate and effective.

Q16. 6 out of 8 students read and re-read the code/task description to understand/reaffirm the task requirements. In my opinion, the purpose of reading the task description is to build up a mental model of the task that is to be performed.

Q17. 3 students think that reading to get a total understanding of the code makes debugging faster (students 1, 2 & 8), and more accurate (students 1, 2 & 5); whereas 3 students think that reading code on as needed basis helps fill in details (students 3, 6 & 7).

Q18. On the first read through, it seems that the primary task (for 6 out of 8 students) is to correlate the code and its structure with the task description. Whereas making sense of the code on its own is a secondary task (for 3 out of 7 students) . This is still true for the second read through (5 out of 7, and 3 out of 6, respectively), but the numbers have been reduced by 1 student, SS8, who starts debugging immediately after the first read through.

Q16-Q19 Summary

Those students who started debugging on the first read through are likely to give fast debug times, because I assumed that the first read through is just for reading and that debugging comes later. If not, then some bugs may already have been solved during the first reading, and it is just a question of writing the error solution out. Rather than finding and solving each error in the subsequent debugging phase which I assumed to be separate from the initial code reading phase. This might account for the fast debugging times for this sort of strategy in the debugging & spotlighting experiments (see §7.2 & §7.3).

From experience I know that some errors just spring out at you on the first read through - usually the glaringly obvious ones - and they are difficult to ignore. It's usually easiest (less drain on remembering to do it later) to fix them there and then.

Investigating Trail Following on Paper & Screen Text - Q24 & Q25

Trail following on paper is preferred by 5 students, although (of the 5) 1 student prefers trail following on screen if the code is short. Students comments indicate that paper text is consulted in order to think things out and to decide what to do, whereas the editor/programming environment is used to fix errors and to test code out immediately.

Differentiating Between Debugging Methods - Q26-28

Students do not appear to be fully conscious of the debugging methods that they use, and what causes one method to be chosen rather than another.

Attitudes Towards the Search Mechanisms - Q30-33

Q30. That 5 students use the search mechanisms either rarely or never is very surprising. It is difficult to believe that any programmer neglects the search mechanisms to this degree, let alone (4 or) 5 out of 8 students. However, 28-30 students (out of 66) had never used the search mechanisms as 1st years, so perhaps this is just a consequence of their initial attitudes, which have not been superceded.

Q33. 7 (of the 8) students think that forward and backward search mechanisms should be provided - supporting Robertson, Davis, Okabe & Fitz-Randolph's (1990) findings. Namely, the reasons for using forward and backward search, and alternating between them is to aid in comprehension and debugging strategies. Especially when trying to get an idea of what the code does, and how.

"Live" Editors & Layout Style - Q34 & Q35

Q34. Opinions varied - 3 students think that a "live" editor like MacPascal's, saves time; and 3 think that it gets in the way sometimes (1 of these students voted for both). Eliminating errors at source gets 2 votes, plus 1 more for enabling immediate detection of bracketing errors.

Q35. Student 5 found the positioning of the blank lines in the experiment code confusing. Having the comments on separate lines rather than to the right of the code also threw him off balance. This is why I think a layout tool is so important - it could remove these obstructions to understanding other people's code, and make unfamiliar programs easier to grasp.

7.1.4 Amendments to Suggested Tools

Reading the students' responses to the new tool concepts brought several amendments to mind.

Summary Tables

For the summary tables, the amendments relate to viewing procedure calls and their parameter lists. The initial idea was to have all the names of the procedures on a list - either in alphabetical or declaration order. Choosing one procedure name would cause its parameter list to pop up on a child menu, in declaration order (squeezed to fit the window but otherwise unchanged).

Having the original procedure call's parameter list available for reference should enable a procedure call to be written out correctly first time. Having the original variable name and data type for reference as to which variable is needed and where (in what order). For example, having a parameter list with slots for 3 variables, "a", "b", and "c", gives 6 possible orderings (abc, bac, bca, cab, acb, cba). There is (usually) only one particular combination (ordering permutation) of these variables within the parameter list that will achieve the desired effect.

The initial concept was to have only one parameter list on view at a time. But this idea could be extended so that more than one parameter list could be chosen and displayed at a time. Or perhaps all parameter lists could be seen all at once. When viewing more than one parameter list at a time, it might be useful to put each parameter list called in a secondary window. Where the main window lists all procedure names, and the secondary window shows the chosen procedure's name and its parameter list.

In this way widely separated procedure names whose lists had been called would be viewable together. This could be useful for comparison of procedure calls when trying to decide between 2 alternative calls. Especially when choosing between predefined procedures (or functions). This strategy could also be applied to choosing between a predefined procedure or function (if they were listed together in the main window), when trying to decide on the more efficient option.

A further possibility would be to insert the chosen procedure call and parameter list into the code at the cursor position, ready for over-writing by the programmer. Or to have the procedure name and parameter list in a window for reference, and the chosen procedure name with an empty parameter list, ready to be filled at the cursor position.

The aim of course is to give the programmer more choice, so that he/she can choose the most useful format and volume of information. Whether it is selecting only one item of information, or comparing many and choosing only one of them.

Spotlighting

One idea is to maintain a "previously spotlighted" word list in addition to a "currently spotlighted" word list. So that the programmer knows which variables "have been" and "are currently" spotlighted at any time. This could be useful in helping the programmer remember which words/variables have already been investigated, and can perhaps be eliminated from the list of words needing further investigation.

The other idea pertains to spotlighting on a multi-colour VDU screen and colour printer.

On a colour system, the colour could indicate the precedence of each spotlighted variable. Having, say, the 1st word spotlighted in red, the 2nd word spotlighted in green, the 3rd in blue, and so on. In this way the programmer could rely on the colour continuity to mark the spotlights already chosen, and would use a new colour for the next word chosen for spotlighting. The number of colours or different words that could be spotlighted would then depend only on the number of different colours supported by the screen (or the colour printer, if choosing a hardcopy printout). So the colour would indicate the spotlighting precedence, and should help to maintain a debugging history - by defining the words/variables already under investigation, by the colours shown. The only other limit on spotlighting on a colour system would be the viewer's perceptual or visual capacity - when the number of colours used, and the density of spotlights exceeds human information processing limits.

7.1.5 Conclusions

The questionnaire was very useful in defining the students attitudes, and the aim of their debugging strategies, and the use of methods and tools in debugging.

Understanding the aim of reading strategies gives a much closer idea of the importance of developing a good model of the task. So that it can be used to accelerate debugging, avoiding having to go back to the original source - the task description, by internalising that information accurately.

Finding out where mental simulation fitted in regarding the debugging stages it is used in and why, and how popular it is gave me encouragement. I think it is a very neglected tool and subject area, simply because it is difficult to test for. But at least this questionnaire has made its importance real, and I hope helped to put it on the map of things that need to be looked into.

7.2 Debugging Experiments

The central part of the debugging experiment consisted of the students applying their debugging skills to 3 separate pieces of "90% complete" buggy Pascal code. These bugs were to be eliminated (by modifying the code) so that the code would compile, and be executed according to the task description provided. All 8 students did these debugging tasks, and the student identifiers remained the same for these tasks, and the spotlighting experiments (of §7.3) as well, to maintain continuity.

These tasks were not very complex, short (1-2 pages each), and contained helpful comments. The purpose of these debugging tasks was to find out exactly how the students went about the debugging task, and to get an idea of how long it took them to "correct" each set of errors, and which errors the students tackled first. Students were advised not to spend more than 15 minutes on each program.

Task 1: Primes program, Task 2: Letter Pairs program, Task 3: Concordance program.

7.2.1 Design of Experiments & Experimental Method(s)

Types of Errors Planted in The Code

- missing variable declarations;
- mis-declaration of variables (either in data type or type of procedural parameter);
- missing variable initialisation or re-initialisation statements;
- initializing or re-initialising a variable to the wrong value;
- using the wrong operators in an expression or assignment statement;
- using the wrong comparators in an expression or condition statement;
- wrong sequencing of code statements (or variable value modification statements);
- faulty conditional statements - logic (AND/OR) errors, or using wrong comparators;
- typos and "finger trouble" errors, such as initialising to 9 instead of 0, or incrementing by 11 instead of 1;
- not paying attention errors - writing out or modifying the wrong (variable's) value eg. writing "prime[num]" instead of "num" (see Primes code, Task 1).

These errors were chosen because they are common/typical errors that one would expect to find in a program under development.

Help Provided

A task or algorithm description was provided for each task, to give each student a good chance of getting a clear and detailed task model and to make debugging as straightforward as possible. Task 2 (Letter Pairs) had further hints on the algorithm and the format of the expected output. Task 3's (Concordance) help sheet gave specific help on pointers, and how to use them to make linked lists.

Appendix 7C contains all 3 pieces of code, with the corrections solutions (hand-written) on each sheet accompanied by the appropriate control error numbers; the task descriptions, and the help sheets given to the students.

Data Requested From Students

Students were asked to write down the start and end times for :-

- reading the task description;
- reading the algorithm hints (for Tasks 2 & 3); and
- debugging each task.

Students were also asked to write down a list of thoughts, actions, and strategies - a written "running commentary" as they went about debugging each task.

Each task description was provided with 2 sets of (15 slot) tick boxes arranged vertically on top of each other. One set for brief, and one set for thorough readings.

The students were asked to put a tick in either the brief or the thorough box next in the sequence. This was intended to capture the reading history of the task description, and to discover the pattern of reading depth and frequency for each task. Unfortunately, the students did not follow the instructions, so the reading history data was useless.

7.2.2 Student Debugging Strategies

Each student was asked to write down his debugging strategy as he went through the code looking for errors. The students were expected to write down something along the following lines (see list below) - accounting for each phase or step in the debugging strategy. The results were very patchy and incomplete. Some students just fixed the errors and made no other comments at all.

List of expected actions/strategies :-

- checking the algorithm line by line;
- checking begin-end loops;
- checking declarations - looking for missing variable names, or alternatively,
- checking each new variable name encountered with the declaration area
 - if it isn't in the declaration area then it isn't declared;
- checking that each variable is initialised correctly;
- checking that each variable is re-initialised correctly;
- checking that variable values are modified in the correct sequence;
- checking that the algorithm is written/executed in the correct sequence;

- checking each conditional statement to see that entry and exit (terminating) conditions are valid, and will be attainable when the program runs;
- reading through to get an idea of what the code does;
- comparing the actual code with the algorithm it is supposed to reflect
 - checking for discrepancies;
- mental execution of code;
- hand simulation of code;
- checking mental model of the task against the code as written.

7.2.2.1 Analysis of Students' Comments While Debugging

The following table is a distillation of the students' common debugging strategies extracted from the students' comment responses, that were written down while debugging the experiment code. The Common Characteristics were deduced from the actual amendments they made to the code itself, and to specific entries on the help sheets (where students ticked a box, or noted start and end times for reading the task description or algorithm hints).

<u>Summary of Student Comment Responses</u>	<u>Students</u>
<u>checking begin-ends</u>	P4 P7
<u>mental execution of code</u>	P4 L8
<u>try numbers in the algorithm</u>	P6 P7
<u>check variable definitions</u>	P6
<u>reading through with no particular strategy</u>	P6 L6 L7 P8 C8
<u>seeking to check & eliminate each section of the program</u>	P6
<u>looked at help information then continued reading and debugging</u>	L6
<u>read error hints and examined code for causes of each error</u>	C6 C7 C8
<u>textbook check up on pointers</u>	C7
<u>looked up syntax of write statements</u>	P6 P7
<u>looked up definition of a constant</u>	P7
<u>looked up definition of a statement</u>	P7

<u>Common Characteristics</u>	<u>Students</u>
<u>adding surplus begin-ends</u>	P1 P4 P5 P7 L7 P8
<u>checking task description or algorithm vs code</u>	A1 A2 P3 L3 A4 A5 A6 L7 C7 L8
<u>reading errors list</u>	A1 A2 A3 A4 C5 A6 A7 A8

Key to coded entries in table.

P = prime, L = letter pairs, C = concordance, A = all programs.

P5 indicates that Student 5 made that comment during debugging of the prime program, whereas A6 indicates that Student 6 made that same comment during debugging of all 3 programs.

As a programmer I always try to pare my code down to the bare minimum, both variable and construct wise. So that the code and usage of variables and data structures is efficient. Thus I tend to be very frugal with BEGIN-END loops especially around IF-ELSE statements.

The students reaction to this aspect of my programming style was to add unnecessary BEGIN—END loops. Regardless of the fact that they had already been told that all necessary BEGIN—END loops were present (but not necessarily in the right places). As a result, some students wasted valuable debugging time on this activity.

The summary of student responses show very little in the way of detailed debugging strategies. Most comments are very superficial and uninformative, so I have attempted to reconstruct a model of debugging strategy using the questionnaire results (see Appendix Q7A) as a guide.

7.2.2.2 Reconstruction of Student Debugging Strategies

Comprehension Strategies Defined by Questionnaire (Q7A)

The questionnaire defined the following 3 steps of comprehension with respect to the code and the task description as important.

Step 1 - (from Q16) students read and re-read the code/task description to understand/reaffirm the task requirements;

Step 2 - (from Q17) the students' aim is to get a total understanding of the code, and to fill in details (or refresh memory) by reading specific sections of the code, as needed;

Step 3 - (from Q18) the students' primary aim is to correlate the code and its structure with the task description.

Step 1 is almost compulsory, in order to get a good idea of how the code is expected to work, and the task carried out. Steps 2 and 3 are usually carried out (approximately) simultaneously with priority swapping from one to the other on a moment by moment basis - dependent on the complexity of (understanding) the code on its own, and the ease or difficulty of relating what the code does to the task (or algorithm) description.

Probable Debugging Strategy

Facts - The students have been told that there are 6 (or 7) errors in each program, and how each algorithm is supposed to work.

The algorithm expected comes from reading the task or algorithm description.

The algorithm as written/executed comes from reading the code itself.

Each student attempts to understand and relate the code to the algorithm and the task at hand - finding out what is going on in the code (as shown by Q18).

As the students read the code they are checking for compliance and deviation from the algorithm expected and the algorithm as written/executed in the code. So they become aware of discrepancies and "problem areas". This means that they can easily find themselves correcting "obvious" errors on the spot rather than waiting till later (as Q19 confirmed). Fixing errors on the spot may also result as an aspect of closure - needing to complete a task in order to gain relief, and to free up cognitive resources for the next task (Winfield 1986).

Error corroboration is usually needed for "problem areas", because some errors are more difficult to pinpoint than others. The most obvious errors are when the wrong operator has been used in a "simple" assignment such as `x := x + 1;` where the "+" should be a "-".

According to Suchman's (1987) theory there may have been a variation in the way the students tackled the debugging tasks as a result of the different types of information supplied.

With the first 2 tasks (the primes and letter pairs programs), the students were given the task description and algorithm hints. The problem was to find the discrepancies between the task/algorithm description and the code. So the students had an open mind, looking for "what is wrong" and were using forward or descriptive debugging. This is a proactive debugging approach.

With the third task (the concordance program), the students were given the task description and background information on how to use pointers to build linked lists. Plus the error hints and compiler messages. The problem was to find the errors/discrepancies by backtracking from the where the error became apparent (as defined by the error hints) back to its source - backward debugging. Trying to find out "what caused the error". In this case the students had definite information as to what was wrong and took a reactive approach.

7.2.3 Results of Experiments - Interpretation & Evaluation

Scoring the debugging experiments proved more problematic than expected, since the students were expected to a) follow instructions as directed, and b) only correct the errors planted in the code.

One of the main problems occurring with a) was that students insisted on adding unnecessary BEGIN-END loops (especially in the primes program - Task 1). Thus debugging time was wasted on this activity - and since students only recorded times for start and end of the debugging period, it was impossible to factor this wasted time out. Students also attempted to solve non-existent or spurious errors.

During debugging students assign an error number to each bug that is found - in the order that it is found/solved; but each student debugs the code in a different order. Therefore I had to assign a nominal number to each control error (or its solution) that I had planted in the text. For instance, if there are 7 errors, then the successive student defined errors are denoted by SE1 to SE7 inclusive; and the control error that each student error corresponds to (if any), is denoted by one of CE1 to CE7 inclusive. For example, say that :-

the student errors SE1 SE2 SE3 SE4 SE5 SE6 SE7 correspond to
the control errors CE5 CE3 CE6 CE1 CE7 CE2 CE4 respectively.

For the purposes of this experiment, each debugging answer was either right or wrong. The control error correctness factor was invented to deal with the problem of defining the degree of rightness or wrongness of the debugging solution, as follows.

Control Error Correctness Factor:

If an error hypothesis is correct it can easily be identified as one of CE1-CE7, say CE6 for example. When the error problem is stated, but the error itself was not solved it would be denoted as CE6½ (expressing a half-solved solution). In some cases the error hypothesis/solution is wrong, but is clearly related to one of the control errors, say CE6, and would be denoted as CE6X. When the error solution is completely spurious it is denoted as CEX. Thus it is easy to see which student errors turned out to be correct, semi-correct, or wrong but related to a specific control error, and those that are completely spurious.

The following tables correlate the reading times for the task description (TD.time) and code reading (CR.time) times, with the debugging times (DB.time) and their means (DB.mean); as well as correlating the student errors (SE1-7) in terms of the control errors (CE1-7). All times are in mm.ss (minutes.seconds).

Number of error solutions follows DB.mean directly (after the "x", as in "5.30x2"), to validate the total debugging time, DB.time. For example, on Task 1, SS1 had a total debugging time of 11.00 (DB.time), and solved 2 errors, so his mean debugging time was 5.30 (5minutes 30seconds).

Task 1: Primes Program

	TD.time	CR.time	DB.time	DB.mean	CE1	CE2	CE3	CE4	CE5	CE6	CE7	Spur
SS1	1.09	6.00	11.00	5.30x2	✓	✓						
SS2	—	3.25	5.00	2.30x2	SE1X							SE2X
SS3	1.05	2.35	15.22	2.12x7	SE5	SE3	SE6	SE1	SE2		SE4	SE7X
SS4	0.43	0.35	10.15	-	No errors found at all.							
SS5	0.50	2.00	9.15	1.51x5		✓		✓	✓	✓	✓	
SS6	0.40	1.50	19.25	4.51x4		SE2½		SE1	SE3½			SE4X
SS7	2.02	5.42	31.40	7.55x4		SE2		SE1		SE5		SE3X
SS8	1.00	7.30	9.30	1.54x5		SE3		SE5	SE2		SE1	SE4X

Task 2: Letter Pairs Program

	TD.time	CR.time	DB.time	DB.mean	CE1	CE2	CE3	CE4	CE5	CE6	Spur
SS1	0.48	3.18	8.28	8.28x1					SE1		
SS2	-	9.09	6.45	3.22x2					SE1		SE2X
SS3	0.38	2.50	21.55	3.39x6	SE1		SE6	SE2	SE3		SE4X & SE5X
SS4	1.05	3.45	3.50	0.46x5	✓		✓	✓	✓	✓	
SS5	1.35	4.00	21.00	4.12x5	✓		✓	✓	✓	✓	
SS6	0.30	1.40	13.00	2.10x6	SE1		SE2	SE5	SE3	SE4	SE6X
SS7	1.20	7.23	25.50	4.18x6	SE2			SE3	SE1	SE4	SE5X & SE6X
SS8	2.15	12.00	31.40	5.17x6	SE3		SE4	SE5	SE2	SE6	SE1X

Task 3: Concordance Program

	TD.time	CR.time	DB.time	DB.mean	CE1	CE2	CE3	CE4	CE5	CE6	Spur
SS1	0.49	2.01	8.15	-	No errors found at all.						
SS2	-	1.16	8.24	-	No errors found at all.						
SS3	0.43	9.20	30.00	6.00x5	SE1	SE5	SE2	SE4		SE3	
SS4	1.00	1.50	12.50	4.17x3	✓			√½		✓	
SS5	0.30	2.15	14.30	2.25x6	SE1		SE4	SE6	SE2	SE3	SE5X
SS6	0.30	4.50	4.25	0.53x5	SE1	SE3	SE4	SE5		SE2	
SS7	1.21	6.35	28.47	7.12x4	SE1		SE2	SE4		SE3	
SS8	1.40	16.20	42.30	8.30x5	SE1	SE4	SE2	SE5		SE3	

Key

TDtime = total task description reading time, CRtime = total code reading time,
 DBtime = total debugging time, DBmean = mean debugging time, Spur = Spurious Errors.
 All times in mm.ss, SE = student error, CE = control error. ✓ = unnumbered solution.
 ½ = error problem stated, but not solved. X = spurious error problem/solution.

There doesn't seem to be a specific correlation between mean debugging time and accuracy. But this may well have been obscured, if much time was spent on finding one specific error, or trying to understand an unfamiliar piece of code.

Checking The Correlation Between Control & Student Defined Errors

I wondered if there might be a common pattern in the order in which code was debugged. I expected the Concordance program to show a definite pattern, since the error hints / compiler messages were ordered with the same precedence that they would occur or become apparent when debugging normally (in real life).

Primes									Letter Pairs							
	SE1	SE2	SE3	SE4	SE5	SE6	SE7	?		SE1	SE2	SE3	SE4	SE5	SE6	?
CE1	-	-	-	-	1	-	-	1	CE1	2	1	1	-	-	-	2
CE2	-	2	2	-	-	-	-	2	CE2	-	-	-	-	-	-	-
CE3	-	-	-	-	-	1	-	-	CE3	-	1	-	1	-	1	2
CE4	3	-	-	-	1	-	-	1	CE4	-	1	1	-	2	-	2
CE5	-	2	1	-	-	-	-	1	CE5	3	1	2	-	-	-	2
CE6	-	-	-	-	1	-	-	1	CE6	-	-	-	2	-	1	2
CE7	1	-	-	1	-	-	-	1	Sp	1	1	-	1	2	2	-
Sp	-	1	1	2	-	-	1	-								

Concordance							
	SE1	SE2	SE3	SE4	SE5	SE6	?
CE1	5	-	-	-	-	-	1
CE2	-	-	1	1	1	-	-
CE3	-	3	-	2	-	-	-
CE4	-	-	-	2	2	1	$\frac{1}{2}$
CE5	-	1	-	-	-	-	-
CE6	-	1	4	-	-	-	1
Sp	-	-	-	-	1	-	-

Key

Sp indicates spurious errors defined by the student, and
 ? indicates student errors that were not numbered.

The first 2 tasks - the primes and letter pairs programs do not show any significant correlations between student and control error numbering.

For Task 3, the concordance program - student error numbering should have followed the control error numbering exactly. If there had been a specific progression in student errors, as they reacted to the error hints relating to CE1-6 directly, the top-left to bottom-right diagonal would have been numerically dense. As it happens, numbers are close to this diagonal, but there are only 2 "direct hits" out of 6. Namely CE1xSE1 scoring 5, and CE4xSE4 scoring 2. This indicates that students did not follow the hints in the order in which they were given.

One reason for this is the nature of the task and the algorithm. Students (especially novices) usually find difficulty with pointers. Their confidence and accuracy when developing and debugging code dealing with pointers only increase when they have much experience with them. A case of practice makes perfect. So this was not an easy task to set them.

7.2.4 Comments on Debugging Experiments

The first set of volunteers, students 1 & 2, did not have access to the help sheets for tasks 2 & 3, because it wasn't thought necessary. On checking their comments regarding task difficulty, it became apparent that they did not have sufficient information to be able to debug properly. The difference in the number of errors solved correctly between students 1 & 2, and the rest of the students makes it quite clear that a full understanding of the algorithm is crucial to efficient and accurate debugging.

7.2.5 Conclusions

Results of the debugging experiments were disappointing, especially after emphasizing how important it was for the students to write down their thoughts, actions and strategies as they went about the debugging task. These debugging tasks were expected/designed to draw out the student's debugging strategies in more detail, but the students only recorded superficial activities, on the whole.

One reason for the lack of debugging data may be that the students' were not fully conscious of their use of debugging strategies because it is operating under (full or partial) automatic processing; or the information may be either too volatile (with a fast decay time in memory), or non-verbal (held at the goal/intention level of cognition).

An alternative explanation may hinge on the fact that debugging is a "doing" (procedural) activity, and for the students to consciously examine what they are doing and attempt to write down a description interferes with (derails) the debugging process; since writing is a complex procedural activity itself.

It obviously requires a different approach to be able to get detail to the required level.

Protocol analysis or talk aloud during debugging would probably be a more useful technique to draw out this data. So that obscure or vague comments could be "questioned" and drawn out on the spot, while the subject is debugging, and the information is fresh and accessible.

The results of the questionnaire came in very useful, and helped to reconstruct a crude model of debugging strategy. That debugging is prompted by perceived discrepancies and is geared towards resolving discrepancies, and restoring the code to the correct or intended action is well-known. Where the mental urge is to correct the code immediately rather than waiting till later (Gray & Anderson 1987, Green, Bellamy & Parker 1987). Waiting until later opposes the usual "see it now, fix it now" strategy or predisposed debugging habit. The latter may also be an aspect of closure - needing to get something completed, and mentally over and done with.

7.3 Spotlighting Experiments

The aim of the spotlighting experiments was to determine the debugging time required to detect, locate and fix each individual error. There were 2 program debugging tasks for each "half" of the experiment: with plain unemphasized text for the letter count and shell sort programs; and 4 sets of typographically emphasized "spotlighted" text for each of the survey and bubble sort programs - with each spotlighted program having a different variable spotlighted within the text.

Variations of the same/similar program algorithms were used for both halves of the experiment, although the algorithms were disguised superficially to make the results comparable. However, the shell sort algorithm is a little more complicated than the one for bubble sort (as shown by the debugging task difficulty ratings at the end of Appendix 7D).

This experiment should yield more concrete data than the previous debugging tasks (of §7.2) since the experiments are geared specifically towards tabulating the time taken to debug each individual error, and can thus produce a more accurate "mean" debugging time value. Unfortunately, the sample size is rather small, with just 7 sets of data to study (from the same students as in §7.1 & §7.2). So the results will be indicative of the effects of different information formats viz plain or typographically enhanced program text, rather than representative.

Another factor is that the tools may need to be used/practised for some time, so that the users become familiar with their operation, and, as a consequence, fully aware of the ways in which they can be used. People need time to develop ways of using a tool efficiently, and understanding its strengths and weaknesses. For example, using spotlighting to detect errors of omission or misplacement. To offset this effect, hints were provided that outlined the ways that spotlighting could be used to detect errors of omission, such as missing declarations or missing initialisation statements. For example, if a variable name is spotlighted and it hasn't been declared, then it will not have a spotlight in the declaration area.

7.3.1 Definition of Hypotheses

Spotlighting

- 1) The use of spotlighting will reduce the time taken to debug a program.
- 2) The use of spotlighting will enhance the quality of interaction during debugging.
- 3) Spotlighting will prove useful in the debugging task(s).
- 4) Spotlighting will make debugging easier.

Layout Style

- 5) Having code in your own preferred layout style increases readability and comprehension.
- 6) Having code in your own preferred layout style makes debugging easier.
- 7) The greater the divergence between the code's layout and your preferred style, the more difficult it is to debug.

The **first hypothesis** requires a series of debugging tasks to be set up and timed on a series of partially complete chunks of programming code, that can be considered as being in the last phase of development and debugging. With comparable tasks done on plain code and on code that has all the features of the plain system plus spotlighting. Each piece of experimental code was accompanied by a description of the task it was to perform, and its input/output requirements, to maximize the students' (accurate) detection and correction of all programming errors planted in the code.

To counteract bias, the subjects were split into 2 groups, where one group worked on the plain code first, while the other group worked on the spotlighting code. After finishing both programs in the first experimental condition, the student was then given the other experimental condition to work on. Thus leaving the student with only 2 sets of code belonging to one experimental condition at a time, to avoid contamination of results (from copying the solutions, or comparing the solutions of one experimental condition against the other). Of course, the students may have remembered solution details from debugging in one experimental condition to the other.

The experimental method used to test the first hypothesis, depended on each student logging his own personal start and finish times for each of the following:

- reading the task description thoroughly;
- reading the code fragment thoroughly; and
- detecting and correcting each bug.

The **remaining hypotheses** required a series of subjective opinions to be elicited. These would be difficult to test for directly, other than by asking specific questions. This meant asking students to answer a questionnaire after the spotlighting experiments, to get their individual subjective responses to the spotlighting tool and the different aspects of layout style.

7.3.2 Design of Experiments

7.3.2.1 Experimental Options for Testing Spotlighting

There were 2 options for the spotlighting experiment. Either paper- or terminal-based experiments; plus a subjective questionnaire to answer after the experiment.

Each proposed experiment had the format <plain system> vs <plain + tools system> .

1. Paper experiment: Plain system has one code sheet, showing the code in its original form, a task description, and a grid for start and finish times of error detection/correction. Spotlighting system has a set of 4 code sheets, each sheet with the original code and a different one of the variables highlighted; a task description, and a grid for start and finish times of error detection/correction.

Advantages: minimal equipment, probably fast to implement.

Disadvantages: non-interactive, perhaps non-representative results.

2. Terminal based experiment: Plain system has very basic editing functions to enable code corrections to be made. Spotlighting system has all facilities of plain system, plus simple mechanisms to enable user to define the word that is to be spotlighted.

Advantages: interactive, results more representative - closer fit to the task.

Disadvantages: time and effort to implement and get working, learning curve to gain skills to implement it, and subjects will require practice with new tools to gain familiarity, skill and confidence in using them. Another problem is how to determine individual debugging times.

NB. On the experimental editing system there would be additional delays caused by invoking the spotlighting utility on each variable. But the plain editing system should have no such delays, since all variations are already built in and accounted for.

Decision

Reasons for deciding to perform paper experiments rather than interactive terminal based experiments are as follows:

1. speed of implementation, low resource requirements, expected high data yield.
2. ability to do a large group of timed experiments relying on each individual to time him/herself, rather than individual experiments timed by me, giving a greater quantity of data from a single time slot, rather than doing each experiment singly and accumulating experimental data one set at a time.
3. considering the time factor vs quality and quantity of data, paper based experiments won hands down.

7.3.2.2 Experimental Tasks

The experiment required 4 pieces of code: 2 for use as the "plain" system; and 2 for use as the "test" spotlighting system. The problem was to find 4 chunks of code that represented 2 sets of pair-wise comparable tasks. The solution was to have 2 code solutions, and to recast the surface structure eg. the variable names, but with minimal changes to the underlying semantics or algorithms.

Each plain debugging task consisted of 1-2 sheets of non-highlighted code laid out in a "standard" familiar form (similar to MacPascal). Each spotlighted debugging task had 4 versions of the same code, each one having a different one of the 4 variables spotlighted (each debugging program used only 4 variables).

7.3.3 Analysing Results Of Spotlighting Experiments

Fact - the programmer analyses each error problem, and writes the corresponding error solution on the program text. Either adding new code, or just moving lines or chunks of existing code around, and/or modifying existing code by using a combination of insertion, deletion and any appropriate copy/cut/paste operations. On paper text, moving a chunk of text is (usually) indicated by "boxing" the text that is to be moved, and "arrowing" where it is to be moved to (or giving the necessary line and/or page numbers instead).

Only 7 students did the spotlighting experiments (all previous volunteers).

Spotlighting data for analysis:

- difference between start and end time for each error/bug gives the debug time;
- the bug solution/correction written on the program text sheet, as per the usual debugging strategy outlined above.

Asking the students to log the start and end times for debugging each individual bug was intended to provide the most accurate debugging times, by eliminating (factoring out) any time spent on distractions between debugging episodes. (The latter was a confounding factor in the debugging experiments of §7.2).

As with the previous debugging tasks (of §7.2.3), the student assigns an error number (shown in the tables as SE1-SE7) to each bug that he finds - in the order that he finds/solves it; but each control error has a nominal number (CE1-CE7). Both numbers are related by the Error Correctness Factor (see §7.2.3) which can be applied to both sets of error notation (either the SE1-7 series, or the CE1-7 series).

Fig. 1 shows the successive debugging times for each student defined error, denoted by SE1 to SE7; and the control error that it corresponds to (if any), denoted by CE1 to CE7. All debugging times are in mm:ss - minutes and seconds.

Fig. 1 Correlating Student Debugging Times & Control Errors, With Means (in mm:ss)

SS1 Letter Count Mean = 1:06 SE 1 1:10 CE 3 SE 2 0:15 CE 1½ SE 3 1:52 CE 4	SS1 Survey Mean = 1:29 SV SE 1 0:30 CE 1½ w SE 2 1:30 CE 4 w SE 3 2:20 CE 2½ w SE 4 1:16 CE 6 w SE 5 1:40 CE 5 w SE 6 1:40 CE 3½ w	SS1 Shell Sort Mean = 2:52 SE 1 5:14 CE 2 SE 2 0:30 CE 6½	SS1 Bubble Sort Mean = 5:00 SV SE 1 5:00 CE 3X i
SS3 Letter Count Mean = 1:28 SE 1 0:35 CE 1 SE 2 1:47 CE 2 SE 3 0:15 CE 5 SE 4 1:20 CE 6 SE 5 1:08 CE 4 SE 6 3:41 CE 7	SS3 Survey Mean = 1:54 SV SE 1 2:56 CE 3 m SE 2 1:55 CE 6 w SE 3 1:03 CE 5 w SE 4 1:30 CE 1 s SE 5 1:23 CE 2 t SE 6 2:40 CE 4 t	SS3 Shell Sort Mean = 2:03 SE 1 1:30 CE 6 SE 2 0:40 CE 5 SE 3 0:20 CE 2 SE 4 4:25 CE 4 SE 5 2:40 CE 3 SE 6 3:59 CE 7 SE 7 0:47 CE 1½	SS3 Bubble Sort Mean = 1:43 SV SE 1 2:33 CE 3 i SE 2 0:45 CE 5 i SE 3 0:28 CE 6 i SE 4 2:47 CE 2 i SE 5 1:44 CE 2X in SE 6 1:17 CE X SE 7 2:29 CE 7 j
SS4 Letter Count Mean = 1:42 SE 1 1:30 CE 4 SE 2 1:05 CE 6 SE 3 2:05 CE 3 SE 4 1:50 CE 5 SE 5 2:00 CE 7 SE 6 1:40 CE 2	SS4 Survey Mean = 2:06 SV SE 1 3:45 CE 5 w SE 2 0:39 CE 3 w SE 3 1:55 CE 6 w	SS4 Shell Sort Mean = 2:11 SE 1 1:04 CE 6 SE 2 6:30 CE 7 SE 3 0:38 CE 5 SE 4 1:40 CE 2 SE 5 1:35 CE 3 SE 6 1:07 CE 4½ SE 7 2:45 CE ??	SS4 Bubble Sort Mean = 3:19 SV SE 1 3:05 CE 3 i SE 2 2:40 CE 4 i SE 3 1:40 CE 5 i SE 4 5:00 CE 2X i SE 5 4:10 CE X
SS5 Letter Count Mean = 1:18 SE 1 0:35 CE 1 SE 2 0:20 CE 3 SE 3 1:05 CE X SE 4 4:15 CE 4 SE 5 0:50 CE 6 SE 6 0:45 CE 5	SS5 Survey Mean = 2:08 SV SE 1 1:55 CE 5 w SE 2 0:30 CE 6 w SE 3 1:20 CE 4 w SE 4 0:50 CE 2 w SE 5 0:55 CE 3 w SE 6 7:20 CE 1 w	SS5 Shell Sort Mean = 1:56 SE 1 1:05 CE 6 SE 2 1:05 CE 7 SE 3 1:15 CE X SE 4 2:50 CE 4 SE 5 1:30 CE 3 SE 6 5:05 CE X SE 7 0:40 CE X	SS5 Bubble Sort Mean = 1:45 SV SE 1 2:25 CE 5 i SE 2 0:55 CE 7 j SE 3 1:10 CE 3 j SE 4 1:05 CE 4 in SE 5 2:50 CE 2 in SE 6 2:05 CE 6 in
SS6 Letter Count Mean = 0:58 SE 1 1:25 CE 3 SE 2 1:40 CE 4 SE 3 0:30 CE 5 SE 4 0:25 CE 6 SE 5 0:50 CE 2 SE 6 0:55 CE X	SS6 Survey Mean = 0:34 SV SE 1 0:25 CE 1 s SE 2 0:20 CE 2 t SE 3 0:45 CE 6 w SE 4 0:25 CE 3 m SE 5 0:42 CE 5 w SE 6 3:45 CE 4 w	SS6 Shell Sort Mean = 1:25 SE 1 1:25 CE 6 SE 2 1:36 CE 7 SE 3 1:25 CE 1 SE 4 1:55 CE 2 SE 5 0:25 CE 3 SE 6 1:46 CE 4	SS6 Bubble Sort Mean = 1:03 SV SE 1 0:50 CE 3 in SE 2 0:28 CE 5 in SE 3 0:53 CE 7 in SE 4 1:25 CE 2 in SE 5 0:45 CE 4 in SE 6 0:55 CE 6 in SE 7 2:05 CE X in
SS7 Letter Count Mean = 1:51 SE 1 1:33 CE 3 SE 2 1:08 CE 2½ SE 3 1:45 CE 4 SE 4 1:10 CE X SE 5 2:08 CE 6½ SE 6 3:19 CE X	SS7 Survey Mean = 2:37 SV SE 1 1:44 CE 2½ w SE 2 2:35 CE 6 w SE 3 1:20 CE 3½ m SE 4 0:30 CE 1½ s SE 5 5:20 CE 5 w SE 6 4:11 CE 4½ m	SS7 Shell Sort Mean = 1:54 SE 1 2:24 CE 7X SE 2 2:02 CE 6X SE 3 1:50 CE 2X SE 4 1:10 CE 5½ SE 5 1:26 CE 4 SE 6 2:30 CE X	SS7 Bubble Sort Mean = 1:20 SV SE 1 0:25 CE 4 t SE 2 0:24 CE 3 t SE 3 0:18 CE 5 t SE 4 1:32 CE 2X in SE 5 4:40 CE 7 in SE 6 0:55 CE 6X in SE 7 1:06 CE 7 j
SS8 Letter Count Mean = 1:13 SE 1 1:00 CE 2 SE 2 0:45 CE 3 SE 3 2:15 CE 4 SE 4 1:30 CE 5 SE 5 0:30 CE 6 SE 6 1:15 CE 7	SS8 Survey Mean = 2:35 SV SE 1 2:30 CE 6 w SE 2 0:45 CE 3 m SE 3 1:00 CE 1 s SE 4 2:00 CE 2 t SE 5 1:00 CE 5 w SE 6 8:15 CE 4 s	SS8 Shell Sort Mean = 4:31 SE 1 16:45 CE 3 SE 2 2:15 CE 5 SE 3 0:30 CE 2 SE 4 8:15 CE X SE 5 1:15 CE 6 SE 6 0:45 CE 7 SE 7 2:00 CE 1	SS8 Bubble Sort Mean = 1:18 SV SE 1 2:30 CE 3 t SE 2 1:45 CE 5 t SE 3 0:45 CE 4 t SE 4 1:00 CE 6 in SE 5 0:45 CE 7 t SE 6 1:05 CE 2 t SE 7 1:15 CE 1 j

Key CE - Control Error, SE - Student Defined Error, SV - spotlighted variable.
Qualifiers appended to error no.s - either X, ½, or first 1 (or 2) letter(s) of the variable's name to specify which spotlighted sheet was used for debugging.
Where X - means wrong error hypothesis; and
½ - means error problem defined but not solved; eg. "wait variable not declared".
Survey - spotlighting signal, time, wait & maxwait variables.
Bubble Sort - spotlighting i, j, temp & inorder variables.

Fig. 2 Master matrix defining the debugging time for each individual control error. Where student defined errors are denoted in debugging order as SE1, SE2, ... SE7; against a nominal ordering of the "control errors" planted in the code, denoted as CE1, CE2, ... CE7. The student error number has an inbuilt correctness component in relation to one of CE1...CE7. Where X denotes a spurious error, and $\frac{1}{2}$ denotes a statement of the error problem, such as "signal variable undeclared" or "wait variable not initialised", but without or instead of an error solution, such as "signal : integer;" or "wait := 0;" either placed or accompanied by an indication of where the missing statement should be placed. Those student error numbers that are shown without a qualifier, indicate a correct solution to that control error.

For example, in the Letter Count program: student 1 correctly defined/solved control error CE3, as his 1st debugging attempt, SE1, with a debugging time of 1:10secs; but only defined control error CE1, as his 2nd debugging attempt, SE2, with a debugging time of 0:15secs. Whereas student 7 defined/solved 2 spurious errors, SE4 and SE6, with debugging times of 1:10 and 1:08secs respectively.

LETTERCOUNT

	CE1	CE2	CE3	CE4	CE5	CE6	CE7	Spurious
SS1	SE2 $\frac{1}{2}$ 0:15		SE1 1:10	SE3 1:52				
SS3	SE1 0:35	SE2 1:47		SE5 1:08	SE3 0:15	SE4 1:20	SE6 3:41	
SS4		SE6 1:40	SE3 2:05	SE1 1:30	SE4 1:50	SE2 1:05	SE5 2:00	
SS5	SE1 0:35		SE2 0:20	SE4 4:15	SE6 0:45	SE5 0:50		SE3X 1:05
SS6		SE5 0:50	SE1 1:25	SE2 1:40	SE3 0:30	SE4 0:25		SE6X 0:55
SS7		SE2 $\frac{1}{2}$ 1:08	SE1 1:33	SE3 1:45		SE5 $\frac{1}{2}$ 2:08		SE4X 1:10 & SE6X 1:08
SS8		SE1 1:00	SE2 0:45	SE3 2:15	SE4 1:30	SE5 0:30	SE6 1:15	
	Mean 0:28	Mean 1:17	Mean 1:13	Mean 2:04	Mean 0:45	Mean 1:02	Mean 2:18	Mean 1:05

SURVEY - spotlighting signal, time, wait & maxwait variables

	CE1	CE2	CE3	CE4	CE5	CE6	Spurious
SS1	SE1 $\frac{1}{2}$ 0:30w	SE3 $\frac{1}{2}$ 2:20w	SE6 $\frac{1}{2}$ 1:40w	SE2 1:30w	SE5 1:40w	SE4 1:16w	
SS3	SE4 1:30s	SE5 1:23t	SE1 2:56m	SE6 2:40t	SE3 1:03w	SE2 1:55w	
SS4			SE2 0:39w		SE1 3:45w	SE3 1:55w	
SS5	SE6 7:20w	SE4 0:50w	SE5 0:55w	SE3 1:20w	SE1 1:55w	SE2 0:30w	
SS6	SE1 0:25s	SE2 0:20t	SE4 0:25m	SE6 3:45w	SE5 0:42w	SE3 0:45w	
SS7	SE4 $\frac{1}{2}$ 0:30s	SE1 $\frac{1}{2}$ 1:44w	SE3 $\frac{1}{2}$ 1:20m	SE6 $\frac{1}{2}$ 4:11m	SE5 5:20w	SE2 2:35w	
SS8	SE3 1:00s	SE4 2:00t	SE2 0:45m	SE6 8:15s	SE5 1:00w	SE1 2:30w	
	Mean 1:53	Mean 1:26	Mean 1:14	Mean 3:37	Mean 2:14	Mean 1:38	

SHELLSORT

	CE1	CE2	CE3	CE4	CE5	CE6	CE7	Spurious
SS1		SE1 5:14				SE2 $\frac{1}{2}$ 0:30		
SS3	SE7 $\frac{1}{2}$ 0:47	SE3 0:20	SE5 2:40	SE4 4:25	SE2 0:40	SE1 1:30	SE6 3:59	
SS4		SE4X 1:40	SE5X 1:35	SE6 $\frac{1}{2}$ 1:07	SE3 0:38	SE1 1:04	SE2 6:30	SEX 2:45
SS5		SE6X 5:05	SE5 1:30	SE4 2:50	SE3X 1:30	SE1 1:05	SE2 1:05	SE7X 0:40
SS6	SE3 1:25	SE4 1:55	SE5 0:25	SE6 1:46		SE1 1:25	SE2 1:36	
SS7		SE3 $\frac{1}{2}$ 1:50		SE5 1:26	SE4 $\frac{1}{2}$ 1:10	SE2X 2:02	SE1X 2:24	SE6X 2:30
SS8	SE7 2:00	SE3 0:30	SE1 16:45		SE2 2:15	SE5 1:15	SE6 0:45	SE4X 8:15
	Mean 1:24	Mean 2:22	Mean 4:35	Mean 2:19	Mean 1:15	Mean 1:16	Mean 2:43	Mean 3:43

BUBBLECOUNT - spotlighting i, j, temp & inorder variables

	CE1	CE2	CE3	CE4	CE5	CE6	CE7	Spurious
SS1			SE1X 5:00i					
SS3	SE6X 1:17in	SE4 2:47i	SE1 2:33i		SE2 0:45i	SE3 0:28i		SE7X 2:20in
SS4		SE4X 5:00i	SE1 3:05i	SE2 2:40i	SE3 1:40i			SE5X 4:10i
SS5		SE5 2:50in	SE3 1:10j	SE4 1:05j	SE1 2:25i	SE6 2:05in	SE2 0:55i	
SS6		SE4 1:25in	SE1 0:50in	SE5 0:45in	SE2 0:28in	SE6 0:55in	SE3 0:53SEin	SE7X 2:05in
SS7	SE7X 1:06j	SE4X 1:32in	SE2 0:24t	SE1 0:25t	SE3 0:18t	SE6X 0:55in	SE5 4:40in	
SS8	SE7 1:15t	SE6 1:05in	SE1 2:30j	SE3 0:45t	SE2 1:45t	SE4 1:00t	SE5 0:45t	
	Mean 1:13	Mean 2:27	Mean 1:13	Mean 1:08	Mean 1:14	Mean 1:05	Mean 1:48	Mean 2:52

The Survey and Bubble Sort programs were used to test the spotlighting concept. Thus each program had 4 sets of program text. Each sheet having a different variable spotlighted. The first 1 (or 2) letter(s) of the variable name were used to indicate which sheet was used to "solve" each bug. Thus Student 1's (SS1) first error, SE1, was denoted CE1½w. This indicates that SE1 corresponds to CE1, and ½ to the fact that the solution was semi-correct (error problem stated but not its solution), and the fact that it operated on the sheet where the "wait" variable was spotlighted. Mean debug times are listed for each student per program.

Fig. 2 shows debug times with respect to control error ordering, CEs vs SEs (control errors vs student errors). This enables a correlation of debugging times for all students and for each individual control error. The error correctness factor was applied to student defined errors in the same way as to control errors. The debugging time, student error numbers vs control error matrix has an additional factor for the spotlighted program text that was debugged. In these cases the letter following the debugging time indicates which of the variables was spotlighted on the sheet that the debugging strategy acted on. Mean debug times are listed for each control error.

For example, in the Letter Count program: student 1 correctly defined/solved control error CE3, as his 1st debugging attempt, SE1, with a debugging time of 1:10secs; but only defined control error CE1 (thus CE1½), as his 2nd debugging attempt, SE2, with a debugging time of 0:15secs. Whereas student 7 defined/solved 2 spurious errors, SE4 and SE6, with debugging times of 1:10 and 1:08secs respectively.

Appendix 7D contains copies of the program text used for the spotlighting experiments, with accompanying task descriptions. Each piece of program text has been 100% debugged, and shows the solutions (hand-written) on each sheet, with the appropriate control error numbers. These sheets show (approximately) what each sheet was expected to look like after it had been fully debugged. With the spotlighting sheets, some control error solutions may appear on more than 1 sheet. Usually as a result of a sequencing error which affects 2 or more variables. Appendix 7D also contains a separate list of control errors and their solutions for each piece of code.

Fig. 3 shows the mean debug time for each student and each program, plus the total debugging time per program. Thus each student's debug time can be compared to his own mean overall debug time, or the mean overall time for that particular program.

Student 8 spent 16:45s on his first error in the Shell Sort program. This is way beyond the expected debug time - being a factor of approximately 10 times slower than usual. Discounting the 16:45s debug episode gives a mean debug time of 2:30, and including it pushes the mean up to 4:32, almost double the debug time. His own personal mean debug time is 1:52 (excluding the 16:45s) - which indicates that this value is a one-off event. The table takes account of this by giving mean values including the 16:45 value,

followed by a "/" and the mean value excluding this value. (All times are in mm:ss, minutes:seconds.)

Fig. 3 Mean Debugging Times Per Student Per Program

	Letter Count	Survey*	Shell Sort	Bubble Sort*	Overall
Student 1	1:06	1:29	2:52	5:00	1:55
Student 3	1:28	1:54	2:03	1:43	1:48
Student 4	1:42	2:06	2:11	3:19	2:18
Student 5	1:18	2:08	1:56	1:45	1:47
Student 6	0:58	0:34	1:25	1:03	1:07
Student 7	1:51	2:37	1:54	1:20	1:54
Student 8	1:13	2:35	4:32/2:30	1:18	2:27/1:52
Total Time	54:06	77:04	100:33/83:48	69:54	301:37/284:52
Mean Time	1:23	1:59	2:24/2:03	1:45	1:53/1:47

Where * indicates the spotlighted programs.

The overall means for each program in ascending order of mean debugging times gives Letter Count, Bubble Sort, Survey and Shell Sort. Faster debugging times indicate "easier" error problems, or errors that are relatively easy to detect and solve.

If spotlighting improves debugging then debug times for errors involving a spotlighted variable should be less than those for a non-spotlighted variable.

The difference between mean debugging times for plain vs spotlighted tasks are:

Letter Count mean is 36secs less than Survey mean;

Shell Sort mean is 18-39secs more than Bubble Sort mean.

For Letter Count 6 students' mean debugging time was less than for Survey, but 1 was higher.

For Shell Sort 2 students' mean debugging time was less than for Bubble Sort, and 5 were higher.

Thus for the Letter Count/Survey pairing, spotlighting is faster in 1/7 cases; and for the Shell Sort/Bubble Sort pairing, spotlighting is faster in 5/7 cases. Thus, overall, spotlighting is faster in 6/14 cases.

Taking into account the fact that the Shell Sort algorithm was slightly more difficult to grasp than the one for Bubble Sort; leads to the opinion that spotlighting did not give any significant reduction in debugging time. Thus the first hypothesis is not proved true.

Fig. 4 Debugging Times in Ascending Order For Each Student, With Totals

Student 1	Student 3	Student 4	Student 5	Student 6	Student 7	Student 8
0:15 CEL 1½	0:15 CEL 5	0:38 CESH 5	0:20 CEL 3	0:20 CES 2t	0:18 CEB 5t	0:30 CEL 6
0:30 CES 1½w	0:20 CESH 2	0:39 CES 3w	0:30 CES 6w	0:25 CEL 6	0:24 CEB 3t	0:30 CESH 2
0:30 CESH 6½	0:28 CEB 6i	1:04 CESH 6	0:35 CEL 1	0:25 CES 1s	0:25 CEB 4t	0:45 CEB 4t
1:10 CEL 3	0:40 CESH 5	1:05 CEL 6	0:40 CESH X	0:25 CES 3m	0:30 CES 1½s	0:45 CEB 7t
1:16 CES 6w	0:45 CEB 5i	1:07 CESH 4½	0:45 CEL 5	0:25 CESH 3	0:55 CEB 6Xin	0:45 CEL 3
1:30 CES 4w	0:47 CESH 1½	1:30 CEL 4	0:50 CEL 6	0:28 CEB 5in	1:06 CEB 7j	0:45 CES 3m
1:40 CES 3½w	1:03 CES 5w	1:35 CESH 3	0:50 CES 2w	0:30 CEL 5	1:08 CEL 2½	0:45 CESH 7
1:40 CES 5w	1:08 CEL 4	1:40 CEB 5i	0:55 CEB 7i	0:42 CES 5w	1:10 CEL X	1:00 CEB 6in
1:52 CEL 4	1:17 CEB Xin	1:40 CEL 2	0:55 CES 3w	0:45 CEB 4in	1:10 CESH 5½	1:00 CEL 2
2:20 CES 2½w	1:20 CEL 6	1:40 CESH 2	1:05 CEB 4j	0:45 CES 6w	1:20 CES 3½m	1:00 CES 1s
5:00 CEB 3Xi	1:23 CES 2t	1:50 CEL 5	1:05 CEL X	0:50 CEB 3in	1:26 CESH 4	1:00 CES 5w
5:14 CESH 2	1:30 CES 1s	1:55 CES 6w	1:05 CESH 6	0:50 CEL 2	1:32 CEB 2Xin	1:05 CEB 2t
22:57	1:30 CESH 6	2:00 CEL 7	1:05 CESH 7	0:53 CEB 7in	1:33 CEL 3	1:15 CEB 1j
	1:44 CEB 2Xin	2:05 CEL 3	1:10 CEB 3j	0:55 CEB 6in	1:44 CES 2½w	1:15 CEL 7
	1:47 CEL 2	2:40 CEB 4i	1:15 CESH X	0:55 CEL X	1:45 CEL 4	1:15 CESH 6
	1:55 CES 6w	2:45 CESH ??	1:20 CES 4w	1:25 CEB 2in	1:50 CESH 2X	1:30 CEL 5
	2:29 CEB 7j	3:05 CEB 3i	1:30 CESH 3	1:25 CEL 3	2:02 CESH 6X	1:45 CEB 5t
	2:33 CEB 3i	3:45 CES 5w	1:55 CES 5w	1:25 CESH 1	2:08 CEL 6½	2:00 CES 2t
	2:40 CES 4t	4:10 CEB Xi	2:05 CEB 6in	1:25 CESH 6	2:24 CESH 7X	2:00 CESH 1
	2:40 CESH 3	5:00 CEB 2Xi	2:25 CEB 5i	1:36 CESH 7	2:30 CESH X	2:15 CEL 4
	2:47 CEB 2i	6:30 CESH 7	2:50 CEB 2in	1:40 CEL 4	2:35 CES 6w	2:15 CESH 5
	2:56 CES 3m	48:23	2:50 CESH 4	1:46 CESH 4	3:19 CEL X	2:30 CEB 3t
	3:41 CEL 7		4:15 CEL 4	1:55 CESH 2	4:11 CES 4½m	2:30 CES 6w
	3:59 CESH 7		5:05 CESH X	2:05 CEB Xin	4:40 CEB 7in	8:15 CES 4s
	4:25 CESH 4		7:20 CES 1w	3:45 CES 4w	5:20 CES 5w	8:15 CESH X
	48:33		44:40	28:00	47:25	16:45 CESH 3
						63:35/46:50
22:57 Total	48:33 Total	48:23 Total	44:40 Total	28:00 Total	47:25 Total	63:35/46:50 Total
1:55 Mean	1:48 Mean	2:18 Mean	1:47 Mean	1:07 Mean	1:54 Mean	1:53/1:47 Mean

Key

CE - means Control Error for each program;

redefined as CEB, CEL, CES & CESH referring to Bubble Sort, Letter Count, Survey & Shell Sort respectively.

Qualifiers appended to error no.s - either X, ½, or first 1 (or 2) letter(s) of the variable's name to specify which spotlighted sheet was used for debugging.

Where X - means wrong error hypothesis; and

½ - means error problem defined but not solved; eg. "signal variable not declared".

Survey - spotlighting signal, time, wait & Maxwait variables.

Bubble Sort - spotlighting i, j, temp & inorder variables.

**Fig. 5 Debugging Times in Ascending Order For Each Program
With Totals and Means in mm:ss**

Letter Count	Survey	Shell Sort	Bubble Sort	
0:15	0:20	0:20	0:18	
0:15	0:25	0:25	0:24	
0:20	0:25	0:30	0:25	
0:25	0:30	0:30	0:28	
0:30	0:30	0:38	0:28	
0:30	0:30	0:40	0:45	
0:35	0:39	0:40	0:45	
0:35	0:42	0:45	0:45	
0:45	0:45	0:47	0:45	
0:45	0:45	1:04	0:50	
0:50	0:50	1:05	0:53	
0:50	0:55	1:05	0:55	
0:55	1:00	1:07	0:55	
1:00	1:00	1:10	0:55	
1:05	1:03	1:15	1:00	
1:05	1:16	1:15	1:05	
1:08	1:20	1:25	1:05	
1:08	1:20	1:25	1:06	
1:10	1:23	1:26	1:10	
1:10	1:30	1:30	1:15	
1:15	1:30	1:30	1:17	
1:20	1:40	1:35	1:25	
1:25	1:40	1:36	1:32	
1:30	1:44	1:40	1:40	
1:30	1:55	1:46	1:44	
1:33	1:55	1:50	1:45	
1:40	1:55	1:55	2:05	
1:40	2:00	2:00	2:05	
1:45	2:20	2:02	2:25	
1:47	2:30	2:15	2:29	
1:50	2:35	2:24	2:30	
1:52	2:40	2:30	2:33	
2:00	2:56	2:40	2:40	
2:05	3:45	2:45	2:47	
2:08	3:45	2:50	2:50	
2:15	4:11	3:59	3:05	
3:19	5:20	4:25	4:10	
3:41	7:20	5:05	4:40	
4:15	8:15	5:14	5:00	
		6:30	5:00	
		8:15		
		16:45		
54:08	77:04	100:33/83:48	69:54	<u>Overall Total</u> 301:37/284:52
<u>Mean</u>	<u>Mean</u>	<u>Mean</u>	<u>Mean</u>	<u>Overall Mean</u>
1:23	1:59	2:24/2:03	1:45	1:53/1:47

If spotlighting really reduces debugging times, then debugging times should have been faster on the letter count vs survey program comparison, as that pair of programs were of the same difficulty, with analogous deep structure algorithms. So, spotlighting may have reduced debugging times, but the result is not sufficiently clear cut to give a definite answer.

Fig. 4 shows the debug times for each student in ascending order. Fig. 5 shows the distribution of debug times for each program, also in ascending order. The mean debug times of Figs 4 & 5 were fed into the table of Fig. 3, for comparison.

In Fig. 6, each matrix defines the ordering of student debugging attempts SE1 to SE7 in relation to the control errors CE1 to CE7 for one piece of code. For example, with the Letter Count program, control error 3, CE3, was solved 3 times as SE1, 2 times as SE2, and 1 time as SE3.

Control errors were numbered in accordance with their (strictly top-bottom textual) position in the code. Thus CE1 appears first in the code, followed by CE2, ..., and CE7 denotes the last error appearing near the end of the code.

It is interesting to see that there is a broad band of numbers appearing on or near the top-left to bottom right diagonal in Letter Count, whereas the Shell Sort matrix has a numerically dense bottom-left to top-right diagonal. The former might indicate that the students solved the Letter Count errors in a top-bottom approach, and in the latter in a bottom-up approach.

Fig. 6 Checking The Correlation Between Control & Student Defined Errors

Letter Count								Survey							
	SE1	SE2	SE3	SE4	SE5	SE6	SE7		SE1	SE2	SE3	SE4	SE5	SE6	
CE1	2	1	-	-	-	-	-	CE1	2	-	1	2	-	1	
CE2	1	2	-	-	1	1	-	CE2	1	1	1	2	-	-	
CE3	3	2	1	-	-	-	-	CE3	1	2	1	1	1	1	
CE4	1	1	3	1	1	-	-	CE4	-	1	1	-	-	4	
CE5	-	-	2	2	-	1	-	CE5	2	-	1	-	4	-	
CE6	-	1	-	1	3	-	-	CE6	1	3	2	1	-	-	
CE7	-	-	-	-	1	2	-								
Sp	-	-	1	1	-	2	-								

Shell Sort								Bubble Sort							
	SE1	SE2	SE3	SE4	SE5	SE6	SE7		SE1	SE2	SE3	SE4	SE5	SE6	SE7
CE1	-	-	1	-	-	-	2	CE1	-	-	-	-	-	1	2
CE2	1	-	3	2	-	1	-	CE2	-	-	-	4	1	1	-
CE3	1	-	-	-	4	-	-	CE3	5	1	1	-	-	-	-
CE4	-	-	-	2	1	2	-	CE4	1	1	1	1	1	-	-
CE5	-	2	2	1	-	-	-	CE5	1	3	2	-	-	-	-
CE6	4	2	-	-	1	-	-	CE6	-	-	1	1	-	3	-
CE7	1	3	-	-	-	2	-	CE7	-	1	1	-	2	-	-
Sp	-	-	-	-	-	-	2	Sp	-	-	-	-	1	-	1

NB. Sp indicates spurious errors defined by the student.

Fig. 7 shows 4 matrices. The 2 on the left show the control error and the name of the spotlighted variable that that error was debugged on. Thus for the Survey program CE1 was debugged on 2 wait and 4 signal sheets. However, 2 students did not swap between sheets - they used only 1 sheet for all the debugging. This leaves a problem - whether to discount the single sheet debugging times for the program concerned, or to ignore it only when it refers to a different variable that spotlighting should have helped with.

Fig. 7 Checking Which Spotlighted Sheet Used for Each Individual Error									
Survey					After discounting single sheet solutions				
	wait	maxwait	signal	time	wait	maxwait	signal	time	score correct sheet
CE1	2w		4s				4s		4/4 signal
CE2	3w			3t	1w			3t	3/4 time
CE3	3w	4m				4m			4/4 maxwait
CE4	3w	1m	1s	1t	1w	1m	1s	1t	1/4 signal
CE5	7w				4w				4/4 wait/maxwait
CE6	7w				4w				4/4 wait
									20/24 = 83%

Bubble Sort					After discounting single sheet solutions				
	i	j	temp	inorder	i	j	temp	inorder	score correct sheet
CE1		1j	1t	1in		1j	1t	1in	1/3 temp
CE2	2i			4in	1i			3in	3/4 inorder
CE3	3i	2j	1t	1in	2i	2j	1t		4/4 i/j
CE4	1i	1j	2t	1in		1j	2t		3/3 i/j
CE5	3i		2t	1in	2i		2t		4/4 i/j/temp
CE6	1i		1t	3in	1i		1t	2in	1/4 i
CE7	1i		1t	2in	1i		1t	1in	1/3 i
Sp	1i			2in				2in	
									17/25 = 68%

Thus for the Survey program the correct sheet was used in 83% of cases, and in 68% of cases for the Bubble Sort program.

In some case more than 1 spotlighted sheet could be used to detect an error, as with sequencing errors involving 2 or more variables.

Looking at the range of variables/spotlighting sheets relevant in identifying an error :-

Survey Program			Bubble Sort Program		
Variable	Relevant Errors	Ratio	Variable	Relevant Errors	Ratio
signal	CE1 & CE4	2/7	i	CE3, CE4, CE5, CE6 & CE7	5/7
time	CE2	1/7	j	CE3, CE4 & CE5	3/7
maxwait	CE3 & CE5	2/7	temp	CE1 & CE5	2/7
wait	CE5 & CE6	2/7	inorder	CE2	1/7

For Bubble Sort, sheets i and j have more chance of being useful than usual (5/7 and 3/7 respectively) whereas with the Survey program each sheet was usually useful for 2

errors at most. However, the nature of the sorting task has upped the odds in favour of i. Thus any student who used the i sheet first off (or throughout) had a head start on any student starting with another sheet.

In this regard, the Survey problem gave spotlighting a more even chance of each sheet being used and solving the relevant 1 or 2 errors on it, before moving to another sheet to solve the next 1 or 2 errors.

7.3.4 Discussion

The Survey program seemed better suited to a spotlighting solution since all variables had at least 4 letters in their names^{*}, and the algorithm was simple and relatively straightforward compared to the sorting tasks. With Bubble Sort, the temp and inorder spotlights were much more eyecatching than the i & j spotlights since the former were 4-6 characters long, and i & j were only 1 character long. So, this was not a good choice of variable names to show spotlighting off well. This problem arose because the algorithms used in the experiments are the original ones learned in my undergraduate programming course. Also, I almost always use i and j as index counters for FOR loops. It is an ingrained habit with me, (and other programmers, eg. Wiedenbeck's 1986 sorting programs also use i and j indices) and it didn't occur to me that they would not be suitable variable names to test spotlighting on.

However, this oversight has shown up one of spotlighting's drawbacks. This means either altering the way that spotlighting is supposed to work on 1-2 letter words. Such as adding an inverse video space before and after the 1-2 letter word. This could also be done for all spotlighted words - it would solve that problem, but it might alter the indentation or code layout. This alteration to the visual appearance of the code, by adding one space to each side of the variable name, may make correct code look wrong, or wrong code look right.

These latter effects could well be counter productive, and sabotage spotlighting's usefulness. Another alternative might be to inverse video whatever characters appeared 1 space to either side of these 1-2 letter words. Thus keeping the code appearing the same line-wise but focussing attention on the intended variable. But even this could have drawbacks. Another possibility would be to inverse video the entire line holding the 1-2 letter word(s) and to put a "flash—code" or "cancel inverse video code" on each 1-2 letter word (doing spotlighting in reverse). Even so the visual stimulus may still be almost negligible for words less than 3 characters long.

^{*} In Bubble Sort, the "i", "j", "temp", and "inorder" variables were spotlighted. In Survey, the "signal", "time", "wait", and "maxwait" variables were spotlighted.

7.3.5 Testing Spotlighting Using Paper Experiments

The main problem was that with automatic spotlighting on a VDU screen the students would have only one point of reference at a time - the screen itself. Whereas indicating the same effect on paper required one spotlighted variable per sheet of paper to indicate "current screen appearance". Thus producing multiple (4) sheets with the same text but with a different variable being spotlighted on each.

One or two students got bored with changing between sheets of paper (and confining themselves to addressing only those errors associated with the spotlighted variable), and simply used one sheet of program text to debug all the errors; thus defeating the whole point of the experiment. I feel that this was due entirely to the usage of a paper experiment - it would have been less likely on a VDU screen-based experiment where the screen would have reflected the cumulative modifications at each stage. Of course, the same effect is not possible with multiple sheets of the same text.

7.4 Results of Post-Spotlighting Questionnaire

The aim was to find out how each student felt about the spotlighting tool in detail. As well as comparing Q37 before and after answers, to see whether there was any change in the order of features. The latter questions referred to particulars about layout preferences. The questions were geared towards testing out hypotheses 2) to 7) inclusive, and gathering any other information that might prove relevant.

Appendix 7B contains the (semi-raw) that is summarized below. The hypotheses are listed in §7.3.1.

7.4.1 Summary of Spotlighting Responses

Q37. Comments after trying the spotlighting concept out were more enthusiastic than beforehand (see previous results of Q37 at rear of Appendix 7A). Students seem to have grasped the uses of spotlighting quite well, since 5 out of 7 of them have commented on how they help to find different errors associated with variables.

Q1. 4 out of 7 students had something positive to say about spotlighting, with only 1 out of 7 students expressing a negative view.

Q2. Defining the ordering of 9 spotlighting features.

1. Focussing attention on a specific item
2. Keeping track of all the locations that involve the specified item
3. Trail following
4. Associating an error with a particular variable
5. Confirming the location of a hypothesized error
6. Showing up variable (non-)declaration errors
7. Showing up (non-)initialisation errors
8. Showing up variable value modification errors
9. Showing up sequencing errors

Q2a. Order of ascending scores gives - most useful 1/6, 2, 7, 3, 5/8, 4, 9 least useful.

Q2b. In contrast, asking students to rank task aspects in order of 9 "slots" of importance (thus eliminating "tied" positions) gave a different ordering altogether.

Order of importance by ascending sum value - most 7,1,4,3,8,6,9,2,5 least important.

On reflection, the Q2a result is probably more accurate, since it depended on a strictly tri-part (yes, maybe or no) choice for each aspect.

However, focussing attention on a specific item, showing up (non-)initialisation errors, showing up variable (non-)declaration errors, and trail following all appear within the first 6 items on both lists. Indicating that these are the most important features that spotlighting offers.

These features all make debugging easier, so this finding supports hypothesis 4).

Q3. Did you find spotlighting helpful in the debugging task?

A positive (Yes) response from 5 out of 7 students was encouraging.

Responses to the previous 3 questions show that spotlighting is regarded as useful in the debugging task. Thus hypothesis 3) is supported.

Q4. Did spotlighting, as applied to this task, conflict with your natural debugging strategy?

I didn't expect spotlighting to interfere with anyone's debugging strategy; so the 1 yes vote (against 6 no votes) was rather disconcerting.

Q5. Did the benefits of spotlighting outweigh this latter consideration?

Similarly, I expected all answers to be yes (not just 2 out of 5); 1 maybe and 2 noes went against my expectations. There were no further comments to clarify the answers.

Q6. If you had 2 alternative debugging strategies, one sequence including spotlighting and the other using your own strategies, which would you choose, and why?

3 students (SS-3,5,6) definitely go for spotlighting plus their own debugging strategies; and 2 (SS-4,8) for their own strategies only; and 1 (SS7) tends towards his own strategies only, but does not rule spotlighting definitely in or out. In the comments, student 8 changes his mind, and decides that spotlighting could be useful after all.

Thus there are between 3 and 5 students who would use spotlighting in addition to their own debugging strategies.

Q7. Would the form of debugging medium affect your decision? For example,

a) would you use spotlighting on a paper system

b) on a screen editing system

The data shows that out of 7 students, 4 would and 2 might use spotlighting if it was an on-screen tool. Whereas out of 5 students, 1 would and 3 might use spotlighting on paper. This is probably the reason why the spotlighting experiments did not get as favourable a reception as expected.

These results indicate that spotlighting's quality of interaction - namely speed and on the spot interaction or usefulness would be greater with a screen-based system, than on paper. **So this supports hypothesis 2) for a screen system, but not for a paper system.**

Of course, the spotlighting tool would have to be tested on a screen system to confirm this indication.

7.4.2 Summary of Layout Responses

Q8. Students did not seem to be hindered by the style of layout used to present the debugging tasks, since 6 out of 7 students rated the difference(s) in layout style between their own, and those used in the experiments as "slight".

Q9. Students gave several reasons for having code in their own preferred style :-

Obviously, readability and being able to line up loops (thus defining scoping control) are important features for comprehension and accurate debugging, scoring 3 and 2 votes respectively. Student 5 thinks that is easier to follow code in your own preferred style. **These answers lend support to hypothesis 5).**

Q10. Having code in your own preferred style: 2 students think that it is easier to follow the flow of the program and helps debugging, while another thinks that it is easier to spot errors in a preferred style. **So this makes 3 votes overall (out of 5), supporting hypothesis 6) and the need for a layout tool.**

Q11. Ranking ease of error spotting in preferred, standard or other layout style. Votes were :-

6 votes for "easier" in a preferred layout style;

6 votes for "average" in a standard layout; and

7 votes for "more difficult" in a non-standard layout.

I think these answers clearly show that it is easier to debug a program if it is in the programmer's own style of layout. These results also show that it becomes progressively more difficult to spot errors as the stylistic differences increase. Thus supporting hypothesis 7.

Q12. The data shows a definite preference (5 to 1 to 1 respectively) for semi-automatic layout over fully automatic or fully manual layout in editors or programming systems. The comment responses reflect the students' need for firm/precise control over layout and its tailorable aspects.

Q13. Votes for editing/programming systems are: 2 for vi, 2 for MacPascal and 3 for semi-automatic. However the 2 students (SS5 & 6) who voted for vi, wish that vi were easier to use.

Q14. There are 3 students who prefer layout support with override facility. Whereas students 7 & 8 don't mind what type of editor, as long as the code ends up in their own preferred style.

However, combining the responses of Q13 and Q14, give the following results:-

<u>Responses</u>	<u>Students</u>
<u>manual layout</u>	<u>4 5</u>
<u>tidy up option</u>	<u>5</u>
<u>auto layout</u>	<u>1 3 7 8</u>
<u>auto layout only if it is in the preferred style</u>	<u>7 8</u>
<u>auto layout only if it can be overridden</u>	<u>3 5 6</u>

This set of responses reinforces my argument for an automatic layout tool that is tailorable to the individual's preferred style.

7.5 Discussion of Results

First of all, the number of volunteers for the experiment was disappointing. A small sample size of 7 does not give a reliable "population" result; although it does give some idea of the variation in people's opinions.

The main problem with choosing the finalists as subjects was that they were a) too short of "spare" time, and b) not as proficient with Pascal as I had been led to believe. Hence a small sample size with possibly highly variable results, rather than a "normal" distribution with mainstream mode and trend (high frequency correlations with common characteristics).

Most of the volunteers indicated that the experiment was too long; I agree. It also required intense concentration - so the quality of results may also have been affected due to tiredness/stress as the experiment continued.

Some questionnaire questions were also misinterpreted, even though every care was taken to eliminate ambiguity and misunderstandings. The length and intensity of the questionnaire/experiment was necessary to extract as much detail as possible. Some comments indicated that the students thought some questions were repeated, but this was only to extract all perspectives on an activity (eg. "What information do the students need to detect a bug?" and "What information do they need to locate a bug?"); or to dissect different phases, and discover basic rules of operation and strategic influences.

In real life, choosing if or when to use spotlighting would be up to the user. In the experiments they had the code spotlighted whether they wanted it or not. This lack of choice and unfamiliarity with the spotlighting concept could well have undone or removed any advantage expected to be conveyed by the tool, due to user resistance or resentment (whether conscious or not).

7.6 Conclusions

Spotlighting

If spotlighting made/makes the task easier then it seems reasonable to suppose that the debugging time spent on an error on spotlighted code would be less than that for ordinary program text.

There doesn't seem to be a consistent reduction ratio for debugging on spotlighting code rather than ordinary. Thus hypothesis 1) fails. This could be due to several reasons.

A bad choice of experiments to test spotlighting with, or a bad choice of experimental equipment to test spotlighting with. It now seems obvious that a screen based experiment would have provided a more realistic setting for spotlighting; doing it on paper just doesn't have the same effect. It could be that when programmers debug on paper they have already decided on a range of possible error hypotheses. But going to debug on paper especially someone else's code with only the task description to go on (and an explanation of the algorithm used), puts them at a major disadvantage. Even having a new tool to help them won't be of much use when they are not familiar with how to use it, and how to use it to best effect.

With a screen based experiment there would probably have been more accurate feedback on qualitative/subjective opinions as to the usefulness of the spotlighting tool. But quantitative data would have been harder to extract and determine.

In contrast, the paper experiments yielded quite good data quantity-wise, but it may not be as reliable as I have assumed due to task disruption while swapping papers. Also there is the fact that each piece of spotlighted program text had the same code; and could not contain the errors already debugged on the same piece of code due to the change of sheet. Some students may have been distracted or put off by this factor. Indeed 1 student went as far as to asking why the other bug solutions weren't shown on each sheet - that is, that when spotlighting variable "x", the code shows all bug solutions for all variables except "x". The fact that that comment is ill considered and contradicts the purpose of the experiments in that the student has to define ALL the solutions for himself - might indicate that at least 1 student didn't fully understand the purpose of the separate sheets for each spotlighted variable.

Several students got fed up changing between the different spotlighted sheets, and just stuck to one sheet when debugging. Either this happened as a result of sheer perversity, or forgetting that a separate sheet was provided for each individual spotlighted variable. In either case 2-3 students stuck to 1 sheet only for the spotlighting experiment - not the same students [S4 & S5 for Survey; and S4 & S6 for Bubble Sort] for both experiments. An alternative explanation might be that they got so engrossed with debugging that they forgot the purpose of swapping the sheets altogether.

Spotlighting on a system with a multi-colour monitor should make it a much easier and more popular tool to use. Using a colour printer to differentiate between spotlights should make debugging on paper much less effort intensive, getting the machine to do the hard work instead.

The questionnaire showed that spotlighting can be relevant to debugging, but that it will probably be of more use as a screen based tool. Leaving spotlighting on paper for difficult/obscure errors or perhaps for longer term study of a piece of text. Thus hypothesis 2) is upheld for screen-wise spotlighting, and is borderline for spotlighting on paper. However, these results will have to be confirmed by user trials or experiments on screen based systems; due to the small sample size, and the fact that the results give an indication rather than a "cast-iron" verdict.

Layout Aids

The response to the layout questions makes it quite clear how important it is to have the code in your own preferred style. It helps readability, makes the code easier to follow, and makes debugging faster and more accurate. Question 11 makes it quite clear that debugging code becomes more difficult the further it differs from the debuggee's own preferred style.

These findings support hypotheses 5), 6) and 7); and shows the need for a layout tool that is geared towards the needs of each individual programmer.

Chapter 8 Summary of Findings & Future Work

8.1 Summary of Main Findings and Conclusions

8.1.1 Questionnaire Design

One of the challenges of questionnaire design is making questions unambiguous, and phrasing the questions to get the required information in the desired detail.

In the questionnaires the rating scales and multi-choice "answers" provided quantitative results in a relatively unambiguous format, making analysis straightforward. In contrast, the open ended "Why did you ... (or Why not)?" questions and the comment answers gave insights into the responses behind the answers (such as the reading strategies). Thus the 2-pronged design strategy provided both quantitative and qualitative results.

8.1.2 Preliminary Data Collection

Observations of 90 first-year student programmers showed 3 main categories of problems/errors: novice programming problems, common programming errors, and MacPascal related problems.

The majority of common programming errors are due to faulty syntax in one form or another. Followed by a combination of algorithm and semantic errors. Typing and spelling errors can contribute to either of the former, and bracketing errors usually arise from miscounting brackets during input typing and checking. Although syntax errors are the most frequent, they usually take less time to fix since the environment provides syntactic help. Whereas most algorithmic and semantic problems have to be solved by the programmer alone, and so take longer. The results of §7.1 (question 15) confirm this.

Few students make use of MacPascal's debugging tools on any regular or semi-regular basis. According to the students' comments, the usual alternative to using MacPascal's debugging tools is to insert "write" statements in the code at the appropriate points. The results of §7.1 (questions 9, 11, 12 & 22) reinforce this finding

A group of 66 student programmers gave opinions on MacPascal. As far as I know, no other programming environment's tools have been assessed in this way. It certainly shows up MacPascal's deficiencies, and points up several improvements and useful features that could be added to make MacPascal more effective and user-friendly. The frequency of use data suggests that some of MacPascal's tools need to be re-thought and redesigned. Especially the search mechanisms.

8.1.3 Spotlighting

The basic argument of the thesis is simple: typographic aids are not being exploited to the full to help users with electronic text-oriented tasks on VDUs. What is needed is careful, considered application of typographic aids to those tasks which would most benefit. Spotlighting is one such tool, aimed primarily at the programming environment. The examples of Chapter 6 gave an idea of how visually effective and useful this tool could be. The whole point of spotlighting is to bring the concept of high profile visibility into the realm of the search mechanisms.

During editing, the cursor position has a dual function: as the focus of visual attention, and the point at which editing operations are possible. The dual action of the cursor position is obvious in action, but is taken for granted unconsciously. Defining this dual function consciously gave the insight into spotlighting's essential feature - it separates the 2 functions, leaving the cursor position as the primary focus of attention and editing operations, but introducing alternative locations to fix visual attention (at the discretion of the user). This insight was a long time coming (4 months ago, during the writing of Chapter 4), but it is so obvious, and explains the solo spotlighting approach of existing search mechanisms succinctly. For existing search mechanisms, 1 cursor means 1 focus of visual attention; whereas spotlighting also has 1 cursor, but with multiple foci for visual attention.

Spotlighting is not a solution looking for a problem, it is a solution to a long standing problem that programmers encounter every time they want to follow a variable's trail through code. Spotlighting's secondary function is to challenge the sequential principle behind existing search mechanisms, and to show that spotlighting's inherently random access principle is complementary. At present the primary goal of search mechanisms is to locate (and jump to) successive instances of the search string one by one. Obviously something needs to be done when 45% of students using MacPascal do not use either the find or the replace tools at all (as shown by Chapter 3's frequency ratings). Some basic need must be unfulfilled with this percentage of students avoiding these particular tools. Spotlighting could help fulfil this need.

Investigating the effect of different typefaces and font size on discriminability, visual distraction, and trail following, when spotlighting program text, may produce some interesting results. It seems common sense to use a clear, plain, but distinct typeface for debugging to minimize the visual distraction from the task, and to focus visual attention the trail following task.

8.1.4 Debugging Strategies Experiments

That the students only wrote down superficial reports of their debugging activity was disappointing, but seems significant: writing as part of the debugging task is natural and easy, whereas attempting to write down the on-going debugging activity is difficult, and seems to interfere with the debugging process, perhaps as a result of divided attention.

8.1.5 Spotlighting Experiments & Questionnaire Results

There are several reasons why spotlighting on paper may have failed to reduce debug times. Either the students were disrupted from the task by having to attend to 4 sheets for each spotlighted program. Thus requiring them to swap from 1 sheet to another, adding the task of finding the correct spotlighted sheet before continuing with the debugging task. The main advantage of automatic spotlighting on a screen is that there would only be one point of reference at a time - the screen itself. Whereas to indicate the same effect on paper requires one spotlighted variable per sheet of paper to indicate "current screen appearance". One or two students got bored with changing between sheets of paper and simply used one sheet of program text to debug all the errors; thus defeating the whole point of the experiment. This would have been less likely on a VDU screen-based experiment where the screen would have reflected the cumulative modifications at each stage. Of course, the same effect is not possible with multiple sheets of the same text.

Students answers to the post-spotlighting questionnaire (§7.4) showed that they thought that spotlighting made debugging easier. Students said that they were more likely to use spotlighting on a VDU screen than on paper. This might well explain why the experiments failed to prove spotlighting conclusively useful regarding reduced debug times with the paper experiments.

The questionnaire data shows that out of 7 students, 4 would and 2 might use spotlighting if it was an on-screen tool. Whereas out of 5 students, 1 would and 3 might use spotlighting on paper. These results indicate that spotlighting's quality of interaction - namely speed and on the spot interaction or usefulness will be greater with a screen-based system, than on paper.

According to the questionnaire data, spotlighting's most useful features were to jump directly from one spotlight to another; and using inverse video to make the location of the search string totally obvious. This shows the need for moving aids that are spotlight-oriented.

Students' ratings of the most important features that spotlighting offers for debugging are: focussing attention on a specific item, showing up (non-)initialisation errors, showing up variable (non-)declaration errors, and trail following.

Also, 5 out of 7 students agreed that spotlighting was helpful in the debugging task; and 3-5 students would use spotlighting in addition to their own debugging strategies.

The questionnaire showed that spotlighting can be relevant to debugging, but that it will probably be of more use as a screen based tool. Leaving spotlighting on paper for difficult/obscure errors or perhaps for longer term study of a piece of text.

Spotlighting on a colour system (with both VDU and printer having multiple colour output, rather than monochrome) will make it a much easier and more popular tool to use. Using a colour printer to differentiate between spotlights should make debugging on paper much less effort intensive, getting the machine to do the hard work instead.

8.1.6 Summary Tables

At present, the onus is on the programmer to extract relevant information from the data declarations. This uses up time, as well as distracting the programmer from the former task. Providing the data in suitable formats would be more efficient, and less prone to human errors. For example, displaying all the integer variables, or calling up the global alphabetic menu listing to name all procedures that use the variable "count", and what data type it has under each procedure.

Summary tables could also be used to display the original declaration of a procedure's parameter list on demand. Having a visual reminder of which variables go where, and in what order, should help the programmer get it right first time. An extension of this idea, would be to put the procedure's name and its original full parameter list onto the line(s) above or below the cursor position. Or even to put the procedure's name and an empty parameter list below, that is blocked with commas to show how many slots need to be filled. Making the task much easier.

8.1.7 Layout Aids

The response to the layout questions makes it quite clear how important it is to have the code in your own preferred style. It helps readability, makes the code easier to follow, and makes debugging faster and more accurate. Being able to line up loops (thus defining scoping control) is another important factor for program comprehension and accurate debugging.

Question 11 (in §7.4) makes it quite clear that debugging code becomes more difficult the further it differs from the debuggee's own preferred style. It is easier to debug a program if it is in the programmer's own style of layout. Code becomes progressively more difficult to spot errors in, as the stylistic differences increase.

These findings show the need for a layout tool that is geared towards the needs of each individual programmer, or for an automatic layout tool that is tailorable to the individual's preferred style.

8.1.8 Final Comments

Throughout my research the emphasis has been on developing tools to increase meaning and make vital information more visible, easier to understand, and easier to get at - requiring less cognitive effort to extract relevant details.

What the user needs is freedom of choice. With alternative tools/methods to accomplish a task. It is well known from work on individual differences that people don't always use tools the same way, or for the same reasons. It is up to tool-designers to make tools as useful and functionally flexible as possible. Giving the user a range of alternative tools to achieve the same or similar tasks.

8.2 Future Work

One goal was to implement the spotlighting and summary tools, and to test them out properly. Another area of interest was investigating mental simulation, the form(s) it takes and how it helps with software development and debugging.

Further work could also be done to find out what it is about each individual's set of layout preferences that make it easier for him to use. Why they are so necessary or particularly useful to each individual. What each user gains from his own set of layout preferences. What correlation there is, if any, between the individual's personal traits or individual differences, and the set of specific layout variations he prefers to use. Ideas for investigating/testing layout are outlined at the end of the chapter.

8.2.1 Spotlighting

On the whole, the spotlighting experiments did not prove spotlighting useful when given to student programmers to debug from a "cold" start on paper. In real life, spotlighting should be available as a programming environment tool, at the editing/developing or code reviewing stage. So it would be better to implement spotlighting on a computer system and ask users' opinions after using it for a fairly long time; or at least long enough to get past the tool's learning curve. This would give a fairer test of spotlighting "for real" rather than under artificial test conditions.

One implementation option is to pilot first and then implement a more sophisticated system - graduating from a basic system to one that meets users' needs more closely, as defined by their feedback after using it.

For example, applying spotlighting to multi-file documents, since program files often contain include files. The compiler opens these up and adds the text inside onto the program file. It should not be too difficult to do the same with the spotlighting tool, it is just a case of being able to access these include files and read them in. A similar method could be applied to a set of files that formed a large report. As long as there is a mechanism (or list) that defines which files are to be searched and spotlighted.

8.2.2 Summary Aids

The summary aid should be useful as a comprehension aid for development or maintenance programmers, when trying to correlate between variable names, their usage and their data types. One issue of implementation concerns updating. Either the summary aid is updated whenever the user invokes a menu listing from it, or a special "update summary information/tables" option is provided which the user can select when ready. The latter gives the user absolute control over the updating decision, instead of the tool working away, updating "secretly" in the background.

The best way to test the summary tables tool is to integrate it with a programming environment, and to see out how users react to it. Finding out how often they use it once they are over the learning curve and how useful they think it is. Gauging their subjective opinions accurately would probably require a short questionnaire, like Chapter 7's post-spotlighting one.

8.2.3 Mental Simulation

In my opinion, mental simulation is a vastly under-rated, yet essential programming tool. The questionnaire of Chapter 7 helped define its uses, regarding program development and debugging tasks, but it is still a research area that needs more attention.

One use of mental simulation is to check through each statement to see that all variable values change when they are supposed to and that guard conditions work properly - activating sub-statements when the condition(s) are met, bypassing them if not.

Making sure that events trigger properly and in the correct sequence - a form of preventative debugging. If the simulation fails to work as intended, then the code must be modified until it does work as intended.

One of the tricky aspects of programming is the placing of initialising, resetting and unsetting of trigger variable values. Misplacing them in the code results in either an infinite loop (one that never stops because the terminating condition is always false) or a redundant loop (where the terminating condition is met before entry to the loop, so that the statements within the loop never get executed). Students voted infinite loops as the most troublesome error (see question 15 in §7.1 or Appendix 7A).

I raised the question of mental simulation at the Psychology of Programming Special Interest Group (PPIG) Workshop held at Loughborough University in January 1992, and got a variety of opinions. The consensus was that it was still an open area. Although people were beginning to realise its importance, it was a difficult subject to identify, quantify, and generally pin down. Mainly because people differed on their interpretations and (internal) representations.

Questions that deserve further investigation:-

- What is the significance of mentally test running sections of code?
- Is the only function of mental simulation to check whether a section of code works correctly?
- Is the only function of mental simulation to find out how a piece of code works, and to correlate this with how it was intended to work?
- Is mental simulation used to iterate code development towards a more efficient or more fully functional code solution?
- How do people achieve mental simulation? What do they "see" and "do"?
- Are some forms of mental simulation better than others?
- Does the quality of the mental simulation relate to the expertise of the programmer?
- Does the quality of the mental simulation relate to the quality of programming and/or debugging skills of the programmer?
- Does the complexity or richness of mental simulations used depend on the expertise of the programmer?
- Is there a hierarchy of mental simulations? and How is it organized?
- Is there a way/method of improving a programmer's mental simulations that will lead to an increase in programming and debugging skills?
- Can "good" forms of mental simulation be learnt by study or practice rather than actual interactive programming/debugging experience?

8.2.4 Exploring Aspects of Layout & Possible Experiments

The purpose of layout (as I understand it) is to reflect control structure and make each individual construct more recognisable. The 4 concepts that I mostly associate with individual differences/preferences in layout style are - readability, comprehensibility, familiarity, and visual rapport. These concepts are central to the issues of layout variations - understanding the interaction between them may give the answer to many puzzles.

Students' answers to the questionnaire of Chapter 7 stated that it was easier to detect bugs in (other people's) code if it was in the same, or similar, layout style to that of the reader. I believe that this is because having a visual rapport with the code (having it in a similar layout style to that used by the reader), aids comprehensibility and makes the mental (internal cognitive) map of the code and its functions easier to build.

The first problem is in detailing the variations in layout styles and cataloguing them. Then it is a question of looking for trends, and grouping stylistic variations in response to different and/or similar situations. Such as line overflow, where the disposition of complex conditions (say for an IF statement) fill more than 1 line of text.

The second problem is defining experiments to test layout itself, or perhaps to explore layout elements individually and in combination. The shadow code experiments are a first attempt to define/explore interaction between layout elements.

8.2.4.1 Cataloguing Layout Styles

The obvious option is to give each subject/student a section of unformatted Pascal code to lay out and then get it handed back in. The students could lay the code out either by hand, giving hand written code; or they could use a manual editor to set the text, and then get it printed out. The problem with hand written code is that the indentation can look variable even when the author "intends" the same indentation, and vice versa. Also it is difficult to judge indentation spacing correctly with hand written code. But with a manual editor, VDU and printer system it is much easier to check spacings and see how the author/programmer lines the code up indentation- and construct-wise.

The code to be laid out should contain all Pascal constructs : if-then, if-then-else, for-to & for-downto, while, repeat-until, case-end, and with. Preferably showing a single statement and a compound (begin-end) statement for each controlling construct.

Another area of layout variation concerns complex condition statements, and where programmers place the AND/ORs in multi-line condition statements (see §6.1.1). In process control software it is not unusual for an if-then condition to cover 2 or 3 lines — the more complex the condition specified, the more space it takes up.

Getting a large set of layout styles to catalogue would enable the percentage frequency of each individual variation of construct layout to be calculated. This would then provide evidence of trends in layout styles. Similar cataloguing of other procedural languages could also be done.

It may be possible to correlate the layout style and the quality of the program coded - since Molzberger's article indicates that good or high quality, "efficient" code has a "meta-style" that his super-programmers recognise. Cataloguing all the layout styles may make this "meta-style" distinguishable from all the others, or at least help to define the particular layout style(s) that gives code maximum readability and comprehensibility.

8.2.4.2 Shadow Code Exploration & Experiments

The first question is how each individual line is seen and comprehended in terms of line length and shape. Not only the shape and position of the line, but of the individual word elements.

The second question is how this fits in with the context of the whole code - or a screenful of code - both visually, cognitively and in relation to the internal representation of the code. For example, do people

- - see individual lines solely in terms of length; or
- - as consisting of individual words or operators; or
- zen := prev * 1.2345; - as consisting of jigsaw shapes - taking the individual outline shape of each word into account.

The shadow code investigations/observations are an attempt at addressing these questions.

8.2.4.3 Shadow Code Observations

On the following page there are examples of shadow code, where all visible alphanumeric characters have been replaced by ■, but the indentation cues (invisible leading spaces/tabs) have been preserved. One of the interesting things to note about the shadow code that leaves punctuation, quotes and round brackets symbols unaltered, is how clearly the round brackets stand out. Making the position of procedure calls (or complex bracketing elements) easy to spot.

An alternative set of experiments investigating the effect on cognitive markers/beacons by reducing different sets of symbols including keywords to ■, could provide interesting information. Such as the effect of putting all alphabetic characters to ■, but not numerics; or putting all alphanumeric characters to ■; or putting all characters from first to last character to ■, either including or excluding spaces.

LAYOUT EXPERIMENT - Comparison of Complete Code vs Its Shadow Indented Code

```

program survey(input, output);
var
    signal, time, vehicles,
    wait, maxwait : integer;
begin
    time := 0;
    wait := 0;
    maxwait := 0;
    vehicles := 0;
    repeat
        read(signal);
        if signal = 2 then
            begin
                time := time + 1;
                wait := wait + 1;
                if wait > maxwait
                    then maxwait := wait;
            end;
        if signal = 1 then
            begin
                wait := 0;
                vehicles := vehicles + 1;
            end;
        until signal = 0;
        writeln(time, 'secs');
        writeln(vehicles);
        writeln(maxwait, 'secs');
    end.

```

Completed Code for Signal Problem

Cognitive markers/beacons, such as keywords, seem to be used as mental "hand-holds" or pivot-points. Analogous to markers on a ski slope which mark the position and help remind you where you are on the course. Allowing a mental slalom around the cognitive map, perhaps; up & down, to & fro, and round the loops.

Going back to the shadow code - at first glance the shadow code looks rather uninformative. But it is easy to spot the procedure calls since they are shown up by the round brackets. IF-THEN statements can also be determined, since the line starts with **■■** (IF) and ends with **■■■■** (THEN). So these can be picked out by deduction and indentation clues. Deduction also gives the position of the BEGIN-END loops, since they have easy to recognise block and punctuation patterns. With BEGIN shown as a line consisting of **■■■■■**, and END as a line consisting of **■■■**. The BEGIN-END pair should (usually) match indentation-wise, giving a further "hint" as to their function and importance.

When ■■ appears as the 2nd block/chunk, it can be deduced as representing := thus defining an assignment statement. Spacing between mathematical symbols varies between programmers — some use one space each side, and others don't, as in:

count := count + 1; or count := count+1;

Sometimes, spacing around mathematical symbols varies within a program text, even for a solo programmer. Sometimes this is due to wanting to keep the "formula" for a statement all on one line, instead of getting line overflow, just for the sake of a couple of spaces or characters.

8.2.4.4 Shadow Code Tasks

The idea is to correlate spatial characteristics of the programmer's cognitive map of a chunk of code to its comprehensibility. 3 sets of layout may be required :

- non-indented code,
- standard code indentation,
- user's preferred style of layout and indentation.

Task : to study a piece of complete code, and its task description for a pre—determined length of time, after which the code is removed and "shadow code" is substituted.

Where the form of the shadowing has 2 possibilities : where all non—space characters (except punctuation) are turned into black blocks :-

"if $a > c$ then $c := a;$ " becomes "■ ■ ■ ■ ■ ■ ■ ■ ■ ■ ;"

Or all characters on a line between the first and last non-space characters become black blocks - resulting in a series of "bar graph" blocks indented according to the original pattern :-

"if $a > c$ then $c := a$;" becomes ".....;"

Thus the first part of the task is concerned with the (simultaneous) comprehension and assimilation of the code. The former on a primarily (conscious) semantic level, and the latter on a (presumably subconscious) level that binds the spatial characteristics (and individual pattern) of each piece of code to its semantic meaning. Thus at the end of the comprehension/assimilation phase, the overall "shape" of the code (macro-level) and its individual (micro-level component) features are tied into the primary meaning and its corresponding sub-ordinate semantic levels.

Now, the first phase of the task should be enhanced by advising the subjects/students that the second part of the task will be concerned with pointing out the location of a variety of distinct code segments. For example, the locations where the "wait" variable is initialised; or the vehicle counter is incremented; and so on. Each such question being dedicated to the task of giving sufficient information for its corresponding coded doppelganger to be located, using the cognitive mental map and the shadow map, in

concert with the spatial and semantic associations existing between them to provide the answers to the questions in spatial format.

So the student is asked to denote the position of several individual statements on the shadow code "text" without access to the original (alphanumeric) program text. This should test the link between memorability and spatial factors. It may also give information on cognitive markers.

8.2.4.5 Example Experimental Method

The signal problem is the example task for the shadow code experiment (see §6.2 for statement of Siddiqi's Signal Problem).

Proposed Method

1. The subject is given the task description and program code to read and assimilate.
2. The program code is taken away and replaced by a shadow code version. Whether the task description is taken away is a moot point. It could act as a distraction from the remembering task. But, on the other hand, it may help prompt the student to remember the sequencing of the code. So it could act as either a disadvantage or an advantage. Perhaps it would be better to remove it and keep things simple.
3. The subject is asked to denote the location of several key statements - where wait variable is initialised, where signal value is read in, position of loop condition, and so on.
4. After say 10-15 minutes the shadow code is handed in for evaluation.

Expected Results

Accuracy of shadow code location to actual location in original program text should give an indication of its memorability. The idea is to do the same type of experiment on indented and non-indented code vs the blocking of all characters or only all non-space characters; and see which combination gives most accuracy. Of course the indented code could be either in standard layout style, or the user's preferred style. This would give a total of 6 test conditions for 1 piece of program text. So the subject population would have to be split into either 3 or 6 groups. Where one half of the subjects work on shadow code that blocks all characters including spaces between "words", and the other half works on shadow code that blocks all characters excluding spaces between "words"; for one experiment, and then the two halves "swap" for a second experiment. That should cut out bias between subjects working with code that does or doesn't have intermediate spaces blocked/blacked out. Numerically speaking, each third of the subject population works on shadow code that is either unindented, with standard indentation, or in the user's preferred style. Of course, to get the preferred style tested, each subject would have to have formatted the "test" code prior to the experiment. This

means that there would have to be a suitable forgetting period between getting the subjects to format the code, and testing it out in the proposed shadow code experiments; as well as for producing shadow code to match each individual subject's layout style.

Example Shadow Code Questions

Mark the shadow code location(s) "answering" the question with the question identifier :- (a), (b), (c), (d), (e), (f) or (g).

- Where is the signal variable's value assigned?
- Show the location of all 'if' tests on the signal variable
- Where is the remaining test on the signal variable's value?
- Show the location of all increment assignments
- Show the location of all initialisation statements on time-related variables
- Where is the maxwait variable's value modified?
- Show the location of any remaining re-initialisation statements

Unindented Code

```

program survey(input, output);
var
signal, time, vehicles,
wait, maxwait : integer;
begin
time := 0;
wait := 0;
maxwait := 0;
vehicles := 0;
repeat
read(signal);
if signal = 2 then
begin
time := time + 1;
wait := wait + 1;
if wait > maxwait
then maxwait := wait;
end;
if signal = 1 then
begin
wait := 0;
vehicles := vehicles + 1;
end;
until signal = 0;
writeln(time, 'secs');
writeln(vehicles);
writeln(maxwait, 'secs');
end.

```

Completed Code for Signal Problem
C A Humphreys

Corresponding Shadow Code

Indented Shadow Code

```

#####(####, #####);
#####
#####, ####, #####,
      ##### ■ #####;
#####
##### ■ ■;
##### ■ ■;
##### ■ ■ ■;
##### ■ ■ ■;
#####
#####(#####);
■ ■ ##### ■ ■ #####
#####
      ##### ■ ■ ■ ■;
      ##### ■ ■ ##### ■ ■;
      ■ ■ ##### ■ #####
      ##### ##### ■ ■ ■ ■;
#####;
■ ■ ##### ■ ■ #####
#####
      ##### ■ ■ ■;
      ##### ■ ■ ##### ■ ■ ■ ■;
#####;
##### ■ ■ ■;
#####{#####, '#####');
#####{#####};
#####{#####, '#####');

```

Shadowing all non-space characters

[illegible]

Shadowing all intermediate characters

Unindented Shadow Code

```

#####(#####,#####);
#####;
#####,####,#####;
####,#####;
#####
#####;
#####;
#####;
#####;
#####;
#####
#####(#####);
#####
#####
#####
#####;
#####;
#####;
#####
#####;
#####;
#####
#####;
#####;
#####
#####;
#####(#####);
#####(#####);
#####(#####);
#####(#####);

```

Shadowing all non-space characters
C A Humphreys

```

#####(#####,#####);
#####
#####,####,#####;
  ####,#####■#####;
#####
#####;
#####;
#####;
#####;
#####;
#####
#####(#####);
#####
#####
#####;
#####;
#####
#####;
#####;
#####
#####;
#####;
#####
#####;
#####(#####, '#####');
#####(#####);
#####(#####, '#####');

```

Shadowing all intermediate characters

Appendix Contents	Page
Appendix 2 Chapter 2 References	1
Appendix 3A	
- Comparison of Student Numbers & Percentages For Each Tool in Tool Order	20-25
Usefulness	21
Frequency of Use	22
Ease of Use	23
Likeability	24
Other Method : Frequency of Use	25
Summary of Means and Rankings for Each Dimension in Tool Order	25
- Comparison of Student Numbers & Percentages For Each Tool	
Frequency of Use - in Order of Decreasing Mean Rating Scale Value	26
Appendix 3B Additional Questionnaire Data	27
Q25. Enhancements and Additional Facilities Frequency	27
Summary of Enhancements Suggested in Questionnaire	28
Q27 Data	29
Q30 Data	31
Appendix 7A Programming & Debugging Strategies Questionnaire Data	32
(A) is the primary information sheet to be read before starting	32
(B) is a reference sheet to help to resolve vague or ambiguous questions	33
Q7A Results & Summarizing Comments for each Question	35
Q7A.1 Experience & Programming Ability - Q1-3	35
Q7A.2 Program Development and Coding Methodology - Q4	35
Q7A.3 Attitudes Towards The Compiler - Q5 & Q7	37
Q7A.4 Development & Debugging Attitudes - Q6, Q8, Q10, Q13 & Q29	37
Q7A.5 Use of Debugging Techniques & Tools - Q9, Q11, Q12 & Q22	39
Q7A.6 Defining The Nature of Errors, Their Frequency & Troublesomeness - Q14-15	41
Error Frequency Ordering	43
Troublesome/Time Consuming Ordering	44
Q7A.7 Investigating Reading Strategies - Q16-19	45
Q7A.8 Information Needed to Detect & Locate Errors - Q20-21	48
Q7A.9 Programming/Software Development Process Diagram - Q23	49
Q7A.10 Investigating Trail Following on Paper & Screen Text - Q24 & Q25	49
Q7A.11 Differentiating Between Debugging Methods - Q26-28	52
Q7A.12 Attitudes Towards the Search Mechanisms - Q30-33	54
Q7A.13 "Live" Editors & Layout Style - Q34 & Q35	55
Q7A.14 Program Visualisation - Q36	56
Q7A.15 Suggestions for New Editing/Debugging Aids - Q37-41	57
Q7A.16 Students' Comments About The Questionnaire	60
Appendix 7B Post-Spotlighting Questionnaire Data	61
Q7B.1 Spotlighting Questions	61
Q7B.2 Layout Questions	66

Appendix 7C Debugging Strategies Experiment Sheets	70
D1 Instructions	70
D2 Task Descriptions for all 3 Tasks	71
Summary Sheet of Control Errors & Their Solutions	72
D3 Primes Code with Control Error Solutions & Algorithm Hint	73
D4 Letter Pairs Code with Control Error Solutions	74
H1 Letter Pairs Algorithm Hint	75
H1 & H2 Concordance Algorithm Hint & Linked List Hints	75-6
D5 & D6 Concordance Code with Control Error Solutions	77-8
D7 Task Difficulty Rating Sheet with Table of Results	79
 Appendix 7D Spotlighting Debugging Task Experiment Sheets	 80
Instructions	80
Summary Sheet of Control Errors & Their Solutions	81
"Plain" Debugging Tasks	82
Letter Count Description	82
Letter Count Code with Control Error Solutions	83
Shell Sort Description	84
Shell Sort Code with Control Error Solutions	85
Spotlighted Debugging Tasks	86
Survey Description	86
Survey Code with Control Error Solutions (4 pieces of code on 2 pages)	87-8
Bubble Sort Description	89
Bubble Sort Code with Control Error Solutions (4 pieces of code, 1 page each)	90-3
Task Difficulty Rating Sheet with Table of Results	94

Appendix 2 - Chapter 2 Reference List

IJMMS = International Journal of Man-Machine Studies.

BIT = Behaviour & Information Technology.

Adelson B & Soloway E. 1985.

The role of domain experience in software design.

IEEE Trans. on Software Engineering 11(11) p1351-1360.

Adelson B & Soloway E. 1988.

A model of software design.

in Chi M T H, Glaser R & Farr M J (Eds); The Nature of Expertise; Lawrence Erlbaum, p185-208.

Allen R B. 1982.

Patterns of manuscript revisions.

BIT 1(2), p177-184.

Alty J L. 1984.

Use of path algebras in an interactive adaptive dialogue system.

INTERACT'84, p351-354.

Arthur J D & Comer. 1987.

An interactive environment for tool selection, specification and composition.

IJMMS 26, p581-595.

Arthur J D & Raghu K S. 1989.

Taskmaster: an interactive, graphical environment for task specification, execution and monitoring.

BIT 8(3), p219-233.

Baecker R & Marcus A. 1986.

Design principles for the enhanced presentation of computer program text.

Proceedings of Human Factors in Computer Systems, ACM: Washington DC, p51-58.

Baecker R M & Marcus A. 1990.

Human Factors and Typography for More Readable Programs.

ACM Press, Addison-Wesley.

Bannon L J & Bødker S. 1991.

Beyond the interface: encountering artifacts in use.

in Carroll J M (Ed); Designing Interaction: Psychology at the Human-Computer Interface; Cambridge University Press, p227-253.

Barstow D R, Shrobe H E & Sandewell E. 1986.

Interactive Programming Environments.

McGraw Hill International Edition.

- Bergantz D & Hassell J. 1991.
Information relationships in PROLOG programs: how do programmers comprehend functionality?.
IJMMS 35(3), p313-328.
- Berlin L M. 1993.
Beyond program understanding: a look at programming expertise in industry.
in Cook C R, Scholtz J C & Spohrer J C (Eds). Empirical Studies of Programmers: Fifth Workshop. Ablex, p6-25.
- Black J B, Kay D S & Soloway E M. 1987.
Goal and plan knowledge representations: from stories to text editors and programs.
in Carroll J M (Ed); Interfacing Thought: Cognitive Aspects of Human-Computer Interaction; MIT Press, p36-60.
- Boehm-Davis D A. 1988.
Software comprehension.
in Handbook of Human-Computer Interaction. Helander M. Editor. New York: North-Holland, p107-121.
- Bourguignon J P. 1984.
PCTE - Portable Common Tool Environment.
ESPRIT '84 p75-84.
- Brooke J B. 1986.
Usability engineering in office product development.
People & Computers II, p249-260.
- Brooks R. 1977.
Towards a theory of the cognitive processes in computer programming.
IJMMS 9(6), p737-751.
- Brooks R. 1983.
Towards a theory of the comprehension of computer programs.
IJMMS 18(6), p543-554.
- Budgen D. 1992.
How to support the work of designers through the presentation of appropriate information.
Paper from the Psychology of (Special) Interest Group (PPIG-4) Workshop, held at Loughborough University, 2-4/1/92.
- Cakir A, Hart D J & Stewart T F M. 1980.
Visual Display Terminals.
J Wiley & Sons.
- Candy L & Edmonds E A. 1988.
Introducing an expert system into an office.
Conference on Man-Machine Systems, Finland, June 1988.

Card S K & Henderson A Jnr. 1987.

A multiple, virtual-workspace interface to support user task switching.
CHI + GI 1987, p53-59.

Card S K, Moran T P & Newell A. 1983.

Psychology of Human-Computer Interaction.
Lawrence Erlbaum Associates.

Carroll J M & Olson S R. 1988.

Mental models in human-computer interaction.
in Handbook of Human-Computer Interaction. Helander M. Editor. New York:
North-Holland, p45-65.

Carroll J M & Rosson M B. 1987.

Paradox of the active user.
in Carroll J M (Ed); Interfacing Thought: Cognitive Aspects of Human-Computer
Interaction; MIT Press, p80-111.

Carroll J M (Ed). 1987.

Interfacing Thought: Cognitive Aspects of Human-Computer Interaction.
MIT Press.

Carroll J M (Ed). 1991.

Designing Interaction: Psychology at the Human-Computer Interface.
Cambridge University Press.

Carver S M & Klahr D. 1986ish.

Children's acquisition of debugging skills in a LOGO environment.
Journal of Educational Computing Research.

Catrambone R & Carroll J M. 1987.

Learning a word processing system with Training Wheels and guided exploration.
CHI + GI '87, p169-174.

Civikly J M. Editor. 1981.

Contexts of communication.
Holt, Rinehart & Winston.

Clarke A A. 1986.

A three-level human-computer interface model.
IJMMS 24(6), p503-518.

Coats R B & Vlaeminke I. 1987.

Man-Computer Interfaces: An Introduction to Software Design & Implementation.
Blackwell Scientific Publications.

Curtis B. 1988a.

Five paradigms in the psychology of programming.
in Handbook of Human-Computer Interaction. Helander M. Editor. New York:
North-Holland, p87-105.

Curtis B. 1988b.

The impact of individual differences in programming.

in van der Veer G C, Green T R G, Hoc J-M & Murray D M (Eds); Working with Computers: Theory versus Outcome; Academic Press, p279-294.

Damodaran L. 1983.

User Involvement in System Design: How to make sure users get the support they need.

Data Processing 25(6), July/Aug 1983, p6-13.

Davies S P. 1989.

Skill levels and strategic differences in plan comprehension and implementation in programming.

in Sutcliffe A & Macaulay L, Editors, People & Computers V. Cambridge University Press.

Davies S P. 1990.

The nature and development of programming plans.

IJMMS 32(4), p461-481.

Davies S P. 1993.

Externalising information during coding activities: effects of expertise, environment, and task.

in Cook C R, Scholtz J C & Spohrer J C (Eds). Empirical Studies of Programmers: Fifth Workshop. Ablex, p42-61.

Détienne F & Soloway E. 1990.

An empirically-derived control structure for the process of program understanding.

IJMMS 33(3), p323-342.

Détienne F. 1990.

Expert programming knowledge: a schema-based approach.

in Hoc J-M, Green T R G, Samurcay R & Gilmore D J (Eds); Psychology of Programming; Academic Press, p205-222.

Détienne F. 1991.

Reasoning from a schema and from an analog in software code reuse.

in Koenemann-Belliveau J, Moher T G & Robertson S P (Eds). Empirical Studies of Programmers: Fourth Workshop. Ablex, p5-22.

Détienne F. 1991.

A schema-based model of program understanding.

in Tauber M J & Ackermann D (Eds); Mental Models and Human-Computer Interaction 2; North-Holland, Elsevier, p225-242.

Diaper D & Winder R. Editors. 1987.

People and Computers III: .

Proceedings of the British Computer Society, Human Computer Interaction Specialist Group, University of East Anglia, 17-20 Sept 1987, Cambridge Univ Press.

- Dillon A & Sweeney M. 1987.
The Application of Cognitive Psychology to CAD.
People and Computers Vol IV, Proceedings of the British Computer Society, Human
Computer Interaction Specialist Group, University of Manchester?, 7-9 Sept 1988.
- Douglas S A & Moran T P. 1983.
Learning text editor semantics by analogy.
CHI'83, p207-211.
- Draper S W. 1984.
The nature of expertise in Unix.
INTERACT'84, p465-472.
- Eason K D. 1982a.
Methods of Planning the Electronic Workplace.
Behaviour and Information Technology 1982 Vol 1 No.2 197-213.
- Eason K D. 1982b.
The process of introducing information technology.
BIT 1(2), p197-214.
- Eason K D. 1984.
Towards the experimental study of usability.
BIT 3(2), p133-144.
- Eason K D. 1987.
Methods of planning the electronic workplace.
BIT 6(3), p229-238.
- Eason K D. 1988.
Information technology and organizational change.
London, Taylor & Francis.
- Eason K D, Harker S D P, Raven P F, Brailsford J R & Cross A D. 1987.
A user centred approach to the design of a knowledge based system.
INTERACT'87, p341-348.
- Eason K. 1983a.
Introduction. Human factors in teleinformatics.
BIT 2(4), p299-300.
- Eisenstadt M. 1993.
Tales of debugging from the front lines. 1993.
in Cook C R, Scholtz J C & Spohrer J C (Eds). Empirical Studies of Programmers:
Fifth Workshop. Ablex, p86-112.
- Feigenbaum E & Feldman J. 1963.
Computers and Thought.
McGraw-Hill, New York.

Fisher C. 1987.

Advancing the study of programming with computer-aided protocol analysis.
in Empirical Studies of Programmers, Second Workshop. Sheppard S, Olson G M &
Soloway E (Eds). Ablex, p198-216.

Fitter M J & Green T R G. 1981.

When do diagrams make good computer languages?
in Computing Skills and the User Interface; Coombs M J & Alty J L (Eds); Academic
Press.

Foley J & Van Dam A. 1982.

Fundamentals of Interactive Computer Graphics.
Addison-Wesley.

Gaines B R & Shaw M L G. 1984.

The Art of Computer Conversation.
Prentice-Hall International.

Galambos J A, Wikler E S & Black J B. 1983.

How to tell your computer what you mean: ostension in interactive systems.
CHI'83, p182-185.

Gardiner M M & Christie B. Editors. 1987.

Applying Cognitive Psychology to User Interface Design.
John Wiley & Sons.

Gardner A, Mayfield T F & Maguire M C. 1984.

Human factors guidelines for the design of computer-based systems.
INTERACT'84, p649-654.

Gellenbeck E M & Cook C R. 1991a.

An investigation of procedure and variable names as beacons during program
comprehension.

in Koenemann-Belliveau J, Moher T G & Robertson S P (Eds). Empirical Studies of
Programmers: Fourth Workshop. Ablex, p65-81.

Gellenbeck E M & Cook C R. 1991b.

Does signaling help professional programmers read and understand computer
programs?.

in Koenemann-Belliveau J, Moher T G & Robertson S P (Eds). Empirical Studies of
Programmers: Fourth Workshop. Ablex, p82-98.

Gilmore D J & Green T R G. 1987.

Are 'programming plans' psychologically real - outside Pascal.
INTERACT'87, p497-504.

Gilmore D J & Green T R G. 1984.

Comprehension and recall of miniature programs.
IJMMS 21(1), p31-48.

- Gilmore D J. 1990.
Expert programming knowledge: a strategic approach.
in Hoc J-M, Green T R G, Samurcay R & Gilmore D J (Eds); *Psychology of Programming*; Academic Press, p223-234.
- Gittins D T, Winder R L & Bez H E. 1984.
An icon-driven end-user interface to UNIX.
IJMMS 21(5), p451-462.
- Goldenson D R & Wang B J. 1991.
Use of structure editing tools by novice programmers.
in Koenemann-Belliveau J, Moher T G & Robertson S P (Eds). *Empirical Studies of Programmers: Fourth Workshop*. Ablex, p99-120.
- Gould J D. 1975.
Some psychological evidence on how people debug computer programs.
IJMMS 7(2), 151-182.
- Gould J D. 1987.
How to design usable systems.
INTERACT'87, pxxxv-.
- Gray W D & Anderson J R. 1987.
Change-episodes in coding: when and how do programmers change their code?.
in *Empirical Studies of Programmers, Second Workshop*. Sheppard S, Olson G M & Soloway E (Eds). Ablex, p185-197.
- Green T R G. 1989.
Cognitive dimensions of notations.
in Sutcliffe A & Macaulay L, Editors, *People & Computers V*. Cambridge University Press.
- Green T R G. 1990a.
The nature of programming.
in Hoc J-M, Green T R G, Samurcay R & Gilmore D J (Eds); *Psychology of Programming*; Academic Press, p21-44.
- Green T R G. 1990b.
Programming languages as information structures.
in Hoc J-M, Green T R G, Samurcay R & Gilmore D J (Eds); *Psychology of Programming*; Academic Press, p117-137.
- Green T R G. 1990c.
The cognitive dimension of viscosity: a sticky problem for HCI.
INTERACT'90 p79-86.
- Green T R G, Bellamy R K E & Parker M. 1987.
Parsing and GNISRAP: a model of device use.
in *Empirical Studies of Programmers, Second Workshop*. Edited by Sheppard S, Olson G M & Soloway E. Ablex, p132-146.

- Green T R G, Sime M E & Fitter M J. 1981.
The art of notation.
in Computing Skills and the User Interface; Coombs M J & Alty J L (Eds); Academic Press.
- Grudin J. 1991.
Obstacles to user involvement on software product development, with implications for CSCW.
IJMMS 34(3), p435-452.
- Gugerty M & Olson G M. 1986.
Comprehension differences in debugging by skilled and novice programmers.
in Soloway E & Iyengar S (Eds). Empirical Studies of Programmers. Ablex, p13-27.
- Guindon R. 1990.
Knowledge exploited by experts during software system design.
IJMMS 33(3), p279-304.
- Hansen W J. 1971.
User engineering principles for interactive systems.
Proceeding Fall Joint Computer Conference, Vol 39, AFIPS Press, pp523-532.
- Harrison M D & Monk A F. Editors. 1986.
People and Computers II: Designing for Usability.
Proceedings of the British Computer Society, Human Computer Interaction Specialist Group, University of York, 23-26 Sept 1986, Cambridge Univ Press.
- Heaton N & Sinclair M . Editors. 1988.
Designing End-User Interfaces: State of the Art Report 15:8.
Pergamon Infotech.
- Helander M. Editor. 1988.
Handbook of Human-Computer Interaction.
New York: North-Holland.
- Hewett T T. 1986.
The role of iterative evaluation in designing systems for usability.
People & Computers II, p196-214.
- Hewett T T. 1987.
The Drexel disk: an electronic "guidebook".
People & Computers III, p115-130.
- Hoc J-M, Green T R G, Samurcay R & Gilmore D J (Eds). 1990.
Psychology of Programming.
Academic Press.
- Holt R W, Boehm-Davis D A & Shultz A C. 1987.
Mental representations of programs for student and professional programmers.
in Empirical Studies of Programmers, Second Workshop. Sheppard S, Olson G M & Soloway E (Eds). Ablex, p33-46.

Hulme C. 1985.

Extracting information from printed and electronically presented text.

in Fundamentals of Human-Computer Interaction. Monk A. Editor. Academic Press, p35-47.

Humphreys C A. 1992.

The design and application of visually oriented tools for use during software development.

Paper from the Psychology of (Special) Interest Group (PPIG-4) Workshop, held at Loughborough University, 2-4/1/92.

Hutchins E L, Hollan J D & Norman D A. 1986.

Direct Manipulation Interfaces.

in User Centered System Design. Norman D A & Draper S W, Editors. Lawrence Erlbaum, p87-124.

Jerrams-Smith J. 1985.

SUSI - a smart user-system interface.

People & Computers I, p211-220.

Johnson P & Cook S. Editors. 1985.

People and Computers I: Designing The Interface.

Proceedings of the British Computer Society, Human Computer Interaction Specialist Group, University of East Anglia, 17-20 Sept 1985, Cambridge Univ Press.

Kessler C M & Anderson J R. 1986.

A model of novice debugging in Lisp.

in Empirical Studies of Programmers. Edited by Soloway E & Iyengar S. Ablex, p198-212.

Kieras D & Polson P G. 1985.

An approach to the formal analysis of user complexity.

IJMMS 22(4), p365-394.

Kindborg M & Kollerbauer A. 1987.

Visual languages and Human Computer Interaction.

People & Computers III, p175-188 #study of cartoons and comics.

Lammers S. 1986.

Programmers at Work.

Microsoft Press.

Laurel B K. 1986.

Interface as Mimesis.

in User Centered System Design. Norman D A & Draper S W, Editors. Lawrence Erlbaum, p67-86.

Lenorovitz A R, Phillips M D, Ardrey R S & Kloster G V. 1984.

A taxonomic approach to the specification of human computer interaction.

INTERACT 84.

- Letovsky S. 1986.
Cognitive processes in program comprehension.
in Soloway E & Iyengar S (Eds). Empirical Studies of Programmers. Ablex, p58-79.
- Letovsky S, Pinto J, Lampert R & Soloway E. 1987.
A cognitive analysis of code inspection.
in Empirical Studies of Programmers, Second Workshop. Sheppard S, Olson G M & Soloway E (Eds). Ablex, p231-247.
- Leventhal L M. 1988.
Experience of programming beauty: some patterns of programming aesthetics.
IJMMS 28(5), p525-550.
- Lewis C & Olson G M. 1987.
Can principles of cognition lower the barriers to programming.
in Empirical Studies of Programmers, Second Workshop. Sheppard S, Olson G M & Soloway E (Eds). Ablex, p248-263.
- Littman D C, Pinto J, Letovsky S & Soloway E. 1986.
Mental models and software maintenance.
in Soloway E & Iyengar S (Eds). Empirical Studies of Programmers. Ablex, p80-98.
- Lodding K N. 1983.
Iconic Interfacing.
IEEE CG&A March/April 1983 11-20.
- Lukey F J. 1980.
Understanding and debugging programs.
IJMMS 12(2), p189-202.
- Maguire M. 1982.
An evaluation of published recommendations on the design of man-computer dialogues.
IJMMS 16(3), p237-261.
- Marr D. 1982.
Vision.
W H Freeman, Chap 1, p8-38.
- Matlin M W. 1989.
Cognition.
Holt, Rinehart & Winston Inc.
- Miara R J, Musselman J A, Navarro J A & Shneiderman B. 1983.
Program indentation and comprehensibility.
Comm ACM 26(11) p861-867.
- Miller G A. 1957.
The magical number seven, plus or minus two: some limits on our capacity for processing information.
Psychological Review 1957, Vol 63, p81-97.

Molzberger P. 1983.

Aesthetics and programming.

CHI'83, p247-250.

Monk A. 1985.

Fundamentals of Human-Computer Interaction.

Academic Press.

Monk A. 1989.

The Personal Browser: a tool for directed navigation in Hypertext systems.

Interacting with Computers: the Interdisciplinary Journal of Human-Computer Interaction 1(2) p190-196.

Moran T P. 1981.

The Command Language Grammar: A Representation for the User Interface of Interactive Computer Systems.

IJMMS 15(1), p3-50.

Mynatt B. 1990.

Why program comprehension is (or is not) affected by surface features.

INTERACT'90, p945-950.

Nanja M & Cook C R. 1987.

An analysis of the on-line debugging process.

in Empirical Studies of Programmers, Second Workshop. Sheppard S, Olson G M & Soloway E (Eds). Ablex, p172-184.

Norman D A & Draper S W, Editors. 1986.

User-Centered System Design: New Perspectives on Human-Computer Interaction.

Hillsdale, NJ: Lawrence Erlbaum Associates.

Norman D A. 1986.

Cognitive Engineering.

in User Centered System Design. Norman D A & Draper S W, Editors. Lawrence Erlbaum, p31-62.

Norman D A. 1987.

Cognitive Engineering - Cognitive Science.

in Carroll J M (Ed); Interfacing Thought: Cognitive Aspects of Human-Computer Interaction; MIT Press, p325-336.

Norman D A. 1988.

The Psychology of Everyday Things.

Basic Books (HarperCollins).

Norman D A. 1991.

Cognitive artifacts.

in Carroll J M (Ed); Designing Interaction: Psychology at the Human-Computer Interface; Cambridge University Press, p17-38.

O'Malley C & Sharples M. 1986.

Tools for management and support of multiple constraints in a writer's assistant.
People & Computers II, p115-131.

Ormerod T. 1990.

Human cognition and programming.

in Hoc J-M, Green T R G, Samurcay R & Gilmore D J (Eds); Psychology of Programming; Academic Press, p63-82.

Pattis R E. 1981.

Karel the Robot: a gentle introduction to the art of programming with Pascal.
John Wiley & Sons.

Payne S J & Green T R G. 1983.

The user's perception of the interaction language: a two-level model.
CHI'83.

Payne S J & Green T R G. 1984.

Task-Action Grammars: a Model of the Mental Representation of Task Language.
Human-Computer Interaction.

Payne S J, Sime M E & Green T R G. 1984.

Perceptual cueing in a simple command language.
IJMMS 21, p19-29.

Pennington N & Grabowski B. 1990.

The tasks of programming.

in Hoc J-M, Green T R G, Samurcay R & Gilmore D J (Eds); Psychology of Programming; Academic Press, p45-62.

Pennington N. 1987.

Comprehension strategies in programming.

in Empirical Studies of Programmers, Second Workshop. Sheppard S, Olson G M & Soloway E (Eds). Ablex, p100-113.

Pennington N. 1987b.

Stimulus structures and mental representations in expert comprehension of computer programs.

Cognitive Psychology 19 p295-341.

Perkins D N & Martin F. 1986.

Fragile knowledge and neglected strategies in novice programmers.

in Soloway E & Iyengar S (Eds). Empirical Studies of Programmers. Ablex, p213-229.

Petre M. 1990.

Expert programmers and programming languages.

in Hoc J-M, Green T R G, Samurcay R & Gilmore D J (Eds); Psychology of Programming; Academic Press, p103-115.

Polson P G, Bovair S & Kieras D. 1987.
Transfer between text editors.
CHI + GI '87, p27-32.

Rasmussen J. 1981.
The human as a system component.
in Human Interaction with Computers; Smith H T & Green T R G (Eds); Academic Press.

Rasmussen J. 1981.
Models of mental strategies in process plant diagnosis.
in Rasmussen J & Rouse W B, (Eds), Human Detection and Diagnosis of System Failures, New York: Plenum Press.

Ratcliff B & Siddiqi J I A. 1985.
An empirical investigation into problem decomposition strategies used in program design.
IJMMS 22(1), p77-90.

Rector A L, Newton P D & Marsden P H. 1985.
What kind of system does an expert need?.
People & Computers I, p239-.

Redmond R T & Gasen J B. 1989.
Measuring change in the programming process.
IJMMS 30(6), p697-711.

Reisner P. 1981.
Formal Grammar and Human Factors Design of an Interactive Graphics System.
IEEE Transactions on Software Engineering Vol SE7 No 2 March 1981, p229-240.

Reisner P. 1984.
Formal grammar as a tool for analysing ease of use: some fundamental concepts.
Human Factors in Computer Systems, edited by Thomas J C & Schneider M L, Ablex Corp, New Jersey.

Reitman Olson J S, Whitten W B & Gruenenfelder T M. 1984.
A general user interface for creating and displaying tree-structures, hierarchies, decision trees, and nested menus.
in Vassiliou Y. Editor. Human Factors and Interactive Computer Systems.
Proceedings of the NYU Symposium on User Interfaces New York, May 26-28, 1982, Ablex Publishing Corporation, p223-244.

Rich E. 1983.
Users are individuals: individualizing user models.
IJMMS 18(3), p199-214.

Riecken R D, Koenemann-Belliveau J & Robertson S P. 1991.
What do expert programmers communicate by means of descriptive commenting?.
in Koenemann-Belliveau J, Moher T G & Robertson S P (Eds). Empirical Studies of Programmers: Fourth Workshop. Ablex, p177-195.

- Rist R S. 1986.
Plans in programming: definition, demonstration, and development.
in Soloway E & Iyengar S (Eds). Empirical Studies of Programmers. Ablex, p28-47.
- Rist R S. 1990.
Variability in program design: the interaction of process with knowledge.
IJMMS 33(3), p305-322.
- Robertson S P & Yu C-C. 1990.
Common cognitive representations of program code across tasks and languages.
IJMMS 33(3), p343-360.
- Robertson S P, Davis E F, Okabe K & Fitz-Randolf. 1990.
Program comprehension beyond the line.
INTERACT'90, p959-963.
- Rogers Y R. 1986a.
An investigation of features important in pictorial representation of abstract concepts.
CHI'86, p353.
- Rogers Y. 1986b.
Evaluating the meaningfulness of icon sets to represent command operations.
People & Computers II, p586-603 #study of the visual perception and subsequent interpretation of the icons.
- Runciman C & Hammond N. 1986.
User programs: a way to match computer systems and human cognition.
People & Computers II, p464-481 #programming user cognitive processes, technique to codify the user's mental model and try to match it to the designer's model.
- Saariluoma P & Sajaniemi J. 1989.
Visual information chunking in spreadsheet calculation.
IJMMS 30(5), p475-488.
- Schindler M. 1982.
Software toolkit for the Microcomputer.
Hayden.
- Schneider M L. 1984.
Ergonomic considerations in the design of command languages.
in Vassiliou Y. Editor. Human Factors and Interactive Computer Systems.
Proceedings of the NYU Symposium on User Interfaces New York, May 26-28, 1982, Ablex Publishing Corporation, p141-162.
- Scholtz J & Wiedenbeck S. 1993.
An analysis of novice programmers learning a second language.
in Cook C R, Scholtz J C & Spohrer J C (Eds). Empirical Studies of Programmers: Fifth Workshop. Ablex, p187-205.

Shackel B. 1984a.

Designing for people in the age of information.
INTERACT'84, p9-20.

Shackel B. 1984b.

Information technology - a challenge to ergonomics and design.
BIT 3(4), p263-276.

Shackel B. 1986.

Ergonomics in design for usability.
People & Computers II, p44-64.

Shneiderman B & Mayer R. 1979.

Syntactic/semantic interactions in programmer behaviour: a model and experimental results.

International Journal of Information Science 7 p219-239.

Shneiderman B. 1980.

Software Psychology.
Winthrop Publishers Inc..

Shneiderman B. 1986.

Empirical studies of programmers: the territory, paths, and destinations.
in Soloway E & Iyengar S (Eds). Empirical Studies of Programmers. Soloway E & Iyengar S (Eds). Ablex, p1-12.

Shneiderman B. 1986.

Seven plus or minus two central issues in human-computer interaction.
CHI'86, p343-349.

Shneiderman B. 1987.

Designing The User Interface: Strategies for Effective Human Computer Interaction.
Addison-Wesley.

Siddiqi J I A & Ratcliff B. 1989.

Specification influences in program design.
IJMMS 31(4), p393-404.

Siddiqi J I A. 1985.

A model of program designer behaviour.
People & Computers I, p369-379.

Soloway E & Ehrlich K. 1984.

Empirical studies of programming knowledge.
IEEE Trans. on Software Engineering SE-10, 595-609.

Soloway E, Adelson B & Ehrlich K. 1988.

Knowledge and processes in the comprehension of computer programs.
in Chi M T H, Glaser R & Farr M J (Eds); The Nature of Expertise; Lawrence Erlbaum, p129-152.

Soloway E, Bonar J & Ehrlich K. 1983.

Cognitive strategy and looping constructs: an empirical study.

Comm ACM 26(11) p853-860.

Sommerville I. 1988.

IPSE user interfaces - issues and requirements.

IEE Colloquium on "The Role of Man-Machine Interaction in Software Engineering Environments", 4th Jan 1988, Digest 1988/1.

Spohrer J C & Soloway E. 1986.

Analyzing the high frequency bugs in novice programs.

in Empirical Studies of Programmers. Edited by Soloway E & Iyengar S. Ablex, p230-251.

Suchman L. 1987.

Plans and situated actions: the problem of human-machine communication.

Cambridge, 1987.

Sutcliffe A G & Old A C. 1987.

Do users know they have user models?.

INTERACT'87, p35-42.

Teitelbaum T & Reps T.

The CORNELL program synthesizer: a syntax-directed program environment.

in Interactive Programming Environments. Barstow D R, Shrobe H E & Sandewell E. Editors. McGraw Hill International Edition. 1986. p97-116.

Teitelman W. 1972.

Automated programming: the programmer's assistant.

in Interactive Programming Environments. Barstow D R, Shrobe H E & Sandewell E. Editors. McGraw Hill International Edition. 1986. p232-239.

Teitelman W. 1977.

A display-oriented programmer's assistant.

in Interactive Programming Environments. Barstow D R, Shrobe H E & Sandewell E. Editors. McGraw Hill International Edition. 1986. p240-287.

Thimbleby H. 1985.

User interface design: generative user engineering principles.

in Fundamentals of Human-Computer Interaction. Monk A. Editor. Academic Press, p165-180.

Thomas J C. 1984.

Organizing for human factors.

in Vassiliou Y. Editor. Human Factors and Interactive Computer Systems.

Proceedings of the NYU Symposium on User Interfaces New York, May 26-28, 1982, Ablex Publishing Corporation, p29-46.

- Thompson P. 1985.
Visual perception: an intelligent system with limited bandwidth.
in Fundamentals of Human-Computer Interaction. Monk A. Editor. Academic Press,
p5-33.
- Treisman A. 1982.
Perceptual Grouping & Attention In Visual Search For Features & Objects.
Journal of Experimental Psychology: Human Perception & Performance 1982, Vol 8
No 2 p194-214.
- Treisman A. 1986.
Features & Objects In Visual Processing.
Scientific American Nov 1986 p106-115.
- Tunnicliffe A. 1987.
The dual & asymmetric brain.
Draft report by PhD research student of Loughborough University, supervised by
Scrivener S A R.
- van der Veer G C, Green T R G, Hoc J-M & Murray D M (Eds). 1988.
Working with Computers: Theory versus Outcome.
Academic Press.
- van Dijk & Kintsch W. 1983.
Strategies of discourse comprehension.
New York: Academic Press.
- van Laar D. 1989.
Evaluating a colour coding support tool.
in Sutcliffe A & Macaulay L, Editors, People & Computers V. Cambridge University
Press.
- van Nes F L. 1984.
Limits of visual perception in the technology of visual display terminals.
BIT 3(4), p371-378.
- van Nes F L. 1986.
Space, colour and typography on visual display terminals.
BIT 5(2), p99-118.
- Vassiliou Y. Editor. 1984.
Human Factors and Interactive Computer Systems.
Proceedings of the NYU Symposium on User Interfaces New York, May 26-28,
1982, Ablex Publishing Corporation.
- Vessey I. 1989.
Toward a theory of computer program bugs: an empirical test.
IJMMS 30(1), p23-46.

- Visser W & Hoc J-M. 1990.
Expert software design strategies.
in Hoc J-M, Green T R G, Samurcay R & Gilmore D J (Eds); *Psychology of Programming*; Academic Press, p235-249.
- Visser W. 1987.
Strategies in programming programmable controllers: a field study on a professional programmer.
in *Empirical Studies of Programmers, Second Workshop*. Sheppard S, Olson G M & Soloway E (Eds). Ablex, p217-230.
- Visser W. 1990.
More or less following a plan during design: opportunistic deviations in specification.
IJMMS 33(3), p247-278.
- Waddington R & Henry R. 1990.
Expert programmers re-establish intentions when debugging another programmer's program.
INTERACT'90, p965-970.
- Wastell D. 1990.
Mental effort and task performance: towards a psychophysiology of human-computer interaction.
INTERACT'90 p107-112.
- Waters R C. 1972.
The programmer's apprentice.
in *Interactive Programming Environments*. Barstow D R, Shrobe H E & Sandewell E. Editors. McGraw Hill International Edition. 1986. p464-486.
- Weiser M & Lyle J. 1986.
Experiments on slicing-based debugging aids.
in Soloway E & Iyengar S (Eds). *Empirical Studies of Programmers* Ablex, p187-197.
- Weiser M. 1982a.
Programmers use slices when debugging.
Comm ACM 25(7) 446-452.
- Weiser M. 1982b.
Program slicing.
IEEE Trans on Software Eng SE-10 352-357.
- Wertz H. 1982.
Stereotyped program debugging: an aid for novice programmers.
IJMMS 16(4), 379-392.
- Wiedenbeck S. 1985.
Novice/expert differences in programming skill.
IJMMS 23(4), p383-390.

- Wiedenbeck S. 1986a.
Processes in computer program comprehension.
in Soloway E & Iyengar S (Eds). Empirical Studies of Programmers. Ablex, p48-57.
- Wiedenbeck S. 1986b.
Beacons in computer program comprehension.
IJMMS 25(6), p697-709.
- Winfield I. 1986.
Human Resources and Computing.
Heinemann: London.
- Winograd T & Flores F. 1986.
Understanding Computers and Cognition.
Ablex.
- Witkin H A, Dyke R B, .. 1962.
Psychological Differentiation.
Wiley 1962 (Erlbaum 1974 reprint).
- Wright P & Lickorish A. 1988.
Colour cues as location aids in lengthy texts on screen and paper.
BIT 7(1), p11-30.
- Wu Q & Anderson J R. 1991.
Strategy selection and change in PASCAL programming.
in Koenemann-Belliveau J, Moher T G & Robertson S P (Eds). Empirical Studies of Programmers: Fourth Workshop. Ablex, p227-238.
- Young R M & Harris J E. 1986.
A viewdata-structure editor designed around a task/action mapping.
People & Computers II, p435-446 #hierarchy of associations & implications, edit assistant eases cognitive effort.
- Young R M. 1981.
The machine inside the machine: users' models of pocket calculators.
IJMMS 15(1), p51-86.
- Young R M. 1985.
User Models as Design Tools for Software Engineers.
Alvey Workshop on MMI/SE Sept 1985.
- Youngs E A. 1974.
Human errors in programming.
IJMMS 6(3), p361-376.
- Yourdon & Constantine. 1979.
Structured Design.
Prentice-Hall.

Appendix 3A - Comparison of Student Numbers & Percentages For Each Tool

Key to Table Headings

actual = actual number of students who scored the rating scale

66 students in total answered the questionnaire

Therefore, 66 - actual = number of students who skipped the question, and didn't vote.

Rating scales ranged from 1 to 5, 1 being the most negative & 5 being the most positive choices on the scale. Minimum vote was 0 students, and maximum vote was 66 students.

mean = sample mean value, with "actual" number of students as divisor

stdv = standard deviation from the mean

mndv = mean deviation

dif2 = (scale[4] + scale[5]) - (scale[1] + scale[2])

= numerical (or percentage) difference between positive and negative sides of the rating scale, showing the size of the swing. Positive values indicate positive swing, Negative values indicate negative swing.

Each tool has a double set of values for each heading.

The first set of values shows student numbers, with the corresponding percentage values below - except for "actual" values which is the same non—percentage "student numbers" value in both cases.

Underlines are used to separate each tool's set of paired values, and to aid reading.

Meanings of Each of the 5-Point Rating Scales For Each Dimension

Scale	1	2	3	4	5
Usefulness	useless	not very useful	Ok	useful	essential
Frequency	never	once or twice	sometimes	often	usually
Ease of Use	difficult	fairly difficult	Ok	fairly easy	easy
Likeability	disliked	mildly disliked	Ok	mildly liked	liked
Other Method	never	once or twice	sometimes	often	usually

Comparison of Student Numbers & Percentages For Each Tool - in Tool Order	Page
Usefulness	21
Frequency of Use	22
Ease of Use	23
Likeability	24
Other Method : Frequency of Use	25
Summary of Means and Rankings for Each Dimension in Tool Order	25
Frequency of Use - in Order of Decreasing Mean Rating Scale Value	26

Comparison of Student Numbers & Percentages For Each Tool - in Tool Order

Usefulness

Name of Tool	Votes actual	Votes per Rating Scale					Mean Data			Swing Data			Majority
		1	2	3	4	5	mean	stdv	mndv	1+2	3	4+5	dif2
Cut	64	0	2	9	33	20	4.11	0.75	0.56	2	9	53	51
%		0.0	3.1	14.1	51.6	31.2				3.1	14.1	82.8	79.7%
Copy	61	0	3	11	30	17	4.00	0.81	0.56	3	11	47	44
%		0.0	4.9	18.0	49.2	27.9				4.9	18.0	77.0	72.1%
Paste	64	0	3	7	32	22	4.14	0.79	0.59	3	7	54	51
%		0.0	4.7	10.9	50.0	34.4				4.7	10.9	84.4	79.7%
Find	36	1	3	13	16	3	3.47	0.87	0.72	4	13	19	15
%		2.8	8.3	36.1	44.4	8.3				11.1	36.1	52.8	41.7%
Replace	38	2	2	13	17	4	3.50	0.94	0.76	4	13	21	17
%		5.3	5.3	34.2	44.7	10.5				10.5	34.2	55.3	44.7%
Check	60	0	1	7	23	29	4.33	0.75	0.64	1	7	52	51
%		0.0	1.7	11.7	38.3	48.3				1.7	11.7	86.7	85.0%
Reset	57	1	2	18	23	13	3.79	0.89	0.72	3	18	36	33
%		1.8	3.5	31.6	40.4	22.8				5.3	31.6	63.2	57.9%
Go	63	0	0	5	7	51	4.73	0.60	0.44	0	5	58	58
%		0.0	0.0	7.9	11.1	81.0				0.0	7.9	92.1	92.1%
Go-go	30	0	5	7	10	8	3.70	1.04	0.89	5	7	18	13
%		0.0	16.7	23.3	33.3	26.7				16.7	23.3	60.0	43.3%
Step	47	2	5	11	21	8	3.60	1.02	0.84	7	11	29	22
%		4.3	10.6	23.4	44.7	17.0				14.9	23.4	61.7	46.8%
Step-Step	48	0	3	9	28	8	3.85	0.76	0.55	3	9	36	33
%		0.0	6.2	18.7	58.3	16.7				6.2	18.7	75.0	68.7%
Stops In	25	0	2	8	12	3	3.64	0.79	0.67	2	8	15	13
%		0.0	8.0	32.0	48.0	12.0				8.0	32.0	60.0	52.0%
Instant	13	0	2	7	4	0	3.15	0.66	0.52	2	7	4	2
%		0.0	15.4	53.8	30.8	0.0				15.4	53.8	30.8	15.4%
Observe	26	0	1	7	12	6	3.88	0.80	0.62	1	7	18	17
%		0.0	3.8	26.9	46.2	23.1				3.8	26.9	69.2	65.4%
Clipboard	39	0	1	11	15	12	3.97	0.83	0.65	1	11	27	26
%		0.0	2.6	28.2	38.5	30.8				2.6	28.2	69.2	66.7%
Font Ctrl	50	0	3	16	24	7	3.70	0.78	0.65	3	16	31	28
%		0.0	6.0	32.0	48.0	14.0				6.0	32.0	62.0	56.0%
Indent Ctrl	10	0	1	6	1	2	3.40	0.92	0.76	1	6	3	2
%		0.0	10.0	60.0	10.0	20.0				10.0	60.0	30.0	20.0%
Layout	66	4	0	18	35	9	3.68	0.92	0.70	4	18	44	40
%		6.1	0.0	27.3	53.0	13.6				6.1	27.3	66.7	60.6%
Highlighting	66	2	4	5	41	14	3.92	0.89	0.55	6	5	55	49
%		3.0	6.1	7.6	62.1	21.2				9.1	7.6	83.3	74.2%
Bracketing	63	1	2	17	35	8	3.75	0.78	0.60	3	17	43	40
%		1.6	3.2	27.0	55.6	12.7				4.8	27.0	68.3	63.5%
Error Msgs	65	2	2	12	33	16	3.91	0.91	0.63	4	12	49	45
%		3.1	3.1	18.5	50.8	24.6				6.2	18.5	75.4	69.2%

Frequency of Use

Name of Tool	Votes actual	Votes per Rating Scale					Mean Data			Swing Data			Majority
		1	2	3	4	5	mean	stdv	mndv	1+2	3	4+5	dif2
Cut %	66	2 3.0	7 10.6	19 28.8	31 47.0	7 10.6	3.52	0.93	0.77	9 13.6	19 28.8	38 57.6	29 43.9%
Copy %	66	5 7.6	11 16.7	18 27.3	25 37.9	7 10.6	3.27	1.09	0.92	16 24.2	18 27.3	32 48.5	16 24.2%
Paste %	66	2 3.0	8 12.1	15 22.7	32 48.5	9 13.6	3.58	0.97	0.80	10 15.2	15 22.7	41 62.1	31 47.0%
Find %	66	30 45.5	18 27.3	13 19.7	5 7.6	0 0.0	1.89	0.97	0.81	48 72.7	13 19.7	5 7.6	-43 -65.2%
Replace %	66	28 42.4	19 28.8	11 16.7	7 10.6	1 1.5	2.00	1.07	0.85	47 71.2	11 16.7	8 12.1	-39 -59.1%
Check %	66	6 9.1	4 6.1	11 16.7	21 31.8	24 36.4	3.80	1.25	1.00	10 15.2	11 16.7	45 68.2	35 53.0%
Reset %	66	9 13.6	20 30.3	23 34.8	8 12.1	6 9.1	2.73	1.12	0.91	29 43.9	23 34.8	14 21.2	-15 -22.7%
Go %	66	3 4.5	1 1.5	7 10.6	10 15.2	45 68.2	4.41	1.04	0.81	4 6.1	7 10.6	55 83.3	51 77.3%
Go-go %	66	36 54.5	14 21.2	7 10.6	7 10.6	2 3.0	1.86	1.15	0.94	50 75.8	7 10.6	9 13.6	-41 -62.1%
Step %	66	19 28.8	16 24.2	28 42.4	1 1.5	2 3.0	2.26	0.99	0.85	35 53.0	28 42.4	3 4.5	-32 -48.5%
Step-Step %	65	17 26.2	13 20.0	23 35.4	11 16.9	1 1.5	2.48	1.10	0.96	30 46.2	23 35.4	12 18.5	-18 -27.7%
Stops In %	64	39 60.9	13 20.3	8 12.5	4 6.2	0 0.0	1.64	0.92	0.78	52 81.2	8 12.5	4 6.2	-48 -75.0%
Instant %	65	52 80.0	6 9.2	6 9.2	1 1.5	0 0.0	1.32	0.70	0.52	58 89.2	6 9.2	1 1.5	-57 -87.7%
Observe %	65	39 60.0	10 15.4	12 18.5	4 6.2	0 0.0	1.71	0.97	0.85	49 75.4	12 18.5	4 6.2	-45 -69.2%
Clipboard %	65	26 40.0	14 21.5	11 16.9	10 15.4	4 6.2	2.26	1.29	1.12	40 61.5	11 16.9	14 21.5	-26 -40.0%
Font Ctrl %	66	16 24.2	13 19.7	18 27.3	13 19.7	6 9.1	2.70	1.28	1.10	29 43.9	18 27.3	19 28.8	-10 -15.2%
Indent Ctrl %	65	55 84.6	7 10.8	3 4.6	0 0.0	0 0.0	1.20	0.50	0.34	62 95.4	3 4.6	0 0.0	-62 -95.4%

Ease of Use

Name of Tool	Votes actual	Votes per Rating Scale					Mean Data			Swing Data			Majority dif2
		1	2	3	4	5	mean	stdv	mndv	1+2	3	4+5	
Cut %	64	0	1	11	15	37	4.37	0.82	0.72	1	11	52	51
		0.0	1.6	17.2	23.4	57.8				1.6	17.2	81.2	79.7%
Copy %	61	0	2	13	15	31	4.23	0.89	0.78	2	13	46	44
		0.0	3.3	21.3	24.6	50.8				3.3	21.3	75.4	72.1%
Paste %	64	0	2	13	15	34	4.27	0.89	0.78	2	13	49	47
		0.0	3.1	20.3	23.4	53.1				3.1	20.3	76.6	73.4%
Find %	36	1	2	19	6	8	3.50	0.99	0.83	3	19	14	11
		2.8	5.6	52.8	16.7	22.2				8.3	52.8	38.9	30.6%
Replace %	38	1	2	17	10	8	3.58	0.96	0.82	3	17	18	15
		2.6	5.3	44.7	26.3	21.1				7.9	44.7	47.4	39.5%
Check %	60	0	0	8	9	43	4.58	0.71	0.60	0	8	52	52
		0.0	0.0	13.3	15.0	71.7				0.0	13.3	86.7	86.7%
Reset %	57	1	2	17	10	27	4.05	1.03	0.90	3	17	37	34
		1.8	3.5	29.8	17.5	47.4				5.3	29.8	64.9	59.6%
Go %	62	0	0	6	6	50	4.71	0.63	0.47	0	6	56	56
		0.0	0.0	9.7	9.7	80.6				0.0	9.7	90.3	90.3%
Go-go %	30	0	0	7	4	19	4.40	0.84	0.76	0	7	23	23
		0.0	0.0	23.3	13.3	63.3				0.0	23.3	76.7	76.7%
Step %	47	1	0	17	15	14	3.87	0.91	0.75	1	17	29	28
		2.1	0.0	36.2	31.9	29.8				2.1	36.2	61.7	59.6%
Step-Step %	48	0	0	17	13	18	4.02	0.85	0.73	0	17	31	31
		0.0	0.0	35.4	27.1	37.5				0.0	35.4	64.6	64.6%
Stops In %	24	0	0	13	4	7	3.75	0.88	0.81	0	13	11	11
		0.0	0.0	54.2	16.7	29.2				0.0	54.2	45.8	45.8%
Instant %	13	0	0	7	3	3	3.69	0.82	0.75	0	7	6	6
		0.0	0.0	53.8	23.1	23.1				0.0	53.8	46.2	46.2%
Observe %	26	0	0	10	10	6	3.85	0.77	0.65	0	10	16	16
		0.0	0.0	38.5	38.5	23.1				0.0	38.5	61.5	61.5%
Clipboard %	38	0	2	14	12	10	3.79	0.89	0.77	2	14	22	20
		0.0	5.3	36.8	31.6	26.3				5.3	36.8	57.9	52.6%
Font Ctrl %	50	0	1	9	18	22	4.22	0.81	0.69	1	9	40	39
		0.0	2.0	18.0	36.0	44.0				2.0	18.0	80.0	78.0%
Indent Ctrl %	10	0	0	4	4	2	3.80	0.75	0.64	0	4	6	6
		0.0	0.0	40.0	40.0	20.0				0.0	40.0	60.0	60.0%

Likeability

Name of Tool	Votes actual	Votes per Rating Scale					Mean Data			Swing Data			Majority
		1	2	3	4	5	mean	stdv	mndv	1+2	3	4+5	dif2
Cut	64	0	1	23	13	27	4.03	0.92	0.82	1	23	40	39
%		0.0	1.6	35.9	20.3	42.2				1.6	35.9	62.5	60.9%
Copy	61	0	2	20	15	24	4.00	0.92	0.79	2	20	39	37
%		0.0	3.3	32.8	24.6	39.3				3.3	32.8	63.9	60.7%
Paste	64	0	2	16	23	23	4.05	0.86	0.69	2	16	46	44
%		0.0	3.1	25.0	35.9	35.9				3.1	25.0	71.9	68.7%
Find	36	2	4	17	11	2	3.19	0.91	0.69	6	17	13	7
%		5.6	11.1	47.2	30.6	5.6				16.7	47.2	36.1	19.4%
Replace	37	1	4	15	14	3	3.38	0.88	0.73	5	15	17	12
%		2.7	10.8	40.5	37.8	8.1				13.5	40.5	45.9	32.4%
Check	60	0	0	19	19	22	4.05	0.83	0.70	0	19	41	41
%		0.0	0.0	31.7	31.7	36.7				0.0	31.7	68.3	68.3%
Reset	56	3	1	31	13	8	3.39	0.94	0.74	4	31	21	17
%		5.4	1.8	55.4	23.2	14.3				7.1	55.4	37.5	30.4%
Go	62	0	0	17	5	40	4.37	0.88	0.81	0	17	45	45
%		0.0	0.0	27.4	8.1	64.5				0.0	27.4	72.6	72.6%
Go-go	30	0	1	16	6	7	3.63	0.87	0.78	1	16	13	12
%		0.0	3.3	53.3	20.0	23.3				3.3	53.3	43.3	40.0%
Step	47	1	8	14	19	5	3.40	0.96	0.82	9	14	24	15
%		2.1	17.0	29.8	40.4	10.6				19.1	29.8	51.1	31.9%
Step-Step	48	1	3	19	11	14	3.71	1.02	0.89	4	19	25	21
%		2.1	6.2	39.6	22.9	29.2				8.3	39.6	52.1	43.7%
Stops In	25	1	0	14	9	1	3.36	0.74	0.59	1	14	10	9
%		4.0	0.0	56.0	36.0	4.0				4.0	56.0	40.0	36.0%
Instant	13	1	0	7	3	2	3.38	1.00	0.78	1	7	5	4
%		7.7	0.0	53.8	23.1	15.4				7.7	53.8	38.5	30.8%
Observe	26	0	0	11	8	7	3.85	0.82	0.72	0	11	15	15
%		0.0	0.0	42.3	30.8	26.9				0.0	42.3	57.7	57.7%
Clipboard	39	0	2	14	12	11	3.82	0.90	0.78	2	14	23	21
%		0.0	5.1	35.9	30.8	28.2				5.1	35.9	59.0	53.8%
Font Ctrl	50	0	0	14	17	19	4.10	0.81	0.68	0	14	36	36
%		0.0	0.0	28.0	34.0	38.0				0.0	28.0	72.0	72.0%
Indent Ctrl	10	0	0	7	2	1	3.40	0.66	0.56	0	7	3	3
%		0.0	0.0	70.0	20.0	10.0				0.0	70.0	30.0	30.0%
Layout	66	5	4	14	14	29	3.88	1.25	1.04	9	14	43	34
%		7.6	6.1	21.2	21.2	43.9				13.6	21.2	65.2	51.5%
Highlighting	66	2	3	8	14	39	4.29	1.04	0.84	5	8	53	48
%		3.0	4.5	12.1	21.2	59.1				7.6	12.1	80.3	72.7%
Bracketing	63	5	6	21	15	16	3.49	1.19	1.01	11	21	31	20
%		7.9	9.5	33.3	23.8	25.4				17.5	33.3	49.2	31.7%
Error Msgs	65	10	5	14	15	21	3.49	1.40	1.21	15	14	36	21
%		15.4	7.7	21.5	23.1	32.3				23.1	21.5	55.4	32.3%

Other Method : Frequency of Use

Name of Tool	Votes actual	Votes per Rating Scale					Mean Data			Swing Data			Majority dif2
		1	2	3	4	5	mean	stdv	mndv	1+2	3	4+5	
Find	48	34	3	8	3	0	1.58	0.98	0.83	37	8	3	-34
%		70.8	6.2	16.7	6.2	0.0				77.1	16.7	6.2	-70.8%
Replace	43	31	4	6	2	0	1.51	0.90	0.74	35	6	2	-33
%		72.1	9.3	14.0	4.7	0.0				81.4	14.0	4.7	-76.7%
Check	43	32	3	5	2	1	1.53	1.02	0.80	35	5	3	-32
%		74.4	7.0	11.6	4.7	2.3				81.4	11.6	7.0	-74.4%
Reset	44	35	3	4	2	0	1.39	0.83	0.61	38	4	2	-36
%		79.5	6.8	9.1	4.5	0.0				86.4	9.1	4.5	-81.8%
Stops In	33	23	3	4	3	0	1.61	1.01	0.84	26	4	3	-23
%		69.7	9.1	12.1	9.1	0.0				78.8	12.1	9.1	-69.7%
Observe	37	26	6	4	0	1	1.49	0.89	0.68	32	4	1	-31
%		70.3	16.2	10.8	0.0	2.7				86.5	10.8	2.7	-83.8%
Clipboard	37	25	7	4	0	1	1.51	0.89	0.69	32	4	1	-31
%		67.6	18.9	10.8	0.0	2.7				86.5	10.8	2.7	-83.8%

Summary of Means and Rankings for Each Dimension in Tool Order

	Usefulness		Frequency		Ease of Use		Likeability		Other Method	
Cut	4.11	4	3.52	4	4.37	4	4.03	6		
Copy	4.00	5	3.27	5	4.23	6	4.00	7		
Paste	4.14	3	3.58	3	4.27	5	4.05	4		
Find	3.47	19	1.89	-12	3.50	17	3.19	21	1.58	- 2
Replace	3.50	18	2.00	-11	3.58	16	3.38	18	1.51	- 4
Check	4.33	2	3.80	2	4.58	2	4.05	4	1.53	- 3
Reset	3.79	11	2.73	- 6	4.05	8	3.39	17	1.39	- 7
Go	4.73	1	4.41	1	4.71	1	4.37	1		
Go-go	3.70	13	1.86	-13	4.40	3	3.63	12		
Step	3.60	17	2.26	- 9	3.87	10	3.40	15		
Step-step	3.85	10	2.48	- 8	4.02	9	3.71	11		
Stops in	3.64	16	1.64	-15	3.75	14	3.36	20	1.61	- 1
Instant	3.15	21	1.32	-16	3.69	15	3.38	18		
Observe	3.88	9	1.71	-14	3.85	11	3.85	9	1.49	- 6
Clipboard	3.97	6	2.26	- 9	3.79	13	3.82	10	1.51	- 4
Font Control	3.70	13	2.70	- 7	4.22	7	4.10	3		
Indent Control	3.40	20	1.20	-17	3.80	12	3.40	15		
Layout Style	3.68	15					3.88	8		
Highlighting	3.92	7					4.29	2		
Bracketing	3.75	12					3.49	13		
Error Messages	3.91	8					3.49	13		

Frequency of Use - in Order of Decreasing Mean Rating Scale Value

Name of Tool	Votes actual	Votes per Rating Scale					Mean Data			Swing Data			Majority
		1	2	3	4	5	mean	stdv	mndv	1+2	3	4+5	dif2
Go	66	3	1	7	10	45	4.41	1.04	0.81	4	7	55	51
%		4.5	1.5	10.6	15.2	68.2				6.1	10.6	83.3	77.3%
Check	66	6	4	11	21	24	3.80	1.25	1.00	10	11	45	35
%		9.1	6.1	16.7	31.8	36.4				15.2	16.7	68.2	53.0%
Paste	66	2	8	15	32	9	3.58	0.97	0.80	10	15	41	31
%		3.0	12.1	22.7	48.5	13.6				15.2	22.7	62.1	47.0%
Cut	66	2	7	19	31	7	3.52	0.93	0.77	9	19	38	29
%		3.0	10.6	28.8	47.0	10.6				13.6	28.8	57.6	43.9%
Copy	66	5	11	18	25	7	3.27	1.09	0.92	16	18	32	16
%		7.6	16.7	27.3	37.9	10.6				24.2	27.3	48.5	24.2%
Reset	66	9	20	23	8	6	2.73	1.12	0.91	29	23	14	-15
%		13.6	30.3	34.8	12.1	9.1				43.9	34.8	21.2	-22.7%
Font Ctrl	66	16	13	18	13	6	2.70	1.28	1.10	29	18	19	-10
%		24.2	19.7	27.3	19.7	9.1				43.9	27.3	28.8	-15.2%
Step-Step	65	17	13	23	11	1	2.48	1.10	0.96	30	23	12	-18
%		6.2	20.0	35.4	16.9	1.5				46.2	35.4	18.5	-27.7%
Step	66	19	16	28	1	2	2.26	0.99	0.85	35	28	3	-32
%		28.8	24.2	42.4	1.5	3.0				53.0	42.4	4.5	-48.5%
Clipboard	65	26	14	11	10	4	2.26	1.29	1.12	40	11	14	-26
%		40.0	21.5	16.9	15.4	6.2				61.5	16.9	21.5	-40.0%
Replace	66	28	19	11	7	1	2.00	1.07	0.85	47	11	8	-39
%		42.4	28.8	16.7	10.6	1.5				71.2	16.7	12.1	-59.1%
Find	66	30	18	13	5	0	1.89	0.97	0.81	48	13	5	-43
%		45.5	27.3	19.7	7.6	0.0				72.7	19.7	7.6	-65.2%
Go-go	66	36	14	7	7	2	1.86	1.15	0.94	50	7	9	-41
%		54.5	21.2	10.6	10.6	3.0				75.8	10.6	13.6	-62.1%
Observe	65	39	10	12	4	0	1.71	0.97	0.85	49	12	4	-45
%		60.0	15.4	18.5	6.2	0.0				75.4	18.5	6.2	-69.2%
Stops In	64	39	13	8	4	0	1.64	0.92	0.78	52	8	4	-48
%		60.9	20.3	12.5	6.2	0.0				81.2	12.5	6.2	-75.0%
Instant	65	52	6	6	1	0	1.32	0.70	0.52	58	6	1	-57
%		80.0	9.2	9.2	1.5	0.0				89.2	9.2	1.5	-87.7%
Indent Ctrl	65	55	7	3	0	0	1.20	0.50	0.34	62	3	0	-62
%		84.6	10.8	4.6	0.0	0.0				95.4	4.6	0.0	-95.4%

Chapter 3 - Additional Questionnaire Data

<u>Q25. Enhancements and Additional Facilities</u>	<u>Frequency</u>
<u>better error messages</u>	<u>8</u>
<u>compiler requests</u>	<u>7</u>
<u>user library of procedures and functions</u>	<u>5</u>
<u>reserved words help/syntax diagrams</u>	<u>4</u>
<u>multiple application windows</u>	<u>4</u>
<u>better file handling</u>	<u>3</u>
<u>lack of speed</u>	<u>3</u>
<u>decent hardcopy facility</u>	<u>2</u>
<u>error output to file/printer</u>	<u>2</u>
<u>colour</u>	<u>2</u>
<u>single keypress window switching</u>	<u>2</u>
<u>(non syntax) semantic checker</u>	<u>2</u>
<u>stack and heap pictures</u>	<u>2</u>
<u>RAM files</u>	<u>2</u>
<u>listing all errors in one go</u>	<u>1</u>
<u>test file creation facility</u>	<u>1</u>
<u>specification -> MacPascal translator</u>	<u>1</u>
<u>easier variable value display</u>	<u>1</u>
<u>more complete/helpful diagnostics</u>	<u>1</u>
<u>type coercion</u>	<u>1</u>
<u>user control of layout</u>	<u>1</u>
<u>better screen moving commands</u>	<u>1</u>
<u>disc editor</u>	<u>1</u>
<u>editing of text window</u>	<u>1</u>
<u>ability to save text window</u>	<u>1</u>
<u>semicolon adding utility</u>	<u>1</u>
<u>trace utility</u>	<u>1</u>

Summary of Enhancements Suggested in Questionnaire

Cut: inverse facility to delete all except prior selected and highlighted block

Better search and replace facilities

Step-step: an infinite loop indicator

Font control: ability to choose new "default" font for program text on subsequent sessions

Compiler wanted rather than interpreter

Error messages :

- should be more accurate/informative and less noisy (or even noiseless)

- either should leave error message until error is fixed

- or should be able to remove message by single keypress or mouse click and be able to call error message back up onto the screen

Should be able to send cumulative error messages to file/printer rather than being limited to the "find and fix one error at a time" mechanism due to the intrinsic nature of the interpreter

Semicolon adding utility

More complete/helpful diagnostics

Reserved words help/syntax diagrams

(non syntax) semantic checker

More comprehensive run-time error checking

Easier variable value display

Trace utility

File handling requirements :

- decent file handling

- RAM files

- assigning of I/O files by the user

- test file creation facility

Saving/editing of text window

Multiwindowing/multiple applications open and on-screen at the same time

Facility to speed up window switching

- single keypress window switching

- automatic switching and resizing on selection of a given window?

User control of text formatting (individualizing layout style)

- ability to change layout code in some sections of code

- option to choose whether to highlight reserved words or not

- option to enter text in "free format" according to user's style

Colour [eg. one colour for reserved words, one colour for variables, another colour for procedure/function calls or declarations]

Stack and heap pictures

- a way of representing the "current memory free/used ratio" graphically

Better screen moving commands in terms of scrolling

- by full/half screen, to go to start/end of file directly, etc.

Decent hardcopy facility - one that enables printing of all [or a selected portion] of the file rather than just the current window's worth

Ability to define and use library(ies) of user defined procedures/functions

Q27 Data

System or Language	Student Id No./Months Spent on System Language
amstrad	30
cp/m rml 480Z	6
cp/m 2.2 & cp/m plus	65
ibm pc	11
research machines 380z	49/12
ibm mvs/xa tso/ispf	31/84
ada pde on vax	8
hp text editor	62/6-12
view for bbc (wp)	32
wordstar	10
any basic	1 2 4 21 27 35 60 61 63
bbc basic	21 27 61 63/48
commodore 64 basic	27
fast basic	1/12
hisoft structured basic on atari	2/5
spectrum basic	27
turbo basic	35/10
vic 20 basic	27
any pascal	1 5 7 10 20 24 25 26 29 39 40 41 42 43 48 57 63 65
acornsoft pascal	7
apple ucsd pascal	5/60
hisoft pascal 80	43 65/36
iso pascal bbc	48 63/24
lsp lightspeed pascal?	7
mt+ pascal on pcw	65/36
oxford pascal	63/18
rm pascal on nimbus	25/6 42/6 48
s-pascal	7
sheffield pascal on prime	57
turbo pascal	1/5 10/18 20 25/6 26 29 39/3 40 41/6
unix pascal	24

NB. Numbers followed by "/" indicate the student identifier followed by the number of months spent on the system/language eg. "63/48" means student 63 spent 48 months (4 years) on BBC Basic.

Numbers not followed by "/" indicate students who spent 1 month per system/language.

System or Language	Student Id No./Months Spent on System Language
any c	2 27 41 64 59
lattice C on amiga	27
metacomco lattice C on Atari	2/7
ms (microsoft?) c	41/3
microsoft c	64
turbo c on ibm	59/12
turbo c professional	64
ciscobol ibm clone	5/24
cobol 8x on vax11/730	58/12
stead (cobol) on rm nimbus	34
autolisp	35/10
codeview optimizing debugger on pc	64
cad systems	62
foxbase	35/8
unix on sun	50 53/½
hisoft devpac assembler on amiga	47/5
65002 assembler lancs univ v1.6	32
z80 assembler on amstrad	15/6
macdraw & macwrite	66/6
macpaint & mackeyboard	66/1
vml	23 44 45 46 52 66
never	16 28 33 36 51

NB. Numbers followed by "/" indicate the student identifier followed by the number of months spent on the system/language eg. "63/48" means student 63 spent 48 months (4 years) on BBC Basic.

Numbers not followed by "/" indicate students who spent 1 month per system/language.

Q30 Data

The following table shows the number of students (at the left margin), and the combination in which they use methods a) b) c) d) and f). The first block shows the student numbers who usually use 1 method only. This is indicated by the definition 1u/0s. 0u/1s indicates no (0) usual method, but 1 sometimes method. 1u/1s indicates 1 method used usually and 1 other method used sometimes, that is 1 principal method and 1 secondary method.

	a	b	c	d	f	
11	u	0	0	0	0	1u/0s group
11	0	u	0	0	0	
2	0	0	u	0	0	
1	0	0	0	u	0	
1	0	0	0	0	u	
1	0	s	0	0	0	0u/1s group
1	0	0	s	0	0	
1	u	u	0	0	0	2u/0s
1	0	0	s	s	0	0u/2s
3	s	s	s	s	0	0u/4s
3	u	s	0	0	0	1u/1s
1	0	u	s	0	0	
1	0	u	0	s	0	
2	0	u	0	0	s	
2	0	s	u	0	0	
2	0	0	u	s	0	
1	s	0	0	u	0	
1	0	s	0	u	0	
1	u	s	s	0	0	1u/2s
1	u	s	0	s	0	
1	s	u	s	0	0	
1	s	u	0	s	0	
1	0	u	s	s	0	
1	s	0	u	s	0	
2	s	s	u	0	0	
1	s	u	s	s	0	1u/3s
1	s	u	0	s	s	
5	s	s	u	s	0	
1	s	s	s	0	u	
2	s	u	s	s	s	1u/4s
1	s	u	u	0	0	2u/1s
1	u	s	s	u	u	3u/2s

Pascal Debugging Strategies Experiment / Questionnaire

(A)

The objective of this experiment is to find out what methods you use to develop and debug code, and to discover the reasons why you do or do not use these methods. Ample space is provided to give these reasons and to make any comments you think are relevant.

The questions are ordered specifically to lead you through all aspects of software development and the debugging process, so that you can consider the methods you use, what effects they might have and what sort of errors you have to deal with.

There are 2 aspects to this experiment, a series of 3 debugging tasks, and a series of questions to be answered. It should take about 1½-2hrs to complete.

Questions

There are 2 types of questions.

- There are "open" questions which are intended to find out your own views/opinions about debugging, and the variety of methods and strategies that you use to detect and resolve different types of bugs.

- There are "multiple choice" questions where the expected answers are shown in square brackets. For example [Y] [M] [N], correspond to the answers Yes, Maybe and No; whereas [U] [S] [N], correspond to the answers Usually, Sometimes and No/Never. To answer No, all you would have to do is to tick the [N] box

In some cases you may have an alternative answer to those presented; if so, I would like you to write down your answer - with a brief explanation (if necessary), so that I will be able to interpret your answer as you intended me to.

Some of the "multiple choice" questions have an answer "table", (or a rating scale eg. ease of use of a tool.....important ☐☐☐☐☐ not important) and you are expected to "tick" the appropriate box(es). For example, to answer the question : Show what you had for breakfast each day, Monday-Friday, in the table below** :

	Monday	Tuesday	Wednesday	Thursday	Friday
eggs	/	/			/
toast		/	/	/	/
cereal			/	/	
coffee		/	/	/	/

** You might want to add an extra row to the table, to indicate that you had orange juice on Monday and Wednesday, as follows :

orange juice / /

Please feel free to add any extra rows or columns, if you think that this will complete the answer "table", more to your satisfaction.

The "open" and "multiple choice" form of questions both appear at the beginning and end of the experiment, so that I can get your views on different aspects of debugging, error types etc., before and after attempting the debugging tasks.

NB. Some of the questions are multi-part, having parts i, ii, and iii, that explore different aspects of the same topic.

Debugging Tasks

- The central part of the experiment consists of you applying your debugging skills to 3 separate pieces of "90% complete" buggy Pascal code, which I want you to fix so that the code will compile, and be executed according to the task description provided.

These tasks are not very complex, are short (1-2 pages each), and contain helpful comments.

The purpose of these debugging tasks is to find out exactly how you go about the debugging task, and to get an idea of how long it takes you to "correct" each set of errors, and which errors you tackle first.

Boxed text is usually provided as reference information, either for specific use in subsequent questions, or simply as information which you might find useful as a memory logger.

(5)

NB. algorithmic error = error that causes the algorithm not to work properly
design error = choosing the wrong algorithm or language construct to implement the design

Information Sheet To give Help With "Vague" or "Ambiguous" Questions

What I am looking for are the reasons behind (the choice of) each answer, and your opinions on a variety of topics affecting software development and debugging styles and methods.

I hope the following list of the reasons behind the questions prove helpful in understanding and answering the questions!

41. To find out how frequently you use each of the 5 methods. Only 1 method should be chosen as "Usual" method, otherwise the percentages won't add up!

41i. To define the circumstances/reasons for using each method, and to get a good idea of what factors affect the decision to use one method rather than another.

41ii. "errors" means all the syntax and semantic errors encountered between developing the software, fixing all the syntax errors and getting it to compile, and getting it to do the required task properly (ie. as required by the task description or specifications).

5. To find out what you think the compiler's main purpose is, and to get you to define the type of errors it detects for you.

7. To see which comments you agree/disagree with about compilers, and to collect any other thoughts generated by this topic.

8. To make you consider how thoroughly you check your code for errors before and after development, and what your expectations are.

9. A means of getting you to state which inspection methods (and their depth/thoroughness) you apply to the various software development stages.

14. What I want you to do is to list all the non-syntactic errors you can think of.

16i & 16ii. "At what stage(s)" means "In what circumstances"

20. What I want you to do, is to state what specific details lead you to believe that an error exists. The 1st information box at the top of the page is a prompt to help you give this information. The 2nd information box defines detect and locate!

21. To find out how the methods/strategies ~~you~~ ^{are} used to deduce that the error lies in a specific section or line of code - how you determine the location of the suspected error.

22. To find out how you locate the source of semantic (design, logic or algorithmic) errors.

30. "At what stage(s)" means "In what circumstances"

36i. What I mean is, when you are developing code, you have ideas - mental images - (which form programming plans) which are joined together to form the eventual program that executes the required task. What I want to know is what form these mental images take. Do you just have a straight copy of the code in your head, or is it like a flow chart detailing the different routes through the code, or is it like a black box where input values go in one end and output results/events come out the other end, or is it more exotic than that, eg. like a branching river whose "child" riverlets cross each other in intricate patterns or what? Please state your answer as clearly as possible or draw a diagram with labels (if possible)!

Design of Programming & Debugging Strategies Questionnaire

The questionnaire had 2 types of questions.

There were "open" questions which were intended to find out each student's views/opinions about debugging, and the variety of methods and strategies that they use to detect and resolve different types of bugs.

There were "multiple choice" questions where the expected answers were shown in square brackets. For example [Y] [M] [N], represent the answers Yes, Maybe and No; whereas [U] [S] [N], represent the answers Usually, Sometimes and No/Never. To answer No, all the students had to do was to tick the [N] box

If the students had an alternative answer to those presented, I asked them to write down their answer - with a brief explanation (if necessary), so that I would be able to interpret their answer correctly.

The "open" and "multiple choice" form of questions both appeared at the beginning and end of the experiment, to get the student's views on different aspects of debugging, error types etc., before and after attempting the debugging tasks.

The variation in question types was intended to draw out students' opinions as fully as possible, by making the questions as explicit as possible.

Some of the questions were multi-part, having parts i, ii, and iii, to explore different aspects of the same topic.

Conventions

The following conventions apply to the resulting data, comments and conclusions drawn from the questionnaires given to the final year students. Specific numeric data that supports a statement/comment are given in round brackets. For example, in Q2, 2 students (25%) defined their favourite language as easy to use.

In data tables/matrices "SS" acts as the student identifier as in question Q4i.

In data tables, numbers appearing under the "Students" heading are the student identifiers (ranging from 1 to 8 inclusive) and NOT totals. This enables direct correspondence between the "comment response" made and the student who made it. For example, in Q3, student 5 considers himself as "very good" at Pascal programming.

In data tables 1-2 letters are used as the category label identifiers for brevity. Usually the method identifier (such as - a,b,c,d,e,), or an amalgam of the first letter of each word defining the category (such as "QD" indicating the "Quite Difficult" category in Q4), and so on.

The following labelling/meaning conventions apply unless stated to the contrary in the text.

Frequency of use labels :-

USN - usually, sometimes, never.

UOSN - usually, often, sometimes, never.

UOSRN - usually, often, sometimes, rarely, never.

Agreement with a statement, posed as a question :-

YMN - yes, maybe, no.

YDN - yes, don't mind, no.

The order category lists the resultant choice of methods (or whatever) by precedence. Thus the first item appearing in an order category is the most important. Items of the same precedence are shown as "a/c" meaning that methods a) and c) have the same precedence.

Q7A Results & Summarizing Comments for each Question

Q7A.1 Experience & Programming Ability - Q1-3

Q1. They have all done 6 projects in the previous (2nd) year, each lasting 5 weeks. With 3 projects in C, 1 in Cobol, 1 in Fortran and 1 in MacPascal. Some students did either a DBase III or Prolog project instead of one of the C projects. One student (SS3) had done an industrial year using RAMIS and SQL database languages, after doing the 2nd year projects.

Q2. Favourite programming language and environment.

Languages	Students
C	2 4 6½ 7 8½
Pascal	1 5 8
SQL	3
ARM Basic V	6

where ½ indicates a 2nd choice - a more common language to "work" or do projects in. Student 1 uses MacPascal, and Student 5 uses Hisoft Pascal.

Not surprisingly, 100% of students' loyalty to programming languages, and in 50% of cases their associated environments is mainly a question of familiarity. With 2 students (25%) describing their favourite language as easy to use. Only 3 students out of 8 identify Pascal (or one of its variants) as their favourite language.

Q3. Programming ability for Pascal.

Ability	Students
very good	5
better than average	1
average	3 4 7 8
worse than average	2 6

In contrast with the previous question, it seems that 6 out of 8 students regard themselves as having average or better than average programming ability in Pascal.

Q7A.2 Program Development and Coding Methodology - Q4

Q4i. Students were asked to define the frequency with which they used each method, below. Choices of frequency were [U]=usually, [O]=often, [S]=sometimes, [N]=never.

With the Usual method being used more than 60% of the time.

- work out the entire algorithm then translate it into code
- work out most of the algorithm then translate it into code, and fill in the missing parts as you go along
- work out a partial algorithm and continue as for b)
- use direct terminal composition - where you express the algorithm in code (on the VDU screen) directly, without working it out on paper first
- use other method(s)

- According to student's comments, e) = top-down modularisation for SS2, 4, 6 & 7 (each of these students scored it as "u"/usually, except SS4 who scored it as "o"/often).

Q4i. UOSN		U	O	S	N	sum
SS12345678						
a ssssnss	a	-	1	6	1	24
b usouonsu	b	3	2	2	1	17
c souousns	c	2	2	3	1	19
d snnnsonn	d	-	1	2	5	28
e nunonuun	e	3	1	-	4	21

Taking u=1, o=2, s=3, n=4 to give sum values above.

Order of ascending sum value gives order of frequency of use as - b,c,e,a,d - with scores of 17, 19, 21, 24, 28 respectively.

Q4ii. Students were asked to choose a method dependent on situations of varying task complexity/difficulty or code length. With 4 categories for each, as follows.
 Task Complexity/Difficulty : Simple, Average, Quite-Difficult, Very-Difficult; and
 Code Length: Short, Medium, Longish, Very-Long.
 (with 60 lines/page) <100 100-300 300-600 >600 lines code

Most students chose only 1 method per category, but some chose more than 1. These multiple methods are shown on the right hand side of the 1st pair of tables. But they also "belong" to the 2nd pair of tables where the room ran out.

Task Complexity						Code Length													
	a	b	c	d	e	order		a	b	c	d	e	order	Multiples					
S	4	1	1	3	-	a, d, b/c	S	3	2	2	3	-	a/d, b/c	S	ba	S	abc		
A	2	4	2	1	1	b, a/c, d/e	M	3	5	2	1	-	b, a, c, d	A	bc	dc	M	bca	da
QD	1	5	2	-	2	b, c/e, a	L	1	5	2	-	3	b, e, c, a	QD	cb	ce	L	cb	ce
VD	2	2	1	-	3	e, a/b, c	VL	1	2	3	-	3	e, c, b, a	VD	none	VL	cb		

These orderings result when equal weight is given to each element in a multiple answer, such as ba, where both b) and a) frequencies are incremented by 1.

Task Complexity						Code Length							
	a	b	c	d	e	order		a	b	c	d	e	order
S	3.5	0.5	1	3	-	a, d, c, b	S	2.3	1.3	1.3	3	-	d, a, b/c
A	2	3.5	1	0.5	1	b, a, c/e, d	M	1.8	4.3	1.3	0.5	-	b, a, c, d
QD	0.5	5	1	-	1.5	b, e, c, a	L	0.5	4	1	-	2.5	b, e, c, a
VD	2	2	1	-	3	e, a/b, c	VL	1	1.5	2.5	-	3	e, c, b, a

These orderings result when equal weight is given to each element in a multiple answer, such as ba, but where the score adds up to 1. Thus both b) and a) frequencies are incremented by $\frac{1}{2}$, rather than 1. A triple answer gives each component method a score of $\frac{1}{3}$ (0.3). The 1.8 score indicates a frequency score of $1 + \frac{1}{2} + \frac{1}{3} = 1.8$ approximately.

There is no major difference between responses to complexity and code length. As expected a) and d) are the predominant development strategies for simple situations. Summarizing strategies gives a) or d) for simple or short tasks; b) for average or quite difficult or long tasks; and e) for very difficult or long tasks. A common sense result, since it is easier to get a simple problem correct in short order, as the solution is usually short and easily managed. But as task and algorithm difficulty increases, the problem has to be split up and details (and bugs) proliferate - so more cautious program development strategies prevail.

Q4iii. Response to the diametrically paired questions as to which method produces least/most errors, and requires least/most debugging time for each of methods a), b), c), d), e), were as follows.

SS	12345678	a	b	c	d	e	order
LE	becbeea	LE	1	2	1	-	4 e b a/c
ME	daddbddd	ME	1	1	-	6 -	d a/b
LDT	becabeaa	LDT	2	2	1	-	3 e a/b c
MDT	daddddadd	MDT	2	-	-	6 -	d a

where e = top-down modularisation.

Thus most ($\frac{6}{8} = 75\%$) of the students think that d), direct terminal composition, produces the most errors, and takes the most debugging time-wise. Whereas the other methods e) b) a) and c) respectively produce less errors and require least debugging time (and in that order).

Q7A.3 Attitudes Towards The Compiler - Q5 & Q7

Q5. The top 7 opinions on the purpose of the compiler, as to why is it useful and what it does, and the type of errors it detects, were as follows :-

Responses	Students						
syntax	1	2	3	4	7	8	
some semantic errors					1	4	7
translating program into machine code					3	5	7
typos						6	7
misspelt commands/keywords						1	2
missing brackets						2	5
spelling mistakes						3	5

Students regard the compiler as most useful for spotting syntax errors (6); detecting some semantic errors (3) and for translating the program into machine code (3). Typos, misspelt commands/keywords, missing brackets, and spelling mistakes all fall into third place with 2 votes apiece. Although it could be argued that misspelt commands/keywords and spelling mistakes go into second place, with 4 votes in total for misspelling variants.

Q7. Given a choice of 2 attitudes a) and b), towards the compiler, the percentage responses were:

62% for a) "The compiler only detects obvious errors, it can't deduce the presence of subtle (or not so subtle) semantic errors resulting from logic, algorithmic or sequencing errors"; and

75% for b) "The purpose of the compiler is to detect obvious errors : typos, misspellings, mixed syntax, missing or misplaced syntactic components.".

So there is a slight preference (75% to 62%) for b) over a).

Q7A.4 Development & Debugging Attitudes - Q6, Q8, Q10, Q13 & Q29

Q6. Students' responses to percentage ratings for confidence of syntactic correctness & correctness task—wise, when they first attempt to compile their code; were as follows :-

for syntactic correctness 60 80 75 80 C language 70 40 20 60

mean=485/8=60.63

for correctness task-wise 80 90 50 60 C language 85 70 75 90

mean=600/8=75.0

Using the ratio p/q , where p = syntactic correctness, and q = correctness task-wise. There are 6 scores where $p < q$ giving a mean ratio of 55 / 82, there are 2 scores where $p > q$ giving a mean ratio of 77.5 / 55, and the overall mean is 60.63 / 75.00.

I was surprised that any students rated syntactic correctness above correctness task-wise. Or rather that it was more difficult to express the intended algorithm in terms of the language than it was to achieve syntactic correctness. Or perhaps the results reflect the difference between the algorithm expressed in code and the way it runs (being syntactically correct), rather than the intended algorithm that would perform the required task (and thus be correct task-wise).

Q8. Students' frequency responses (either usually, sometimes or never) to the question, "Do you look for errors before attempting to compile your code? How thoroughly?" were :

- a) check each section/chunk of code before you develop and/or extend it
- b) check each section/chunk of code after you develop and/or extend it
- c) expect (or are confident) that what you have done is correct
- d) expect the compiler to detect any trivial errors that have been made?

Q8. USN abcd

SS12345678		U	S	N	sum
a uuunsuss	a	4	3	1	13
b usunnsus	b	3	3	2	15
c susssuuu	c	4	4	-	12
d susuuusu	d	5	3	-	11

Taking u=1, s=2, n=3, to give the above sum values.

Order of ascending sum values is d, c, a, b.

Most (7) students check their code before, and (6) after development.

It is split approximately 50/50 between usually and sometimes. ["usually" to "sometimes" ratio is 4 to 3 for a), and 3 to 3 for b).]

SS4 doesn't check the code either before or after development. He just modifies as he goes.

SS5 checks code before but not after.

Q10. Ranking of post-syntactic debugging strategies by frequency of use (with choices of usually, sometimes or never).

- a) checking that code is correct "task-wise" after eliminating the syntactic errors;
- b) only eliminating design, logic and algorithmic errors as you become aware of them, during testing or as a result of run-time errors.

SS12345678		U	S	N	sum
a snsusuus	a	3	4	1	11
b ussuuuuu	b	6	2	-	10

There is a (6 to 3 "usually") preference for b) rather than a). Indicating a tendency towards a reactive approach rather than a proactive one. Also it is (usually) easier to set up test data, and to see what errors become apparent. Instead of trying to eliminate ALL errors by relying wholly on mental visualisation or simulation of how the code will execute.

Q13. Defining students' primary debugging strategy.

- a) to eliminate bugs as you become aware of them
- b) to prevent as many errors as possible from occurring in the first place

SS12345678	
bbbbaacb	2a 5b 1c

Again the students regard error prevention, b), as being better than cure, with 5 votes to 2 respectively. Having experience as programmers, they have obviously learnt that error elimination is much, much more time and effort consuming than error prevention.

Combining the results of Q10 & Q13, shows that the students' priority regarding errors is : error prevention, error elimination (in response to error presentation or detection), and checking for task correctness after syntactic elimination.

Q29. Which do you think is more difficult to correct? [a] or [b]
a) an inadvertent "typo" that alters the intended execution of the code
b) an algorithmic error (the chosen algorithm doesn't fit the task requirements).
Give reasons for your answer.

SS12345678

babaa*aa, where * = "could be either", Totals are 5a 2b 1a/b

As expected most students (72%, 5 out of 7) agree that an inadvertent typo is more difficult to correct than an intended algorithm that doesn't meet the task requirements. Student 8 gives the answer "a) ... as b) can be compared with [the] spec, whereas with a), you know what you INTENDED to type and tend to read it as it SHOULD be ... reading over the mistake."

Errors of type a) can take weeks to find, due to the reading-over effect. Getting someone else to look at the code for you, usually gets over the problem, and the typo is found and fixed pronto. Sometimes describing what the code does to a third party can give a fresh perspective, and thus lead to the error being found.

Q7A.5 Use of Debugging Techniques & Tools - Q9, Q11, Q12 & Q22

9. Defines the inspection techniques used to test for errors before (and after) attempting to compile the code. The table shows the distribution of scores for each individual method

	during development	after development	after compilation
q=quick read through	4	5	2
s=slow read through	5	4	6
m=mental simulation of code	5	2	6
h=hand simulation of code	1	3	6

Actual responses were:-							reading		simulation			
							qUs	qrs	mUh	mrh		
During Development	-	q	s	qm	sm	2qsm	smh	7	2	5	1	
After Development	-	2q	s	qs		qsh	qmh	smh	7	2	5	2
After Compilation	-	q		sm	mh	qsh	4smh		7	1	7	5

Table checking for numerically significant component task pairings.

	q	s	m	h	qs	qm	qh	sm	sh	mh	qsm	qsh	qmh	smh
During Development	4	5	5	1	2	3	-	4	1	1	2	-	-	1
After Development	5	4	2	3	2	1	2	1	2	2	-	1	1	1
After Compilation	2	6	6	6	1	-	1	5	5	5	-	1	-	4

During development q, s and m are the main single component strategies, scoring 4, 5, and 5 respectively. With 4 pairings of sm, 3 of qm and 2 of qs; with 2 triples of qsm. There are 7 students reading through, with 5 of them using mental simulation as well.

After development the main strategies are reading through rather than mental or hand simulation - given by 5q, 4s, 2m, and 3h. There are 7 students reading through, with only 3 of them using mental or hand simulation.

The predominant debugging strategy after the compiler defines errors is 4smh - a slow read through with mental and/or hand simulation. There are 7 students reading through, with 5 doing both mental and hand simulation as well, while the other 2 choose to do either mental or hand simulation.

This indicates a higher degree of problem solving during actual development and in response to compiler defined errors.

11i. Frequency and preference ratings for use of common debugging methods. Where frequency range is Usually, Often, Sometimes, Rarely, Never; and preference range is 1-6, where '1' indicates the individual's favourite debugging method, and '6' the least preferred.

The table below shows the mean values derived from the students' rating scores. For frequency of use these mean values correspond to the Usually -- Never ranges, as shown. Thus c) inserting write statements' mean score is 1.62, corresponding to somewhere in the range often-usually (o-u).

	mean frequency of use	mean order of preference
a) mental simulation of code	3.00 s	3.25
b) hand simulation of code	2.25 s-o	2.87
c) inserting write statements	1.62 o-u	1.75
d) tracing variable values by hand/eye	2.50 s-o	3.62
e) tracing variable values by search mechanisms	3.50 r-s	4.40
f) tracing variable values by debugger	3.00 s	4.00

Order of frequency of use is c,b,d,a/f,e.

Order of preference is c,b,a,d,f,e.

Looking at the mean values, c), inserting write statements, wins on both counts; with b), hand simulation, lagging 0.4 behind frequency-wise, and 1.1 behind preference-wise.

One reason for this preference for c) is that adding write statements is easy and straightforward, and is under direct user control - so its actions are known and predictable.

Whereas d), tracing variable values by hand/eye, and a), mental simulation of code, seem to be interchangeable. Since d) is 0.5 smaller than a) frequency-wise, and d) is 0.3 larger than a) preference-wise. Thus d) is used more frequently than a), but is preferred less.

In both cases alternative methods of tracing variable values, by debugger and search mechanisms, f) and e) respectively, come in last.

ii. Defining the stage(s) where each of the above debugging methods are used and why.

Before Testing	Students	After Testing	Students
a)	1 2½ 8	a)	1 2½ 3 4½ 5 6 7½ 8
b)	2 4	b)	1 2 3 5 7 8
c)	1 4 5	c)	1 2 3 4 5 6 7 8
d)		d)	1 3½ 4½ 6 7½ 8
e)		e)	1 3½ 7 8
f)		f)	1 3½ 5½ 6½ 7 8

where ½ indicates 2nd choice or a method only used when necessary.

The order for "after testing" is the same as the order for preference (see 11i. above). Again c) inserting write statements comes out on top. a) and c) are often used in conjunction with other methods, as shown in the table below.

Comment Responses	Students
use debugger as last resort	5 6
use a) b) c) during initial debugging	3
d) e) f) are used if a) b) c) do not find errors	3
use a) & b) to locate the error	5
a) & c)	6
a) & d)	6
visualisation instead of b)	6
c) used to narrow down problem	5

Q12. Finding out which (programming environment) debugging tools (if any) are used most frequently and why or why not.

Responses	Students
never uses debugging tools	2 4
rarely uses debugging tools	3 5
uses own debugging methods	2 3 5
uses write statements	3 5
uses debugger as a last resort	5 6
TurboPascal's breakpoints and variable tracing	7

Most comments indicate that students are either not really aware of the debugging tools provided or they don't have enough information to use them. Hence they do not feel either confident or comfortable in using them. Also there is the problem of variation in debugging tools on different systems. So the effort spent learning one tool is wasted when the student has to change to a new system.

Q22. Describe the methods/tools you use to locate each type of (non-syntactic) error and the features of each method ie. how it helps you locate such errors.

Responses	Students
write statements	2 4 5 6 7
hand simulation	1 4 8
mental simulation	4 8
compare intended code to actual code	1 8
trial & error modifications	6
breakpoints	7
checking value/state of variables	7

Inserting write statements wins with 5 votes; hand simulation is next with 3 votes; while mental simulation, and comparing the intended code to the actual code, tie with 2 votes each.

Summary of Debugging Methods & Tools

It seems that using write statements is the most frequently used and best liked debugging method. No other tool(s) seem able to match it for flexibility.

Q7A.6 Defining The Nature of Errors, Their Frequency & Troublesomeness - Q14-15

Q14. The most common semantic, logic and/or algorithmic errors that students check for are :-

Responses	Students
faulty procedure/function calls	1 2 3 4 5 7
variable faults	1 3 5 7 8
faulty conditional statements	5 6 8
language errors	1 5
algorithm errors	1 7
typos	6 8
missing semicolons	6 8

Overall votes for each type of error are : faulty procedure/function calls (6), variable faults (5),

faulty conditional statements (3), and 2 votes each for : language errors, algorithm errors, typos, and missing semicolons.

The results of this question reinforce the need for some way of reducing the number of procedure/function call faults. The summary tool should answer this need.

Spotlighting should be effective in resolving some of the errors associated with variables.

15i. The following table lists (23) types of errors; the columns numbered 1 to 5 correspond to the time of detection of each error. The numbers within the columns represent the total number of students who voted for that stage of detection. (NB. Parameter list errors are the same for procedures and functions. For procedures read procedures or functions.

Labelling Key for Stages of Detection and Error Ordering:

- 1 = pre-compile (errors detected by you),
- 2 = during compilation (errors detected by the compiler),
- 3 = post compile, before testing (errors detected by you),
- 4 = during execution - run-time, design and algorithm errors, etc.
- 5 = post execution - errors deduced from output values or events, or lack of them

Types of Errors	1	2	3	4	5
Undeclared variables, types or constants	1	8			
Misspelt names (eg. variables, types, constants, reserved words)	2	6	1		
Redundant declarations of variables, types or constants	3	2	2	1	
Inappropriate data typing of variables (eg. using real instead of integer)	1	4	3		
Using round brackets, (), instead of square brackets, []		7	1		
Missing or extra bracket in an expression		8	1	1	
Incorrect placing of brackets in an expression		3	1	1	3
Inappropriate declaration of a procedural parameter list (eg. using wrong variables or data types)		5		1	2
Inappropriate declaration of a procedural parameter list (eg. under- or over-use of "var" parameters)		4		1	3
Incorrect format of procedural parameter list call (too many or too few parameters)	1	6	2		
Incorrect content of procedural parameter list call (mis-ordering/transposition of variables in parameter lists)	1	3	2	3	
Incorrect initialisation or termination of variable values		3	2	4	
Incorrect modification of variable values				3	5
Incorrect choice of loop variable value ranges (eg. in "for" statement)	1		6	3	
Incorrect choice of selection or loop construct to give required effect (eg. using "if" rather than "while")				4	4
Inappropriate choice of loop variable		2	5	3	
Infinite loop(s)				8	2
Redundant loop(s)	2	1	1	1	2
Inappropriate placing of "end" statements		5	2	1	
Missing "else" statement(s) to complement an "if" statement		3	2	4	
Incorrect sequencing of variable value assignments (eg. modification, initialisation or termination of variable values)	1	3	2	4	
Incorrect sequencing of control structures	1	3	3	4	
Incorrect sequencing of procedure or function calls	1		4	4	
Run-time errors (divide by zero, under- or over-flow of values)				7	2

15ii. The next task was to give opinions as to which 12 errors were encountered most frequently : 1 indicates the most frequent error, 2 the 2nd most frequent error etc.

[If 2 errors were equally frequent, the same number was to be used for each, and the next number was skipped. For example, if 2 errors shared the 5th most frequent error value, then the next most frequent error number would be the 7th.]

15iii. Then the same was done for the ordering of the 12 most troublesome/time-consuming errors.

Orderings for error frequency and the most troublesome/time-consuming errors follow.

EF = Order of most frequently occurring bugs
TC = Order of most troublesome/time-consuming bugs

EF TC Error Frequency Ordering

- 1 13 Missing or extra bracket in an expression
- 2 20 Undeclared variables, types or constants
- 3 23 Misspelt names (eg. variables, types, constants, reserved words)
- 4 4 Incorrect placing of brackets in an expression
- 5 22 Using round brackets, (), instead of square brackets, []
- 6 24 Inappropriate data typing of variables (eg. using real instead of integer)
- 7 20 Redundant declarations of variables, types or constants
- 7 7 Inappropriate placing of "end" statements
- 9 8 Incorrect modification of variable values
- 10 13 Incorrect initialisation or termination of variable values
- 11 16 Inappropriate choice of loop variable
- 12 18 Missing "else" statement(s) to complement an "if" statement
- 13 12 Incorrect choice of loop variable value ranges (eg. in "for" statement)
- 14 5 Inappropriate declaration of a procedural parameter list
(eg. using wrong variables or data types)
- 14 1 Infinite loop(s)
- 16 6 Run-time errors (divide by zero, under- or over-flow of values)
- 17 10 Incorrect sequencing of control structures
- 18 19 Redundant loop(s)
- 19 17 Incorrect format of procedural parameter list call
(too many or too few parameters)
- 20 13 Incorrect choice of selection or loop construct to give required effect
(eg. using "if" rather than "while")
- 21 3 Inappropriate declaration of a procedural parameter list
(eg. under- or over-use of "var" parameters)
- 22 9 Incorrect sequencing of variable value assignments
(eg. modification, initialisation or termination of variable values)
- 23 11 Incorrect sequencing of procedure or function calls
- 24 2 Incorrect content of procedural parameter list call
(mis-ordering/transposition of variables in parameter lists)

EF = Order of most frequently occurring bugs

TC = Order of most troublesome/time-consuming bugs

EF TC Troublesome/Time Consuming Ordering

- 14 1 Infinite loop(s)
- 24 2 Incorrect content of procedural parameter list call
(mis-ordering/transposition of variables in parameter lists)
- 21 3 Inappropriate declaration of a procedural parameter list
(eg. under- or over-use of "var" parameters)
- 4 4 Incorrect placing of brackets in an expression
- 14 5 Inappropriate declaration of a procedural parameter list
(eg. using wrong variables or data types)
- 10 6 Run-time errors (divide by zero, under- or over-flow of values)
- 7 7 Inappropriate placing of "end" statements
- 9 8 Incorrect modification of variable values
- 22 9 Incorrect sequencing of variable value assignments
(eg. modification, initialisation or termination of variable values)
- 17 10 Incorrect sequencing of control structures
- 23 11 Incorrect sequencing of procedure or function calls
- 13 12 Incorrect choice of loop variable value ranges (eg. in "for" statement)
- 1 13 Missing or extra bracket in an expression
- 10 13 Incorrect initialisation or termination of variable values
- 20 13 Incorrect choice of selection or loop construct to give required effect
(eg. using "if" rather than "while")
- 11 16 Inappropriate choice of loop variable
- 19 17 Incorrect format of procedural parameter list call
(too many or too few parameters)
- 12 18 Missing "else" statement(s) to complement an "if" statement
- 18 19 Redundant loop(s)
- 2 20 Undeclared variables, types or constants
- 7 20 Redundant declarations of variables, types or constants
- 5 22 Using round brackets, (), instead of square brackets, []
- 3 23 Misspelt names (eg. variables, types, constants, reserved words)
- 6 24 Inappropriate data typing of variables (eg. using real instead of integer)

Q7A.7 Investigating Reading Strategies - Q16-19

Q16i. Defining the stage(s) and reasons for reading (or re-reading) the code/task specification. These questions relate to research on comprehension and reading by Pennington (1987), Nanja & Cook (1987), Gugerty & Olson (1986), and Holt, Boehm-Davis & Shultz (1987).

Responses	Students
re-reading to understand/reaffirm task requirements	1 4 5 6 7 8
constantly re-reading to know what was going on	1
fast read followed by re-read to make sure nothing was missed	2

In my opinion, the purpose of reading the task description is to build up a mental model of the task that is to be performed - a task model.

ii. Responses to "At what stage(s) if any did you compare the code/task specification with your mental image of the task, (ie. your own personal view of how the task should be performed). Why?" cannot be summarized. So, in this case the students speak for themselves :-

SS1 ——— Hardly, apart from with Task (1) as this was the only task I had encountered before. With this I tried to remember how I had carried out the task previously.

SS4 ——— Near the end of each debugging session, to see which one would perform the task better

SS5 ——— After I read the program through once.

It's difficult to spot other peoples' errors

SS6 ——— This only occurred in primes. I compared the algorithm printed in the program with my view, derived from the specification. This was to pin down any mismatches (I was confident of my version of the algorithm)

SS7 ——— I did not, as it would be confusing

[implies that his mental image was at odds with that projected by the code]

SS8 ——— I had to occasionally alter my mental model on re-reading the task specification ... but it was usually only re-read due to reasons in 16i.

{When I couldn't quite follow what was happening in the code, I often had to refer to the task description which reminded me and usually helped me to see what the code was doing}

Q17i. Defining the usual code reading strategy, prior to debugging - with choices of usually, sometimes or never. And asking what effect this has on debugging - whether it gives faster or more accurate debugging results. Reading choices were :

a) reading code on an as-needed basis

b) reading to get a total understanding of how the code works

SS12345678 U S N

a snuusuu a 4 3 1

b uuususnu b 5 2 1

Responses	Students
debugging is faster if you apply strategy b)	1 2 8
debugging is more accurate if you apply strategy b)	1 2 5
strategy a) helps fill in details	3 6 7
a quick read through gives a general overview	3

There is a slight bias (5 to 4) towards b) rather than a); with 3 students considering that strategy b) makes debugging faster (SS1, 2 & 8) and more accurate (SS1, 2 & 5); and a further 3 students (SS3, 6 & 7) thinking that a) helps fill in details.

ii. Do you think it is necessary to read through all the code, to get an overall view of the code structure and all its active elements, before you start debugging? Why/why not?

SS12345678

nyynnnnyy 4y, 4n.

This split of opinion (4 yes to 4 no) is very significant. It relates to Pennington's cross-referencing of application and domain models, and the divergence in approach taken by her problem solvers.

Q18. Attempts to define the primary goal of code reading on the first and second readings of the code, and to see if there are any significant differences for choices a) and b).

- i. What were you doing when you read the code through for the first time?
- ii. What were you doing when you read the code through for the second time?

a) attempting to make sense of the code on its own

[U] [S] [N]

b) attempting to correlate the code (and its structure) with the task description

[U] [S]

[N]

Putting the 1st and 2nd read through tables alongside each other enables easy comparison of the results. Taking $u=1$, $s=2$, $n=3$ to give sum values gives :-

18i. USN

Q18ii. USN

First read through

Second read through

First Read					Second Read						
SS12345678	U	S	N	sum	SS12345678	U	S	N	sum		
a nsusuuss	a	3	4	1	14	a ssuusu--	a	3	3	2	15
b uuuuus	b	6	2	-	10	b uusuusu-	b	5	2	1	12

On the first read through, it seems that the primary task (for 6 out of 8 students) is to correlate the code and its structure with the task description. Whereas making sense of the code on its own is a secondary task (for 3 out of 7 students). This is still true for the second read through (5 out of 7, and 3 out of 6, respectively), but the numbers have been reduced by 1 student, SS8, who starts debugging immediately after the first read through.

This question also refers to Pennington and the cross-referencing of program and application/task models.

iii. Gauging the importance of using method (a) on the first read-through, and method (b) for subsequent readings. And whether they thought that methods (a) and (b) formed a natural comprehension strategy, responses were :-

SS12345678

y-yynyyn- 4y 3n

A mixed response with 4 in favour and 3 against. A much larger proportion was expected to vote Yes, since it seemed the logical progression.

Responses	Students
a) then b)	1 3 5 6
b) then a) if task description is hard to follow	6
b) on both readings	2
a) and b) both together	8

iv. As to which is more important of a) and b), and why; responses were -

SS12345678

ababb-ba 3a 4b

There is a fairly even split (3 votes to 4) between understanding the code on its own and correlating the code with the task description. Again this reinforces Pennington's summing up of the situation.

Further comments were not forthcoming.

Q19. The response to "Did you find yourself trying to debug the code on the first read-through?", was 5 usually, 2 sometimes and 1 never (as shown in the table). Reasons below -

SS12345678
snuuuuusu 5u 2s 1n

SS2 wanted to make sure of understanding the problem/code before attempting to debug it. Students 1, 5 & 7 were also trying to make sense of the code rather than debugging it. But all the rest were raring to go and saw no point in "wasting time".

Summary Q16-Q19

Those students who started debugging on the first read through are likely to give fast debug times, because I assumed that the first read through is just for reading and that debugging comes later. If not, then some bugs may already have been solved during the first reading, and it is just a question of writing the error solution out. Rather than finding and solving each error in the subsequent debugging phase which I assumed to be separate from the initial code reading phase. From experience I know that some errors just spring out at you on the first read through - usually the glaringly obvious ones - and they are difficult to ignore. It's usually easiest (less drain on remembering to do it later) to fix them there and then.

This might account for the fast debugging times for this sort of strategy in the debugging & spotlighting experiments.

Q7A.8 Information Needed to Detect & Locate Errors - Q20-21

Q20. What specific information/evidence do you need to be able to detect* an error and why? (ie. What is it that does or doesn't happen that makes you aware that an error exists?)

Responses	Students
when output events/values differ from those expected	4 5 6 7 8
when program does not behave as expected	2 4 6 8
relying on compiler to detect and report errors	2 4 7
relying on run-time error messages	5 6
when the program goes into an infinite loop	3 5
knowledge of the syntax of the language	3
knowing how the semantics tie in with the task	3
need to know for what input values the program should terminate	5
need a set of sample inputs and manually calculated results to detect wrong output values	5
unexpected output & anything that happens unexpectedly	6
when program crashes	7

The top 5 responses give the usual signs that the code has a bug. The remaining responses give a mixture of finer and coarser interpretations of what means are needed to detect a bug. The former being "knowing how the semantics tie in with the task", and the latter "when program crashes"! Perhaps he assumes there are no bugs if the program doesn't crash. It seems a rather extreme means of detecting errors.

Responses appearing for both Q20 & Q21.

Responses	Students
basic knowledge of the language	1
need to know all errors that are detected by the compiler	1
need to be aware of all post-execution errors	1

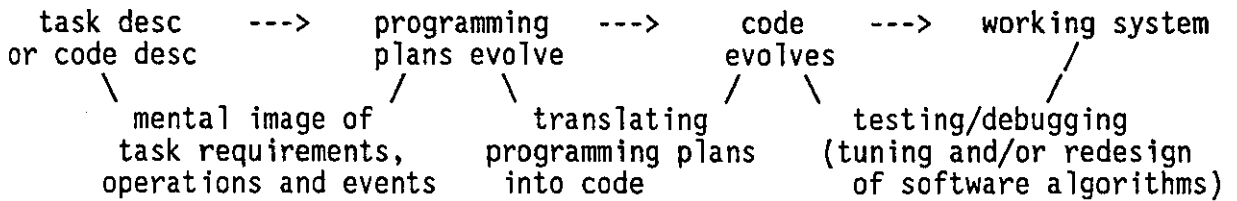
Q21. What specific information do you need to be able to locate* an error (to associate it with a particular section of code, or specific variable), and why?

Responses	Students
knowing which sections/procedure relate to each output	2 5 8
need a basic knowledge of the code	5
inserting extra write statements to find the procedure/block holding the error	2
knowing values of variables, to identify which ones are causing the problem	6
knowing the state and the variable values just before the error occurs helps locate its origin	4
if an output value differs from its expected value, then you need to know where the variable was initialised and which procedures change its value	5
using knowledge of incorrect output to "backtrack" to faulty section of code	8
detail of where and why errors have occurred	1
need to know which part of the algorithm has gone wrong	7
program stopping (crashing or exiting) at a certain point	8
clear error messages from the compiler	3

The responses have been ordered roughly according to importance from a programmer's viewpoint. Accurate "working" knowledge of the code (how it interacts) is vital, without it debugging capability is decimated. Knowing the value of principal variables is just as important.

Q7A.9 Programming/Software Development Process Diagram - Q23

Q23. Do you agree with the following programming/software development process diagram? ie. do you think it reflects the main aspects of the software development task?



Responses	Students
Yes	1 2 3 4 8
mostly agree	5 6 7

On the whole a favourable response to my diagram showing the process of transforming a task or code description into a fully working system under software control. But there are a few provisos:

Comments :

SS2 — Yes, this is an accurate description of the planning involved, right up to the final working system.

SS3 — Yes, if mental images are transformed into physical specifications at the "programming plans evolve" stage.

SS5 — Mostly agree, but programming plans evolve and code evolves often occurs in parallel [confirmed by Pennington & Grabowski 1990]

SS6 — I basically agree, though this diagram is incomplete (no feedback loops)

SS7 — I agree to a certain extent. I think it is also important however, to determine how the program is going to be debugged before starting any coding, as it gives you an idea what to look out for

SS8 — Yes, I think this fits in very well with the way most people develop a software program

Q7A.10 Investigating Trail Following on Paper & Screen Text - Q24 & Q25

Q24i. Ratings as to whether trail following (ie. following a variable's trail through the code) is an important debugging technique. For a 1-5 rating scale, with 1 as important, and 5 not important.

important	1	2	3	4	5	not important
SS1	2	3	4	5	6	7
SS2	1	2	3	4	5	6
SS3	1	2	3	4	5	6
SS4	1	2	3	4	5	6
SS5	1	2	3	4	5	6
SS6	1	2	3	4	5	6
SS7	1	2	3	4	5	6
SS8	1	2	3	4	5	6
SS9	1	2	3	4	5	6
SS10	1	2	3	4	5	6
SS11	1	2	3	4	5	6
SS12	1	2	3	4	5	6
SS13	1	2	3	4	5	6
SS14	1	2	3	4	5	6
SS15	1	2	3	4	5	6
SS16	1	2	3	4	5	6
SS17	1	2	3	4	5	6
SS18	1	2	3	4	5	6
SS19	1	2	3	4	5	6
SS20	1	2	3	4	5	6
SS21	1	2	3	4	5	6
SS22	1	2	3	4	5	6
SS23	1	2	3	4	5	6
SS24	1	2	3	4	5	6
SS25	1	2	3	4	5	6
SS26	1	2	3	4	5	6
SS27	1	2	3	4	5	6
SS28	1	2	3	4	5	6
SS29	1	2	3	4	5	6
SS30	1	2	3	4	5	6
SS31	1	2	3	4	5	6
SS32	1	2	3	4	5	6
SS33	1	2	3	4	5	6
SS34	1	2	3	4	5	6
SS35	1	2	3	4	5	6
SS36	1	2	3	4	5	6
SS37	1	2	3	4	5	6
SS38	1	2	3	4	5	6
SS39	1	2	3	4	5	6
SS40	1	2	3	4	5	6
SS41	1	2	3	4	5	6
SS42	1	2	3	4	5	6
SS43	1	2	3	4	5	6
SS44	1	2	3	4	5	6
SS45	1	2	3	4	5	6
SS46	1	2	3	4	5	6
SS47	1	2	3	4	5	6
SS48	1	2	3	4	5	6
SS49	1	2	3	4	5	6
SS50	1	2	3	4	5	6
SS51	1	2	3	4	5	6
SS52	1	2	3	4	5	6
SS53	1	2	3	4	5	6
SS54	1	2	3	4	5	6
SS55	1	2	3	4	5	6
SS56	1	2	3	4	5	6
SS57	1	2	3	4	5	6
SS58	1	2	3	4	5	6
SS59	1	2	3	4	5	6
SS60	1	2	3	4	5	6
SS61	1	2	3	4	5	6
SS62	1	2	3	4	5	6
SS63	1	2	3	4	5	6
SS64	1	2	3	4	5	6
SS65	1	2	3	4	5	6
SS66	1	2	3	4	5	6
SS67	1	2	3	4	5	6
SS68	1	2	3	4	5	6
SS69	1	2	3	4	5	6
SS70	1	2	3	4	5	6
SS71	1	2	3	4	5	6
SS72	1	2	3	4	5	6
SS73	1	2	3	4	5	6
SS74	1	2	3	4	5	6
SS75	1	2	3	4	5	6
SS76	1	2	3	4	5	6
SS77	1	2	3	4	5	6
SS78	1	2	3	4	5	6
SS79	1	2	3	4	5	6
SS80	1	2	3	4	5	6
SS81	1	2	3	4	5	6
SS82	1	2	3	4	5	6
SS83	1	2	3	4	5	6
SS84	1	2	3	4	5	6
SS85	1	2	3	4	5	6
SS86	1	2	3	4	5	6
SS87	1	2	3	4	5	6
SS88	1	2	3	4	5	6
SS89	1	2	3	4	5	6
SS90	1	2	3	4	5	6
SS91	1	2	3	4	5	6
SS92	1	2	3	4	5	6
SS93	1	2	3	4	5	6
SS94	1	2	3	4	5	6
SS95	1	2	3	4	5	6
SS96	1	2	3	4	5	6
SS97	1	2	3	4	5	6
SS98	1	2	3	4	5	6
SS99	1	2	3	4	5	6
SS100	1	2	3	4	5	6

So trail following is seen as an important debugging technique. With 5 votes for trail following on paper. With one student also voting for trail following on a VDU providing the code is short.

With comments defining in what circumstances trail following is used on :

- VDU text (using an editor and the search mechanisms), and
- printed listings (using eyes/hands/pens).

Responses	Students
prefer trail following on paper	1 2 3 5 7
prefer trail following on VDU if code is short	1

4 students definitely prefer trail following on paper because they are not restricted by the screen size, where they only have access to 24 lines of code; and they feel they can spend more time checking on paper. The latter may be due to the restriction of how much time a student can spend on a VDU, due to the many students to one VDU ratio.

Q24ii. Advantages & disadvantages of working on a) screen text b) paper text; and what effects these differences have.

Screen

Advantages	Students
can alter code immediately	1 2 8
can use editor's search mechanisms	4 8
applies to small pieces of code	3 5
can test code immediately	2
easier to trace variable values	4
(vi) ability to jump to matching brackets in an expression	5
focusses attention on a few statements	5
edited code remains readable	6

Disadvantages	Students
cannot study code thoroughly	1
cannot get an overview	5
cannot scribble thoughts in margins	6
screen size sets up distance effects (body of loop can be distant from conditions)	5
sore eyes, back pain	2
reading screen text is stressful	8

Paper

Advantages	Students
easier to see whole code	2 5 7
easier browsing	4 5 8
applies to larger codes	3 5
easier to follow route (mentally) with pen & paper	4 7
use debugger to mark the page before printing	3
more time can be spent on checking	1
can scribble thoughts in margins	6
can shuffle pages around and view more than one at a time	8
no waits while editor looks for "next" page	8
can see several screenfuls at once	5
reading paper text less stressful	8

Disadvantages	Students
none	1
code can become illegible after a long session	6
human eye searching not efficient	8

With screen text (and having an editor on hand) the main advantages seem to be speed related. How fast something can be fixed or found or tested; and taking advantage of the small screen size (and search mechanisms) to narrow down on specific sections of the code. Principal disadvantages also relate to the small screen size and the "window" effect of an editor - cutting off access to anything beyond the screen's horizon, and the physical stresses associated with VDU work.

In contrast, working on paper offers a relaxed, less hurried, more time to check things out atmosphere; and speed and on the spot "now" decisions seem of less importance. As stated previously, a high student to system ratio would reinforce the "hurry up, do it now" nature associated with screen text. Paper text has a more relaxed attitude. It can be taken away and studied "at leisure" away from the hustle and bustle of a terminal room. Paper text's main advantage is its flexibility and the fact that you can "scribble thoughts in the margins" and revise decisions as often as required.

The disadvantage being of course that after a lot of scribbling, code can become illegible and searching for something by eye is not as efficient as machine search. However one student has found a way of getting around this - he uses the debugger to mark things before printing it out. This is obviously an alternative to spotlighting, and in my view expresses the need for the spotlighting tool.

Q25. Investigating the usual debugging strategy(ies) students apply

- i. to paper text, and
- ii. to screen text (during editing).

Paper Text

Responses	Students
examining specific erroneous sections	6 8
for hand simulation	2 4
using coloured pens to outline errors & write solutions	1
highlighting a variable trail	5
drawing vertical lines to match up indentation	5
follow passing of control from one procedure to next	2
checking that code fits task description	4
print out procedure name & relevant variable values	7
write error solutions on paper then type into computer	1

Screen Text

Responses	Students
inserting write statements	2 6
correcting misspellings or obvious errors	3 8
enables immediate error solution/correction	1
syntax errors	4
keeping a specific line on centre of the screen	5
moving cursor vertically up or down (to determine "reach" of a control structure)	5
display values of various variables	7
finding errors defined by compiler	8

The overall impression coming across is that students go to paper text to think things out and decide what to do; and they go to the screen/editor to write up alterations and test them out. Most mental or hand simulation is best suited to an unrestricted view of the code. So it is hardly surprising (from Q9) that it is associated with a slow read through of the code. Paper text provides the ideal conditions for these activities, of course.

Comparing the lists above, paper seems to provide the best conditions for thinking things out, and screen text (and editor) for fixing errors and testing them out, or getting accurate variable value printouts (via write statements or other means) as the code runs.

The responses to these 2 questions (Q24 & Q25) validate the reasons why programmers prefer to use paper text for debugging (whether for mental or hand simulation).

Q7A.11 Differentiating Between Debugging Methods - Q26-28

Q26. Which of the following methods do you use to detect/locate most errors, and how/why?

- a) discrepancies between your mental image of what the program should do, and what it actually does [U] [S] [N]
- b) discrepancies between the code/task specification and the program code itself (or what it actually does) [U] [S] [N]

Q26. USN

SS12345678 U S N

a sss-uuus a 3 4 -

b uusussu b 4 4 -

There is a fairly even split between the 2 strategies, but b) has a 4 to 3 voting advantage over a).

Student 5 makes the distinction that this question was aiming to draw out. Namely, that the mental image should encapsulate the code or task description. But with an unfamiliar task situation the programmer's understanding may be either incomplete or inadequate, so the mental image is flawed and the code will reflect this. Student 8's view though is flawed, he doesn't seem to realise that it is the mental image that drives/fuels the programming plans, and the way the code develops [as per the Adelson & Soloway 1985 Sketchy Model].

Reasons :

SS2 — Once you know what the program should do, and what it actually does it is easier to know what is wrong, and whereabouts in the code it is wrong.

SS5 — It really depends on how much experience I have with the problem area. If I have some experience then I use a) as it is faster, but if I have little experience of that particular method or similar sorts of algorithm then I have to resort to b) which is slower.

1/ because you have to examine each line of code to see what it does and then fit that with the specification, whereas if you have experience of similar algorithms you just look for a general pattern ie. with concord [concordance program] I looked for the chaining through of the linked list and particular assignments at the places I expected them.

SS7 — Your mental image of what the code should do will correspond to the specification and is therefore related to b)

SS8 — b) the spec is what the code should be doing (rather than your mental image, which could itself, have 'bugs') ... if a certain part of the spec is not being executed properly, can find the part of code which supposedly performs that part of the spec and compare it to what it SHOULD be doing

Responses	Students
difficult to keep a mental image of programs and what it should do	1
use b) to locate errors quickly	1
c) insertion of print statements	2
mainly use option b)	4
a) as it is faster if you have experience of the problem area [task]	5
b) if you have little experience of that particular method or similar sorts of algorithm [ie. task situation] but b) is slower	5
use a) mostly	6
b) type errors are usually big, and easy to identify	6

Lukey (1980) describes 2 types of debugging; he says that "Tentative debugging progresses by interpreting debugging clues and tracking them back to source. Whereas [descriptive] debugging progresses by detecting and resolving all significant discrepancies between the program code (or its description which tells what the program actually does) and the program specification (which tells what the program should do)."

- Q27. a) Do you think that Lukey is right?

b) Is tentative debugging better than descriptive debugging?

c) Is descriptive debugging better than tentative debugging?

d) Are tentative and descriptive debugging complementary?

Do you think that there are other types of debugging? If so what are they?
- [Y] [M] [N]

[Y] [M] [N]

[Y] [M] [N]

[Y] [M] [N]

SS12345678	Y	M	N
a mmymyyym	a	4	4
b ymyymyyn	b	5	2
c mmnnmnnn	c	-	3
d yyymymyy	d	6	2

Opinion is split (4 yes & 4 maybe) as to whether Lukey is right or not. However, they (5 to 0) definitely think that tentative debugging is better than descriptive debugging, and 6 agree that they are complementary debugging-wise.

Holistic approach - seeing how the code correlates to the task description and fixing any discrepancies either in design approach (specification -> program plans) stage or design translation (program plans -> code) stage (eg. missing declarations etc.). Top-down hierarchical search - main program, then sub-procedures in order of calling.

Isolated approach - following specific items through the code. Sequential text search - not following the structure, but the text sequence, usually top-down.

- Q28. Do you think that the description of the Holistic Approach, is the same as Lukey's [descriptive] debugging description?

What do you think the Isolated Approach corresponds to? Any other comments?
- [Y] [M] [N]

SS12345678
myn-yyyn 4y 1m 2n

Responses	Students
Holistic approach = descriptive debugging	1½ 2 5 6 7
Holistic approach ≠ descriptive debugging	1½ 3 8
Isolated approach = tentative debugging	2 5 6 7
Isolated approach = variable tracing	3 8

Summary Q26-28

Curiosity prompted these questions as to whether the students consciously realised that they use different debugging methods dependent on the nature of the error. Like whether the error can be definitely linked to something specific being wrong, either an algorithm or variable. Or whether it's something more vague, like an intuitive feeling that something is not quite right, but it cannot be pinned down easily. Such as a program crashing for an unknown reason.

The students have responded to the questions but I don't think they are fully conscious of the differences in the methods and their use of them. Due to the "intuitive" or "expertise" features of the debugging activity. Choosing which to use may not be a conscious decision - it is usually dictated by the circumstances and what "bug evidence" is available at the time.

Q7A.12 Attitudes Towards the Search Mechanisms - Q30-33

Q30. At what stage(s) do you use the editor's search mechanisms? Can they help with debugging? How?

Responses	Students
never used them	4 5
rarely use them	1 2 7
used for jumping to required position eg. start of procedure	2 8
correcting spelling mistakes	3
correcting typos	6
correcting simple/obvious errors	8

That 5 students use the search mechanisms either rarely or never is very surprising. Presumably they correct these types of (renaming) errors manually instead. It is difficult to believe that any programmer neglects the search mechanisms to this degree, let alone (4 or) 5 out of 8 students. However, as 1st year students 28-30 of them had never used the search mechanisms, so perhaps these results are a natural consequence of their initial attitudes.

Q31i. Are the search mechanisms adequate for the tasks you undertake whilst programming and debugging? Why or why not?

Yes 1 2 6 8

There were no comments made in response to this question.

ii. What modifications/improvements would you like to make to them?

No-one answered this question!

Q32. Do you think it is important to be able to see all locations involving a given word (eg. a specific variable) in screen text at one time? Why? [Y] [M] [N]

SS12345678

mmmnyn-n

1y 3m 3n

Responses	Students
to correct a variable that is wrong	1
to check procedure calls and variable passing	2

Not a promising response. I thought that they would have realised and understood the importance of this facility by now.

They (all or most of them) may have misinterpreted the question. I may have assumed that the concept of "seeing all locations at one time" equates to (or describes) a spotlighted word situation; whilst they seem to visualise the question as if it refers to the chosen word being within (non-spotlighted) bland homogeneous text, and therefore invisible.

Some (Students 2, 6 & 4?) may have misinterpreted the question as in a "grep" of all lines. Resulting in a group of lines bunched together with no surrounding background text, so each line appears out of context.

Q33. Do you think it is important that tools should provide both forward and backward search mechanisms? Why? [Y] [M] [N]

SS12345678

yyyyyy 7y 1n

Responses	Students
using both or flicking between forwards and backwards search	5 6
backward search	1 5 8
uses page-up and page-down keys	4

A definite (7 to 1) response supporting Robertson, Davis, Okabe & Fitz-Randolph's (1990) findings. Namely, the reasons for using forward and backward search, and alternating between them to aid in comprehension and debugging strategies. Especially when trying to get an idea of what the code does, and how.

Q7A.13 "Live" Editors & Layout Style - Q34 & Q35

Q34. Do you prefer to use a "live" editor such as the MacPascal editor, which catches simple errors (eg. missing or extra bracket errors) on the spot? [U] [S] [N]

SS12345678

unssnssu 2u 4s 2n

Responses	Students
saving time	4 7 8
gets in the way sometimes	3 6 7
eliminating errors at source	7 8
enables immediate elimination of missing or extra bracket errors	1
find live editor irritating	2
prevents layout flexibility	3
needs on/off flexibility ie. under user control	5

3 students think that a "live" editor like MacPascal's, saves time; and 3 think that it gets in the way sometimes (1 of these students voted for both). Eliminating errors at source gets 2 votes, plus 1 more for enabling immediate detection of bracketing errors.

Q35. How much does your preferred layout style differ from that used in the experiments?

Responses	Students
nil	8
slightly	1 3 4 5 6 7
moderately	2

Responses	Students
none	3 8
trivial	1 6 7
within tolerance level	2 4 5

Student 5 found the positioning of the blank lines in the experiment code confusing. Having the comments on separate lines rather than to the right of the code also threw him off balance. This is why I think a layout tool is so important - it could remove these obstructions to understanding other people's code, and make unfamiliar programs easier to grasp.

Q7A.14 Program Visualisation - Q36

Q36i. How do you visualise a program that you are developing?

Some students misinterpreted the question again! They obviously describe the code's visual appearance, rather than how they represent the code inside their own heads. What I was hoping for was a range of verbal outlines describing the shape or form the code has mentally. Especially when they are developing code themselves, or attempting to comprehend someone else's code.

Instead they give me a pat answer on well structured code and aspects relating to its readability (Students 1,2,3). Student 4's answer is closer - at least it describes one method of translating from the internal mental representation to the external paperised form. The remaining students' answers are ambiguous and may or may not refer to an internal representation. Judge for yourself :-

SS1 — Well structured, loads of comments, variables with easy to understand names.

SS2 — I try to produce a well structured, easy to read piece of code.

SS3 — As a rough whole and then detailed sections

SS4 — I visualise using data-flow diagrams and sometimes Structured English

SS5 — Depends sometimes on the global level ie, how it all fits together and at the individual procedure level. Has to flow well and look right [re. Molzberger (1983)]. If it gets to the stage where I'm having trouble following what I've written, I delete the whole section and try breaking the problem down in a different way.

SS6 — I don't really. I suppose the best answer is that I visualise lines of code in my head

SS7 — I visualise what the output should look like, how the program should behave and what potential interaction errors can occur

SS8 — I see it as 'chunks' of function ... usually these correspond to procedures/functions ... which may be linked conditionally with other chunks

ii. How do you perceive errors or bugs in your code? (eg. like a block restricting the flow of the program)?

Students 4, 5 & 8 elaborate slightly more in response to the thought of a bug in the code. They see bugs as interrupting the "flow" of the program. This description gives a sense of movement, progression and life. In a way, I suppose a program is "alive" while it is running, and dormant (or at least in a state of suspended animation) otherwise.

SS1 — Most errors tend to be silly mistakes, although when I am writing some code I know it's going to be incorrect so I tend to totally comment it so that it is not compiled. [wrapping comments around the faulty parts??]

SS2 — Things put in the way to prevent me from finishing the task!

SS4 — Like a block restricting the flow of the program

SS5 — (other than as a pain in the butt!)

I like to think of the different sections of the program as a system of interlocking gear wheels. A procedure or section with an error is like a gear wheel with missing teeth

SS6 — Again, this is something I don't consider

SS7 — I am seldom overjoyed about them

SS8 — No, more like a deviance from the desired flow of the program

Comment This question relates to Molzberger's (1983) article on aesthetics and programming, in which he investigated "trance programmers" and their approach and attitudes towards code (theirs and other people's). One aspect that struck me, was how his programmers were able to "decide" whether a program would work or not by looking at its visual characteristics - the way it was laid out, and the flow of the program. Even though the programming styles and layout characteristics varied from one programmer to another - there seemed to be a "meta-style" of program disposition. Programs that expressed this "meta-style" were usually found to be practically bug-free, and to be very efficient - no matter how long they were. This may of course have been due to the effects of the "altered state of consciousness" achieved by some of his "trance programmers", during extra-long (24+ hours) code development and programming sessions.

Q7A.15 Suggestions for New Editing/Debugging Aids - Q37-41

In the following questions, each aspect of the tool was rated on expected usefulness, on a 5-point scale. With '1' being the "useful" rating, '3' being the "average" usefulness rating, and '5' being the "not useful" rating.

Q37. Imagine that all of the following modifications were made to the find/search mechanisms. Rate each feature on its usefulness :

- a) Wrapping each instance of the search string in inverse video, to form a spotlight, that makes its location within the current screen totally obvious
- b) The ability to jump directly from one spotlight to another
- c) The ability to jump forwards or backwards between spotlights (eg. use command "+3s" to move forward 3 spotlights, "-5s" to move back 5 spotlights)
- d) A counter indicating how many instances of the current word there were altogether, and which "position" the current spotlight holds eg. "3/5" would indicate a total of 5 instances of the word altogether and that the current spotlight is the 3rd one
- e) The ability to spotlight 2 or more words at the same time

How would these modifications assist in the programming/debugging tasks? (ie. how would you use it, and to do what?)

After analysis, the order of usefulness was most useful - b,a,d,c,e - least useful, by ascending sum value.

Jumping directly between spotlights, b), gets the highest score, 2.00, corresponding to fairly useful. With a) wrapping each instance of the search string in inverse video, to form a spotlight, next with 2.25, falling within the range useful—fairly useful. This range is shared by d), the counter mechanism showing the "position" of the current spotlight in relation to the total number of spotlights for that variable. The other 2 aspects of spotlighting - c) the ability to jump forwards or backwards between spotlights, and e) the ability to spotlight 2 or more words at once, have mean values of 3.00 and 3.13 respectively, putting them in the average (or slightly less) usefulness category.

The responses also show that features a) and b) are received more warmly than the others.

Positive Responses	Students
able to jump forward and back between spotlights (one at a time)	5
useful when debugging somebody else's code that is not well understood	7
using a) & b) to locate the variable being tracked quickly	6
d) gives an idea of where you are in the program	6
d) the counter defines the total number of variable instances	1

Negative Responses	Students
undecided about usefulness of spotlighting	2
spotlighting 2 words can get confusing	4
c) is of dubious value	5
e) is pointless	6

Q38. Imagine a new editing tool that could put a complete line of screen text (80 characters) in inverse video. Creating an inverse video "bar" that could be moved up or down the screen, using the cursor controls, so that you could examine a particular line more closely. How useful do you think such a tool would be? Why? - Give reasons (ie. how would you use it, and to do what?)

```
useful 12345 not useful
SS12345678 score 1 2 3 4 5 sum mean
34352452 freq - 2 2 2 2 28 3.5
```

The scores are spread but there is not a lot of enthusiasm about the tool. The mean value is slightly less than average usefulness, so this reflects a non-committal response at best.

However Students 5 & 8 have the most interesting use for an inverse video bar.

SS5 — It would draw the eye to a particular line. Would be useful when trying to simulate how the code will run in my head, could step the inverse line through the code to keep track of where I've got to

SS8 — It would allow you to concentrate on ONE line ... this is sometimes difficult, with a lot of other code around .. and enables the eye/mind to focus-in on the line of interest

Q39. Imagine a new tool that could produce a series of (vertical) menus, listing all the variables used within the current piece of code under development (so that you could scroll down each menu gathering information) that could be invoked whilst editing or viewing the code from within the editor. Rate the following features : (NB. for "procedure" read "procedure or function")

After analysis, order by ascending sum values gives - most useful f,h,g,b,c,a/e/i,d least useful.

Thus usefulness ordering is as follows, where sum value precedes the identifier.

- 10 f) Ability to list all words that are not already declared or pre-defined, eg. typos, misspellings, undeclared variables etc.
- 13 h) Ability to list all the user-defined procedure names alphabetically, with a sub-menu to show the parameter list of each
- 14 g) Ability to list all the user-defined procedure names in declaration order, with a sub-menu to show the parameter list of each
- 15 b) Ability to list all the declarations (as seen in the code) for a specific procedure squashed into a menu
- 17 c) Ability to list ALL variables alphabetically, with sub-menus for each variable name defining parent procedure(s) and the variable's data type
- 19 a) Ability to list ALL variables in declaration order, giving each procedure/function its own sub-heading, followed by all its declarations as seen in the code, "squashed" into a (long vertical) menu
- 19 e) Ability to do all the above with types and constants
- 19 i) Ability to provide an alphabetic listing of all the pre-defined procedures, with sub-menus to show the parameter list for each
- 22 d) Ability to list ALL variables (alphabetically) according to data type, with sub-menus defining parent procedure(s), and the status of the variable : local; or value or variable procedural parameter

The summary tool concept got a mixed reception as the comments below indicate.

SS1 — Nice features and some may be useful and/or helpful but I'm not sure if it's necessary to implement them.

SS6 — a) Not sure what this means

b) Would help identify duplicate definitions, name clashes

c) d) e) Not a particularly useful way of organising it. d) perhaps less useful than c).

f) Very useful - these are common errors

g) h) i) a useful reference of how to use the procedures (especially libraries)

SS7 — Good idea, it is often useful to quickly relate variables to procedures

Q40. Imagine a formatting program, that could present you with all the different ways of laying out each Pascal structure, in the form of a series of option menus. So that you could choose a complete set of layouts, one for each construct, to mimic your own pattern of laying out code. Rate the following :

a) The ability to transform any piece of formatted or unformatted code into your own personal style of layout

b) The ability to transform your code, laid out according to your own preferences into another style eg. into the "in-house" style

c) The ability to get the screen editor to lay out your code, as you edit/modify it, into your own pattern, thus eliminating manual laying out

SS12345678 sum score 1 2 3 4 5

a 22221221 14 a 2 6 0 - -

b 21212221 13 b 3 5 - - -

c 22214321 17 c 2 4 1 1 -

Ascending sum value gives b,a,c in order of usefulness.

Although the results are close, I expected either c) or a) first, with b) trailing well behind. I expected the students, as programmers, to go for c) - a tool that would do all the layout thinking for them, and in their own style; rather than option b). As I thought that this would relieve the laying out burden considerably.

Comments :

SS1 — All useful, but I think most are present in applications nowadays?

[balderdash!] eg. MacWrite.

If you had to implement, I think b) would be the easiest and probably the most useful to implement.

SS2 — This would be useful if several people were working on the same program - the finished program could then be all in the same style instead of in several different styles.

SS4 — The third option would probably be the most useful.

The second sounds pretty useful as well, especially if the program in question has to move to a different number of people who have different layout styles

SS5 — a) would be very useful and [could] considerably speed up debugging

c) might be useful if it could be turned on and off since I sometimes vary my code layout ie. short if statements without an else I like to keep [it] all on one line eg.

if x=5 then write('X = ', x);

SS6 — All these are useful ideas, though the third [c)] could be unnerving (code jumping around etc. as it is being typed).

SS7 —

SS8 — The latter I think especially useful .. anything which gives the desired formatting (and hence readability) with as little effort as possible MUST be good

Q41. Do you have any ideas for new editing/debugging tools, or modifications to existing tools?

Students 4, 5 and 6 were the only ones to have any useful ideas :-

SS4 — To have a help screen for error messages, which would describe the error in much more detail. This would be especially useful for those errors which use complicated language and which nobody understands

SS5 — I like editors that have simple windowing ability ie. can split screen into two halves horizontally so that you can keep a procedure declaration on screen whilst examining all the places where it is called.

SS6 — A facility to attach a run-time module to a program, so that, in response to a 'hot key' pressing, a window would pop up, allowing inspection/adjustment of variables, 'program counter' (position in source) etc. Something like Unix debug, but much easier to use

Q7A.16 Students' Comments About The Questionnaire

SS3 — Too long, and many questions seemed to ask the same or very similar things, as you may be able to tell from my answers. Also, some things seemed too precise to be answered for a general editor/debugger.

SS4 — Too long

SS6 — It is far, far too long. It took well over twice the time you suggested it would. Also, I had trouble providing specific answers to many of the questions. I am unsure of my methods

I would have preferred to be able to type answers (into the word processor document, as I have such awful writing

SS7 — It was very difficult to debug the routines from a cold start. My knowledge of Pascal is a bit rusty and it was not clear what type of error to look for.

This took a lot longer than the estimated 2 hours

The Post-Spotlighting Questionnaire contains 14 new questions: 7 for spotlighting, and 7 for layout; plus a restatement of Q37 from the the previous questionnaire, to see whether the students had changed their minds about the usefulness of spotlighting after tackling the spotlighting experiments.

NB. Student 2 did not take part in the spotlighting experiments or this questionnaire.

Q7B.1 Spotlighting Questions

Q37. Results: Comparing the order of usefulness (by ascending mean sum value) -

BEFORE most useful b,a,d,c,e least useful (2.00,2.25,2.63,3.00,3.13)

AFTER most useful b,a,e,d,c least useful (2.00,2.29,2.86,3.00,3.29).

Although the mean sum values are pretty close in range, there is a definite swing towards b) and a) being the most useful features of the spotlighting tool. Whereas c), d) and e) are regarded as being of average usefulness, since they are really only viable on-screen. While b) and a) are applicable to both screen and paper texts. Thus the students' order of features in terms of usefulness after using tackling the spotlighting experiments is :-

b) The ability to jump directly from one spotlight to another

a) Wrapping each instance of the search string in inverse video, to form a spotlight, that makes its location within the current screen totally obvious

e) The ability to spotlight 2 or more words at the same time

d) A counter indicating how many instances of the current word there were altogether, and which "position" the current spotlight holds eg. "3/5" would indicate a total of 5 instances of the word altogether and that the current spotlight is the 3rd one

c) The ability to jump forwards or backwards between spotlights (eg. use command "+3s" to move forward 3 spotlights, "-5s" to move back 5 spotlights)

Summary of Comments BEFORE spotlighting experiment

Positive Responses	Students
able to jump forward and back between spotlights (one at a time)	5
useful when debugging somebody else's code that is not well understood	7
using a) & b) to locate the variable being tracked quickly	6
d) gives an idea of where you are in the program	6
d) the counter defines the total number of variable instances	1

Negative Responses	Students
undecided about usefulness of spotlighting	2 4
spotlighting 2 words can get confusing	5
c) is of dubious value	6
e) is pointless	6

Summary of Comments AFTER spotlighting experiment

Responses	Students
locating variable instances within the code	4 5 8
non-declared variables	6 7 8
uninitialised variables	6 8
highlighting questionable areas of code	3
detecting errors by "missing" spotlights (error detection by spotlight omission)	7
variable tracing	8
setting of variables (sequencing of variable value modifications)	8
locating a specific point in a program	8
ability to spotlight 2 variables, and enable parallel debugging	8
where SS8's definition of parallel debugging is: "whereby you [can] concentrate on more than one variable whilst only reading through the code once".	

Comments after trying the spotlighting concept out are more enthusiastic. Students seem to have grasped the uses of spotlighting quite well, since 5 out of 8 of them have commented on how they help to find different errors associated with variables.

Q1. List the task aspects that the spotlighting tool helped with, and explain how or why.

Positive Responses	Students
spotting non-declared variables	7 8
obvious for spotting statement omissions (error detection by spotlight omissions)	7
variable tracing	8
initialising and setting of variables (sequencing of variable value modifications)	8
locating specific variables enables close inspection of surrounding context	8
highlighting questionable areas of code	3
allowing concentration on one variable at a time (to some extent)	3
checking places where you expect a variable to appear	5
Negative Responses	Students
only helps with explicit statements	6
where anything depends on more than one variable it cannot help	6
having more than one variable spotlighted at once could get complicated	6

4 out of 7 students had something positive to say about spotlighting, with only 1 out of 7 students (SS6, above) expressing a negative view.

Q2. Spotlighting's tasks aspects (features) are :-

1. Focussing attention on a specific item
2. Keeping track of all the locations that involve the specified item
3. Trail following
4. Associating an error with a particular variable
5. Confirming the location of a hypothesized error
6. Showing up variable (non-)declaration errors
7. Showing up (non-)initialisation errors
8. Showing up variable value modification errors
9. Showing up sequencing errors

Q2a. Which tasks aspects did spotlighting help with? Choices are yes, maybe no.
(YMN)

Task	1	2	3	4	5	6	7	8	9
y	7	6	3	1	1	7	5	1	-
m	-	1	3	5	6	-	2	6	4
n	-	-	1	1	-	-	-	-	3

sum 7 8 12 14 13 7 9 13 20 if y=1, m=2 & n=3.

Order of ascending scores gives - most useful 1/6, 2, 7, 3, 5/8, 4, 9 least useful.

Personally, I think that task 9, showing up variable sequencing errors, is much more important, and should be ranked within the first 4 places; because sequencing errors are subtle and very difficult to spot.

The following list shows the resultant usefulness ordering of the task aspects, preceded by the sum value (ranging from 7-21) to show aspects in "tied" positions.

- 7 1. Focussing attention on a specific item
- 7 6. Showing up variable (non-)declaration errors
- 8 2. Keeping track of all locations of a specified item
- 9 7. Showing up (non-)initialisation errors
- 12 3. Trail following
- 13 5. Confirming the location of a hypothesized error
- 13 8. Showing up variable value modification errors
- 14 4. Associating an error with a particular variable
- 20 9. Showing up sequencing errors

Q2b. In contrast, asking students to rank task aspects in order of 9 "slots" of importance (thus eliminating "tied" positions) gave a different ordering altogether.

Order of importance by ascending sum value (ranging from 9-81) - most 7,1,4,3,8,6,9,2,5 least.

- 29 7. Showing up (non-)initialisation errors
- 30 1. Focussing attention on a specific item
- 31 4. Associating an error with a particular variable
- 32 3. Trail following
- 34 8. Showing up variable value modification errors
- 35 6. Showing up variable (non-)declaration errors
- 39 9. Showing up sequencing errors
- 42 2. Keeping track of all locations of a specified item
- 43 5. Confirming the location of a hypothesized error

Q2b.

Task	1	2	3	4	5	6	7	8	9
most	1	3	-	1	-	-	1	-	1
	2	-	-	1	1	-	1	2	1
	3	1	1	-	2	-	1	1	-
	4	-	1	-	-	2	1	1	2
avg	5	-	1	3	1	1	-	1	-
	6	1	1	-	3	-	-	1	1
	7	-	-	2	-	2	1	1	-
	8	-	3	-	-	2	-	-	1
least	9	2	-	-	-	-	2	-	1
Sum	30	42	32	31	43	35	29	34	39

Order of importance by ascending sum value - most 7,1,4,3,8,6,9,2,5 least.

On reflection, the Q2a result is probably more accurate, since it depended on a strictly tri-part (yes, maybe or no) choice for each aspect.

However, focussing attention on a specific item, showing up (non-)initialisation errors, showing up variable (non-)declaration errors, and trail following all appear within the first 6 items on both lists. Indicating that these are the most important aspects that spotlighting offers.

Q3. Did you find spotlighting helpful in the debugging task? YMN

If so, what aspects did it help you with?

SS1345678

yymymy

5y 2m

A positive response from 5 out of 7 students is encouraging.

Responses	Students
<u>finding non-declaration errors</u>	<u>6 7 8</u>
<u>checking where you expect a variable to appear</u>	<u>5 7</u>
<u>helped spot errors more easily</u>	<u>1</u>
<u>finding where a variable next appeared.</u>	<u>1</u>
<u>finding non-initialisation errors</u>	<u>6</u>
<u>finding variable modification errors</u>	<u>6</u>
<u>variable tracing</u>	<u>8</u>
<u>initialising and setting of variables</u>	
<u>(sequencing of variable value modifications)</u>	<u>8</u>
<u>locating specific variables enables close inspection of surrounding context</u>	<u>8</u>

Q4. Did spotlighting, as applied to this task, conflict with your natural debugging strategy? YMN

SS1345678

nnnnnny 1y 6n

I didn't expect spotlighting to interfere with anyone's debugging strategy; so the 1 yes vote (against 6 no votes) was rather disconcerting. Student 8's comment below, gives an indication of the problem.

SS8 — I jumped about the code a lot more, looking at the highlighted areas ... this made me feel that I may be missing other problem areas or not getting a full overview of the program (in order to put things in context)

Comment In real life, choosing if or when to use spotlighting would be up to the user. In the experiments they had the code spotlighted whether they wanted it or not. This lack of choice and unfamiliarity with the spotlighting concept could well have undone or removed any advantage expected to be conveyed by the tool, due to user resistance or resentment (whether conscious or not).

Q5. Did the benefits of spotlighting outweigh this latter consideration? YMN

SS1345678

nn-yy-m 2y 1m 2n

Similarly, I expected all answers to be yes (not just 2 out of 5); 1 maybe and 2 noes went against my expectations. There were no further comments to clarify the answers.

Q6. If you had 2 alternative debugging strategies, one sequence including spotlighting and the other using your own strategies, which would you choose, and why?

3 students (SS-3,5,6) definitely go for spotlighting + their own debugging strategies; and 2 (SS-4,8) for their own strategies only; and 1 (SS7) tends towards his own strategies only, but does not rule spotlighting definitely in or out. In the comments, student 8 changes his mind, and decides that spotlighting could be useful after all.

Thus there are between 3 and 5 students who would use spotlighting in addition to their own debugging strategies.

Student 7 tends to rely on the compiler's error messages and program execution behaviour more than most. He seems to be totally dependent on the programming environment's facilities, and has problems when deprived of a live system to test and run code on.

Q7. Would the form of debugging medium affect your decision? For example,
a) would you use spotlighting on a paper system [Y] [M] [N]
b) on a screen editing system [Y] [M] [N]
Give reasons :-

SS1345678 Y M N
a nm-my-m a 1 3 1
b nymymy b 4 2 1

The data shows that out of 7 students, 4 would and 2 might use spotlighting if it was an on-screen tool. Whereas out of 5 students, 1 would and 3 might use spotlighting on paper. This is probably the reason why the spotlighting experiments did not get as favourable a reception as expected.

Positive Responses	Students
doesn't matter which medium is used for highlighting	1
only useful on paper when a few (1 or 2) words are of interest	3
spotlighting only as needed on-screen (no wasted paper)	3
easier to flick between spotlighting different variables on-screen	5
on paper can see how one (or each) particular instance of a spotlighted variable sits within the whole program	6

Negative Responses	Students
paper wastage if doing many reprints	3 5 8
more than 1-2 words spotlighted on paper could confuse	3 5 8
on screen, you can only see a small part of the program	6
on paper, once highlighted, cannot de-highlight words and then spotlight others	8
as proportion of spotlighted to non-spotlighted words increases it will become more difficult to differentiate between the latest spotlighted word, and those previously spotlighted	8

Paper wastage is obviously a major concern with students. However, using the printer to put spotlights onto paper would relieve them of lost time, and the initial manual effort and eyestrain expended on manual spotlighting. Additional spotlighting could always be done manually anyway. However the use of a colour printer would make manual spotlighting practically obsolescent, and enable more than 2 variables (or words) to be spotlighted at a time. For example, red could be used for the 1st selected word, green for the 2nd, and blue for the 3rd, and so on. So the number of spotlights that would be differentiable would depend on the palette available on the printer.

Q7B.2 Layout Questions

Q8. How much does your own layout style differ from that used in this experiment?
What form does the difference take? eg. indentation, construct layout, etc.

Tick-Box Responses	Students
none	3
slightly	1 4 5 6 7 8

Comment Responses on Differences in Layout Style	Students
none / basically the same	6 8
use more begin-ends for extra clarity	4
slight difference in construct layout of some if-then statements	5
comments and variables names	7

Votes for layout style differences are: 6 slightly, 1 none. However Student 6 says the difference between experimental code layout and his own is "none"; so votes may change to 5 for slightly, and 2 for none.

Q9. Does having the code in your preferred style help you work on it or not? If so, how?

Responses	Students
better readability	3 7 8
can clearly see (by lining up) where loops start & end	6 8
easier to follow code	5

Obviously, readability and being able to line up loops (thus defining scoping control) are important features for comprehension and accurate debugging, scoring 3 and 2 votes respectively. Student 5 thinks that is easier to follow code in your own preferred style.

Q10. Is it easier to spot errors in your preferred style of layout than in a standard format?

Responses	Students
more begin-ends helps to define start & end of loops	4 6
easier to follow flow of program	5 8
helps debugging	5 8
easier to spot errors in a preferred style	1

These answers, again, clearly show the need for a layout tool. Students did seem to use too many begin-end statements in the debugging and spotlighting tasks. They didn't seem to be able to grasp/appreciate the "natural scoping" of code, extended by some constructs, when they appeared in a slightly more complex form than usual. Like nesting two IF or FOR statements that acted on a single statement. It seemed obvious to me where scoping started and ended, and I found it strange that they couldn't grasp it. However, this may have come about as a result of getting used to programming in C, where curly brackets are used instead of BEGIN-ENDs as in Pascal. So they could have been translating C's scoping habits into Pascal.

Q11. Ranking ease of error spotting in preferred, standard or other layout style. Votes were :-

6 votes for "easier" in a preferred layout style;
6 votes for "average" in a standard layout; and
7 votes for "more difficult" in a non-standard layout.

I think these answers clearly show that it is easier to debug a program if it is in the programmer's own style of layout. These results also show that it becomes progressively more difficult to spot errors as the stylistic differences increase.

Q12. Which do you prefer: systems that lay your code out automatically, like MacPascal?; or systems that leave the layout to you entirely ie. manual layout; or semi-automatic systems that help with layout by providing automatic indentation, with an editor such as vi? Tick preference boxes below, and Explain why:

Automatic layout (eg. MacPascal)	[Y] [N] [don't mind]
Semi-automatic layout (eg. vi with auto-indent "on")	[Y] [N] [don't mind]
Manual layout (eg. vi with auto-indent "off")	[Y] [N] [don't mind]

SS1345678	Y N D
a dddndy	a 1 2 4
s yyydydn	s 5 1 1
m ddyddn	m 1 1 5

The data shows a definite preference (5 to 1 to 1 respectively) for semi-automatic layout over fully automatic or fully manual layout in editors or programming systems. The comment responses reflect the students' need for firm/precise control over layout and its tailorable aspects.

Responses	Students
on/off auto-indent control at user's request on selected code	1 6 8
tailorable layout - able to specify indentation for each construct	6 8
prefer semi-automatic, as long as layout style is tailorable	6
semi-automatic layout allows flexibility and saves time	3
tidy up at user's request	5
prefers to lay out code manually	5
anything to reduce the indenting burden is welcome!	8
finds MacPascal's constant reformatting of code confusing	5

Using vi's auto-indent while adding code (manually) allows flexibility in layout style. It is up to the user how the code is laid out. Once the code is set in its pattern, it stays that way. Unless it is changed deliberately. Either manually (adding/deleting/modifying code) or by executing editing commands. So the layout style is completely under user control. This means that the layout style can be varied for the same construct under different conditions.

Such as the variation in layout for an IF-THEN clause for a single statement, or for a compound statement. Once laid out the code is fixed, whereas MacPascal "relays" the code (from the point of modification downwards) whenever you add or remove code. Adding or deleting a level of control can cause a lot of reformatting. MacPascal does it all for you. But with an editor like vi, the user has to do it himself, but because of this, the layout style remains his own. MacPascal is faster of course, and saves time, but it enforces its own layout style, and the user has little say in the matter.

Q13. In your opinion, which editing system best supports your layout preference? Why?

Responses	Students
automatic (MacPascal)	1 8
semi-automatic	3 4 7
vi, if it were easier to use	5 6

Votes for editing/programming systems are: 2 for vi, 2 for MacPascal and 3 for semi-automatic. However the 2 students (SS5 & 6) who voted for vi, wish that vi were easier to use. Comments from SS1, 5, 6 & 7 sum it up :-

SS1 — Automatic probably, because I like to indent a lot and highlight keywords (as in MacPascal)

SS5 — Difficult to say. Both vi and MacPascal have their problems, but if vi were easier to use then I would choose it.

SS6 — vi (or similar). It provides the right level of layout facilities without getting in the way. I don't like vi itself very much - I prefer something a little easier to use (ie. which doesn't force me to remember so many cryptic commands)

SS7 — Semi-automatic. Takes the pain out of indenting

Q14. Does it matter to you whether you lay the code out yourself or it is done for you?

Responses	Students
prefer layout support with override	3 5 6
manual	4 5
either - as long as code is laid out in preferred style	7 8
prefer layout support with a tidy-up option	5

So 3 students prefer layout support with override facility. Whereas students 7 & 8 don't mind what type of editor, as long as the code ends up in their own preferred style.

I have prepared 3 chunks of code that are "90% complete" development-wise. The problem is, that each piece of code contains some errors, mostly semantic, logic or algorithmic errors. It is up to you to determine the nature of each error, and to fix it by "correcting" the code as you would on one of your own printouts, but please take care that "arrows" indicate clearly where each code insertion or modification is to go!

**The title for each task tells you how many errors there are in each program. Please number each error that you find, eg. ①, ②, etc.

The purpose of these debugging tasks is to find out how you go about the debugging task, and to get an idea of how long it takes you to "correct" each set of errors, and which errors you tackle first. Thus it is important that you note down your thoughts, actions and/or strategies (to the right of the code, in the space indicated) as you debug each piece of code, as well as the time you start and finish each new action.

I also want you to note down the start and end times for the first reading of the task description, the first reading of the code, and the times at which you start/finish debugging each piece of code. Please state start/end times as hh:mm:ss, and remember to check that you have written the time(s) down whenever you start or finish one of these actions! I will need this timing information as a guide to the way that time is spent whilst debugging.

Each piece of code has a task description that tells you what the code is expected to do. All the task descriptions are on one reference sheet. A table is also placed underneath each task description, for you to note how often you refer to the task description, whilst debugging. It is essential that you fill it in whenever you consult the task description, no matter how briefly. This will tell me how important the task description is to the debugging process, and what part it plays during debugging.

Helpful information : The primes task has a hint as to how the main algorithm works, and the concordance task has a selection of error messages and error evidence, in the order you would expect them to be generated if the code was run and tested in a Pascal environment. NB. These hints appear at the end of the code, so that you can read them, after having your first look through the code. There is a double-sided sheet that also contains information as to how the algorithms for tasks 2 & 3 are intended to work. I suggest that you read the task description first, then the code, then the algorithm hints, to get the best initial understanding. After that it's up to you.

The following is a list of procedure declarations that are used within the code, with a brief explanation so that you know what they do, and you don't get put off when you find them in the code. I have treated them like pre-defined procedures, ie. so that they do not need to be declared before use, just like "writeln", "readln", etc. (to save unnecessary clutter).

```

procedure openfile(var thisfile : text; name : string);
{ opens the file "name", and enables its contents to be read from thisfile }

procedure closefile(var thisfile : text; name : string);
{ closes the file "name", whose contents now correspond to thisfile }

procedure rewritefile(var thisfile : text; name : string);
{ opens the file "name" for rewriting as thisfile }

procedure ask(question : string; var answer : boolean);
{ writes the question to the screen, and returns answer value of true
  if the user's response is y/Y for yes, otherwise returns value false }

procedure readname(message : string; var name : string; var ok : boolean);
{ reads in the name of the file to be used, using the message string to
  complete the prompt given to the user; returns ok value true if
  a name is given, false if quit is entered instead of name }

procedure writedot(var itemcount : integer);
{ writes a dot and increments the itemcount variable to indicate to the user
  that another entry has been processed. A linefeed is issued every 80 dots }

procedure readword(var infile : text; var nuword : string);
{ reads nuword in from infile }

```

Please tick only one column for each reading, so that I can tell exactly which "readings" were brief and which were (more) thorough. Thank you. (D2)

Remember to number errors in the order in which you find them, and to write down your intended actions/strategies/thoughts throughout the debugging of the code, no matter how trivial you think they are!

NB. Questions 14 & 15 can be used as a prompt list of (most) likely errors that you will find in the debugging tasks.

Debugging Task 1 - Primes - Contains 7 Errors

Primes Task description

Reading Time Start : :

Write a program that lists out all the prime numbers between 2 and 1001, in ascending order, in a simple table 80 columns wide, giving each prime number a field width of 5 (ie. 16 columns, each 5 spaces wide). Also state the number of prime numbers in that range.

NB. If P is a prime number then P cannot be divided exactly by any other (integer) number, other than itself and 1. For example, 7 is a prime number because $1 \times 7 = 7$, and 2, 3, 4, 5 and 6 do not divide into 7 without remainder. Likewise, 2 and 3 are also prime numbers, but 4 is not since $4/2 = 2$ (ie. 2 is a factor of 4, since $2 \times 2 = 4$).

Reading Time End : :

No. of times	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Brief reading																				
Thorough reading	✓																			

Debugging Task 2 - Letter Pairs - Contains 6 Errors

Letter Pairs Task Description

Reading Time Start : :

Write a program to count the frequency of letter pairs eg. 'ea', 'le', 'at', etc. in a text file, and print a (26x26) matrix containing the frequency values for each letter pair. The matrix should have a label for each individual letter (eg. a, b, c, ... z), across the top and down the side, to make reference to the frequency values straightforward.

Reading Time End : :

No. of times	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Brief reading																				
Thorough reading	✓																			

Debugging Task 3 - Concordance - Contains 6 Errors

Concordance Task Description

Reading Time Start : :

Write a program that will read the words from a text file and store them in a linked list, placing each word in "dictionary" order in the list (ie. a's at the top and z's at the bottom), and keeping count of how many times each word occurs, and the number of words in the input file in total. When all the words have been read, the total word count should be written out; followed by the complete list of words with the frequency count for each word.

Reading Time End : :

No. of times	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Brief reading																				
Thorough reading	✓																			

Debugging Tasks

Control Errors in Primes Code

- 1) wrong initial value
- 2) missing declaration
- 3) wrong initial value on loop
- 4) wrong operator used
- 5) missing initialisation
- 6) not paying attention error
- 7) wrong sequencing of code

Error Solution

firstnum's value should be 2
 count : integer;
 mult's initial value should be 2
 + should be * (to give num * mult)
 count := 0;
 should write "num" value, since prime[num] = true
 writeln statement should be inside the begin-end loop

Control Errors in Letter Pairs Code

- 1) wrong initial value
- 2) not paying attention error
- 3) wrong variable value modification
- 4) missing re-initialisation
- 5) wrong upper bound on loop range
- 6) not paying attention error

Error Solution

all matrix elements should be initialized to value 0
 thisch is read from standard input instead of from the named file, should be read(infile, thisch);
 increments matrix value by 11 instead of 1
 prevch value is not updated, so need to add
 prevch := thisch; at end of loop
 for loop only executes once since initial value = end value, upper bound should be chz
 writes out matrix[across,across] value repeatedly, should write matrix[across,down] value instead

Control Errors in Concordance Code

- 1) missing declaration
- 2) wrong variable initialise
- 3) missing re-initialisation
- 4) wrong re-initialisation value
- 5) inappropriate declaration
- 6) wrong operator used

Error Solution

count missing from wordrec, need count : integer;
 should be q := top; in addword (not lastq := top)
 only the last word inserted remains on the list, need to add p↑.next := q; in procedure insert
 infinite loop caused by q := top↑.next, should be q := q↑.next;
 top should be declared as a var parameter, otherwise the wordlist does not get built up
 - should be + (so count := count + 1)

Concordance Errors Hints/Evidence Discovered by Compiler and During Testing

- 1) Undeclared variable "count" in procedure insert and addword.
- 2) Q not initialised in procedure addword.
- 3) Procedure addword is not working - after reading in the file, the wordlist is still empty.
- 4) Procedure writelist goes into an infinite loop, writing out the first element of the wordlist repeatedly.
- 5) Only the last word inserted into wordlist remains, all others have "disappeared".
- 6) Words (read in from infile) repeated more than twice have negative count values.

Reading Time Start : :

Debugging Time Start : :

program primes(input, output);

Thoughts, Actions & Strategies List

const

{ define range of prime no.s and array bounds }

firstnum = 1; → 2;

CE1

lastnum = 1001;

var

prime : array[firstnum..lastnum] of boolean;

num, mult : integer;

count : integer;

CE2

begin

{ initialise by assuming all are prime no.s }

for num := firstnum to lastnum do

prime[num] := true;

{ "remove" all multiples of the current prime no. }

for num := firstnum to lastnum do

if prime[num] then → 2;

CE3

for mult := 1 to (lastnum div num) do

prime[mult * num] := false;

CE4

count := 0; → *

writeln('The prime numbers between 2 and 1001 are :-');

{ write out each prime number }

for num := firstnum to lastnum do

if prime[num] then

begin

count := count + 1;

write(prime[num] : 5);

end;

{ allow 16 prime no.s per line, then send writeln }

if (count mod 16 = 0) then writeln;

writeln;

writeln('There are ', count,

' prime numbers between 2 and 1001 altogether');

end.

Reading Time End : :

Debugging Time End : :

Algorithm Hint :

The aim of the program is to start with the smallest (prime) no. in the array, and to remove all multiples of this no. up to the maximum limit (eg. 1001). So starting with the value 2, 2 is a prime no. so "remove" the values given by: 2x2, 3x2, 4x2, ... 500x2 from the array. That is, to make false all array values corresponding to the elements 4,6,8,...1000. Then go to the next prime no. in the array (ie. the next one with the value true) and remove its multiples, in the same way. At the end of the cycle, the list of prime no.s corresponds to those array elements whose values are true.

Reading Time Start : :

Debugging Time Start : :

program letterpairs(input,output);

Thoughts, Actions & Strategies List

const

```

numwidth = 4;
space = ' ';
cha = 'a';
chz = 'z';

```

type

letter = cha..chz;

var

```

matrix : array[letter,letter] of integer;
across, down : letter;
thisch, prevch : char;
infile : text;
name : string;
ok : boolean;

```

begin

```

for across := cha to chz do
  for down := cha to chz do
    matrix[across, down] := 0;

```

readname('input file', name, ok);

if ok then

begin

openfile(infile, name);

prevch := space;

while not eof(infile) do

begin

read(thisch);

if eoln(infile) then readln(infile);

{ increment frequency count iff both thisch and prevch are letters }

if [thisch, prevch] <= [cha..chz] then

matrix[thisch, prevch] := matrix[thisch, prevch] + 1;

end;

closefile(infile, name);

write(space : 2);

for down := cha to chz do { write column headings }

write(space : numwidth - 1, down);

writeln; writeln;

for across := cha to chz do

begin { write out each row of frequency values }

write(space, across);

for down := cha to chz do

write(matrix[across, down] : numwidth);

writeln;

end;

end;

end.

Reading Time End : :

Debugging Time End : :

NB. [thisch, prevch] <= [cha..chz] is the short way of writing
(thisch in [cha..chz]) and (prevch in [cha..chz])

Information to Help You Understand The Algorithm(s) Intended To Execute The Tasks

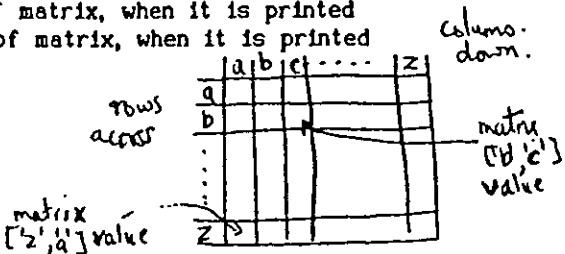
Letter Pairs

numwidth (=4) is the no. of spaces allocated to each frequency value element contained in matrix when it is printed out.
thisch is the current character read from infile
prevch is the previous character read from infile (ie. last cycle's thisch)
across is the for loop variable for the "row" elements of matrix, when it is printed
down is the for loop variable for the "column" elements of matrix, when it is printed

Algorithm

"initialisation"

all matrix elements become zero
prevch is initialised to space
"counting the letter pairs"
while infile is not finished
 read current character from infile
 if both thisch and prevch are letters (ie. both are in ['a'..'z']) then the matrix count value for the letter combination [prevch, thisch] is incremented by 1
 prevch then takes on the old value of thisch, so that the next character can be read in (eg. if prevch='g', and thisch='e' then matrix['g','e'] := matrix['g','e'] + 1; and then prevch := thisch, ie. prevch='e')
 and the loop continues until end of file.



"writing out entire (26x26) matrix values with row and column headings" (see diag ↑ of example printout)

Writing out column headings : write 2 leading spaces (for row headings on subsequent rows), followed by all 26 letters - giving 3 spaces followed by the appropriate letter for each column, in turn (from a to z)

Then write out each row's frequency values on a new line, allocating 4 spaces per value, and putting the "row" label at the front of each row's values.

No. of times		1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	
Brief reading																						
Thorough reading		✓																				

NB. [thisch, prevch] <= [cha..chz] is the short way of writing (thisch in [cha..chz]) and (prevch in [cha..chz])

Concordance 2 pages of code

Main Program

Infile is opened for reading, wordcount is initialised to zero, and wordlist is initialised to nil (ie. wordlist is empty). As each new word is read in from infile it is added to wordlist, in the correct position (as per "dictionary" order). As each word is processed, a dot is written to the screen so the user can see that the program is working ok, and the wordcount value is incremented by 1. When all the words have been read in from infile, the file is closed, and writelist is then used to print out all the words in wordlist, starting with the first word.

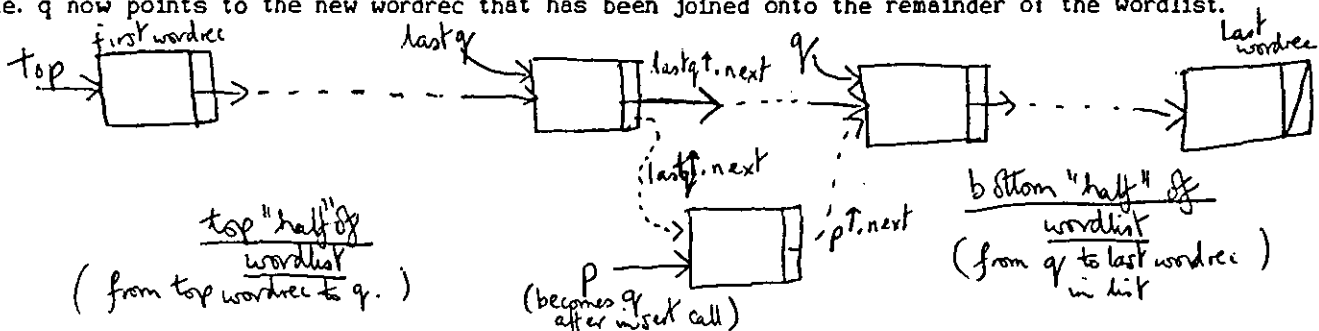
Procedure Addword

top is the pointer to the top of wordlist
q is the pointer which moves down the list of words, one record at a time
lastq points to the record where q pointed to on the previous cycle around the while loop
The purpose of this procedure is to look through the list of words, starting at the top of the wordlist, comparing the new word with each word in the list. When a word is reached that occurs later in the alphabetic ordering than new word ie. when new word < current word, then the new word is placed in front of the current word using "insert".
[eg. if new word is 'bat' and current word is 'ball' then new word > current word, but if new word is 'bat' and current word is 'colour' then new word < current word.]
If wordlist is empty (top=nil) then obviously, the new word is inserted at the top of the list, as there are no other records to compare it to.
If the word already appears in the list then the frequency "count" of that word is incremented.

No. of times	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Brief reading																				
Thorough reading	✓																			

Procedure Insert

p is the pointer that is (or will be) pointing to the new wordrec record
q is the pointer that is pointing to the remainder of the word list
"new(p)" creates a new wordrec attached to the pointer p
The purpose of insert is to fill all the fields of this new wordrec (pt.word, pt.count and pt.next) with the appropriate initial values, and then to "hook" it into/onto the correct position in the wordlist, in front of the record pointed at by q. q then takes on p's value ie. q now points to the new wordrec that has been joined onto the remainder of the wordlist.



To insert the record pointed to by p, in front of the record pointed at by q requires 2 statements, (in the following order) :

```
pt.next := q; { the next pointer now points to the same record as q }  
q := p; { q now points to the same record pointed to by p }
```

However, unless q points to the top of the wordlist, the "front" end of the wordlist (from top to the record that used to point to the record pointed at by the "old" q (before insertion of the new record) will have to be re-attached to the newly inserted record, otherwise the inserted record will not be accessible.

This is cured by applying lastqt.next := q; which completes the linkage between the previous record and the new record which is already attached to the remainder of the wordlist (by the previous execution of the pt.next := q; statement). (in addword after the insert statement)

No. of times	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Brief reading																				
Thorough reading	✓																			

Procedure Writelist

q is the pointer which moves down the list of words, one record at a time
q starts at the top of the word list, and writes out the word-string and count values held in each record, until the end of the list is reached.

No. of times	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
Brief reading																				
Thorough reading	✓																			

Copy of Error Hints Shown At End of Concordance Code

Note that the main program is error-free - the bugs are elsewhere, as described below.

Errors Hints/Evidence Discovered by Compiler and During Testing

- in The Order They Would Be Discovered During "Real" Debugging

- Undeclared variable "count" in procedure insert and addword.
- Q not initialised in procedure addword.
- Procedure addword is not working - after reading in the file, the wordlist is still empty.
- Procedure writelist goes into an infinite loop, writing out the first element of the wordlist repeatedly.
- Only the last word inserted into wordlist remains, all others have "disappeared".
- Words (read in from infile) repeated more than twice have negative count values.

(2 pages of code)

Reading Time Start _: _:

Debugging Time Start _: _:

```

program concord(input, output);
type

```

Thoughts, Actions & Strategies List

```

    link = ↑wordrec;
    wordrec =
        record
            word: string;
            next: link;
        end;
var
    infile: text;
    name1, nuword: string;
    ok: boolean;
    wordcount: integer;
    wordlist: link;

```

(CE1)

```

{ ----- }
# procedure insert
# ----- )

```

```

procedure insert(var nuword: string; var q: link);
{ creates new wordrec, sets up initial count and word values, then
  appends q's list to end of new wordrec, and passes it back as q }
var p: link;
begin
    new(p);
    p.count := 1;
    p.word := nuword;
    q := p;
end; { of procedure insert }

```

p↑.next := q;

(CE3)

```

{ ----- }
# procedure addword
# ----- )

```

```

procedure addword(var nuword: string; top: link);
{ inserts new word at "dictionary" correct position in list }
var lastq, q: link;

```

[var top: link]

```

begin
    if (top = nil) or (top.word > nuword)
    then insert(nuword, top) { insert new word at top of list }
    else { cycle through list until new word > current word }
    begin
        lastq := top;
        while (q <> nil) and (q.word < nuword) do
        begin
            lastq := q;
            q := q.next;
        end;
        if (q <> nil) and (q.word = nuword) { if new word already in list }
        then q.count := q.count + 1 { then increment counter }
        else { insert new word in front of current word }
        begin
            insert(nuword, q);
            lastq.next := q;
        end;
    end;
end; { of procedure addword }

```

(CE2)

(CE6)

Debugging Time End _: _:


```

{ ----- }
* procedure writelist
* ----- )

procedure writelist(var top : link);
{ write out entire contents of the list, from the top, down }
begin
  q := top;
  while q <> nil do
    begin
      writeln(q↑.word, q↑.count : 5);
      q := top↑.next;
    end;
end; { of procedure writelist }

{ ----- }
* MAIN PROGRAM
* ----- )

```

```

begin
  readname('input file', name1, ok);
  if ok then
    begin
      openfile(infile, name1);
      wordcount := 0;
      wordlist := nil;
      while not eof(infile) do
        begin
          readword(infile, nuword);
          addword(nuword, wordlist);
          writedot(wordcount);
        end;
      closefile(infile, name1);
      writeln('There are ', wordcount, ' words altogether. ');
      writelist(wordlist);
    end;
end.

```

Reading Time End : :

Debugging Time End : :

Note that the main program is error-free - the bugs are elsewhere, as described below.

Errors Hints/Evidence Discovered by Compiler and During Testing

Undeclared variable "count" in procedure insert and addword.

Q not initialised in procedure addword.

Procedure addword is not working - after reading in the file, the wordlist is still empty.

Procedure writelist goes into an infinite loop, writing out the first element of the wordlist repeatedly.

Only the last word inserted into wordlist remains, all others have "disappeared".

Words (read in from infile) repeated more than twice have negative count values.

Debugging Tasks Comment Sheet & Difficulty Rating For Each Task

*NB. Task requirements means all the things you have to consider/evaluate in order to develop the software so that it will fulfill the task description, and all the constraints (whether stated or not) imposed by the Pascal language, (and the machine in some cases) and the way that the algorithm(s) cause the task to be executed. In essence, the task description defines what is to be done, and the task requirements define how to do this whilst meeting all the (sometimes conflicting) constraints imposed by the language, machine and task.

Task 1 - Primes

Ease of understanding task description easy ☐☐☐☐☐ difficult

*Ease of understanding task requirements easy ☐☐☐☐☐ difficult

Ease of understanding code itself easy ☐☐☐☐☐ difficult

Ease of understanding how the coded algorithm(s) fulfills the task easy ☐☐☐☐☐ difficult

Comments :

Primes

SS 1345678	Score	1	2	3	4	5	Sum	Mean
TD 1222121	TD	3	4	0	0	0	11	1.57
TR 1323221	TR	2	3	2	0	0	14	2.00
UC 3354233	UC	0	1	4	1	1	23	3.29
UA 2333243	UA	0	2	4	1	0	20	2.86

Task 2 - Letter pairs

Ease of understanding task description easy ☐☐☐☐☐ difficult

*Ease of understanding task requirements easy ☐☐☐☐☐ difficult

Ease of understanding code itself easy ☐☐☐☐☐ difficult

Ease of understanding how the coded algorithm(s) fulfills the task easy ☐☐☐☐☐ difficult

Comments :

Letter Pairs

SS 1345678	Score	1	2	3	4	5	Sum	Mean
TD 1122221	TD	3	4	0	0	0	11	1.57
TR 2222231	TR	1	5	1	0	0	14	2.00
UC 2223244	UC	0	4	1	2	0	19	2.71
UA 2123343	UA	1	2	3	1	0	18	2.57

Task 3 - Concordance

Ease of understanding task description easy ☐☐☐☐☐ difficult

*Ease of understanding task requirements easy ☐☐☐☐☐ difficult

Ease of understanding code itself easy ☐☐☐☐☐ difficult

Ease of understanding how the coded algorithm(s) fulfills the task easy ☐☐☐☐☐ difficult

Comments :

Concordance

SS 1345678	Score	1	2	3	4	5	Sum	Mean
TD 1342131	TD	3	1	2	1	0	15	2.14
TR 3324341	TR	1	1	3	2	0	20	2.86
UC 5344453	UC	0	0	2	3	2	28	4.00
UA 5334452	UA	0	1	2	2	2	26	3.71

Instructions for Timed Debugging Tasks Experiments

There are 2 pieces of code in each half of this debugging experiment. As in the 1st experiment, a task description and/or algorithm is provided for each piece of code, so you should have no difficulty in understanding what the code is supposed to do. It is your job to spot all the discrepancies in the code (ie. the errors) and to correct each one, so that the code will work according to the task description/algorithm. However, for this experiment, it is vital that you note down the start and finish times for detecting EACH error, and remember to NUMBER each error that you find. So that I can tabulate each error's "debugging time" and work out a mean value for each error across the group as a whole.

Types of Errors

There are NO missing semicolons, or missing/mis-matched begin-end loops.

However, errors may be found in declarations, and inappropriate (or missing) initialisation, modification or re-initialisation statements.

There are also sequencing errors where the statement is correct/appropriate but it is in the wrong place, and so destroys the smooth running of the code, and the pattern of expected events.

[Assignment of values to variables - these are most likely to cause trouble (in any form of procedural programming), because if the value of the variable is modified wrongly, or the correct value is assigned to the wrong variable, then the events that depend on the correct variable getting the correct value "on time" will throw the rest of the algorithm(s) out of kilter (ie. into confusion eg. infinite loops etc..)]

Errors are usually associated with specific (ie. the spotlighted) variables, their values, and/or any language constructs or expressions that contain them.

[NB. for "spotlight" read "specific variable under investigation".]

Errors can be detected by examining those lines containing a spotlight, and/or the lines between spotlights. However, errors of omission can be detected by the lack of expected statements in "critical" positions (eg. missing declarations or initialisation statements).

For example

- if a variable isn't declared then its spotlight won't appear in the declaration area.
 - if a variable isn't initialised then its spotlight won't appear in the position where it should have been initialised.
- etc.

"Plain" Debugging Tasks

Control Errors in Letter Count Code	Error Solution
1) missing declaration	ch : char;
2) missing initialisation	wordlen := 0;
3) wrong sequencing of code	read(ch) should be first statement inside repeat loop
4) not paying attention error	in the else clause, the const variable should be incremented not the vowels variable
5) wrong comparator used	should be if wordlen > maxwordlen
6) wrong sequencing of variable value modifications	move wordlen := 0; below if statement, so that maxwordlen is modified correctly
7) missing output statement	writeln('Word count = ', wordcount);

Control Errors in Shell Sort Code	Error Solution
Proc sort errors :	
1) missing declaration	temp : integer;
2) wrong initialisation value	should be alldone := true; (not false)
3) wrong initialisation value	should be n := m + jump;
4) wrong comparator used	< > instead of >
5) assignment back to front	row[m] := temp; should be temp := row[m];
main loop errors :	
6) wrong initial value on loop	for loop should start at 1, for i := 1 to count do
7) writing wrong output value	writing out value of inrow[i+1] instead of inrow[i]

Spotlighted Debugging Tasks

Control Errors in Survey Code	Error Solution	Relevant Sheet
1) missing declaration	signal : 0..2; or signal : integer;	signal
2) missing initialisation	time := 0;	time
3) missing initialisation	maxwait := 0;	maxwait
4) wrong sequencing of code	swap read(signal); and "repeat"	signal
5) wrong sequencing of variable value modifications	move wait := wait + 1; above if stat	wait or maxwait
6) missing re-initialisation	wait := 0; when vehicles increments	wait

Control Errors in Bubble Sort	Error Solution	Relevant Sheet
1) mis-declaration of variable	should be temp : integer;	temp
2) missing re-initialisation	inorder := true; above for loop	inorder
3) wrong initialisation value	j := i + 1; (not j + 1)	i or j
4) wrong comparator used	should be if num[i] > num[j]	i or j
5) wrong sequencing of variable value modifications	move temp := num[j]; up 2 lines	temp, i or j
6) wrong upper bound on loop range	for i := 1 to maxels (not max)	i
7) not paying attention error	write(num[i] : 6); (not i)	i

"Plain" Debugging Tasks

Letter Count Description

The aim of this program is to read in text from the input stream, and to count and eventually print out the total no. of vowels and consonants, the total no. of words and to give the length of the longest word in the text.

Algorithm

The algorithm starts by initialising all the variable values. The essence of the main loop is to read each character in, and to increment either the vowel or consonant count, if ch is a letter, or to increment the word count if ch is not a letter or a hyphen, '-', between words (eg. co-operate). When the end of the word is detected, the wordlen variable is returned to zero, after having checked its value against maxwordlen, and if necessary updated maxwordlen's value.

All the required values are printed out on detecting eof.

variables

ch = current character being evaluated

vowels = total no. of vowels (a, e, i, o, u) so far

consts = total no. of consonants so far

wordlen = length of current word

maxwordlen = length of longest word so far

wordcount = total no. of words read so far

** there are 6 errors associated with the ch, wordlen, consts/vowels variables

Letter Count									
Error	Start		Finish						
No.	hh	mm	ss	hh	mm	ss	hh	mm	ss
1	:	:	:	:	:	:	:	:	:
2	:	:	:	:	:	:	:	:	:
3	:	:	:	:	:	:	:	:	:
4	:	:	:	:	:	:	:	:	:
5	:	:	:	:	:	:	:	:	:
6	:	:	:	:	:	:	:	:	:

	1			2			3			4			5			6		
	hh	mm	ss	hh	mm	ss	hh	mm	ss	hh	mm	ss	hh	mm	ss	hh	mm	ss
Start	:	:	:	:	:	:	:	:	:	:	:	:	:	:	:	:	:	:
Finish	:	:	:	:	:	:	:	:	:	:	:	:	:	:	:	:	:	:

```

program lettercount(input,output);
var
  vowels, consts, wordlen,
  maxwordlen, wordcount : integer;
  ch : char;

```

```

begin

```

```

  vowels := 0;

```

```

  consts := 0;

```

```

  maxwordlen := 0;

```

```

  wordcount := 0;

```

```

  repeat

```

```

    if ch in ['A'..'Z']

```

```

      { demote ch to lower case }

```

```

      then ch := chr(ord(ch) + 32);

```

```

    [read(ch):

```

```

    if ch in ['-','a'..'z'] then

```

```

      case ch of

```

```

        'a', 'e', 'i', 'o', 'u' :

```

```

          begin

```

```

            vowels := vowels + 1;

```

```

            wordlen := wordlen + 1;

```

```

          end;

```

```

        '-' : { do nothing } ;

```

```

      else

```

```

        begin

```

```

          vowels := vowels + 1;

```

```

          wordlen := wordlen + 1;

```

```

        end;

```

```

      end { of case }

```

```

    else { ch is space, comma or fullstop, }

```

```

      begin { so word is ended }

```

```

        [wordlen := 0:

```

```

        wordcount := wordcount + 1;

```

```

        if wordlen < maxwordlen

```

```

          then maxwordlen := wordlen;

```

```

        end;

```

```

      until eof;

```

```

      writeln('Vowel count = ', vowels);

```

```

      writeln('Consonant count = ', consts);

```

```

      writeln('Maximum word length = ', maxwordlen);

```

```

    end.

```

[read statement
needs to
move up]

CE1

CE2

CE3

CE4

[incrementing
wrong]
variable

CE6

[more
down]

CE5

writeln('Word length = ', wordlen)

CE7

Shell Sort Algorithm

The aim of the algorithm is to sort an array of integer values, read in from input, into ascending order, and then to print them out.

Main loop algorithm :

The values are read into the array inrow, from the input stream, one at a time, by incrementing the count variable, until the end of file character is read. If count is larger than 1, then it is worth while sorting the array. However, first of all the last number in the array must be eliminated, by reducing the value of count by 1, since the last "value" is the eof marker, not a proper integer value.

The array is then sorted according to the shell algorithm, and the values are printed out in ascending order, 6 "spaces" per value, 12 values per line.

Algorithm for sub-procedure sort :

jump is the current "distance" between the pair of array elements of inrow, num[m] and num[n], that are being compared and if necessary, swapped, since $n = m + \text{jump}$. Swapping only occurs if $\text{num}[m] > \text{num}[n]$. The variable "temp" is used to hold one of these array values during the swap.

Now the first time round the loop jump is half the length of the array, and 2 elements are compared, one from the bottom half of the array and the other from the top half of the array. For example, say that inrow has 30 elements, then $\text{length}=30$, and the first time round the loop jump is 15, so elements [1] and [16], [2] and [17], ... [15] and [30] are compared/swapped, one pair at a time. Now once all these pairs have been compared and there are no more to be swapped, the jump size is halved again ($\text{jump}=7$) and the comparison and swapping of values ([1] and [8], [2] and [9], ... [23] and [30]) continues until these pairs are also in order. At each iteration, the higher values are moving to the top end of the array, and the lower values are moving downwards. This iteration continues to use smaller and smaller jump sizes ($\text{jump}/2$, $\text{jump}/4$, $\text{jump}/8$, etc.) until the jump size is 1 element, and every element's value is being compared/swapped with its neighbour. When this last loop finishes the array is sorted in ascending order, ready for printing.

**** 7 errors altogether : 2 associated with the main loop variable i; and 5 associated with the variables m, n, temp, and alldone belonging to sub-procedure sort**

Shell Sort											
Error	Start			Finish							
No.	hh	mm	ss	hh	mm	ss					
1	:	:	:	:	:	:					
2	:	:	:	:	:	:					
3	:	:	:	:	:	:					
4	:	:	:	:	:	:					
5	:	:	:	:	:	:					
6	:	:	:	:	:	:					
7	:	:	:	:	:	:					

	1			2			3			4			5			6			7			
	hh	mm	ss	hh	mm	ss	hh	mm	ss	hh	mm	ss	hh	mm	ss	hh	mm	ss	hh	mm	ss	
Start	:	:	:	:	:	:	:	:	:	:	:	:	:	:	:	:	:	:	:	:	:	:
Finish	:	:	:	:	:	:	:	:	:	:	:	:	:	:	:	:	:	:	:	:	:	:

```

program shellsort(input,output);
const maxlength = 1000;
type
  index = 1..maxlength;
  rowtype = array[index] of integer;
var
  inrow : rowtype;
  count : 0..maxlength;
  i : index;

  { -----procedure sort----- }
  procedure sort(var row : rowtype; length : index);
  var
    jump, m, n : index;
    alldone : boolean;
    temp : integer;
  begin
    jump := length;
    while jump > 1 do
      begin
        jump := jump div 2;
        repeat
          alldone := false;
          for m := 1 to (length - jump) do
            begin
              n := m + jump;
              if row[m] > row[n] then
                begin
                  temp := row[m];
                  row[m] := row[n];
                  row[n] := temp;
                  alldone := false;
                end;
            end;
          until alldone;
        end;
      end;
    end; { of procedure sort }

  { -----MAIN PROGRAM----- }

begin { shellsort }
  count := 0;
  while not eof do
    begin
      count := count + 1;
      readln(inrow[count]);
    end;
    if count > 1 then
      begin
        count := count - 1;
        { as last "number" read in is eof character }
        sort(inrow, count);
        for i := 0 to count do
          begin
            write(inrow[i] : 6);
            if (i mod 12) = 0 then writeln;
          end;
        end;
      end;
    end; { shellsort }

```

Handwritten annotations and corrections in the code:

- `temp : integer;` is written above the printed line.
- `alldone := false;` is written above the printed line, with `True;` crossed out.
- `n := m + jump;` has `m` circled and an arrow pointing to `n`.
- `if row[m] > row[n] then` has `<` circled and an arrow pointing to `>`.
- `temp := row[m];` is written to the right of the code, with an arrow pointing to the assignment.
- `row[m] := temp;` is crossed out.
- `write(inrow[i] : 6);` has `i` circled.

Circled labels (CE1) through (CE7) are placed near various lines of code.

Spotlighted Debugging Tasks

Survey Task Description

A detector is placed to count traffic, which sends signals back to a computer, which processes this information, in order to print out certain information. The input signals are as follows :

0 indicates the end of the survey period

1 indicates that a vehicle has passed the detector

2 indicates that another second has passed (time intervals are in seconds)

The detector is set up so that it only sends one signal to the computer at a time, so there is no problem with simultaneous signals. The information required at the end of each survey period is as follows : the total time elapsed during the surveying session, the vehicle count for this period, and the maximum time period when no vehicles were detected.

Algorithm

It is clear that the vehicle count increments whenever the signal is 1, and that the time and wait values increment (and the maxwait value may be altered, to reflect the maximum wait value) when the signal is 2. But the wait value must be reset to zero whenever a vehicle passes (ie. when signal=1) since this terminates the previous vehicle-less interval.

variables

signal is the current signal value

time is the time elapsed since the survey period began

wait is the length of time between successive "vehicle" signals being received

maxwait is the maximum value of "wait" during the current survey period

vehicles is the current vehicle count (ie. no. of "vehicle" signals received)

**** there are 6 errors associated with the signal, time, wait and maxwait variables.**

Survey Problem

Error	Start	Finish
No.	hh:mm:ss	hh:mm:ss
1	:	:
2	:	:
3	:	:
4	:	:
5	:	:
6	:	:

	1	2	3	4	5	6
	hh:mm:ss	hh:mm:ss	hh:mm:ss	hh:mm:ss	hh:mm:ss	hh:mm:ss
Start	:	:	:	:	:	:
Finish	:	:	:	:	:	:

program survey(input, output);
 var *signal*, *time*, *vehicles*, *wait*, *maxwait* : integer; [*signal* : 0..2;]
 begin
 wait := 0;
 vehicles := 0;
 read(*signal*); (CE1)
 repeat
 if *signal* = 2 then (CE4)
 begin
 time := *time* + 1;
 if *wait* > *maxwait*
 then *maxwait* := *wait*;
 wait := *wait* + 1;
 end;
 if *signal* = 1 then
 begin
 vehicles := *vehicles* + 1;
 end;
 until *signal* = 0;
 writeln('Time-span=', *time*, 'secs');
 writeln('Vehicle-count=', *vehicles*);
 writeln('Max-wait=', *maxwait*, 'secs');
 end.

program survey(input, output);
 var
 time, *vehicles*, *wait*, *maxwait* : integer;
 begin
 wait := 0; *time* := 0; (CE2)
 vehicles := 0;
 read(*signal*);
 repeat
 if *signal* = 2 then
 begin
 time := *time* + 1;
 if *wait* > *maxwait*
 then *maxwait* := *wait*;
 wait := *wait* + 1;
 end;
 if *signal* = 1 then
 begin
 vehicles := *vehicles* + 1;
 end;
 until *signal* = 0;
 writeln('Time-span=', *time*, 'secs');
 writeln('Vehicle-count=', *vehicles*);
 writeln('Max-wait=', *maxwait*, 'secs');
 end.

```

program survey(input, output);
var
  time, vehicles, wait, maxwait : integer;
begin
  wait := 0;
  vehicles := 0;
  read(signal);
  repeat
    if signal = 2 then
      begin
        time := time + 1;
        if wait > maxwait
          then maxwait := wait;
        wait := wait + 1;
      end;
    if signal = 1 then
      begin
        ← wait := 0;
        vehicles := vehicles + 1;
      end;
  until signal = 0;
  writeln('Time-span=', time, 'secs');
  writeln('Vehicle-count=', vehicles);
  writeln('Max-wait=', maxwait, 'secs');
end.

```

(CE5)

(CE6)

```

program survey(input, output);
var
  time, vehicles, wait, maxwait : integer;
begin
  wait := 0; ← maxwait := 0;
  vehicles := 0;
  read(signal);
  repeat
    if signal = 2 then
      begin
        time := time + 1;
        if wait > maxwait
          then maxwait := wait;
        wait := wait + 1;
      end;
    if signal = 1 then
      begin
        vehicles := vehicles + 1;
      end;
  until signal = 0;
  writeln('Time-span=', time, 'secs');
  writeln('Vehicle-count=', vehicles);
  writeln('Max-wait=', maxwait, 'secs');
end.

```

(CE3)

Bubblesort Algorithm

The integer numbers are read into an array, which can hold up to 1000 numbers in all. "maxels" defines the actual no. of values held in the array. The purpose of the algorithm is to sort this array of values into ascending order. Successive pairs of array elements, num[i] and num[j], are compared.

For example, comparing num[1] with num[2], then comparing num[2] with num[3], etc. If num[i] > num[j] then the 2 values are swapped using the intermediate variable "temp" to hold one of the numbers (which would otherwise be lost). The comparison of array elements starts with the lowest element, num[1], and finishes with the highest element, num[maxels]. Each cycle of comparison and value swapping, brings the array closer to being in order. However the comparison/swapping loop cannot terminate until all of the array elements are in ascending order. When order has been achieved, the sorted array is written out, 6 "spaces" per value, 12 values per line.

NB. the ordering has to be checked on each cycle, even if only 2 values were swapped on the previous cycle, until no values need to be swapped.

variables

maxels = no. of array elements to be sorted.

i = initial index used to read the values into the array,

j, j are used as indices for comparing successive pairs of array elements;

eg. if i is 1 then j is 2, in general $j = i + 1$

temp holds one of the values to be swapped.

inorder = boolean variable indicating whether all elements of the array are in ascending order or not

** there are 7 errors associated with the i, j, temp and inorder variables

Bubble Sort									
Error	Start		Finish						
No.	hh	mm	ss	hh	mm	ss	hh	mm	ss
1	:	:	:	:	:	:	:	:	:
2	:	:	:	:	:	:	:	:	:
3	:	:	:	:	:	:	:	:	:
4	:	:	:	:	:	:	:	:	:
5	:	:	:	:	:	:	:	:	:
6	:	:	:	:	:	:	:	:	:
7	:	:	:	:	:	:	:	:	:

	1		2		3		4		5		6		7		
	hh	mm	ss	hh	mm	ss	hh	mm	ss	hh	mm	ss	hh	mm	ss
Start	:	:	:	:	:	:	:	:	:	:	:	:	:	:	:
Finish	:	:	:	:	:	:	:	:	:	:	:	:	:	:	:

```

program bubblesort(input, output);
const
  max = 1000;
var
  num : array[1..max] of integer;
  maxels, i, j, temp : 0..max;
  inorder : boolean;

begin
  i := 0;
  while (i <= max) and not eof do
    begin
      i := i + 1;
      readln(num[i]);
    end;
  maxels := i - 1; { last "num" is eof }
  inorder := false;
  if maxels > 1 then
    while not inorder do
      begin
        for i := 1 to (maxels - 1) do
          begin
            j := i + 1;
            if num[i] > num[j] then
              begin
                inorder := false;
                num[j] := num[i];
                num[i] := temp;
                temp := num[j];
              end;
            end;
          end;
        maxels := maxels - 1;
      end;
    end;
  for i := 1 to max do
    begin
      write(i: 6);
      if (i mod 12) = 0 then writeln;
    end;
  end.

```

```

program bubblesort(input, output);
const
    max = 1000;
var
    num : array[1..max] of integer;
    maxels, i, j, temp : 0..max;
    inorder : boolean;

begin
    i := 0;
    while (i <= max) and not eof do
        begin
            i := i + 1;
            readln(num[i]);
        end;
    maxels := i - 1; { last "num" is eof }
    inorder := false;
    if maxels > 1 then
        while not inorder do
            begin
                for i := 1 to (maxels - 1) do
                    (CE3) begin
                        j := i + 1;
                        (CE4) if num[i] > num[j] then
                            begin
                                (CE5)
                                inorder := false;
                                num[j] := num[i];
                                num[i] := temp;
                                temp := num[j];
                            end;
                    end;
                end;
            end;
        for i := 1 to max do
            begin
                write(i : 6);
                if (i mod 12) = 0 then writeln;
            end;
        end.

```

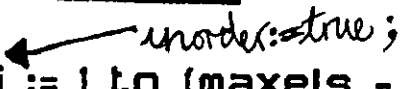
```

program bubblesort(input, output);
const
    max = 1000;
var
    num : array[1..max] of integer;
    maxels, i, j, temp : 0..max;
    inorder : boolean;
    temp : integer; (CEI)
begin
    i := 0;
    while (i <= max) and not eof do
        begin
            i := i + 1;
            readln(num[i]);
        end;
    maxels := i - 1; { last "num" is eof }
    inorder := false;
    if maxels > 1 then
        while not inorder do
            begin
                for i := 1 to (maxels - 1) do
                    begin
                        j := j + 1;
                        if num[i] < num[j] then
                            begin
                                inorder := false;
                                num[j] := num[i];
                                num[i] := temp;
                                temp := num[j]; (CES)
                            end;
                        end;
                    end;
                for i := 1 to max do
                    begin
                        write(i : 6);
                        if (i mod 12) = 0 then writeln;
                    end;
                end.

```

```

program bubblesort(input, output);
const
  max = 1000;
var
  num : array[1..max] of integer;
  maxels, i, j, temp : 0..max;
  inorder : boolean;

begin
  i := 0;
  while (i <= max) and not eof do
    begin
      i := i + 1;
      readln(num[i]);
    end;
  maxels := i - 1; { last "num" is eof }
  inorder := false;
  if maxels > 1 then
    while not inorder do
      begin
         inorder:=true; CE2
        for i := 1 to (maxels - 1) do
          begin
            j := j + 1;
            if num[i] < num[j] then
              begin
                inorder := false;
                num[j] := num[i];
                num[i] := temp;
                temp := num[j];
              end;
            end;
          end;
        end;
      end;
    end;
  for i := 1 to max do
    begin
      write(i : 6);
      if (i mod 12) = 0 then writeln;
    end;
  end.

```


Debugging Tasks Comment Sheet & Difficulty Rating For Each Task

Survey Problem

Ease of understanding task requirements easy ☐☐☐☐☐ difficult

Ease of understanding code itself easy ☐☐☐☐☐ difficult

Ease of understanding how the coded algorithm(s) fulfills the task easy ☐☐☐☐☐ difficult

Comments :

Survey

SS 1345678	Score	1	2	3	4	5	Sum	Mean
TR 1111211	TR 6	1	0	0	0	0	8	1.14
UC 2112311	UC 4	2	1	0	0	0	15	1.57
UA 2112311	UA 4	2	1	0	0	0	15	1.57

Bubble Sort

Ease of understanding task requirements easy ☐☐☐☐☐ difficult

Ease of understanding code itself easy ☐☐☐☐☐ difficult

Ease of understanding how the coded algorithm(s) fulfills the task easy ☐☐☐☐☐ difficult

Comments :

Bubble Sort

SS 1345678	Score	1	2	3	4	5	Sum	Mean
TR 1122121	TR 4	3	0	0	0	0	10	1.43
UC 3223352	UC 0	3	3	0	1		20	2.86
UA 3222332	UA 0	4	3	0	0		17	2.43

Letter Count

Ease of understanding task requirements easy ☐☐☐☐☐ difficult

Ease of understanding code itself easy ☐☐☐☐☐ difficult

Ease of understanding how the coded algorithm(s) fulfills the task easy ☐☐☐☐☐ difficult

Comments :

Letter Count

SS 1345678	Score	1	2	3	4	5	Sum	Mean
TR 1112221	TR 4	3	0	0	0	0	10	1.43
UC 2113321	UC 3	2	2	0	0		13	1.86
UA 2112322	UA 2	4	1	0	0		13	1.86

Shell Sort

Ease of understanding task requirements easy ☐☐☐☐☐ difficult

Ease of understanding code itself easy ☐☐☐☐☐ difficult

Ease of understanding how the coded algorithm(s) fulfills the task easy ☐☐☐☐☐ difficult

Comments :

Shell Sort

SS 1345678	Score	1	2	3	4	5	Sum	Mean
TR 2232342	TR 0	4	2	1	0		18	2.57
UC 4333352	TR 0	4	2	1	0		18	3.29
UA 3323351	TR 0	4	2	1	0		18	2.86

