Quality Aspects of the Program Development Process used by Learner Programmers.

A thesis submitted in partial fulfilment

of the requirements of the

University of Abertay Dundee

For the degree of Doctor of Philosophy

Geoffrey Robert Lund

University of Abertay Dundee

31st January 2002

I certify that this thesis is a true and accurate version of the thesis approved by the examiners.

Signed ..[redacted] Date ..14-3-'02.

(Director of Studies)

Abstract.

Much research in the area of computer programming education has examined the product (program) produced by the novice, measured it and sought ways to improve it. Little regard has been given to the process by which the novice has produced the product. This is in sharp contract to the main teaching in software engineering that stresses the importance of process rather than the product. This thesis initially developed and validated a set of metrics that allowed the measurement of the personal software development process (PSDP). These metrics allow comparison between different personal software development processes. In this thesis an experiment is reported where a group of novices were given feedback during the development of the program that sought to improve the PSDP. The results showed a significant improvement in the PSDP is achieved against a control group. Investigation into the relationship between the process and the product indicates that there is no correlation between the process metrics and the product metrics save for the measurement of correctness; a program developed well tends to be more correct than one that is not. Other product quality measures are unaffected by the quality of the process. This replicates results recorded in the literature. The thesis concludes by proposing a unified framework of programming knowledge that includes 4 levels of knowledge (syntactic, semantic, schematic and strategic) each with two levels (declarative and procedural). The work in this thesis is used to justify the inclusion of strategic knowledge in the framework. This work has implications for deliverers of computer programming education be they lecturers or providers of computer aided learning packages in providing a framework for the learning of novice programmers and especially emphasising the importance of the personal software development process.

Acknowledgements.

I would like to thank all the people who have helped me in their many and various ways throughout the time it has taken me to complete this project and especially my supervisors who spent a lot of time in meetings and outside discussing various issues with me, Dr. Leona Elder, Dr. Louis Natanson, and my Director of Studies Dr. Colin Miller.

Thanks are also due to many colleagues, past and present, at the University of Abertay Dundee who have helped in various ways during the course of this research. I would like to thank my long-suffering room mate who, I hope, I did not drive to early retirement; Dr. Andy Wakelin. Special thanks to Mr Sewart Gardener who helped with the technicalities of VMS and did not switch off the machine before I backed up my files!

Finally thanks to all my family who I ignored on many an occasion when I was working on this thesis.

Geoffrey Lund

January 2002

Contents.

Appendix B The Data

Subject Cohort A – Weather Problem Results
Subject Cohort B – River Problem Results
Subject Cohort B – Tennis Balls Problem Results
Subject Cohort C – Traffic Problem Results
Subject Cohort C – Type Wear Problem Results

Appendix C Papers by the Author

Lund, G.R. 1995. The Program Development Process. *In Innovations in Computing Teaching.* SEDA Paper 88.

Lund, G.R. 1995 Controlling Plagarism in Student programs IN: *Proceedings of 3rd. Annual Conference on the Teaching of Computing.* 29th August – 1st September, Dublin, Ireland.

Lund, G.R., Elder, L., Natanson,L.D., and Miller, C.J., 1997 The Importance of Process In: *Proceedings of 5th. Annual Conference on the Teaching of Computing.* 26th - 29th August Dublin, Ireland.

Appendix D Programs and Scripts.

`PC` VMS script to capture the program versions.
`CntLn2.cpp` C++ program to analyse a single program.
`PRIV` VMS script to control the analysis of a whole PSDP

List of Figures and Tables.

Figures:

Tables:

1. Introduction.

1.1 Introduction.

"It may seem odd in an age of home computers, when so many academics have their own workstation, when most computer users do a bit of programming without any training, that we find it difficult to teach programming to computer scientists"
(Bornat 1990.)

Although the quotation above is over 10 years old the sentiments still hold true. There has been much research work being carried out over the past twenty or so years, (for example McAlpin 1992) with the aim of improving computer programming education. Much of this work is targeted towards different approaches to programming, the creation of automatic marking schemes or using IT during the delivery of a course. The work reported here takes a novel approach. This approach borrows from software engineering the idea of looking at the development process as a way to improve the product.

Software engineering was seen as the solution to the software crisis of the late 1970's. One of the main lessons learnt from the software engineering discipline is that the process by which the software is developed must be managed in order to create a quality product. In order to be able to manage a process efficiently the process must be capable of being measured. Possible measurements of the software development process include the time taken for the development, the cost of development or the number of lines of code produced per week during the

development.

This research sought to measure and manage the way novice computer programmers produce computer programs. The aim of such measurement was to seek improvements in the quality of the programs novices produce. The work started by validating metrics for the novice software development process. It explored the relationship between the novice software development process and the final software product and then moved on to consider if the novice software development process may be improved during the development process. The work carried out led to a greater understanding of what knowledge novices need to become expert programmers. This experimental work gave the author a deeper understanding of the novice programmer. This led to a discussion of the framework of knowledge needed to be acquired by a novice to become an expert programmer.

This work is important as it adds to the understanding of the knowledge needed for novice programmers to become experts. It provides an educational framework upon which traditional lecture courses may be built. However there are an increasing number of on-line learning / training packages available that purport to teach computer programming. There is less flexibility in an on-line computer based learning package for the delivery of the material to adapt to the needs of the students. Therefore it is more important in these on-line computer based learning packages that there is a sound educational foundation to the package writer's understanding of expertise needed to program a computer. A thorough understanding of the knowledge needed for a novice programmer to become an expert is fundamental to computer programming educators when building their courses.

This research looked at a number of aspects of the way the novice programmers develop their programs. In particular it focussed on:

- illustrating the personal software development process employed by novices,
- measuring this personal software development process,
- comparing the quality of the personal software development process with the quality of the final computer program developed,
- investigating how to improve the personal software development process of these novice programmers,
- relating work on the personal software development process to current understanding of what constitutes expertise in computer programming.

There are other significant research projects working in this area. Ceilidh (now Coursemaster) Benfield et.al. (1993b) is a major development project that started out attempting to automatically mark student programs. This has now developed into a major software system that can manage student coursework submission, automatically marking and checking for plagiarism in a number of computer languages. Coursemaster concentrates on analysing and managing the final product and does not consider the way in which the novice got to this program. Indeed anecdotal evidence suggests that some students focus on maximising their mark rather than developing a "good" programming method. The automatic marking system provided is widely used and based on very early work by Rees (1982). More sophisticated automatic marking schemes based on program structure rather than counts have recently been developed, for example Thorburn & Rowe (1999).

"The Programmer's Apprentice" Rich & Walters (1988) is another major research project that looked at capturing the expertise of a programmer. This project focused on trying to use artificial intelligence methods to capture programming expertise. Here again the focus was on the expertise encompassed in the product (the program) rather than on any expertise in producing this product. This project has not been reported in the literature in recent years.

Current understanding of the framework of programming expertise is based on the work of Solway and Ehrlick (1984) and further developed by Davies (1989, 1990a, 1990b, 1991a, 1991b, 1994). Current understanding of what constitutes programming expertise is encompassed in the framework models of McGill & Volet (1997) and Mayer (1997). These recent works make some reference to the strategic knowledge shown by expert programmers. The strategic knowledge of an expert programmer is his/her expertise in the personal software development process. This aspect of expertise has not been studied by others and does not feature strongly in either of the two frameworks for programming expertise.

A number of small studies have been reported that have looked at the software development process. These include Bishop-Clarke (1992), Grove (1999) and Parrish et.al. (1997). Bishop-Clarke used protocol analysis looking at the process eight subjects used to develop a small program. This method is very time consuming for the researcher and the method of protocol analysis cannot be scaled up to tens of subjects writing larger programs. However, Bishop-Clarke identified common features of the software development employed by these subjects and some limitations in the subject's understanding of how to develop a program. Grove

attempted to improve computer programming in students by basing the software development on the personal software process of Humphrey (1995). He claimed that focussing on the personal software process improved the programming expertise of his subjects. Grove was unable to measure objectively adherence to the personal software process model and thus could not state whether or not his methods produced real improvements in the subject's programming ability. Parrish et.al. looked at the development of a program over time. They used the compiler to capture a copy of every program submitted for compilation. They attempted to measure the quality of the software development by measuring the elapsed time from start to finish of the development, the number of compiling attempts and the total amount of time spent on the computer during development. They could not correlate these usage patterns with the final product grade. However they report a first attempt to measure the software development process, a foundation upon which this research is based.

In recent years there has been a growth in online training of computer and IT skills including computer programming skills. To be most effective these packages must address all aspects of programming skills. The underlying framework of programming expertise must be understood to enable computer packages and other forms of training to be designed correctly. A number of the current packages are very sophisticated in their operation but they tend to ignore the major aspect of computer programming, the software development process.

1.2 Project Aims.

Overall Aim: To explore the personal development process employed by novice programmers in order to seek to improve the quality of computer programs.

The aim of this thesis is to investigate how novices develop computer programs. The sequence of tasks a programmer tackles in the development of a computer program is called the personal software development process (PSDP). The main hypothesis of this thesis is that the quality of the PSDP is reflected in the quality of the final product and that learning to follow a quality PSDP (learning how to program) is an important aspect of learning to program. In pursuit of this aim a number of research objectives have been established.

Research Objectives:

1. To visualise the personal software development process employed by novice and expert programmers
2. To establish a set of metrics that measure the quality of the personal software development process employed by novice programmers.
3. To analyse the correlation between the personal software development process and the program product.
4. To evaluate improvements in the personal software development process and software product gained from using simple feedback mechanisms.
5. To synthesise current frameworks of programming expertise into a unified framework that includes a strategic knowledge factor, that is expertise in the personal software development process.

In pursuit of the aim of this thesis the first step was to investigate the personal software development process employed by novice programmers. Various methods were used to help visualise this process. However visualisation only provides an subjective view of the process. To allow further scientific study there is a need to be able to measure the process. Thus the next step was to validate a set of metrics that allow the personal software development process to be measured.

The relationship between the personal software development process and the software product was explored. It seemed reasonable that a good quality personal software development process and a high quality product would correlate and this was investigated. There is potential for any correlation found to be exploited by seeking to improve the personal software development process and consequently improving the final software product.

The work carried out in pursuit of these objectives has provided information on the personal software development process and the programs written by novice programmers. The final objective of this research was to relate this information to current understanding of how novices learn to program.

1.3 Structure of the Thesis.

The thesis continues in chapter 2 with a review of related areas of research. The area of research covered crosses the boundary between software engineering and psychology of programming and of necessity therefore the literature review covers a

wide spectrum. The first sections deal with software engineering issues of quality and metrics. This leads onto the specific topic of how metrics have been used to measure novice computer programs. The final sections deal with the literature concerning the acquisition of programming skills.

Chapter 3 discusses the technicalities of the methods of data collection. A large amount of data was collected from each subject during the experimental phase of this research. The methods of collecting this data and the first stage of the analysis, reducing the data to a summary file, are described in this chapter.

The purpose of chapter 4 is to illustrate and measure the personal software process employed by novice programmers and to compare this to the process used by experts. The personal software development process is illustrated for a number of subjects. This leads to a model of an expert personal software process. Various process metrics that measure adherence to the model are presented and validated.

Chapter 4 uses the metrics established in the previous chapter to examine whether the personal software development process may be improved using simple feedback. An experiment is described that sought to deliver some simple feedback to a group of novice programmers. The experiment used to test this hypothesis is fully described and the results discussed. A discussion of further feedback mechanisms is included here.

Chapter 6 examines the relationship between the program and the personal software process. The criteria and metrics used to measure quality in novice programs are

discussed and established. The correlation between the program quality and personal software process quality is calculated and discussed and the findings are related to other work reported in the literature.

Finally chapter 7 discusses a framework for expertise in computer programming. This chapter brings together the work from the literature and the experimental results to put forward a new general framework of expertise in computer programming. The chapter argues for the validity of this general model using both work reported in the literature and the experimental results.

A summary of the thesis is given in chapter 8. Each research objective as stated above is evaluated against the research results and a discussion of future work to be carried out in this area is given.

2. Review of Related Work.

2.1 Software Metrics

Software metrics came to prominence in the software crisis of the 1970's. The perceived problem at that time was that software was not built either as specified or reliably as required by clients (Brooks 1987). Software engineering planned to introduce an engineering ethos into software development with the aim of improving the quality of software by controlling the development process.

The underlying principle of quality control in any production engineering environment, of which software production is one, is that a high quality product will result from a high quality production process. The focus is directed towards the process of production rather than the product itself. This approach is embodied in standards such as ISO9000 and ISO9000-3. This quality methodology, where the process is as important as the product was applied to computer programming education in Lund (1994) and was fundamental to this research.

The basis of quality is to seek to control the process and therefore the product. However *"you can't control what you can't measure"* (De Marco, 1982 re-statement of an earlier quotation from Lord Kelvin). The measurement of software is an important issue. The research reported here is founded on the ability to measure the software produced by novice programmers.

Software metrics are generally split into two categories, those that measure some

aspect of the product and those that measure some aspect of the process. Early metric work focussed on the product. Halstead (1972) proposed a number of measurements that he collectively called Software Science based on counts of various factors within the program text. This metric is relatively easy to calculate and hence gained popularity despite no evidence that the measurements did measure program complexity. McCabe (1976) put forward a measure of complexity, cyclomatic complexity, based on the loops and branches in a program. The aim of both these authors was to find some measurement of the maintainability and reliability of software. Since these two attributes do not lend themselves to measurement directly, complexity was seen as a convenient proxy for them. McCabe's and Halstead's metrics both gained popularity. However little or no validation of the work had been carried out up to the late 1980's according to Shepperd (1988). Later attempts to validate the metric and relate it to actual program maintainability have not been successful, (Shepperd and Ince 1994).

According to Fenton (1992) there has been a *"shift away from traditional work on software measurement which concentrates on proposing specific metrics without any real thought for what these were supposed to be measuring"*. There is a need for a proper system of metric validation but validation cannot be carried out without a clear understanding of what the metric is to be used for. The goal-question-metric approach of Rambach and Basili (1987), attempts to tackle this problem of metric validation. Their method initially focuses on the purpose of the measurement. For example, in this work, the purpose of the process metrics is to improve the software process employed by novice programmers. The overall goal is refined by a series of stages until a set of metrics emerges that support the goal. The metrics are then

verified using experimental results. These experimental results are used to back up the theoretical work rather than taken as validation in themselves. Shepperd (1992) claims that this more scientific approach to metric development *is "beginning to pay dividends".*

The approach due to Rambach and Basili (1987) is goal driven. Shepperd argues that in addition some model upon which to base the metrics is needed. Ejiogu (1993) takes up this approach. He proposed five principles that are fundamental to the formal validation of metrics. These rely on a model being available. This model must accurately show the specific attributes of software that are of interest and, be useful as the basis of metrics. Ejiogu argues, as does Shepperd, that validation must be done from first principles, relating metrics to a model and hence the software, as opposed to validating one metric against another thus perpetuating validation errors of the past.

The five principles given by Ejiogu (1993) are:

- The target attribute of software behaviour must be clearly defined. This essentially sets the "goal" for the metrics, (as Rambach and Basili 1987)
- The measure under consideration must have the properties of a "mathematical metric". A "mathematical metric" is an abstraction of a ruler i.e. the metric can be related to some scale which allows comparison.
- There must be practical evidence that the metrics when applied to software measure the intended features of the model.
- The results from any measurements must be able to be related to the target attribute of software behaviour.

- The factors making up the metric are clearly defined.

The ideas of goal based software metric formulation tied to a model give a scientific basis on which to establish software metrics. In chapter 4 work to establish metrics that measure the personal software process employed by novice programmers is discussed. This work is goal based, to improve the personal software process employed by novice programmers, and also dependent on an incremental model of software development also described in chapter 3.

2.2 Measuring the Novice Software Product

Ever since computer programming has been taught as an academic subject computer programs have been measured or marked. Often marks are based on the correctness of the program supplemented by a subjective grade for program style or layout. These marks take a long time to generate manually. There has therefore been much work reported on automatic marking schemes (for example Rees 1982, McAlpin 1992, Benford et.al. 1993, Hung et.al. 1993,) with a view to creating an objective marking scheme. However prior to discussing which objective measures are used to mark programs the criteria by which marking is carried out need to be examined.

2.2.1 Measurement Criteria.

What are the criteria used to evaluate computer programs? Howatt (1994) and Benford et.al. (1993) have both put forward reasonable criteria by which a novice program could be measured. The criteria presented may be split into dynamic

criteria, those issues that affect a program during execution, and static criteria that relate to the quality of the code itself.

The static criteria put forward are concerned with the quality of the code itself. The criteria put forward by both authors are:

1. The layout or style of the program. This is an important criterion that relates to the readability of the program code. This is a very subjective criterion and a great deal of work (see section 2.2.2) has gone into creating an objective measurement for this criterion.
2. The design or structural quality of the program. This criterion is arguably even more subjective than program style. Again much work is reported on attempting to measure this objectively.
3. Complexity of program code. This can be measured using one of the complexity metrics discussed in section 2.1, not withstanding the criticism of these metrics reported earlier. There is a belief that complexity is inversely associated with maintainability.

The dynamic criteria cover aspects of correctness and efficiency.

1. The correctness of a program. This can be evaluated by running the program against a test plan.
2. Efficiency is concerned with execution time and the memory requirements of the code. This criterion has diminished in importance with the increase in processing power and the low cost of computer memory.

Table 2.1 below describes in brief the criteria used by Ceilidh to assess the quality of

a novice programmer's program. It is similar to that put forward by Howatt (1994) and will be used as the basis for the later discussion of quality factors. It was also used as the basis for the measurement of novice programs described later in this research.

Area	Criteria	Description
Static	Program style	Measures how well the program is structured. It includes the use of comments
	Program Design	Measures how well the programming language has been used in the program
	Complexity	Measures the complexity of the code
Dynamic	Correctness	Measures how much of the requirements specification has been achieved correctly.
	Efficiency	Measures the program speed and space requirement.

Table 2.1 Criteria for measuring the quality of a novice program.

2.2.2 Measuring the Programming Style of Computer Programs.

Programming style is taken to include features such as layout, use of comments, suitability of variable names and general programming readability. These factors are difficult to measure. Early attempts by Rees (1982), Meekings (1983), Rosenthal (1983), and Lovegrove and Rees (1984) base their systems on a sequence of counts. A number of features of the program text are counted including number of lines of code in the program text, number of reserved words used, number of lines of comment in the program, number of blank lines, number and types of loop, number of selection statements. The work was not based on the criteria outlined in table 2.1 but rather the idea that a novice program must match some ideal or expert solution. The reason for picking one count over another was its accessibility rather than being derived from any background model. Indeed this is one criticism Hannemyr (1983) made of this early work.

In later work Redish and Smyth (1986, 1987) put forward their own automatic marking scheme based again on counts. However more importantly they attempted to evaluate their and other automatic marking schemes by proposing 8 criteria which the quality measurements could be judged. These criteria are:

- Programs must be evaluated against a specification.
- A model program must be available against which a student program is evaluated.
- The factors chosen should represent a wide range of program properties.
- Factors of correctness and efficiency must be included.
- The measure of program quality must be a metric in the formal sense i.e. satisfy the formal mathematical definition of a distance function
- The measure used has a certain sensitivity.
- Programs that are *"similar"* generate the same mark.
- For a given problem there is a fixed range of marks given for a "correct" solution.

This was the first attempt to provide a theoretical background to the work of establishing a valid automatic marking scheme.

The most widespread automatic assessment scheme used within the UK today is Ceilidh (now CourseMaster), Benford et.al. (1993a, 1993b, 1994). The metrics used in that software are derived from the early work of Rees(1982), but there is some underlying model upon which to base the counts. The program style measures used in Ceilidh have been validated as much by the extensive use of the software as by the original experimental data. The criteria used to measure program style in Ceilidh and used in this research are:

- % indented lines
- % blank lines
- average number of characters per line
- average number of spaces per line
- average identifier length
- % identifier with good length
- % comment lines

Other attempts to use counts to evaluate the program style have been reported. These include SPROUT, (Rimmer, Pardoe and Vickers 1995), and ICCASAS, (McAlpin et.al. 1995).

Oman and Cooke (1989), Lake and Cooke (1990) use a more sophisticated method of analysing program style. Their style analyser is built into a compiler. Style errors are noted as well as compiling errors. However they have not made any attempt to use this information to measure the style of a novice program. Schorch (1995) also reports a similar style analysis method. It is interesting to note that these two authors do not attempt to measure the quality of the program style but provide feedback to the programmer when poor style is noted.

Whilst there may be arguments about which counts to use to measure the programming style, and the original counts put forward by Rees(1982) have been modified since his early work, the way these counts are used has not altered. The overall aim of all the authors who proposed counts was to produce a single mark that measures the program style. This mark is obtained by transforming each count to an

individual mark and then combining the individual marks into a single mark using some weighting. Hence:

$$ps = \Sigma\ f(c_i) * w_i$$

where

ps = mark for Program Style
C_i = count for factor i
f = function that transforms counts to a mark
w_i = relative weighting of the counts i ($\Sigma w_i = 1$)

The function that transforms the raw count to mark is a trapezium function. Figure 2.1 below shows a typical trapezium function used to convert the counts to marks.

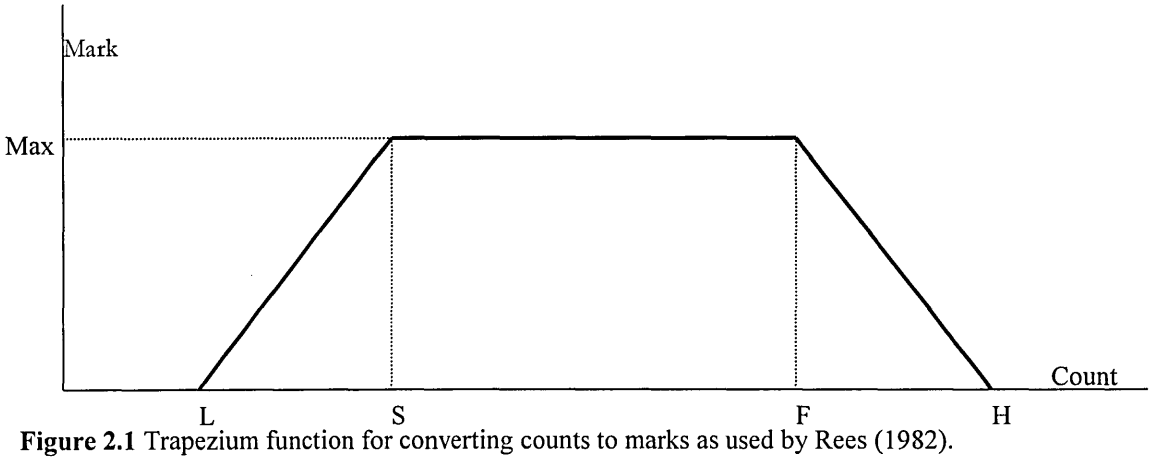

Figure 2.1 Trapezium function for converting counts to marks as used by Rees (1982).

The four parameters, L, S, F and H dictate the marks given for a particular count. The parameters S and F give the range that would earn the maximum marks. These values are obtained from the model answer that is assumed to exist in all cases. A count of less than L and more than H would obtain zero marks. A count value between L and S or between F and H would gain a mark between 0 and Max based on the linear interpolation between these two values. The choice of the parameters L, S, F, and H are problem dependent and can be used to tune the marking scheme. This method was used in this research to convert program style counts to marks and ultimately, an individual value for the program style of a novice program.

2.2.3 Measuring the Programming Design of Computer Programs.

The second criterion used to measure the static quality of computer programs is that of program design. This measures how well the program language has been used in the program. A better way to look at this is to see how well the program code meets established programming practice. The basis of good programming is given in various programming books most notably Kernighan and Plauger (1974). There are a number of language dependent utilities that provide the programmer with feedback concerning the correct or incorrect use of the language. One such facility is the `lint` utility in Unix. A possible measure of the quality in program design can be related to the number of "error" messages generated by `lint`. This is indeed the method used in Ceilidh by Benford et.al. (1993a). However this limits the metric to C programs and a different "`lint`" tool would have to be written for different programming languages. The style analysers developed by Oman and Cook (1989) and CAP developed by (Schorch 1995) give comments on the program design as well as the style. By filtering the design comments from the style comments these utilities could also be used to count the design errors in a program and hence provide a program design mark for that program.

Another way to look at program design is again to compare the code with a model solution. Again counts of various factors can be taken, compared with the expert solution and converted via the trapezium function to an objective mark. Possible factors include:

- Sub program (function or procedure) length.
- Number and type of sub program constructs.
- Number and type of loop constructs.

- Number and type of selection statements.
- Number of identifiers

These are identified from the work of Lovegrove and Rees (1985) and Redish and Smyth (1986) as measuring the program design. These features are used in more recent automatic assessment schemes such as ICCASAS, McAlpin et.al. (1995). The criteria listed above were used as the basis for the program design metric in the work described in chapter 5.

Thorburn and Rowe (1997) have developed a more sophisticated method. They again compare the novice programmer's work to an expert solution. This comparison is not based on a simple series of counts but on the solution plan used to solve the program. This has the advantage that it attempts to capture the whole structure of the program rather than picking a few features to concentrate upon. It is of necessity more complex than methods based on counts.

2.2.4 Program Complexity.

The third criterion, complexity, is often given less importance in marking programs. It can easily be calculated through Halstead's (1977) software science, McCabe's (1976) complexity metric or Henry and Kafura's (1984) information flow complexity metrics. There is concern whether any of these complexity measurements actually captures the complexity of a computer program, (Shepperd and Ince 1994). However McCabe's widely used complexity measure is often used as the measurement for complexity in marking schemes, eg Ceilidh.

2.2.5 Measurement of Correctness of Computer Programs.

Correctness can be measured by running the program through a set of test data and comparing the results to the expected results. The proportion of correct results will give a sensible measure of program correctness. This is done in Jackson(1991) and Ceilidh (Benford et. al. 1993). The problem with this method is that it is difficult to detect automatically whether the output of a program is correct unless a full and complete specification of the output required is given. Any deviation for this output will result in the test failing when in actual fact the answer was correct and only the formatting was wrong. It seems harsh to base such weight on data formatting. To get over such problems various authors have reported methods to parse the output from novice programs, for example Jackson (1991), Ceilidh (Benford et.al 1993), and CAPE (Edmunds 1990). This parsing of output is carried out in an attempt to neutralise the effect of different formats of essentially correct solutions.

Another way to neutralise the effect of formatting is to use a pre-written output sub-program that the novice programmers use for the output the data. Its advantage over parsing the output is that the pre-written routine has direct access to the calculated variables whereas parsing can be confused by some output formats and certain test cases. This method is only appropriate in certain situations but using pre-written output sub-programs were used in this research and are described more fully in chapter 3.

Whilst the counting of tests passed is a widespread way to measure the correctness of a program a different system is proposed by Conway (1978). He proposes an eight-point scale starting at point 1 *"a syntactically correct program"* running to point 8

where a program gives the *"correct answer for all possible input"*. This gives a measure of correctness between 0 and 8. Within each level some test data is still needed and some parsing of output or use of an output sub-program is needed to establish the correctness at each level.

The correctness metric used in this study is fully described in chapter 4. It combines the idea of counting correct cases with the idea of levels of correctness from Conway.

2.2.6 Efficiency Measurement of Novice Programs.

Efficiency is not regularly used as an issue in program measurement. Current computers are much faster than the early machines. Hence efficiency is less of an issue in the quality of a program than maintainability, for example. However really inefficient programs should be penalised. Time and memory usage can be used as measurements of efficiency which values are then compared to the expert solution and converted into marks. Efficiency has not been used as a factor in the quality measurements of programs in this study.

2.2.7 Automated Marking Schemes.

A number of automatic marking schemes have been reported in the literature.

The early work of Rees (1982), Meekings (1983) and Lovegrove and Rees (1984) presents a number of factors that are used to measure the quality of student programs. These early attempts tended to pick aspects of the program that were amenable to measurement rather than being derived from any underlying model. There was little

in the way of validation reported and even Rees in his paper shows that some of the factors used were of questionable merit. The advantages of such schemes are that they are easy to apply; the disadvantage being there is little evidence to show that they actually measure what is intended.

Whereas the early work concentrated on static aspects of programs the work of Jackson (1991) looked at the correctness issues and provided the ground work for the parsing of output to check its correctness.

McAlpin et.al. (1995) report a system of automatically assessing the style of Modula-2 programs. The theoretical basis of their automatic marking is derived from the early work. They claim that the automatically generated marking scheme is *"useful in the teaching, learning and assessment of programming".* There is some evidence given that the marking scheme correlates with manually derived marking.

The Ceilidh system is the most widely used automatic marking system in UK universities with over 40 universities using the system, (Benford et.al., 1993a, 1993b, 1994, Foxley et.al., 1996). It has a clear model by which it assesses a program. The five components of the assessment are style, design, complexity, correctness and efficiency as discussed earlier in this section. The Ceilidh system has been in use over a number of years that coupled with the large number of universities in which the software is used, gives Ceilidh a level of validation not available to any other system. The methods employed in Ceilidh were used in this research. In particular the assessment of a novice program was based on both the Ceilidh criteria and assessment methods.

Pardoe & Vickers (1994) report on a scheme, SPROUT, where novice programs are compared to an expert solution. The system is not as extensive as Ceilidh in that the underlying model of assessment is less complex. Even so, later work reported by Rimmer et al. (1995) claims that a comparison between their automatic marking system and manual marking is *"favourable"*. The most important aspect of this work is that they have made extensive efforts to validate the automatic marking scheme. This signals a start to the more sophisticated approaches to automatic marking.

The schemes reported so far have used simple methods of assessing program code using counts of various factors but it can be argued that this does not capture the assessment criteria well. In an attempt to address this problem, Thorburn & Rowe (1997) focus on the program design. They examine the structure of the code to derive a solution plan and use this plan to create an assessment of the program design. They report an "*acceptable*" accuracy when compared to manual marking. This system gives a radically different view of the program that is nearer to the way human markers would assess the code. The system provides a start to the next generation of automatic marking systems, moving away from counts and moving onto viewing the program in terms of programming plan and deriving the assessment from these.

2.2.8 Summary.

The majority of automatic marking schemes are based on software metric work. It was not the role of this research to establish novel automatic marking schemes. Rather it investigated methods of measuring the software process as opposed to the software product. Therefore where the need arises for program quality measurement

the established methods will be used. A fuller description of the exact methods used in this research is to be found in chapter 5.

2.3 Measuring Novice Processes.

In the previous section the various ways to measure the quality of a program written by a novice program were discussed In this section the focus is changed to measuring the software process employed by novice programmers. Prior to looking at how the software process is measured it is useful to look at the process itself.

2.3.1 Personal Software Process.

To make any sensible measurements of the programming process some understanding of how novices undertake a programming task is needed.

The Capability Maturity Model (CMM) developed by the Software Engineering Institute at Carnegie-Mellon University has been successfully used to improve the software development processes in a number of organisations (Herbsleb et. al., 1997). A cut down version of this model applicable to people rather than organisations forms the basis of two books by Humphrey (Humphrey,1995 and Humphrey, 1997). This version of the model, when applied to people, Humphrey named the personal software process (PSP). There are two aspects to this process. Firstly there is a detailed personal software development process (PSDP) which models the way a programmer develops the finished program from the requirements specification. The second aspect is the supporting documentation that allows the

programmer to manage software development. Figure 2.2 taken from Humphrey (1997) clearly shows the PSP process he uses in his work.

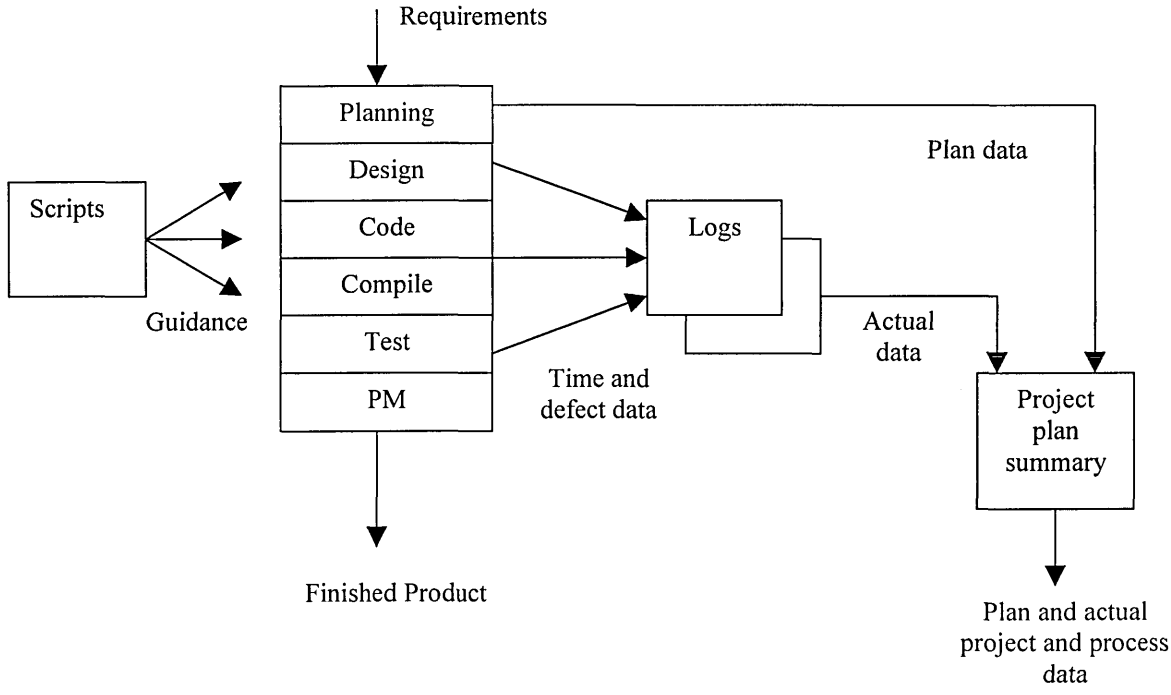

Figure 2.2 PSP Process Flow Taken From Humphrey (1997).

The objective of the research reported in this thesis was to establish metrics for the software process that are employed by novice programmers. To achieve this some underlying model of how a novice develops a program was needed. Figure 2.2 above illustrates the PSP proposed by Humphrey and clearly shows the central role of the software development process model. The PSDP model used by Humphrey is a simple waterfall model, (see Sommerville 1996). This model lacks detail that was needed for this study. In chapter 3 an incremental model of the PSDP used by novices is discussed. This model is used as the underlying model from which the novice software process metrics are developed. The work of Humphrey highlights the need for a model when indulging in this type of research. Another interesting aspect of Humphrey's work is that the metrics that he uses are simple: the lines of code (LOC) and time taken. More complex metrics were used to describe the more

complex PSDP model within this research.

Two authors Grove (1999) and Hou and Tomayko (1999) report on using the work of Humphrey with the aim of improving the programming skills of novices. Whereas the research reported in this thesis focused on the underlying model and metrics to provide feedback and thus improve the process, the above authors concentrate on documenting the PSDP as a way to improve the programming process employed by novice programmers. Novices were asked to supply documentation to support their PSDP. Grove (1999) reports that using documentation helps the students to see the importance of the design and review stages, and to gain a better view of the overall PSDP. Hou and Tomayko (1999) also report that the documentation helps novices in the design stage of development. Hilburn and Towhidnejad (1998) also report using PSP in teaching programming to novices. Their aim is to emphasise quality. They report, as do Grove (1999) and Hou and Tomayko (1999), on the difficulties in keeping correct and timely data. The method relies on novices recording accurately their actions and time spent on these actions. Such a manual system is always going to be open to problems of data collection. In the research reported here data was collected through automatic methods which alleviates the problems these researchers have encountered.

2.3.2 Novices Process Metrics

Whilst much work has been reported on measuring novice programs little is reported on measuring how novices develop their programs.

McGill and Volet (1996) have looked at the development of programs using twelve

subjects. The subjects were observed during the development of computer programs. The observers gave a subjective mark for the way in which the program development was carried out and the authors report a positive correlation between this mark and the final product mark.

Both Grove (1999) and Hou and Tamayko (1999) use lines of code as their process metrics. This metric is objective and can be automatically calculated. Although the authors claim an improvement in the development process caused by the use of the PSP, their metric does not capture this improvement. This indicates either that improvement did not occur or that the metric is not sophisticated enough.

Bishop-Clark (1992), use protocol analysis to determine the strategies used by a novice in developing programs. Although limited in scope (to eight subjects) it does reinforce ideas gleaned from other work. In particular the idea that novices devote little time to planning, (Allwood, 1996), and that they follow an opportunistic strategy, (Webb, Ender and Lewis, 1986).

Parrish et al. (1997) attempt to measure the computer usage patterns and correlate their metric with the quality of program. They use three metrics to measure the PSDP: the time spent on the project, the number of compiles used and the date programming was started. These metrics gave them a better understanding of how students develop their programs and objective measurements of this process. Parrish et al. (1997) calculated the correlation between these process metrics with the final grade marks. This showed no significant correlation. The authors conclude that this is due to the simplicity of their choice in metric and are investigating using different

metrics that capture the process in a better way. The choice of metric is surprising and perhaps reflects the ease of metric calculation rather than any relationship to a model of programming. Thus it is not surprising that there was no correlation. This work does however show that other research teams consider this to be an important issue in novice programming.

2.4 Learning to Program

Expertise in programming like any other skill has to be built up over time. This section addresses two questions relating to programming knowledge namely:

- What constitutes expertise in computer programming?
- How do novices acquire this programming expertise?

2.4.1 Programming Expertise

Programming expertise is not simply a matter of learning the syntax of a programming language. The ability to take a problem, produce a solution method, translate this into a computer program, and to implement the program are all features of the expert programmer's skill. Indeed the skills needed by an expert programmer have been much studied, for example Hoc et al (1990).

Bayman and Mayer (1988) put forward a model of expertise in programming that includes three related types of programming knowledge: syntactic, conceptual and strategic. This model encapsulates earlier work of Shneiderman (1976), Shneiderman and Mayer (1979) and Linn (1985). Shneiderman and Mayer (1979)

proposed a model of programming knowledge that separated syntactic and semantic knowledge. The former being the knowledge set related to the rules of the language; the latter is the knowledge of how to apply these rules. The later work of Linn (1985) added the concepts of design skills and problem solving skills.

Mayer (1997) in his review of programming knowledge brought together the earlier models to produce a four-stage framework for the organisation of programming knowledge. This framework separates syntactic, semantic, schematic and strategic knowledge. Table 2.4 below summarises his framework.

Knowledge	Definition	Common Tests
Syntactic	Language units and rules for combining language units	Recognise whether a line of code is correct
Semantic	Mental model of the major locations, objects, and actions in the system	Rating pairs of terms for relatedness; providing thinking aloud protocol
Schematic	Categories of routines based on function	Recalling program code or keywords; Sorting routines or problems; recognising or naming routines
Strategic	Techniques for devising and monitoring plans	Providing thinking aloud protocols; answering comprehension questions

Table 2.2 Mayer (1997) framework for programming expertise.

A similar but more complex framework is presented in McGill & Violet (1997). They use the 3 categories of knowledge, syntactic, conceptual and strategic, but split the first two categories into declarative knowledge and procedural knowledge. Declarative knowledge is defined as knowledge about something, whereas procedural knowledge refers to the use of this knowledge. This is based on Anderson's (1983) ACT (architecture of cognitive tasks) model of skill acquisition. The first stage is the declarative stage in which factual knowledge is learnt. The second stage is knowledge compilation where domain specific knowledge is

grouped. This leads to procedural learning where these groupings are fine tuned leading to the ability to apply the declarative knowledge. McGill and Volet (1997) use the first two layers of Anderson's model to define syntactic and semantic knowledge, as does Mayer. The third layer of McGill and Volet's model is an amalgamation of the top two layers in Mayer's model. Table 2.5 shows McGill and Volet's model.

	Declarative Knowledge	**Procedural Knowledge**
Syntactic Knowledge	Knowledge of syntactic facts relating to a particular language.	Ability to apply rules when programming.
Conceptual Knowledge	Understanding of and the ability to explain the semantics of the actions that take place as a program executes.	Ability to design solutions to programming problems
Strategic / Conditional Knowledge	Ability to design code, and test a program to solve novel problems.	

Table 2.3 A framework of programming expertise, McGill & Volet 1997.

These two models of the programming knowledge framework are similar and are used as the basis for discussion of programming knowledge in chapter 6.

The next sections deal with the individual categories of programming knowledge following the Mayer (1997) model.

2.4.2 Syntactic programming knowledge.

This category represents the knowledge of the syntax of a particular language and the ability to apply these rules. Wiedenbeck (1985) describes a study which compared the syntactic knowledge of experts and novices reporting that experts were 25% faster and made 40% less syntax errors than novices. She concludes that experts have automated their syntactic skill to concentrate on higher level skills. Schmidt (1986) who studied time taken to read programs by novices and experts, corroborated

the findings of Weidenbeck.

2.4.3 Semantic programming knowledge.

Semantic programming knowledge relates to the understanding a person has of what goes on inside the computer as a result of a line of code. Procedural semantic knowledge allows a novice to apply their understanding to the solution of simple problems. This knowledge is often referred to as the students having *a "mental model of the workings of programs"* (Gentner and Stevens 1983). Research looking at this aspect of programming focuses on the "mental model" of the working of programs. Goodwin and Samuti (1986) conclude that training in these mental models improves novice performance.

2.4.4 Schematic Programming Knowledge.

This third kind of knowledge is the ability to recognise and use functional units of code frequently called "chunks" (Mayer 1979) or "plans" (Gilmore and Green, 1988, Rist 1989, Solway and Erhlict, 1984). A programming plan is a pattern of code used to solve typical problems that can be used in larger programs.

It was Adelson (1984) who noted that experts could recall groups of lines of code as against novices who could recall individual lines. The experts organised the lines into chunks whereas the novices focused on syntactic aspects of the code. This theory is based on experimental evidence and provides some initial evidence concerning programming plans.

Solway and Ehrlich (1984) present a straightforward view of the relationship

between expertise and programming plans. They argue that expertise in programming consists of building up a number of programming plans. This idea is not unique to computer programming and is also used to explain natural language understanding (Carberry and Pope 1993). The programming plans have been used as the theoretical background to a number of automated programming tutors. Predominant amongst these is the Bridge system (Bonar and Cunningham, 1988), which aims to improve the learning process. A number of other automated systems for learning to program based on programming plans have been reported e.g. Johnson (1988), Huff and Lesser (1998), Rowe and Thorburn (2000).

Rist (1991) reports on experiments that examined the strategies that expert and novices use to generate programs. In his conclusion, he supports the view that whilst novices are still creating plans experts are merely recalling these plans.

The evidence recorded here assumes that expertise is proportional to the number of plans a programmer has available to them. Davies (1989, 1990a, 1990b, 1991a, 1991b, 1994) has reported a sequence of work which looks at the programming plans of experts and novices. He gives experimental evidence that intermediate programmers have just as many plans available as experts. He also points out that the experts have learnt not only the plans themselves but also a means to apply these plans in a more organised way. Davis (1994) concludes that this plan knowledge is organised in a hierarchical manner in experts that allow them to make better plan choices than intermediates. This suggests that there is a distinction between declarative knowledge and procedural knowledge as reported by McGill and Volet (1997). This point is discussed in chapter 6 in which further models of programming

knowledge are discussed.

2.4.5 Strategic Knowledge – Problem solving.

The strategic knowledge available to experts is their problem solving ability. Problem solving in general has been widely studied in psychology. Polya (1957) puts forward a number of stages involved in problem solving, these being: understand the problem; devise a plan; carry out the plan; and check the results. The expertise needed to solve programming problems is similar in nature to general problem solving. Chi et al (1988) study problem solving in a wide range of subject areas and show that experts use a detailed problem solving strategy as opposed to the shallower approach taken by novices.

This final aspect of programming knowledge has not been studied in as much depth as the other aspects. A lot of work concerning syntactic, semantic and schematic knowledge have been carried out by looking at the understanding of program code, or debugging code or testing knowledge. These features of programs are easily accessible. To examine strategic knowledge the researcher must observe the programmer during program development. This activity is harder to carry out than an examination of the finished program. Davis and Castell (1992) studied design of novice programs. They highlight problems of analysing the design process due to the difficulty in separating the design activity from its context.

A number of authors suggest that the program development process is different in novices and experts. In a study looking at debugging programs, Spoher and Solway (1986) make the point that there is little in text books about *"how to put the pieces*

together". They conclude that novices *"should be given a whole new explicit vocabulary for learning how to construct programs"*. Adelson et al. (1984) report a study in which expert and novice software designers were studied whilst designing a non-trivial program. They concluded that the novices had less well-developed strategies for using whatever knowledge they had.

2.4.6 Summary

Programming expertise can be classified as being syntactic, semantic, schematic and strategic. Much work has been carried out on the first three classes and has been briefly reported here. To study strategic knowledge researchers need to examine the programmer during development of the program. The data to carry out this research is less accessible then for the first 3 categories. This work reports an investigation into various aspects of the strategic knowledge of expert and novice programmers. It looks at how experts and novices develop programs and how that informs on the strategic aspects of the model of programming expertise.

2.5 General Summary.

This chapter has reviewed the relevant literature that informs on the subject area of this thesis, namely computer programming education. Within each section of the review a number of important conclusions have been made. These will be used later in this report and are summarised below:

- Software metrics are used to measure some aspect of the quality of computer programs. These metrics are split into product metrics and process metrics.

- Software metrics must be based on some underlying model. Metrics are developed from this model with a particular goal in mind. Any metric used must be validated experimentally to ensure that it measures the correct features of the model. This method of metric development is used in chapter 3 of this thesis to develop metrics that measure the process employed by novice programmers.
- There is an established set of criteria against which a program can be measured. These criteria are program style, program design, complexity, correctness and efficiency. These were used as the basis of the product measuring within this research.
- There is reasonable agreement on how to measure the quality of novice computer programs. The methods are largely based on counts. Whilst there is some criticism of these methods they are currently in use and must be considered the best methods available. These established methods were used in this research to measure the quality of the computer program. Details of the actual factors included in the program measurements used here are given in chapter 4.
- Measuring the software development process employed by novice programmers must be carried out against a model of that process. Work in this area is based on a simple waterfall model of the PSDP. A more sophisticated model is employed in chapter 3 of this research.
- Various authors have attempted to carry out experiments to find out more about the PSDP employed by novice programmers. These have been hampered by the difficulty in collecting data and the subjective nature of that data. Attempts at measuring the PSDP have not been reported.
- Programming knowledge can be categorised as syntactic, semantic, schematic, strategic. Various authors have attempted to explain the expertise in computer

programming based largely on the first three of these categories. Strategic programming knowledge is concerned with expertise in the PSDP. The experimental evidence presented in chapters 4 and 5 of this thesis is used to add to the current understanding of programming expertise especially in the area of strategic knowledge.

3. Data Collection Techniques.

3.1 Introduction.

This research examined the personal software development process (PSDP) employed by novice programmers. It was, therefore, essential to have some way to make visible the PSDP without altering the process. There are potentially three methods to collect data about the process employed by a programmer.

The first way to capture data about the PSDP employed by a programmer is to request that the programmer record what they have done. This was the approach taken by Grove (1990) who required novices to keep a logbook of their activities. This has two disadvantages. There is potential for novices to mask the truth by entering what they think is expected and perhaps ignoring problems. This problem increases in importance if the novices think they are being assessed on the log as well as on the final program. A second problem is that the actual act of keeping a record of their activities may provide the novice with a stimulus to consider and hence improve the process rather than merely reporting on the process.

A second method to capture data about the PSDP of the novice is to observe the process and then carry out a protocol analysis of the novice programmer. This was the method used by Bishop-Clark (1992) and McGill & Volet (1996). In both of these studies the novices were observed during their development of a program and questioned afterwards. This method can only be used on a small number of subjects due to the time involved in data capture. Also it is only applicable to an

experimental environment where the subjects are restricted for a period of time rather than allowing subjects to develop programs in the manner they wish to.

Both of the data capture methods described above can be used to collect data but cannot be extended to automatic calculation of process metrics. It is essential that the process metrics can be calculated automatically from the data collected. Future developments of this work may involve metrics being calculated part way through a development in order to provide related feedback to be given to the novice. The process metrics may also be used to generate a "process" mark for the development and this is added to an automated marking scheme. In either case it is important that the method of data capture and metric calculation does not rely on either a student or an instructor entering data but on the automatic methods.

This chapter describes the automatic data capture method used in this research and how the many versions of the program were reduced to a single file of data that can subsequently be used in the calculation of process metrics.

3.2 Collecting the Raw Data.

As noted above there were two major factors affecting the method of data collection, namely the method used must be automatic and non-invasive. Secondly the method must not affect the PSDP employed by the novices. The method used was to intercept and record the source code at every compilation step. Thus the record of the program development is the incremental sequence of source code versions

submitted to the compiler. This method mirrored that used by Parrish et.al. (1997) in their research concerning computer usage patterns in an educational environment.

There is an ethical issue to be addressed with this form of data collection. The novices were all informed that data was being collected during their development of programs. They were assured that the data collected as part of the research would not be used either in their assessment nor would their name be associated with the data in any research reports, papers, etc.

3.2.1 Source Code Capture.

The novices in this study were using a VMS computer and learning to program using the Pascal language. VMS is a command driven operating system and to compile and run a Pascal program a user must enter the following three commands:

```
pascal prog
link prog
run prog
```

A special VMS command was written that combined these three commands into one:

```
pc prog
```

The novices were introduced to this command at the start of their course so it was not something special that they had to do for the purpose of the experiments. When the time came for collecting data the `pc` command was amended to copy the source code to a secure location. This change was invisible to the novices. The pc command followed the algorithm given in figure 3.1

```
pc prog

if parameter missing then exit
if prog.pas does not exist then exit

copy prog.pas to secure location

compile prog.pas
if errors then
        display error file page by page
        exit
else
        link prog.obj and libraries
        run prog.exe
        delete  *.obj files
        purge  *.exe files
        purge  *.lis files
        exit
```

Figure 3.1 Outline of pc command

A full listing of the pc command is given in appendix D.

This command gives the novice programmer extra value over and above that provided by VMS, namely a single command rather than three, page display of errors rather than the default scrolling, and tidying up of file space. This encourages students to use this command rather than looking for the VMS commands.

3.2.2 Capturing Program Output.

Not only was the source capture method described above used to copy the source files to a secure location but also to provided a means of capturing the output from a program. This output was used to deliver feedback to students. As part of the specification of the Weather Station problem, River problem and Traffic problem (problems 1, 2, 3 in Appendix A) the novices were given the interface to a procedure that displayed the data and results. They were requested to use this procedure in

their programs. The procedure details are given in Figure 3.2 below:

```
PROCEDURE Display-Data (
    N:INTEGER ;      {week number}
    data:a_table ;   {raw data table}
    min:a_list ;     {min values of temperature and
                     wind speed each day}
    max:a_list;      {max values of temperature and
                     wind speed each day}
    av:a_day;        {average values of temperature and
                     wind speed each time}
    ovmin:a_rec;     {overall minimum temperature and
                     wind speed}
    ovmax:a_rec;     {overall maximum temperature and
                     wind speed}
    ovav:a_rec;      {overall average temperature and
                     wind speed}
)

where
    a_rec = RECORD
              temp : REAL  ;
              wind : REAL
            END  ;
    a_list = ARRAY[1..7] of a_rec  ;
    a_day = ARRAY[1..6] of a_rec  ;
    a_table = ARRAY[1..7, 1..6] of a a_rec  ;
```

Figure 3.2 Specification of `Display_ Data( .. )` Procedure

The `pc` command linked this supplied procedure into the novice's program. One problem is that standard Pascal does not allow this facility of linking to external procedures, however, VMS Pascal has been extended to allow this facility. If the language used were C then the ability to use external functions is standardised in the language. The major reasons for supplying this procedure was not to make the task of the novice simple but to allow automatic correctness measurements to be made and to facilitate the feedback mechanism. Details of both of these are given in the following sections.

3.2.3 Summary.

The `pc` command outlined here allowed a full history of the development to be stored. In some cases this was more than 100 versions of the program. Whilst this sequence gives a full account of the PSDP it is raw data and not useful for analytical purposes. The next section describes how these sequences of program versions are reduced to numbers suitable for analysis.

3.3 Automating Correctness Checking.

To analyse a program it is essential to have some measure of the correctness of the program. This correctness measure needs to be objective and is here created automatically. Many authors have attempted to assess the correctness of a program by attempting to parse the output of a program in order to isolate the answer and then check it, for example Jackson (1991), Edmunds (1990) and Benford et.al. (1993). This method is difficult to implement in a problem with complex multi-valued output as in this experiment. In the work described here a different approach was taken.

For the purposes of the experiments reported in this thesis correctness is measured as the proportion of the maximum number of tests passed. For the Weather Station problem (problem 1, Appendix A) sixteen stages were recognised. These tests are given in table 3.1.

Test	Description
1	Program Compiles
2	Opens File
3	Reads File OK
4	Display data correctly
5	Calculate average temperature (column)
6	Calculate average wind speed (column)
7	Calculate minimum temperature (row)
8	Calculate minimum wind speed (row)
9	Calculate maximum temperature (row)
10	Calculate maximum wind speed (row)
11	Calculate overall average temperature
12	Calculate overall average wind speed
13	Calculate overall minimum temperature
14	Calculate overall minimum wind speed
15	Calculate overall maximum temperature
16	Calculate overall maximum wind speed

Table 3.1 Tests for weather problem

To generate automatically the test count a program that compiled was not linked with the standard `display_data(..)` procedure but with a special `display_data(..)` procedure. This procedure displayed the data as used by the novice and in addition evaluates the correctness of the program by examining the parameters passed to the procedure and checking these against the correct values.

The advantage of this method of correctness checking is that the correctness checking software has direct access to the variables calculated in the program and is not affected by data formatting problems. There is no problem of parsing and potentially misunderstanding the output of a program. The main disadvantage is that the novice must use the `display_data(..)` procedure in their program. This was not a problem in the experiments recorded here. The novices found using the procedure an advantage to their program development.

In the experiments described in Chapter 5 feedback in the form of number of tests

passed was given to a group of novices to see if this affected their PSDP. The method of supplying feedback to these students was similar to the method described above. The `display_data(..)` procedure was amended to give correctness feedback to the novices as well as displaying the results.

3.4 Analysing the Data.

For each subject, the PSDP used is available for future analysis. There may be a hundred or more versions of a program captured during the development. Whilst this sequence of program versions gives a full account of the PSDP it is raw data and not useful for analytical purposes. The raw data is analysed by reducing each source file version to a set of numbers and thus the development represented by the sequence of these sets of numbers.

Each program is processed and reduced to a sequence of counts. The counts used are detailed in Table 3.2.

Version Numbers	• Sequence number starting at 1 for first program and incrementing by 1 for each subsequent program • Generation number as taken from the filename
Run Characteristics	• Number of compilation errors • Number of tests passed
Lines of Code	• Total number of lines of code • Lines of comments • Blank lines • Lines containing code
Code Characteristics	• Number of procedures • Number of functions • Number of while statements • Number of repeat statements • Number of until statements • Number of for statements • Number of if statements • Number of case statements

Table 3.2 Counts used in the processing of novice programs

The whole sequence of novice programs is processed, once the development is complete, under the control of a VMS macro PRIV, see Appendix D. Each version of the program is processed by the program `Cntln2`, again given in Appendix D. This program produces, for each program, a line of numbers in a `.new` file. The counts are grouped and described in Table 3.2.

- The version numbers are obtained from calling`PRIV` macro and passed to `Cntln2` program through the parameters.

- The number of compilation errors is found by recording the output from the compiler. There is a compilation error limit of 30 set by the compiler so that no program may have more than 30 compilation errors, subsequent errors are

unlikely to be of any significance. Currently, this number is typed into `Cntln2` program by the user, but it would be possible to capture the output from the compiler and pass this to `Cntln2` via another parameter.

- The correctness of the program is gained by linking the compiled code with the version of `display_data(..)` that checks the correctness of the program (see section 3.3). The number of tests passed (maximum 16) is output. This value is also typed into `Cntln2` by the user, but again it would be possible to capture the output from the program, via a file, and submit it directly to `Cntln2`.

- The program `Cntln2` scans the program text to evaluate the line counts. A comment line is defined as a line that only contains a comment. A line of code is any line that contains code. Note that a line containing both code and a comment is counted as a code line. Thus each line contributes to the total number of lines plus one of the other categories, comment line, code line, and blank line.

- The program `Cntln2` also counts the specific Pascal reserved words. This is done simply by comparing each word to the target list. A match results in an increment to the count.

Once the whole sequence of programs has been analysed under the control of the `PRIV` macro a `.new` file is available. It is from this file that the various metrics used later in this thesis are calculated.

In the following chapter a number of metrics are identified as potential metrics for measuring the novice program development process. For each novice these metrics are calculated by the program `metric.cpp` given in Appendix D. This program reads in the `.new` file for a given development process and calculates all the candidate metrics outputting the values. These values, together with the novice code letters, are entered into a spreadsheet. The spreadsheets are given in Appendix B. These spreadsheets have one row for each novice and one column for each potential metric. It is from this data that the later analysis is taken.

3.5 Summary.

This chapter has outlined the technical methods used to capture and analyse data collected in pursuit of the study of the PSDP used by novice programmers. The key issues are:

- Data is captured by copying the program to a secure location prior to every attempt at compiling the code.
- The sequence of programs is reduced to a file of counts to be used in later analysis.
- Each program is reduced to a line of numbers representing lines of code, compile and run characteristics, reserved word counts and sequence numbers so the PSDP history for a program is reduced to a file of counts.
- Correctness of a program if found by linking a special procedure to the program

which evaluates the correctness of the program via the program variables.

- The metrics are calculated from the file representing the PSDP and entered into a spreadsheet that is used in further analysis of the data.

The data captured in this way is used to calculate process metrics as described in the next chapter.

4. Measuring the Novice Program Software Process

4.1 Introduction.

The personal software development process (PSDP) is the sequence of tasks employed by a programmer whilst writing their program. The main hypothesis in this thesis is that the quality of this PSDP is reflected in the quality of the final product and, that learning to follow a quality PSDP is an important aspect of learning to program. In order to pursue this hypothesis a study of the PSDP was required. This involved visualising the individual PSDPs employed by a number of programmers in order to build up a model of the PSDP. This model once established was used as the basis for measuring the quality of a given PSDP.

This chapter uses experimental data to visualise the PSDPs from a number of programmers with varying levels of expertise. Taken together with reported work this information is used to propose a model of the PSDPs employed by programmers in a learning environment.

To measure the quality of a PSDP a set of metrics needs to be developed. The creation of these metrics is reported in section 4.5. The goal – question – metric approach due to Rambach and Basili (1987) is used in the proposal and validation of a set of metrics used to measure the PSDP of learner programmers. These metrics are based on the model of the PSDP previously established.

4.2 Data Collection.

The PSDP used by novice programmers was captured by taking a copy of the program prior to its compilation. The full development was therefore represented by a sequence of programs. These were reduced to a file (the `.new` file) of counts representing the version numbers, compilation errors, correctness, lines of code, and reserved word counts for each program in the sequence. Full details of the data collection and its subsequent analysis is given in chapter 3.

There were three phases in the data collection used in this study. The first set of data was collected according to the methods described above in order to visualise the personal software development. From this initial stage a number of graphs were drawn that show the development of the programs over time (or rather over the version number). From this data set a number of possible process metrics were considered. Full details of this activity are described in the next section. In accordance with good practise these metrics were validated using a second set of data. This second set of data was collected using a different set of students and a different but isomorphic problem. From the analysis of the data some metrics were validated as being useful in the measurement of the personal software development process and some not. Details of the validation of process metrics are given in section 4.5. The third data set was collected from a third set of students again developing a solution to a problem isomorphic to but different from the first and second problems. The students in this group were treated differently to the first two sets; they were given feedback during the time of the program development. The metrics developed for the first set of students and validated with the second set of

students are used for this third set of students to see if there has been an improvement in the individual's personal software process. Full details of the feedback experiment are given in chapter 4.

4.3 Visualisation of the Personal Software Development Process.

The aim of this part of the research was to gain some insight into the way programmers both novice and expert developed computer programs. To achieve this a group of students and experts were asked to write a program to solve a problem. Full details of the problems are given in Appendix A. During the development of the problem each version of the program submitted to the compiler was archived using the method described in the previous section. Each version of the program is analysed and the whole development reduced to a file of data as described in the previous section. This is referred to as the `.new` file.

The data in this `.new` file was used to illustrate the PSDP employed by a subject. For each subject three graphs were drawn. The first was a graph of the number of lines of code against version number showing the development of the program through time. A second graph showing the cumulative correctness against version number was also drawn. Cumulative correctness was defined as the maximum number of tests passed by this or previous versions of the program. This will be a monotonic non-decreasing function. The cumulative correctness was chosen rather than the correctness of the current version, as that graph would be a series of peaks. This is because with most subjects a large proportion of versions did not compile and

hence no tests were passed. A third graph of number of compiler errors against version number was also plotted.

The graphs were compared and similarities between different groups of subjects noted.

The data collection used in this study uses three groups of subjects. The data collected from the first group of subjects is used to illustrate the personal software development process and to suggest various process metrics. In this study there were 18 students (novices) and 6 programming lecturers (experts) used in the study.

Expert Group A:

There were 6 expert programmers in the experiment. The graphs below are those taken for a single programmer.

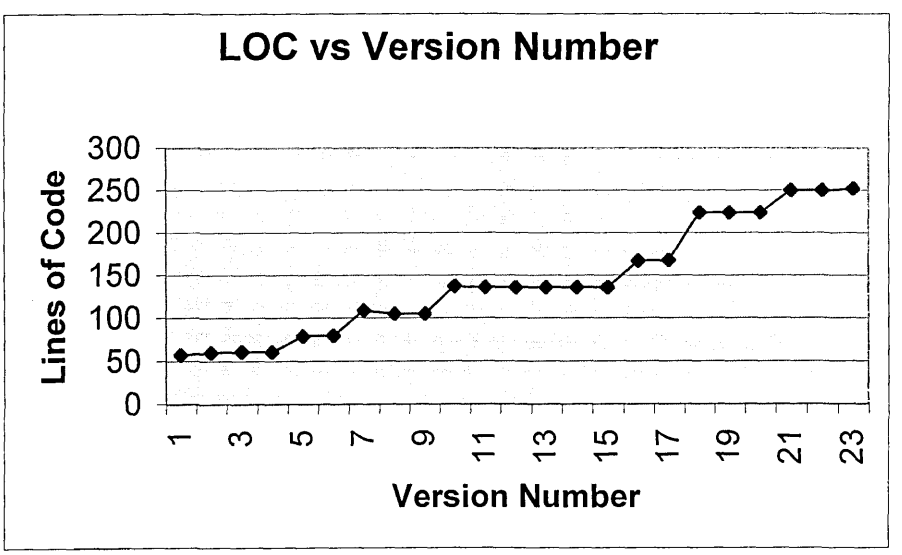

Figure 4.1 Graph of Lines of code against version number for expert programmers.

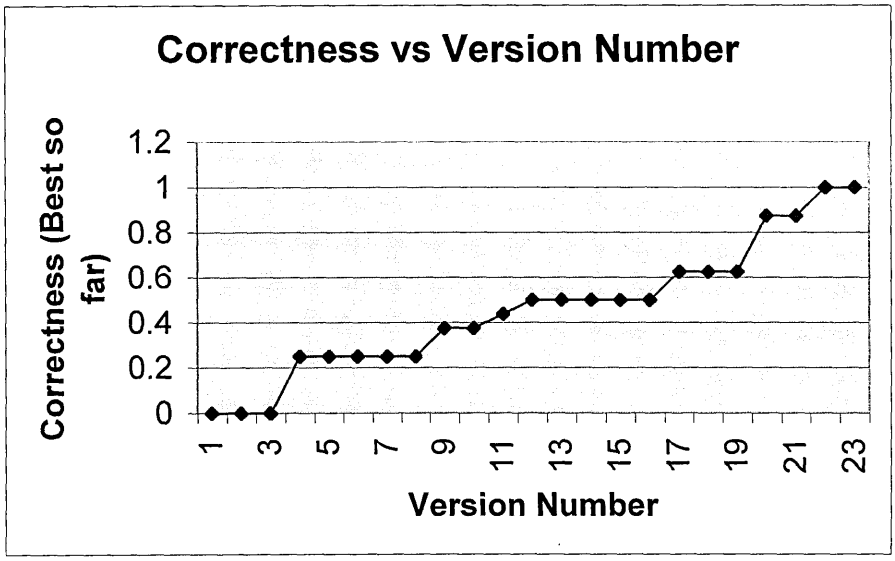

Figure 4.2 Graph of cumulative correctness against version number for expert programmers.

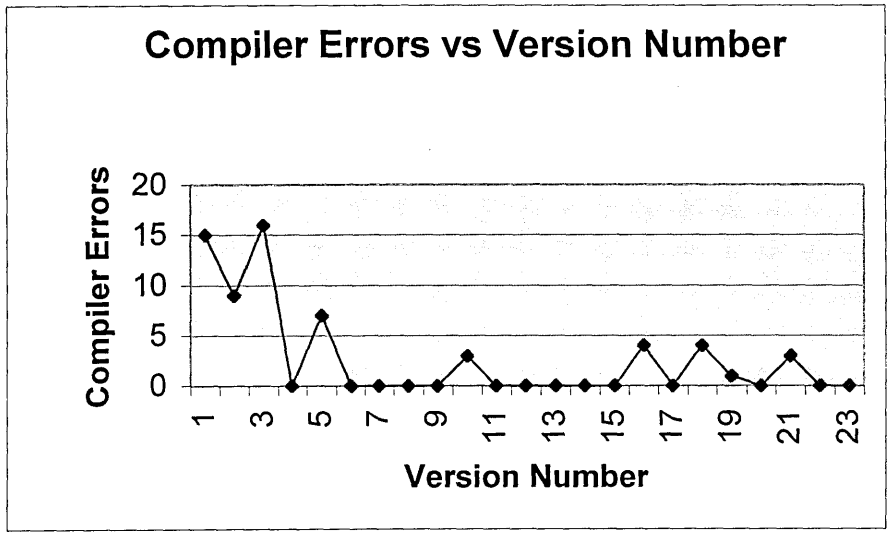

Figure 4.3 Graph of compilation errors against version number for expert programmers

These figures above illustrate the PSDP for an expert programmer and reveal a number of points:

- The lines of code graph (Figure 4.1) increases in a number of stages. Within each stage there is a relatively fixed number of lines of code; between each stage a substantial number of lines of code are added.

- The correctness graph (Figure 4.2) also increases in stages. These stages mirror the lines of code stages with the appropriate lag. This indicates that the subject used each stage to add part of the specification that they got working before proceeding to the next stage.
- The compilation error graph shows two important features. The number of compilation errors was small especially after the initial stage. Also the number of compilation errors from one version to the next tended to reduce unless a new stage was initiated.

These points suggest that these experts operate a strategy of partitioning the problem into a number of smaller tasks. Each additional task involves adding further lines of code and eradicating the errors in this code until it matches the objective of the task. Thus these experts employ a clear methodical way of working through the problem specification.

Novices.

The novices within this study showed a wide diversity of behaviour when developing their program. In order to capture some of the different behaviours of the novices they were categorised into three groups. The first split is between those novices who worked towards a (near) successful solution to the problem and those that did not. The former group are further split into those who followed a development pattern similar in nature to that employed by the experts and those that used a different development method, see Table 4.1 below:

Group	Description	Number of subjects in initial study	Number in Validation study
Experts Group A	Expert programmers – computer programming lecturers.	6	-
Novice Group B	Novices who developed their program to (near) full specification and used a development method similar to that used by experts	4	6
Novice Group C	Novices that developed their program to (near) full specification but did not use similar development methods to those used by experts	8	6
Novice Group D	Novices whose program failed to meet the specification	6	17

Table 4.1 Classification of expert and novice programmers.

Novice Group B.

The first group of novices are those who successfully completed the program specification and showed characteristics in their PSDP similar to the experts. These novices showed evidence of an incremental solution to the programming problem. There were 4 from a total of 18 novices in the initial set of novices and 6 out of a total of 29 in the second set of novices who fell into this category.

The figures below show the PSDP for a particular novice in this category.

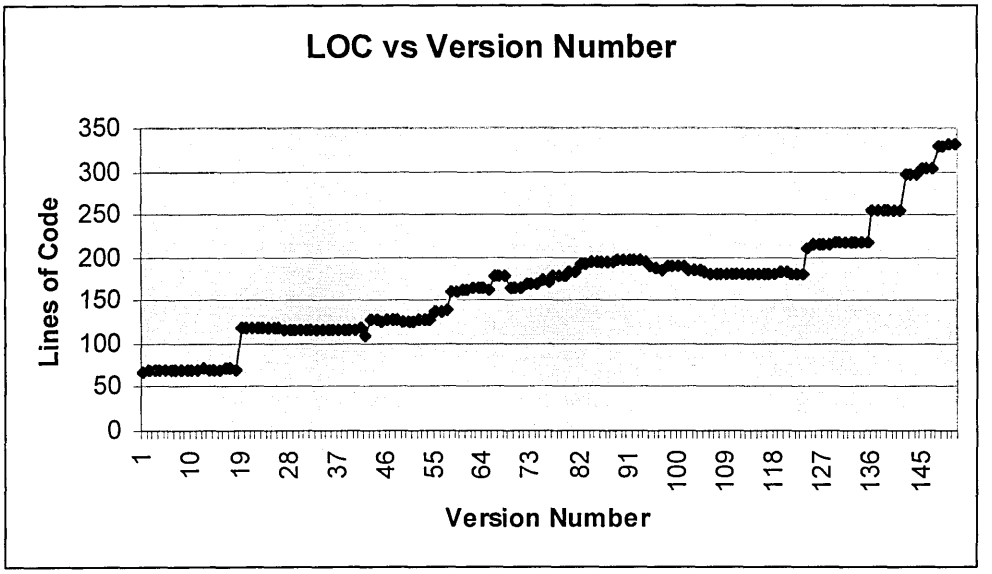

Figure 4.4 Graph of Lines of code against version number for a novice group B programmer.

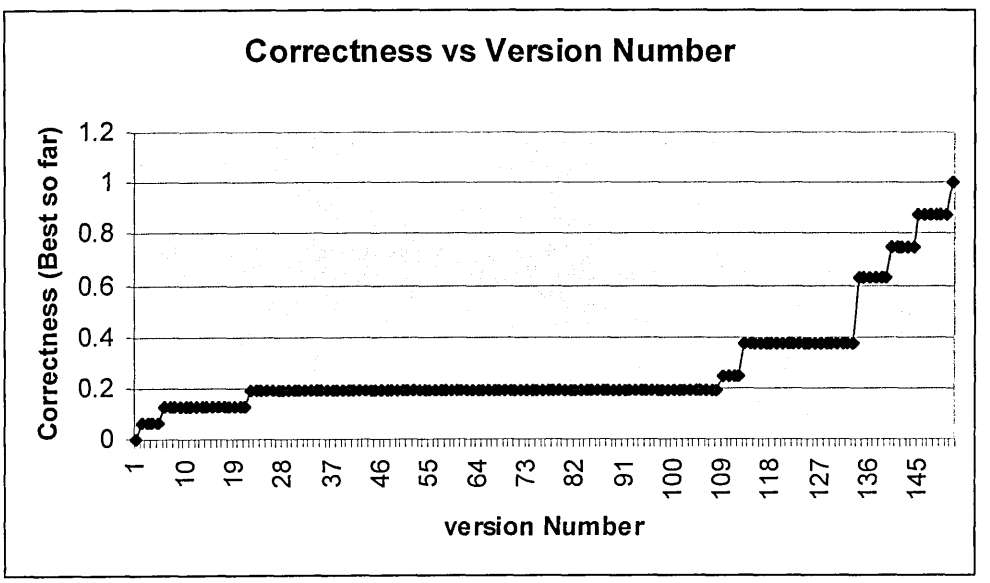

Figure 4.5 Graph of Correctness against version number for a novice group B programmer.

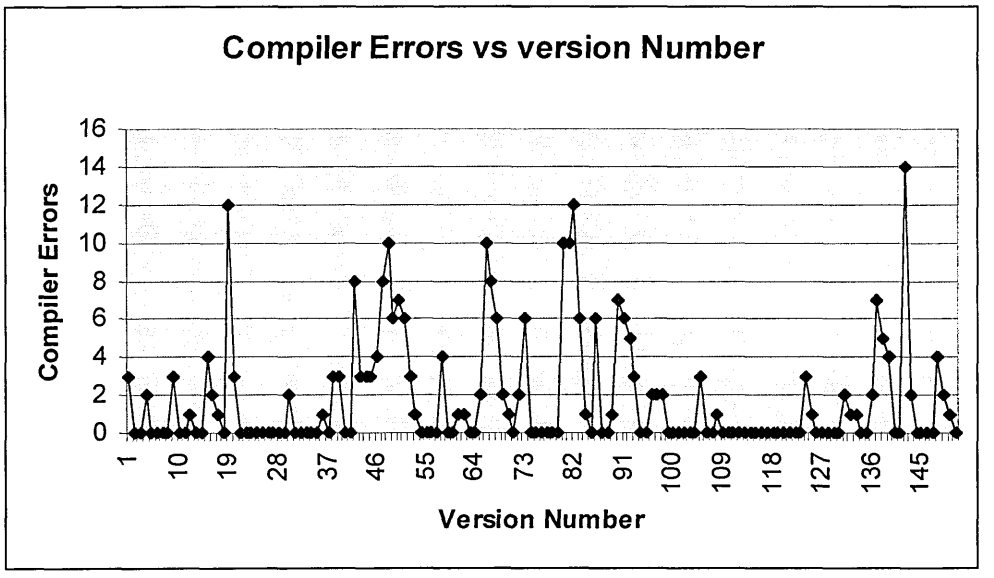

Figure 4.6 Graph of Compiler Errors against version number for a novice group B programmer.

The figures above illustrate the PSDP for a Group 1 novice and reveal a number of points:

- The lines of code graph (figure 4.4) increases in stages. In this example 8 or 9 stages can be identified. There are more stages than the 7 stages in the expert solution. Other novices in this category show fewer stages than do the experts.
- The correctness graph (figure 4.5) also builds in stages which mirror the stages in the lines of code graph. In this way this student is emulating the expert development. This particular student has a close link between correctness stages and lines of code stages, in other novices this link is not as strong.
- The compilation error graph (figure 4.6) does not show the same reduction in compilation errors that the expert does. This could be explained by the lack of familiarity with the language.
- The actual number of versions in this example is much greater than that of the expert shown earlier, (in this example it is approximately a ten fold increase).

Novice Group C.

This second group of novices is characterised by a fairly flat" lines of code" graph but they have stages in the correctness graph. This indicates a behaviour whereby the code is all added at the start and then "debugging" is undertaken to gain a correct program. There are 8 out of the 18 subjects in the initial study group in this category. The graphs below show the PSDP for a particular novice in this category.

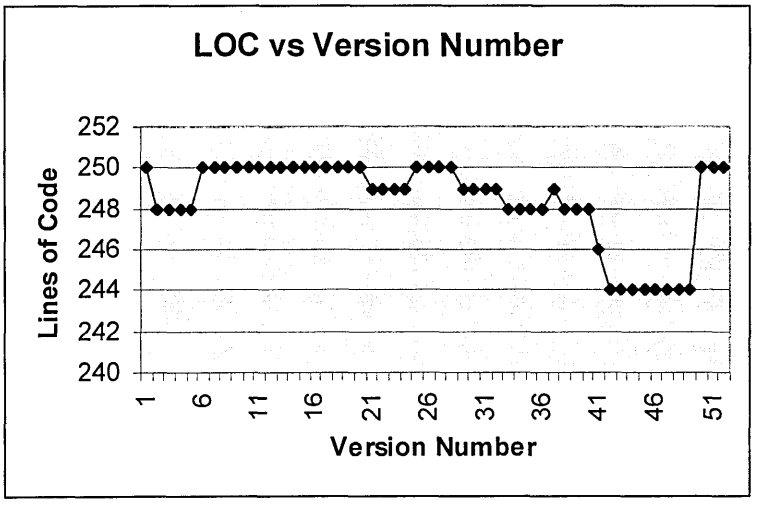

Figure 4.7 Graph of Lines of Code against version number for a novice group C programmer.

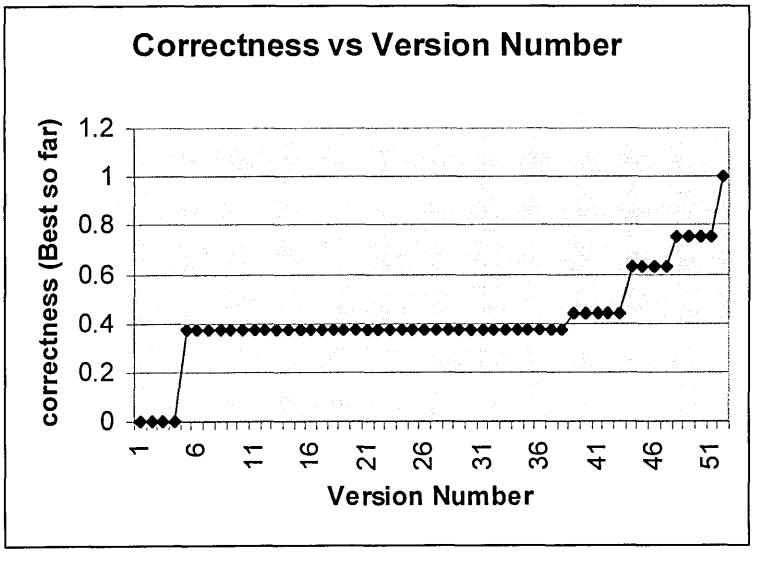

Figure 4.8 Graph of Correctness against Version Number for a novice group C programmer.

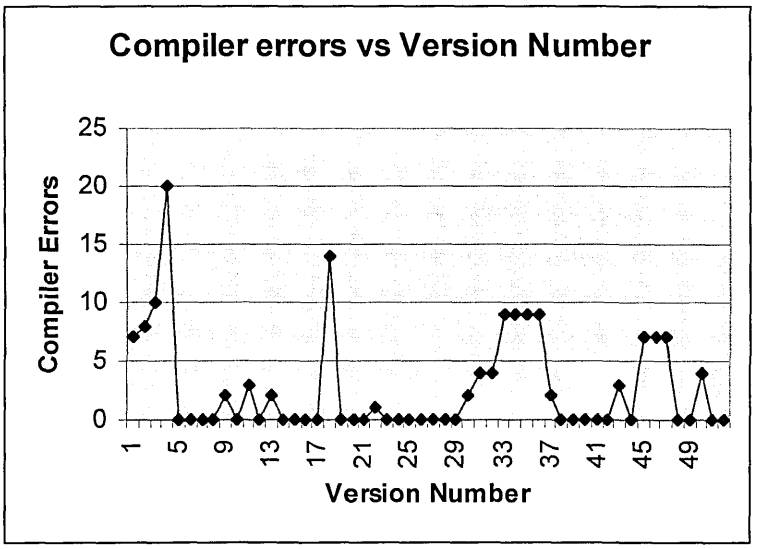

Figure 4.9 Graph of Compiler Errors against Version Number for a novice group C programmer

These graphs above illustrate the PSDP for a group 2 novice. They illustrate a number of points:

- The lines of code graph (figure 4.7) is flat indicating code is added before any compilation is attempted. Only small variations of lines of code (+ / - 10 lines) occur throughout the development.
- The correctness graph (figure 4.8) shows the program becoming more correct over time in stages. This particular subject had some difficulty between version 6 and version 36 in making any progress. After that progress was swift.
- The compilation error graph shows a larger number of compilation errors than the example in novice group 1. There could be two reasons for this, either their knowledge of the language syntax was inferior or, the larger number of lines of code produced more errors.
- A similar number of versions to the previous novice group is used to gain a solution.

Novice Group D.

This final set of subjects were those who did not complete the task. The example gives a subject who made no progress towards a correct program. Other examples in this category make slight progress towards a correct program. The graphs below show the PSDP for a particular novice in this category.

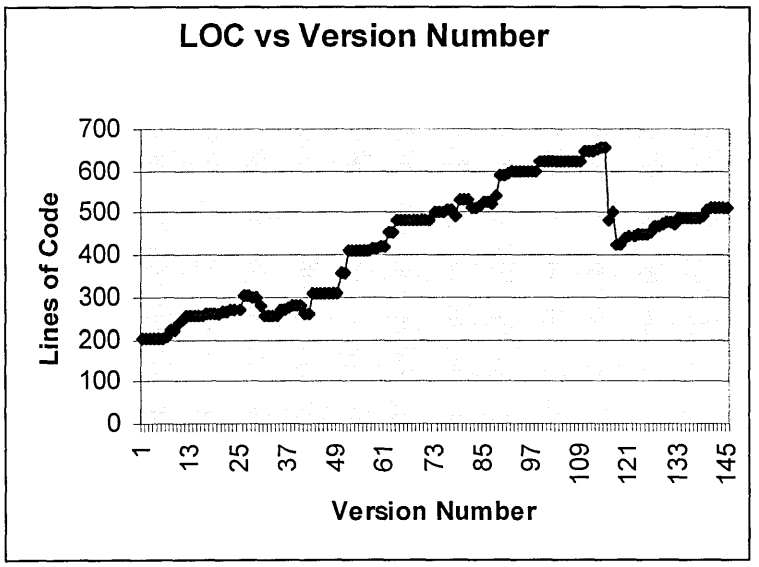

Figure 4.10 Graph of Lines of Code against version number for a novice group D programmer.

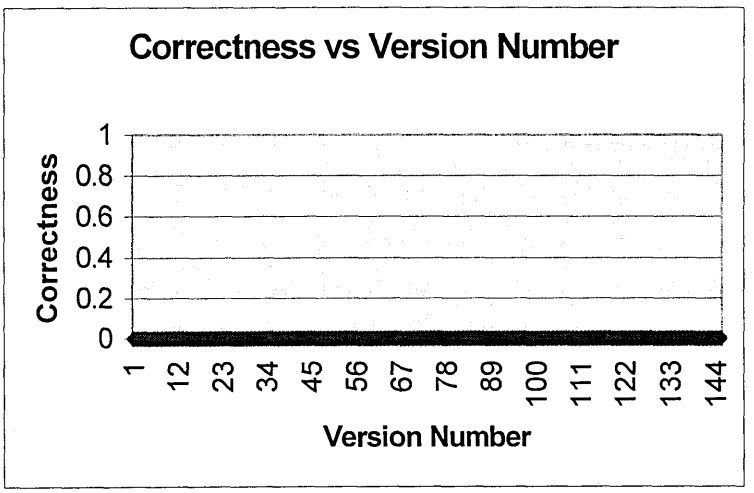

Figure 4.11 Graph of correctness against version number for a novice group D programmer

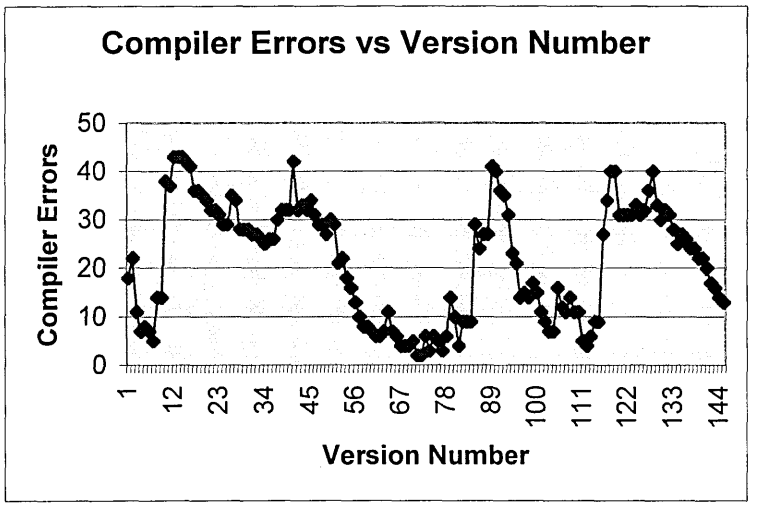

Figure 4.12 Graph of compiler errors against version number for a novice group D programmer.

- The lines of code graph (figure 4.10) shows an increase in code from one version to another despite not getting the previous version to meet its aim. A larger amount of code is written than in any of the previous categories.
- The correctness graph (figure 4.11) indicates that this subject did not gain any correctness for the program. Other subjects show slight increases in correctness (up to 0.125 perhaps).
- The number of compilation errors (figure 4.12) is much larger than any of the previous examples. This indicates a lack of syntactical knowledge of the programming language.
- A lot of effort has been put in by the student as measured by the number of versions submitted to the compiler. This subject is not lacking in commitment rather lacking in knowledge.

Four groups of subjects were identified at the start of this section namely experts and three groups of novice. The visualisation has identified a number of characteristics of each of these groups. The points that have been drawn out for the selected novices are indicative of the group that they represent. The general points were used to develop a model of the PSDP.

4.4 Model of Personal Software Development Process (PSDP).

Authors who have reported on how novices develop programs have put forward iteration enhancement as the basis of software development. Incremental development or iterative enhancement is based on work of Basili & Turner (1975) and Brooks (1987), and described in Hutchens and Katz (1996). In this methodology the programmer splits the overall program into a number of stages. Each stage adds functionality to the program until the full requirements are fulfilled. Brooks talks about growing rather than building a program.

Evidence for this behaviour is seen in the graphs presented in the previous section. This model of incremental program development may be expressed in pseudo-code as shown in figure 4.13 below.

```
Split the problem into stages
Repeat
      Add code for next stage in development
      Loop
            Loop
                  Compile
                  Exit if no compile errors
                  Correct compiler errors
            End Loop
            Run program
            Exit if code satisfies stage of development
            Update code to correct errors
      End Loop
Until full specification complete
```

Figure 4.13 Description of the proposed PDSP model

This model can be used to describe the behaviour of the subjects in the previous sections. Four subjects were presented, each a representative of a group of subjects.

The expert subject followed the model exactly. The problem was split into stages. The code for the next stage was added to the program. Any compilation errors were corrected quickly. The program was tested and potentially altered until that particular stage was correctly implemented. The next stage was then added. This process was repeated until the program was complete.

Novice group B students follow a similar PSDP to the experts. In the example given in section 4.3 the number of stages used in the development was greater than in the expert solution but the subject still followed the model.

The novice group C subject entered all the code prior to attempting to get any part of the program working. This behaviour again fits the model with a single rather than multiple stage development.

The novice group D subject did not complete the task. The behaviour of the subject did not fit the model for two reasons:

1. The subject added code even though they did not complete a stage
2. The subject added code even though they did not reduce the compiler errors to 0.

In short they did not follow the well-defined process embodied in the model and failed to complete the task in any meaningful manner.

The evidence presented in section 4.3 visualises the development of 24 subjects. The model presented here clearly explains the behaviour of the subjects who successfully complete the program. It is hence reasonable to use this model as the basis for the

measurement of the PSDP.

4.5 Establishing a Set of Core Process Metrics.

The question to be answered in this section is "How can the PSDP be measured?". This section describes the study that sought to establish a set of core metrics to measure the PSDP. The methods used to establish the metrics follow the five principles of Eijogu(1993) and discussed in section 2.1. Moreover the validation follows the methods proposed by Shepperd (1992). The metric was derived from the model described in the previous section, and was then validated against experimental data.

4.5.1 Investigation Process.

Five features of the PSDP model were identified as being suitable for measurement. For each of these features candidate metrics were put forward. These candidate metrics were derived from the model. Only metrics that satisfy Eijogu's(1993) mathematical metric properties were considered.

The validation process to establish a metric used in this research was based on the ability of the candidate metric to discriminate between groups of novice. Moreover the analysis to establish a metric was repeated over two sets of novice (two separate cohorts of students) to ensure the results were repeatable.

To validate a particular candidate metric the following method was used. An initial

set of data was collected from 24 experts and novices using the methods discussed in chapter 3. Each subject was classified as either expert, or novice group B, C or D according to the criteria discussed in section 4.2 and summarised in table 4.2 below.

Group	Description	Initial Set of subjects	Validation Set of subjects
A	Experts	6	-
B	Incremental Solution Novices	4	6
C	Debug Novices	8	6
D	No Solution	6	17

Table 4.2 Details of the subjects in the experiment to establish a set of novice process metrics.

The candidate metric was calculated for each subject. The subjects were then ranked in order of the candidate metric values. This list was used to allocate a predicted group for each subject according to a particular metric. The number of subjects in each predicted group was kept the same as the numbers in the actual group. The numbers in each group are given in table 4.2. Using these values the top 6 would be predicted to be group A, the next 4 group B, the next 8 group C and the final 6 group D. The tables used in section 4.6 (table 4.5, 4.6 and onwards) were generated from this data. Each novice has an actual group and a predicted group and makes a contribution to one cell in each table. A metric that predicts the groups perfectly would have values down the leading diagonal of the table and zero elsewhere. Any variation from this pattern indicates a wrong predicated classification for a subject.

This process is repeated for a second set of subjects to validate the candidate metric. The second data set consists of novice programmers who were again classified as group B, C, or D novices according to the criteria discussed in section 4.2. This second set of students was similar in nature to the first. The students came from similar backgrounds and the experiment was carried out at a similar time in their

educational experience. The problem given to the second set of students was different from that given to the first set for educational reasons. However the problem is in fact isomorphic to the original problem (details of the problems are given in appendix A). The data from this second set of subjects was processed in the same way as the first; the versions of the program were collected, analysed and the data reduced to a file of counts representing the process (a `.new` file). The metrics calculated and the subjects ranked to product their group from the metric values.

Some candidate metrics that discriminated between groups within the first data set also discriminated between groups within the second data set and were viewed as suitable metrics to measure the particular aspect of the PSDP. Some candidate metrics that could discriminate between the groups within the first data set could not discriminate between groups within the second data set and were discarded.

4.5.2 Measuring Process Quality.

There are five aspects of the PSP model that it would be useful to measure. For each of these categories a number of potential metrics are discussed, analysed and validated.

The five aspects are:

- how long the development took to complete
- how successful the subject was at compiling their code
- how successful the subject was at getting their code to run
- how well the problem was split into stages
- how much progress was made at each step in the development

For each of these categories a number of potential metrics were considered. Many were discarded as they failed to discriminate between the different groups in the first data set. The results presented below discuss only those candidate metrics that had potential and could discriminate between groups within the first data set.

Fundamental to developing these metrics is the ability to measure the correctness of a program. This would be used not only in measuring the correctness of the final program but also the correctness of each version of the program submitted to the compiler. Section 2.2.5 contains a discussion concerning how other authors measure the correctness of a program. The method used here is that used in Ceilidh (Benford et.al., 1993) and Jackson (1991). A set of tests was established. A fully correct program will pass all the tests. The tests vary in complexity from "program compiles successfully" to the most difficult test where each of the problem requirements is satisfied. The scale is based on the 8 point correctness scale of Conway (1978), again described in section 2.2.5.

The metric `sp` is used to measure the *proportion of specification* completed and used to measure the program correctness.

Area	Symbol	Name	Description
Correctness	`sp`	Proportion of specification complete	`sp = t / T` where `t` = number of tests passed `T` = total number of tests.

Table 4.3 Definition of Specification Proportion.

4.6 Results.

4.6.1 Length of Development Results.

The project elapsed time or time taken on a project could measure the length of development. Due to the nature of student work all subjects will start the project and end it at approximately the same time although subjects may have spent differing amounts of time on the project. Both Parrish et. al. (1997) and Grove (1998) tried to capture the amount of time spent on the project by getting students to record their time. The authors report that time is not a significant metric in measuring quality of novice programmer process. As argued earlier, recording the time would interfere with the PSDP and therefore invalidates the results.

There are two potential metrics investigated here. One is the count of the number of versions submitted for compiling during the development. The second is derived from the first and is a measure of the estimated number of versions for successful completion. In cases where the subject completed the project the two metrics are identical. If, for example, a subject completed 0.5 of the project in 50 versions then the estimated versions to completion is 100 (=50 / 0.5). These metrics are described in table 4.4 below:

Area	Symbol	Name	Description
Length of Development	N	Number of versions	Number of versions of the program submitted to the compiler
	Nc	Versions to completion	Extrapolation of N to estimate how many versions would be needed to complete the project. Nc = N / sp

Table 4.4 Definition of potential length of development metrics.

For each novice in this investigation the metrics (N and Nc) were calculated. The novices are ranked according to the particular metric value and the predicted group for each novice is found. Tables 4.5 and 4.6 below show the numbers of novices in each predicted group for each actual novice group.

Number of Versions (N)

First Set of Subjects

Actual Group \ Predicted Group	A	B	C	D
A	4	0	1	1
B	2	1	1	0
C	0	1	4	3
D	0	2	2	2

Validation Set

Actual Group \ Predicted Group	A	B	C	D
A	-	-	-	-
B	-	0	2	4
C	-	0	2	4
D	-	6	2	9

Table 4.5 Classification of groups of subject on the basis of number of versions metric

Number of Versions to Completion (Nc)

First Set of Subjects

Actual Group \ Predicted Group	A	B	C	D
A	6	0	0	0
B	0	1	3	0
C	0	3	4	1
D	0	0	1	5

Validation Set

Actual Group \ Predicted Group	A	B	C	D
A	-	-	-	-
B	-	3	3	0
C	-	3	1	2
D	-	0	2	15

Table 4.6 Classification of groups of subject on the basis of number of versions to completion metric

The results given in Table 4.5 and 4.6 show that the experts can be distinguished from the novices using the Nc metric. The experts take fewer versions to complete their development than do the novices. This is as would be expected.

However the data does not clearly distinguish between the groups of novice. The number of versions a novice takes to complete the program development cannot help predict how well the development was carried out.

It is disappointing that the length (or expected length) of development does not separate out the groups of novices. This is very much in line with the result of Parrish et.al. (1997) who found no correlation between number of compiles and project success. Some students who expend much effort on the project succeed while others fail. Some students gain success after a smaller number of compiling attempts while others give up.

No metric measuring the length of development was used in future work due to the unreliability of such metrics.

4.6.2 Compiling Success Results.

One aspect of programming knowledge captured in the model of a programmer's personal software development process is the ability to add syntactically correct code and the ability to correct any compiler errors. This is represented by the innermost loop of the model. It seems reasonable that better programmers make fewer mistakes and take less time (fewer versions) to correct any mistakes that they make.

Three potential metrics are proposed to measure the compiling expertise of the programmer. The first measures the proportion of versions that compile correctly. The second measures the proportion of versions that have fewer compilation errors than the previous version. The final proposed metric is the average number of versions it takes to achieve a clean compilation.

Area	Symbol	Name	Description
Compiling Expertise	cr	Compiling ratio	Proportion of versions that compiled with 0 errors
	cpi	Compiling performance index	Proportion of versions which had fewer compiler errors than the previous step
	asc	Average steps to compile	Average number of versions taken to achieve a clean compilation

Table 4.7 Definition of potential compiling expertise metrics.

For each novice in this investigation the cr metric is calculated. The novices are then ranked according to the metric value and the predicted group for each novice is found. Table 4.7 below shows the numbers of novices in each predicted group for each actual novice group.

Compiling Ratio (cr)

First Set of Subjects

Actual Group \ Predicted Group	A	B	C	D
A	0	2	4	0
B	1	2	1	0
C	4	0	3	1
D	1	0	0	5

Validation Set

Actual Group \ Predicted Group	A	B	C	D
A	-	-	-	-
B	-	3	3	0
C	-	1	3	2
D	-	2	0	15

Table 4.8 Classification of groups of subject on the basis of compiling ratio.

This metric (cr) can distinguish group D novices from the others in both sets of data. However groups A, B, and C are confounded. The metric cannot distinguish between these groups of programmers. There may be many reasons for these results. Group 4 subjects, those novices who did not develop a correct solution, had difficulty with the language syntax and can thus be separated from the rest. For the other groups the ratio comes out at a similar figure as better programmers take fewer steps

thus tending to increase the `cr` figure.

For the metric `cpi` a similar table was generated using the same methods as for the `cr` metric

Compiling Performance Index (`cpi`)

First Set of Subjects

Actual Group \ Predicted Group	A	B	C	D
A	3	0	3	0
B	1	1	2	0
C	1	3	2	2
D	1	0	1	4

Validation Set

Actual Group \ Predicted Group	A	B	C	D
A	-	-	-	-
B	-	2	3	1
C	-	2	1	3
D	-	2	2	13

Table 4.9 Classification of groups of subject on the basis of compiling performance index.

This metric (`cpi`) does not clearly distinguish between the groups. In the first experiment some experts are confused with novices, but half are not. Similarly some group D subjects are confused with group C subjects but the majority are not. The group B and C subjects are confused. Perhaps this last finding is not surprising and the grouping has been carried out on the basis of their ability to solve the problem rather than on their syntactic knowledge.

The third potential metric `asc` is investigated in the same way as the previous two metrics.

Compiling Performance Index (`asc`)

First Set of Subjects

Actual Group \ Predicted Group	A	B	C	D
A	2	1	2	1
B	1	1	2	0
C	3	2	2	1
D	0	0	2	4

Validation Set

Actual Group \ Predicted Group	A	B	C	D
A	-	-	-	-
B	-	3	1	2
C	-	3	3	0
D	-	0	2	15

Table 4.10 Classification of groups of subject on the basis of average steps to compile a program.

The performance of the `asc` metric to discriminate between groups is not good. There is a lack of discrimination between groups A, B, and C. Group D does seem to be discriminated from the others.

The three metrics discussed are not good at discriminating between the groups of subject. However, there was a need in the work following to have some metric that could measure the compiling expertise. `cpi` was used as this metric because of it's ability to distinguish group D from the others and it can distinguish experts (group A) better than `cr`.

4.6.3 Running Success Results.

The aspect of programming investigated here was to find a metric that measures how successful a programmer was at getting the program to run. This aspect of program development is represented by the middle loop in the model of the PSDP in figure 4.14. This aspect of the PSDP is concerned with ensuring the logic at each stage of development is correct. The metrics put forward here are chosen to be independent of the compiling aspects of the PSDP. Two metrics are considered here. The first is the run performance indicator that considers only those steps that have clean compiles (no compiler errors) and calculates the percentage that made some progress towards full correctness. The second attempts to penalise those subjects who had few versions that compiled and thus made significant progress in each stage. The metric chosen is the run performance index (`rpi`) multiplied by the proportion of the full specification achieved (`sp`).

Area	Symbol	Name	Description
Run Stage Success	`rpi`	Run Performance Indicator	From all those steps that compiled what proportion had a greater % correctness than any previous step.
	`rrpi`	Relative Run Performance Indicator	`rrpi = rpi * sp`

Table 4.11 Definition of potential run stage success metrics.

To validate a metric in the area of running success similar methods are used as were used to validate the compiling metric. For each novice the `rpi and rrpi` metrics were calculated. The novices are then ranked according to the metric value and the predicted group for each novice is found. Table 4.12 below (and 4.13 later) shows the numbers of novices in each predicted group for each actual novice group.

Run Performance Index (`rpi`)

First Set of Subjects

Actual Group \ Predicted Group	A	B	C	D
A	4	2	0	0
B	0	2	0	2
C	0	0	6	2
D	2	0	2	2

Validation Set

Actual Group \ Predicted Group	A	B	C	D
A	-	-	-	-
B	-	1	3	2
C	-	4	1	1
D	-	1	2	14

Table 4.12 Classification of groups of subject on the basis of run performance indicator.

As can be seen from the results the discrimination between the groups is not clear. One reason for this is that some group D subjects have a good `rpi` figure, comparable with the experts, because they only succeeded in compiling a program two or three times thus artificially inflating the figure due to the low numbers. If the group D students are removed from the data then this metric perfectly discriminates the experts from the novices and there is only 1 misclassification between the novice groups.

In order to militate against novices who only succeeded in compiling a program two or three times and hence gained a high `rpi` figure, the `rrpi` metric is investigated. This metric multiplies the `rpi` figure by the proportion of the specification complete (`sp`). This penalises those subjects who only managed to get a low proportion of the specification completed.

The investigation into the use of the `rrpi` metric follows the same method as that used above.

Relative Run Performance Index (`rrpi`)

First Set of Subjects

Actual Group \ Predicted Group	A	B	C	D
A	6	0	0	0
B	0	2	2	0
C	0	2	5	1
D	0	0	1	5

Validation Set

Actual Group \ Predicted Group	A	B	C	D
A	-	-	-	-
B	-	4	2	0
C	-	2	3	1
D	-	0	1	16

Table 4.13 Classification of groups of subject on the basis of their relative run performance indicator.

The `rrpi` figure clearly distinguishes between the experts and the novices. There is also a high level of discrimination between the group D novices and the other groups (B and C). Though there is still some confusion between groups B and C. There is good reason for this. The graph of "version number" against "best correctness value so far" are similar for these groups of novice.

The metric `rrpi` will be used to measure the ability of a programmer to achieve their goal within a stage of a program. The metric can distinguish between experts, group 4 novices and the rest. To distinguish between groups B and C some measurement of the stages in development needs to be obtained.

4.6.4 Stages in Development Results.

The stages used in the development of a program with respect to lines of code added and correctness were clearly seen in the graphs produced earlier in this chapter. A third area where stages in development can potentially be seen is in the increase in number of procedures and functions used in a program. This section describes the investigation of potential metrics that can discriminate between groups using stages in the PSDP. Four terms are now defined:

- Lines of code are defined as a count of the number of lines in the program source file.
- A stage in the lines of code graph is defined as ten or more additional lines of code added to the program.
- A stage in the tests graph is defined as one or more additional tests passed.
- A stage in the (potential) number of procedures and functions graph is defined as the addition of an extra procedure or function within the program code.

Three metrics were initially considered:

- `sloc` – number of stages in the lines of code graph during development.
- `srn` – number of stages in the correctness graph during the development
- `spf` – number of times procedures or functions were added to the program code.

None of these metrics discriminated between the groups of subjects, other than a tendency of group C and D subjects to have a lower value for these metrics than group A and B subjects did. Part of the problem is that many of the group C and D subjects submitted their program to the compiler, got compilation errors that they did not fix but added further code. Thus the value of the `sloc` metric was artificially inflated as it measures the stages in the lines of code graph and does not take account

of whether these stages achieve anything. To counter the effect of adding code when the previous code did not work amended metrics were considered.

Area	Symbol	Name	Description
Stages in Development	`sl`	stages in lines of code graph	Number of stages (additional 10 loc between one step and the next) during development after % tests > 15%
	`st`	stages in % running graph	Number of stages (additional 1 test passed) after % tests > 15%
	`sf`	stages in function graph	Number of times procedures or functions added to the code after % tests > 15%

Table 4.14 Definition of potential stages in development metrics.

To validate a stages in development metric similar methods are used as were used above. For each novice the `sl`, `st` and `sf` metrics were calculated. The novices are then ranked according to each metric value and the predicted group for each novice is found. Table 4.15 below (and 4.16, 4.17 later) show the numbers of novices in each predicted group for each actual novice group

Stages in LOC (sl)

First Set of Subjects

Actual Group \ Predicted Group	A	B	C	D
A	3	1	2	0
B	3	1	0	0
C	0	2	5	1
D	0	0	1	5

Validation Set

Actual Group \ Predicted Group	A	B	C	D
A	-	-	-	-
B	-	5	1	0
C	-	1	4	1
D	-	0	1	16

Table 4.15 Classification of groups of subject on the basis of the stages in lines of code.

The results for the first set of subjects do not look very promising. However if all the experts are removed from the data the first `sl` discriminates well between the

novices. This is shown clearly in table 4.15a. which is a repeat of table 4.15 but with the expert (group A) subject data removed. There is only one novice wrongly classified by this metric.

Stages in LOC (sl)

First Set of Subjects

Actual Group \ Predicted Group	A	B	C	D
A	-	-	-	-
B	-	4	0	0
C	-	0	5	1
D	-	0	1	5

Validation Set

Actual Group \ Predicted Group	A	B	C	D
A	-	-	-	-
B	-	5	1	0
C	-	1	4	1
D	-	0	1	16

Table 4.15a Classification of groups of subject on the basis of the stages in lines of code, with experts removed from the data.

In the validation data set there are only 2 subjects wrongly classified. The overall goal of this research is to examine the behaviour of novices and not experts, so the problem of the experts being confused with novices is not considered being very important. One reason why the experts were confused with novices is that as the experts have more experience their stages in the development of their programs becomes larger and hence the number of stages fewer.

The investigation produced the following results for the `st` metric:

Stages in % program correct (st)

First Set of Subjects

Actual Group \ Predicted Group	A	B	C	D
A	2	2	2	0
B	4	0	0	0
C	0	2	5	1
D	0	0	1	5

Validation Set

Actual Group \ Predicted Group	A	B	C	D
A	-	-	-	-
B	-	5	1	0
C	-	1	4	1
D	-	0	1	16

Table 4.16 Classification of groups of subject on the basis of the stages in correctness.

Again the results for the first set of subjects do not appear to discriminate between subjects. However if the expert data is removed, as was done for the previous metric, then this metric, `st`, discriminated between the novices with the same power as does `sl`. The data for the validation set is the same as the previous metric in that two

subjects are wrongly classified.

The experiment produced the following results for the `sf` metric:

Stages in functions added (sf)

First Set of Subjects

Actual Group \ Predicted Group	A	B	C	D
A	3	1	2	0
B	2	1	1	0
C	1	2	4	1
D	0	0	1	5

Validation Set

Actual Group \ Predicted Group	A	B	C	D
A	-	-	-	-
B	-	4	2	0
C	-	2	2	2
D	-	0	2	14

Table 4.17 Classification of groups of subject on the basis of the stages in functions added.

These results do not show a metric that can clearly discriminate between the categories of subject and even when the experts are removed the discrimination with respect to the first set of data is worse than that exhibited by metrics `sl` and `st`. This metric will not be used further within this research. The `sl` and `st` metrics will both be used later in this work.

4.6.5 Overall Success Rate Results.

The final measure looked for is some way to characterise the whole development. The question is whether some progress has been made at each step in the development. It seems reasonable that better programmers will make more progress at each step of the development than do novice programmers. Two overall success rate metrics were considered.

Area	Symbol	Name	Description
Overall success rate	`tpi`	total performance indicator	Proportion of versions that made progress over the previous steps during the development
	`rtpi`	relative total performance indicator	`rtpi = tpi * sp`

Table 4.18 Definition of potential overall success rate metrics.

Both these metrics are based on a notion of measuring the progress towards the overall goal that each version of the program made. In this instance progress is defined as either the version having fewer compilation errors than the previous version or the version passing more tests than all previous versions. The first metric in table 4.17 above, `tpi`, measures the proportion of versions that make progress, as defined previously, over the previous versions. It was found that this metric was artificially high for those novice programmers who failed to get a working program (group D novices). For example if a novice has 20 compilation errors in the first version and reduced this figure by 2 in each of the next 10 versions and the final version is incorrect then the `tpi` figure is 100%, even though the program may not do anything of value.

Validation of overall success rate metrics used the same methods as have been used previously. The candidate metrics were calculated for each novice. The novices were ranked and the predicted group found. The predicted and actual group pairs were tabulated and the following results found:

Total Performance Indicator (tpi)

First Set of Subjects

Actual Group \ Predicted Group	A	B	C	D
A	6	0	0	0
B	0	0	4	0
C	0	1	3	4
D	0	3	1	2

Validation Set

Actual Group \ Predicted Group	A	B	C	D
A	-	-	-	-
B	-	2	1	3
C	-	2	2	2
D	-	2	2	12

Table 4.19 Classification of groups of subject on the basis of the stages in correctness.

The `tpi` metric confuses group D subjects with other groups. The definition of progress gives them credit for progress when in reality the progress towards the overall goal is poor. That aside the metric discriminates between the other groups (1 misclassification between groups B and C). The experts have the best `tpi` figure.

To solve the problem of group D indicating progress when there was no real progress the `rtpi` metric was used in the method rather than the `tpi` metric above.

Relative Total performance Indicator (rtpi)

First Set of Subjects

Actual Group \ Predicted Group	A	B	C	D
A	6	0	0	0
B	0	3	1	0
C	0	1	7	0
D	0	0	0	6

Validation Set

Actual Group \ Predicted Group	A	B	C	D
A	-	-	-	-
B	-	4	2	0
C	-	2	3	1
D	-	0	1	16

Table 4.20 Classification of groups of subject on the basis of the stages in correctness.

The `rtpi` metric discriminates between the groups save for 1 subject in the first set of data and three subjects in the validation set of data. This metric will be used as a metric for overall progress during the rest of the work reported in the next chapters.

4.7 Process Marking.

There are numerous automated marking schemes, see section 2.2, that are used to mark a program developed by a novice programmer. These marking schemes use a linear combination of metrics to establish a "mark" for the quality of the student's program. With the establishment of metrics that measure aspects of the PSDP in section 4.6 above, it is possible to combine these metrics to produce a "process mark" for each subject. In this section a combined process mark was calculated for each subject and the discriminatory power of this metric shown using the classification systems used with the individual metrics.

The "process mark" is calculated as a linear combination of the four elements considered in section 4.6 above. A weighting of 0.25 is given to each of compiling success (`cpi`), running success (`rrpi`) and total success (`rtpi`). The final 0.25 is split evenly between the two metrics that measure the stages in development `sl` and `st`. The `cpi`, `rrpi` and `rtpi` metrics give a percentage value and are simply weighted together in the calculation. The `sl` and `st` values are absolute values that depend on the problem under consideration. These values are converted to a percentage by comparing the values to a "model" solution using a trapezium function. This is the technique that is used to calculate the program mark by Rees (1982) and Benford et.al. (1993). The overall process mark is calculated as:

```
p = 0.25*cpi + 0.25*rrpi + 0.125*sl' +
    0.125*st' + 0.25*rtpi
```

where

p = Overall process mark
cpi = Compiling progress indicator
rrpi = Relative run progress indicator
Sl' = Transformed value of stages in LOC
st' = Transformed value of stages in testing
rtpi = Relative total progress indicator.

The values for each subject were calculated and used to predict which category the subject should be in. These results are presented in table 4.20 below.

Overall Process Mark (p)

First Set of Subjects

Actual Group \ Predicted Group	A	B	C	D
A	5	1	0	0
B	1	3	0	0
C	0	0	7	1
D	0	0	1	5

Validation Set

Actual Group \ Predicted Group	A	B	C	D
A	-	-	-	-
B	-	4	2	0
C	-	2	3	1
D	-	0	1	16

Table 4.21 Classification of groups of subject on the overall process mark.

The interesting point that can be taken from this table is that it is identical to the table for rtpi indicating that the linear combination of metrics gives the same discrimination, as does the rtpi metric. This gives further evidence to the argument that rtpi is in some way an overall process metric.

In a marking scheme that sought to include some element of process then the rtpi metric can be included as a measurement of the quality of the overall novice's PSDP.

4.8 Summary.

The purpose of this chapter was to establish a set of metrics that can be used to measure the personal software development process (PSDP) used by novice programmers. The first step was to visualise the PSDP used by novices and experts. Studies were carried out to capture data during the development of software by novice programmers. The data once collected was reduced to counts and graphs of version number against lines of code, correctness and compilation errors drawn. These graphs showed similarities between groups of novices allowing 3 groups of novices and one expert group to be defined.

A model of the PSDP was proposed and validated by comparing the expected behaviour of the novice against the actual behaviour. The proposed model did in some way describe how novice programmers developed programs.

From the model a number of characteristics were established from which a set of metrics were chosen to be validated. For each characteristic a number of potential metrics were proposed and using experimental data these metrics were validated (or not). The following set of metrics were established and were used during the rest of this work:

Area	Metric	Name
Length of Development	None	
Computing Success	`cpi`	Compiling progress indicator
Running Success	`rrpi`	Relative run progress indicator
Stages in Development	`sl`	Stages in LOC.
	`st`	Stages in testing
Overall Success	`rtpi`	Relative total progress indicator

Table 4.22 Validated set of PSDP metrics.

5. Feedback as a Method of Process Improvement.

5.1 Introduction.

The overall theme of this research was to examine the hypothesis that the expertise of a novice programmer may be improved by focussing the learning onto the program development process. The work discussed in the previous chapter allowed the personal software development process (PSDP) to be measured. Discussions from the literature and reported in chapter 2 indicate methods of how a program may be measured. It is now feasible to measure both the process and the product for an individual software development. The experiment described here was designed to investigate whether simple feedback, in the form of a correctness score for each version of a program, can improve the PSDP of a novice programmer. A successful outcome of this experiment would show that there is potential in the argument that the software development process is an important aspect of expertise in computer programming.

The experiment was designed as a pre-test-post-test control group design that is widely used in educational research. Two groups of novices are considered. Both groups are tested before the treatment. They are then asked to develop a second program. The treatment group is given feedback during this second development. An analysis of variance was used to analyse the results. These results are analysed in terms of the improvement to the process metrics identified in chapter 3 namely the compiling performance index (`cpi`), relative run performance index (`rrpi`), stages in lines of code (`sl`) and stages in testing (`st`) and relative total performance index

(rtpi).

The feedback under consideration in this chapter relates to program completeness as indicated by the number of test cases successfully completed. When a novice runs a program not only are the results printed on the screen but the number of tests passed is also reported to the novice. The simple feedback does no more than emphasise to the novice the correctness of the program and informs them whether they have completed the given stage in the program development. This feedback, if successful, would be expected to have an effect on the process metrics.

- cpi – compiling performance index – no change in this metric was expected. This aspect of the PSDP model was not addressed by the particular feedback. Indeed if there were significant changes to this figure then this would indicate a major flaw in the experimental design.
- rrpi – relative run performance indicator – should be improved if the feedback was achieving the expected effect. The feedback influences the novice's awareness of the correctness of their program and was designed to improve the novice's awareness of this aspect of the development. This should show itself by improving this metric during the development process.
- sl – stages in the lines of code and st – stages in the testing – these metrics measure the planning ability of the novice. They should not be directly affected by the feedback given. This is because the feedback is designed to improve within each stage rather than between stages.
- rtpi – relative total performance index – this metric is defined in terms of progress which in turn is defined as compiling or correctness improvement from one version to another. The latter aspect of this metric should be affected by a

process improvement and hence it is expected that this metric will be increase given the feedback.

5.2 Experimental Method.

5.2.1 Introduction.

This experiment was carried out to test the hypothesis that process intervention can improve the PSDP of a novice programmer. This experiment was based on observing two groups of students during their development of two programs. The two groups of students are a treatment group who are given feedback to help in their program development and a control group who have not been treated. Each group was asked to develop two programs. By analysing the change in performance of the control and treatment groups between the first and second developments the effectiveness of the treatment can be determined.

5.2.2 Experimental Design.

In a pre-test-post-test-control group experimental design subjects are tested before treatment to provide a base line for the analysis. Each subject in this experiment was requested to develop a program (pre-test program). This first program was considered appropriate to the current state of the novice learning as dictated by the lecture course. No feedback was given with this program to either groups of novice. At a time later (6 weeks), the novices were requested to develop a second program (post-test program). This program was again considered appropriate to the current state of the novices' learning. As the course had developed further this problem was

more complex than the first problem. During the development of this second program, novices in the treatment group received feedback.

5.2.3 The Participants.

Two separate groups of novices are needed for this experiment. For educational reasons it was decided to use two different cohorts of students in the experiments. The main advantage of using two separate cohorts of students is that there was little chance of information crossing over between the two groups of students. A single cohort of students split into two groups would have a greater danger of interaction between the groups. Indeed it would be possible for some students in the control group to see the feedback given to the treatment group of students and use their command to receive the feedback. This cross over between the two groups would have invalidated the experiment. Within each cohort all the students are subject to the same educational experience. From an ethical point of view there is no danger of some students claiming they had an inferior educational experience.

The disadvantage of using two cohorts of students is that they may have different qualifications and educational experience. Greater control of groups, both in terms of numbers and background, can be gained if a single cohort is split rather than relying on two cohorts. In this experiment there was no reason to suspect any differences between the cohorts. The admission policy to the course and hence the module was not altered between the two intake years in question. Furthermore the lecturer delivering the module and the module content did not alter between the cohorts. The programs given to the two groups of novices, pre and post-test programs, were different to ensure no cross fertilisation (plagiarism) between the groups. Although

two different problems were used in the pre-test and in the post-test, the pairs of problems were isomorphic. The program context was different but the essential problems the same.

The control group of students is 23 in number. They tackled two problems 6 weeks apart and the changes to process metrics calculated. There are 20 students in the treatment group of students. They tackled a different but similar pair of problems, again 6 weeks apart, but one year later. Again the changes in the process metrics are calculated for each student. For each process metric an analysis of the changes in the process metric measurements is carried out. The analysis uses the analysis of variance technique.

5.2.4 The Problems Used in the Experiment.

Two problems were given to each subject within the experiment.

Problem One: A set of data was read into a one-dimensional array. The maximum value, minimum value, average value and standard deviation are calculated and output

Problem two: A set of data was read into a table from a file. Each data item consisted of a row number, column number, variable 1 value, and variable 2 value. For each of the variables the row and column minimum and maximum are calculated together with the corresponding overall values.

As the experiment took place over two academic years there were different problems

for each group. The difference between the problems was the context in which the problem was set. In all other aspects the problems were the same. Details of the two problems are given in appendix A.

5.2.5 The Treatment.

The treatment group was given feedback during the development of their second program. This treatment was automatic testing of the program. For a program that completely satisfies the test criteria the treatment out put is shown in figure 5.1.

```
Passed test  1 : Program compiles
Passed test  2 : File accessed
Passed test  3 : Data read correctly
Passed test  4 : Data displayed
Passed test  5 : Calculate minimum east flows
Passed test  6 : Calculate minimum west flows
Passed test  7 : Calculate maximum east flows
Passed test  8 : Calculate maximum west flows
Passed test  9 : Calculate average east flows
Passed test 10 : Calculate average west flows
Passed test 11 : Calculate overall minimum east
Passed test 12 : Calculate overall minimum west
Passed test 13 : Calculate overall maximum east
Passed test 14 : Calculate overall maximum west
Passed test 15 : Calculate overall average east
Passed test 16 : Calculate overall average west
Passed 16 out of 16 tests
```

Figure 5.1 Sample output of treatment.

For students with less than perfect solutions the feedback only outputs the tests passed. This can be seen in table 5.2 below.

```
Passed test  1 : Program compiles
Passed test  2 : File accessed
Passed test  3 : Data read correctly
Passed test  4 : Data displayed
Passed test  5 : Calculate minimum east flows
Passed test  9 : Calculate average east flows
Passed test 10 : Calculate average west flows
Passed test 15 : Calculate overall average east
Passed test 16 : Calculate overall average west
Passed 9 out of 16 tests
```

Figure 5.2 Sample output for a novice with less than complete specification

5.2.6 The Computer Environment.

All students developed their programs in the same computer environment. Once the students edited the source file of their program they executed a command that:

- copied the program to an archive for future analysis
- compiled the program reporting any errors
- linked the program to produce a .exe program for those program that compiled
- executed the program and generated the results
- tidied up the student directory of intermediate files.

To allow feedback to be given both groups of students were provided with a procedure (`output_data(`*`parameters`*`)`) that was used to output the data and results. Students were required to use this library procedure in their program. There were two versions of this procedure available in the library. The control group of students used the normal version that output the results to the screen. The second version was used by the treatment group and again output the results to the screen. This second version also generated the tests passed output.

5.3 Results.

The results for the process metrics are collated and discussed individually. In all cases the null hypothesis is that the treatment has no effect on the PSDP employed by the novice. The alternative hypothesis is that the process was improved by the treatment. A significant result in the analysis indicates that the alternative hypothesis is true and the treatment improved the PSDP for that group of students.

The results for the five process metrics are considered in order. For each subject, the difference in the process metrics between the pre-test and the post-test was calculated. There are two opposing factors affecting this difference. It is affected by the expertise in PSDP gained by the student over the time between the two tests; this would lead to a positive difference between the pre and post metrics. The problem tackled in the post-test is more difficult than that in the pre test which would lead to a negative difference being recorded. If the null hypothesis is true and the feedback has had no effect then there will not be any difference between the two samples. If however the null hypothesis is false and the treatment has had an effect then the treatment group will have, on average, higher difference values than the control group. The improvement (or otherwise) of each metric was calculated for each subject and the values between the groups compared using an analysis of variance.

5.3.1 `cpi` Improvement.

The raw data of the difference in `cpi` scores for each subject is illustrated in the chart below, figure 5.3. The individual data items for the treatment group are shown at the top and for the control group at the bottom.

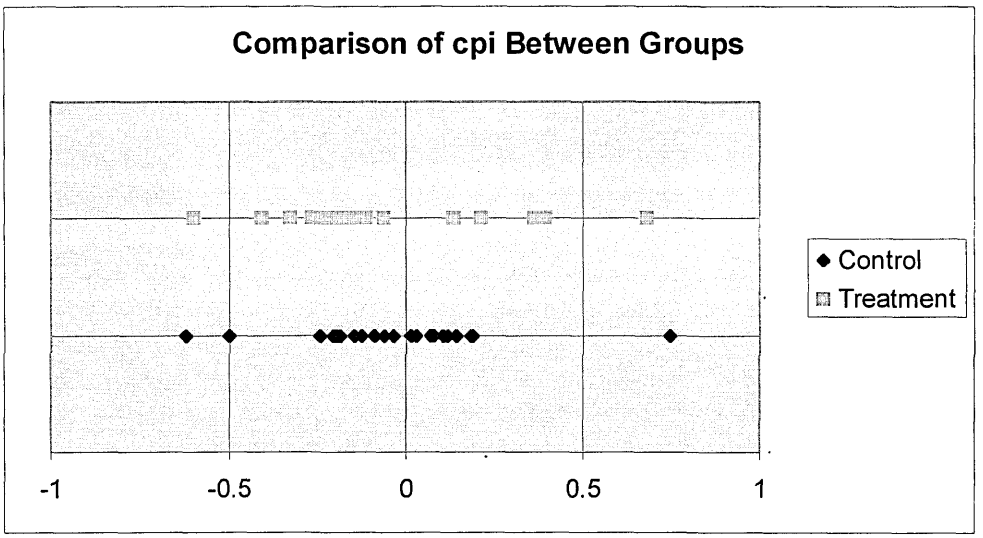

Figure 5.3 cpi Comparison of the control and treatment groups.

There is a clear overlap between the data items from the two groups. The clustering for the control group is at a higher value than that or the treatment group. A statistical analysis of the data shows whether the differences seen are significant.

Compiling Performance Index (cpi)

SUMMARY

Groups	*Count*	*Sum*	*Average*	*Variance*
Control	23	-1.08633	-0.04723	0.071419
Treatment	20	-1.654	-0.0827	0.090513

ANOVA

Source of Variation	*SS*	*df*	*MS*	*F*	*P-value*	*F crit*
Between Groups	0.013458	1	0.013458	0.167661	0.684331	4.078544
Within Groups	3.290962	41	0.080267			
Total	3.304419	42				

Table 5.1 Statistical analysis of the cpi metric

The results above show an F statistic value of 0.16766 that is well below the critical value of 4.078544 indicating that there is no significant difference between the two groups. The treatment has had no significant effect on the `cpi` metric.

The feedback to the subjects in this experiment provides information regarding the correctness of their program once it has compiled. It did not provide any extra feedback that can help the subject solve compilation errors. Thus it would be surprising if this metric showed any significant improvement between the groups. Indeed if there was a significant improvement in the treatment groups then the cause of this difference would not be the feedback and must have been some other external factor thus invalidating the experiment. As the result is not significant it provides reassurance that the experiment is well designed.

The average `cpi` difference for both groups is negative showing that on average the compiling performance was slightly worse between the pre test problem and the post test problem. It is likely that the reason for this apparent diminution in compiling performance by the subjects was that the post-test problem was more difficult than the pre-test problem. The statistical analysis above indicates that the diminution of the cpi figure is similar in both groups of subject.

5.3.2 Relative Run Performance Index (`rrpi`)

The relative run performance index (rrpi) measures the progress made towards the goal of a correct program at each version of development. This metric is affected by how long (how many steps) the subject takes to make progress in their program. This is the area targeted by the feedback given. The individual data items are shown in figure 5.4 below.

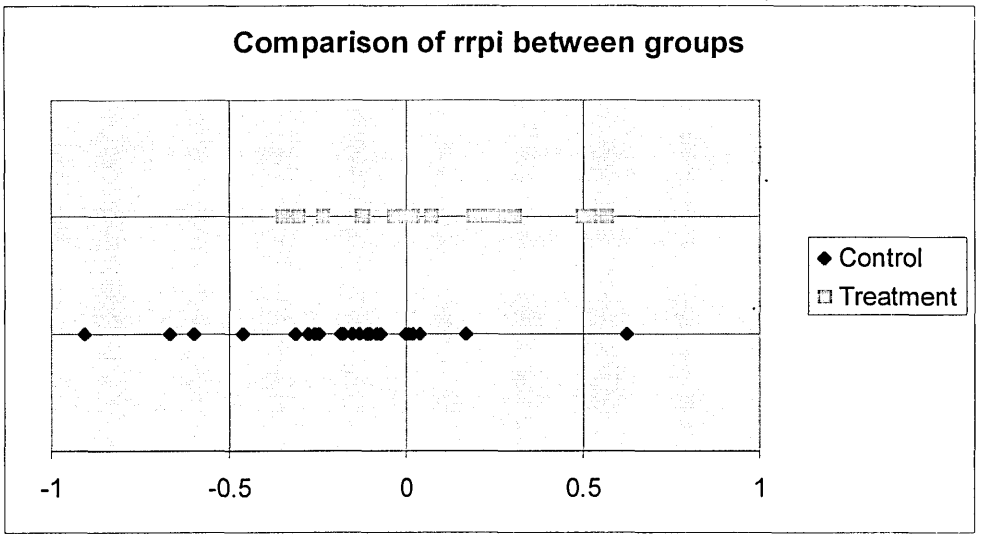

Figure 5.4 `rrpi` Comparison of the control and treatment groups.

There is a clear clustering of data in each group and moreover the clustering in the treatment group has shifted higher than in the control group. This would suggest, by inspection, that the treatment had a positive effect on this metric. Table 5.2 below gives a statistical analysis of the data.

Relative Run Performance Index (rrpi)

SUMMARY

Groups	*Count*	*Sum*	*Average*	*Variance*
Control	23	-3.95311	-0.17187	0.092239
Treatment	20	2.263975	0.113199	0.07073

ANOVA

Source of Variation	*SS*	*df*	*MS*	*F*	*P-value*	*F crit*
Between Groups	0.869365	1	0.869365	10.56707	0.002304	4.078544
Within Groups	3.373118	41	0.082271			
Total	4.242483	42				

Table 5.2 Statistical analysis of rrpi metric.

The results above show an F statistic value of 10.56707 that is well above the critical value of 4.078544 indicating that there was a significant difference between the two

groups. The treatment has had a significant effect on the `rrpi` metric.

The average `rrpi` difference for the control group was negative, which indicates that for this group of subjects, their `rrpi` metric value lowered over the time of the experiment. One explanation of this point is that the second problem was harder than the first and the students found it more difficult to make progress with their second problem. It does not indicate that the programming ability of the control group students reduced over the period of the experiment just that they were unable to cope as well with the second more complex problem. The treatment group showed a positive increase in their `rrpi` figure over the time of the experiment. These students coped better with the second program than the first. The only difference between these groups was the feedback given to the treatment group thus it can be concluded that the treatment had a highly significant positive effect on the students.

The treatment given provides feedback on program correctness but this does not tell the student how to develop their program. Nevertheless there was a measurable improvement in the process. One explanation of this is that the students did have the knowledge of how to develop programs but were less skilled in applying that knowledge. The feedback on the correctness helped them focus and apply their knowledge. A second, related explanation is that the students lacked the knowledge to test the program and the feedback that automatically tested the program provided this knowledge thus improving the PSDP.

5.3.3 Stages in Lines of Code (`sl`) and Stages in Correctness (`st`)

In chapter 3 stages in lines of code (`sl`) and stages in correctness (`st`) were

established as part of the metric set that measure the development of a program. One feature of these metrics is that the value (number of stages in lines of code or number of stages in correctness) is related to the program under development.

In this experiment the "expert" value of `sl` for the first program is 4 and for the second program is 7. Thus the expected change in `sl` is 3. Figure 5.3 below illustrates the spread of the values for the control and treatment group.

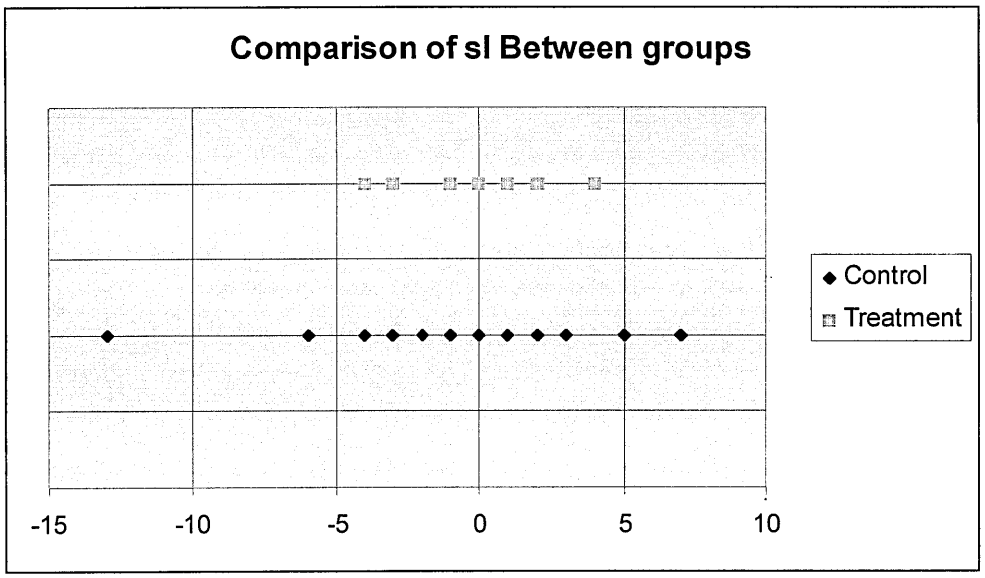

Figure 5.5 `sl` comparison of the control and treatment groups.

Stages in lines of code (sl)

SUMMARY

Groups	*Count*	*Sum*	*Average*	*Variance*
Control	23	-25	-1.08696	15.53755
Treatment	20	-17	-0.85	4.976316

ANOVA

Source of Variation	*SS*	*df*	*MS*	*F*	*P-value*	*F crit*
Between Groups	0.600657	1	0.600657	0.056435	0.813405	4.078544
Within Groups	436.3761	41	10.64332			
Total	436.9767	42				

Table 5.3 Statistical analysis for the sl metric

The results above show an F statistic value of 0.056435 with an associated probability of 0.813405 (81.34%) This indicates that there is no significant difference between the groups in the `sl` metric.

As noted previously the expected change for `sl` from an expert programmer is 3. In this experiment the mean change in `sl` is –1.087 for the control group and –0.85 for the treatment group. The statistical analysis summarised in table 5.3 reveals that the difference between the groups is not significant; the treatment had no effect on the `sl` mean value. It must be noted here that these average values show a reduction in the stages rather than an increase. An expert would have a `sl` figure of +3. One explanation for this is that the second more difficult problem has overwhelmed the students who have been unable to split the problem into stages properly. These students still lack the sophisticated problem solving skills available to expert programmers. The most notable feature of the illustration of the raw data is the

change in variation between the 2 groups. However this variation is not statistically significant.

The same analysis was carried out for the `st` metric. In this experiment the "expert" value of `st` for the first program is 4 and for the second program is 7. Thus the expected change in `st` is 3. Figure 5.4 below illustrates the spread of the values for the control and treatment group.

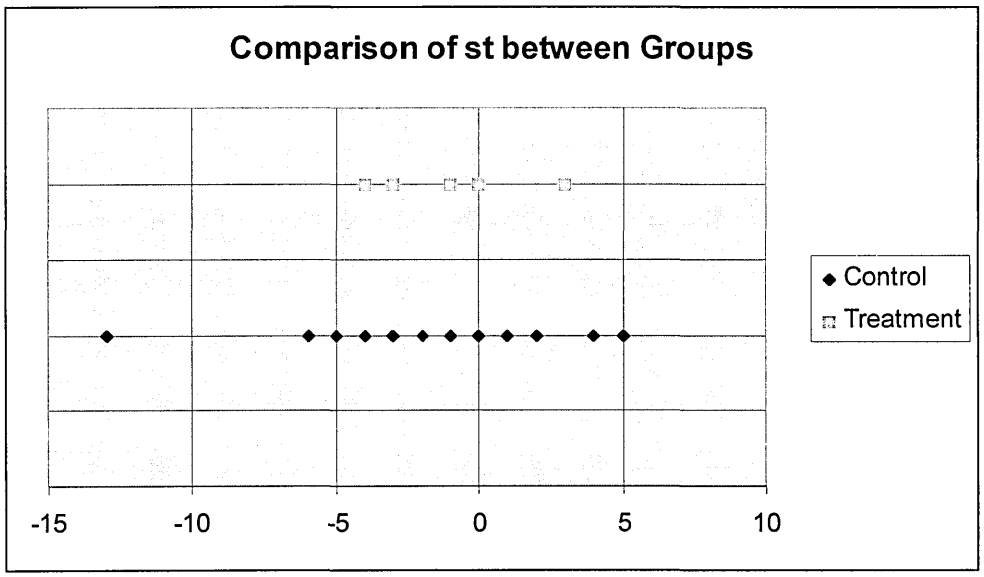

Figure 5.6 st comparison of the control and treatment groups

The diagram above has the similar characteristics as that for `sl` given in figure 5.3, namely that there does not look to be any change in the mean value between the two groups but the variation within the group is less for the treatment group. An analysis of variance was carried out on the data. The results are shown in table 5.5.

Stages in testing (st)

SUMMARY

Groups	*Count*	*Sum*	*Average*	*Variance*
Control	23	-42	-1.82609	15.05929
treatment	20	-29	-1.45	4.681579

ANOVA

Source of Variation	*SS*	*df*	*MS*	*F*	*P-value*	*F crit*
Between Groups	1.513094	1	1.513094	0.147617	0.702808	4.078544
Within Groups	420.2543	41	10.25011			
Total	421.7674	42				

Table 5.4 Statistical analysis for the st metric

The results above show an F statistic value of 0.147617 with an associated probability of 0.702808 (70.28%). This indicates that there is no significant difference between the groups in the `st` metric.

These figures show that the average improvement in `st` for the control group is –1.826 and for the treatment group is –1.45. The statistical analysis above indicates that there is no significant difference between these mean values. As is the case with the `sl` metric, the figure for an expert programmer would be expected to be +3. However both these groups do not achieve this figure indicating that the subject group are not yet experts. The difference between the variances of the two groups is also not significant.

The results for the metrics `sl` and `st` are as expected. The treatment gave simple feedback indicating the correctness of the program. These metrics measure how the novice split the program into development stages. This was not expected to be altered by the feedback and indeed it was not.

5.3.4 Relative Total Performance Index (`rtpi`)

The final metric in the set that measures the PSDP is the relative total process index (`rtpi`). Figure 5.5 shows the raw data from the experiment.

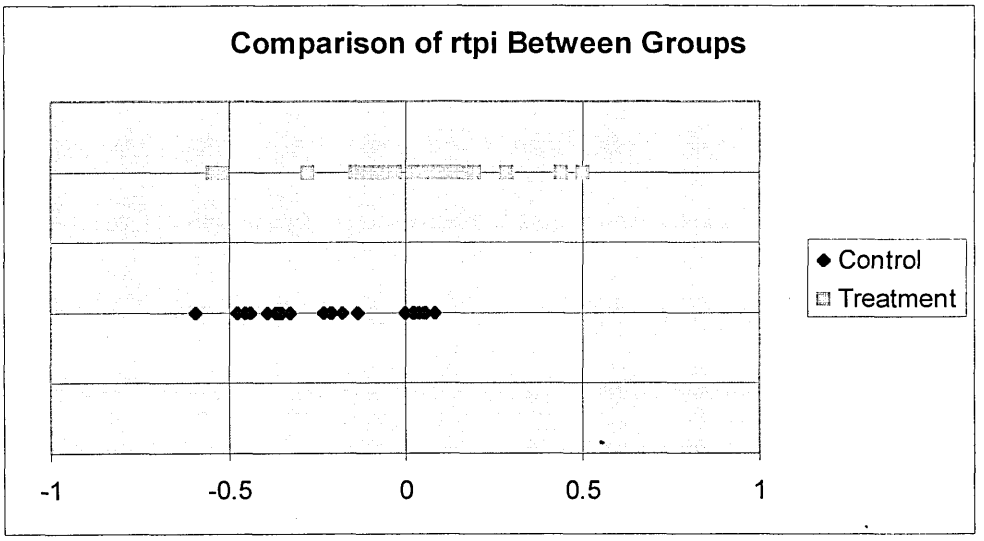

Figure 5.7 rtpi comparison of the control and treatment groups.

The data shown above shows that the clustering of the treatment values was "higher" than that for the control group. There is also some visual evidence that there was a greater spread in the treatment group rather than the control group. A statistical analysis of the data is shown in table 5.6 below.

Relative Total Performance Index (rtpi)

SUMMARY

Groups	*Count*	*Sum*	*Average*	*Variance*
Control	22	-4.68181	-0.21281	0.042423
Treatment	21	0.70545	0.033593	0.07734

ANOVA

Source of Variation	*SS*	*df*	*MS*	*F*	*P-value*	*F crit*
Between Groups	0.652324	1	0.652324	10.97157	0.001939	4.078544
Within Groups	2.437688	41	0.059456			
Total	3.090011	42				

Table 5.5 rtpi comparison of the control and treatment groups

The results above show an F statistic value of 10.97157 which when compared to the F value of 4.078544 indicates that this is a significant result. There is a significant difference between the treatment group and the control group in the improvement in the `rtpi` metric.

The table above shows that there is a significant difference in mean between the two groups of subjects showing that the treatment made a significant improvement to the process metrics for the treatment group of students. This result mirrors the result for the `rrpi` metric confirming that the treatment has improved the PSDP employed by the subjects.

5.4 Conclusion.

The experiment reported above has tested the hypothesis that providing feedback during the PSDP can have a positive impact on the students' learning. The five metrics developed in chapter 3 were used to measure the differences in the PSDP of the students taking part in the experiment. The students were split into a control group and a treatment group. The experiment was designed so that the only difference between the two groups was the feedback given to the treatment group. The feedback given to the treatment group of students involved automatically testing the program and reporting the program correctness. Data was collected for the two groups of students and analysed. The results from the experiment are summarised below:

- The treatment gave the students feedback on the correctness of their program and as such it was not expected to influence the compiling performance metric (`cpi`). This indeed was the case; there was no significant change in the `cpi` figure between the two groups of students.
- The treatment focussing on the correctness was intended to provide information that the student could use to improve their program development. The relative run performance index (`rrpi`) measures how many steps have made progress and hence should have been affected by the treatment. This indeed was the case with a highly significant change in the mean `rrpi` figure for the treatment group over the control group. This indicates that the treatment had a beneficial effect on the subjects.
- The metrics `sl`, steps in lines of code, or `st`, steps in testing, were designed to measure how many stages the student used to develop their program. This is not a

factor that should be affected by the treatment. The results show that there was no significant change in either of these metrics between the control and treatment groups.

- The final metric developed in chapter 3 measures the progress of each step over the previous steps. This is the relative total progress index, `rtpi`. The results show a highly significant improvement in this metric for the treatment group over the control group.

The results show a consistency in that significant improvements were made to the run performance metrics (`rrpi` and `rtpi`) but no significant change was made to the compiling performance (`cpi`) or the stages in development metrics (`sl` or `st`).

The treatment does no more than report on the correctness of the program. Thus it is reasonable to argue that the treatment merely provided the subjects with some insight into the correctness of the program. However the results show that this treatment has a significant effect on the PSDP used by the subjects. It has been argued that the treatment did not give the subjects knowledge of how to develop their program, for by the very nature of the feedback this is unrealistic. However it is concluded that the treatment allowed the subjects to apply what knowledge they had concerning how to develop programs. A fuller discussion of the implications of these results to the understanding of the knowledge acquisition of novice programmers is given in chapter 6.

5.5 Future Feedback Work.

5.5.1 Introduction.

The experiment reported in the previous sections has established that improvements can be made to a novice's PSDP by giving automatic feedback during the development of the program. The feedback given in these experiments was based on the progress of the program towards correctness. This was chosen because it was seen as relevant, accessible and simple to deliver. There is no claim that this feedback is optimal in any sense. The next stage in the development of this research is to investigate the potential for feedback in improving the PSDP for novices.

The feedback given in the previous section proved to be beneficial to novices. A number of questions follow from this experiment:

- Is the improvement found due to the feedback related to the semantic content or would any supportive feedback have a positive effect? This question addresses the issue of whether there is a placebo effect in the feedback. This is a possibility as the feedback given in section 5.4 was only be relevant to a subset of the subjects within the experiment. Novices who did not obtaine an error free compilation were not helped from the feedback.
- Assuming that the semantic content of the feedback is significant then the next question to be addressed is : what is the optimal feedback for a given individual? This question is difficult to answer and the investigation will look for feedback giving a "better" rather than an "optimal" result.
- The feedback given in section 5.4 was given to all novices independent of their classification and was not specifically targeted. The third question to be addressed here is whether feedback targeted at an individual is necessary or are

the individuals able to select their feedback from a range given.

This selection proposes experiments that will explore these issues.

The experiment reported in section 5.4 found a significant improvement in the process metrics when novices were given feedback relating to the completeness of their program. This feedback produced a distinct improvement in the process metrics. Different feedback scenarios are considered in this section. The experiments proposed here are designed to look for an improvement in the process metrics as least as large as the effect found in section 5.4.

The completeness feedback of section 5.4 could only help novices who achieved a clean compilation of the program. It allowed these novices to measure where they were on the development path by displaying the correctness of a program in terms of a percentage finished figure. The feedback did not help those novices who had difficulty in compiling a program nor helped a novice improve their PSDP by suggesting a next stage in the development. The proposed experiments discussed here will examine the issue of targeted feedback to investigate whether specific feedback can achieve a better improvement in a novice's PSDP that the simpler correctness feedback employed in the previously reported experiments.

5.5.2 Experimental Overview.

The first effect to investigate is to what extent it is the feedback content rather than it's existence that has a positive effect on the novice's PSDP. Computer systems can be characterised as giving no response if an action was successful and a usually long

and complex error message if the computer encountered a problem. The action of providing some positive feedback may have a positive effect on the novice thus improving the PSDP i.e. is there a placebo effect generated by the feedback mechanism itself rather than the feedback content? It is this effect that will be investigated first. Whilst it is expected that the "null" feedback will have some effect on the novice programmer it is not expected that this feedback has such a positive effect at the "correctness" feedback reported earlier. If this is so then there is a need to consider targeted feedback in order to seek to maximise thePSDP improvement.

There are three categories of novice identified earlier in this thesis, labelled B, C, and D, classified by their behaviour during the development of a program from their PSDP. The important features that determine this classification is their skill in using the programming language and skill in problem solving. It seems reasonable that the feedback requirement for a novice in one category of novice are different to that required in another category. Novices in category D have difficulty in compiling programs and feedback tuned to this requirement may have a greater effect than the "correctness" feedback. Novices in category C are able to compile code but have difficulty in devising a solution strategy for the given problem so guidance in the next step in the program development is indicated. Category B novices require fine-tuning to their problem solving skills to bring them up to expert levels. An extensive set of experiments is required to test the effectiveness of each feedback mechanism on each novice category and to show the effectiveness or otherwise of the feedback across the range of novices. The issues involved in such experiments are discussed in section 5.5.5.

Providing feedback in the form discussed above, feedback tailored to an individual novice's need is difficult to deliver since it is necessary to identify the category to which the novice belongs prior to the delivery of feedback. A simpler would be to offer all three forms of feedback to all novices and let the individual choose which feedback they use. Moreover the improvement looked for in these novices is an improvement greater than that found with the "correctness" feedback for any lesser effect would mean this treatment being superseded by the "correctness" treatment.

5.5.3. Experimental Environment.

The experiments outlined above are designed to investigate the effect of a treatment. To do so it is necessary to analyse the differences between treatment groups and a control group, as was carried out in section 5.4. For this comparison to be valid it must be reasonable to argue that there is no in built bias in the experiment and that the control and treatment groups are comparable. Additionally it is important to address ethical issues of experiments involving people.

The experimental method of section 5.4 used one cohort of students as a control group and a second cohort of students as the treatment group. The disadvantage of such a method is that it will take two years to carry out any experiment and if three alternative treatments were being investigated then this would take four years to complete. Although it can reasonably be argued that two consecutive cohorts of students are comparable, there may be many changes (change in entrance qualifications, course structure, lecturer etc.) outwith the control of the experimenter that can occur over four years. Hence the methods used in section 5.4 cannot practically be used to discriminate amongst several treatments.

The advantages of using multiple cohorts are that the educational experience for each student in the cohort would be the same. The aim of education is to do the best for each student. Using student experience for experimental purposes could compromise this. However if each cohort were given an identical treatment and the experimentation was designed to improve their experience then experimentation with student subjects would not compromise their education. This was the argument used in the earlier experiments reported in this chapter. It is possible to compare two treatments using these methods but a quicker result could be obtained using a single cohort split between control and treatment groups. However there are potential dangers in this approach in that the control group could be given a lesser educational experience than the treatment group, or if the treatment proved disastrous, the treatment group could argue that they were disadvantaged. Hence any testing of novices from a single cohort cannot be carried out as part of their normal educational experience though could be carried out in addition to the normal classroom activity.

The feedback experiments, section 5.4, permitted the students to develop programs in their own time. This can only be allowed if a single cohort is given a single treatment since there is a possibility of interaction between the groups. If a single group of novices were to be used for an experiment that is in addition to their classroom activity then, to control interaction between the novices, the experiments must be undertaken in a controlled environment. The controlled environment gives an unnatural environment in which to develop a computer program and this strangeness may itself affect the PSDP. It is concluded that this is the only way to carry out these experiments in a valid way and so to cater for this problem of unfamiliarity with the artificial controlled environment some prior experience. It is argued that the

unfamiliar surroundings are the same for the control and treatment groups and will not affect the analysis of the experimental data.

Finally there is the issue of number of novices required for the experimentation.

		Treatment actual effect	
		No Effect	Effective
Results indicate	No Effect	**OK**	**Type 1 error**
	Effective	**Type 2 error**	**OK**

Table 5.6 Possibilities from an experiment analysed by statistics.

Table 5.6 summarises the possible outcomes of any experiment that investigates the effect of a treatment and analysed by statistics. There are two types of error that must be considered in the design of such experiments. Type one errors are when there is actually an effect but the experiment was not powerful enough to measure this effect. Type two errors are when an effect was concluded but there was not an actual effect. In the experimental design there must be a balance between these two error types as they are related; reducing the probability of concluding that there was an effect when there was no effect makes the experiment less powerful and increases the probability of detecting a real effect. In the proposed experiments the type two error will be taken to be 5%, the normal figure used in educational research. The power of an experiment is defined as the minimum treatment effect that can be detected by the experiment. This is affected by the mean and standard deviation of the data and the size of the sample. The experiment reported in section 5.4 showed an improvement in the `cpi` metric of 0.04 and, by assuming similar mean and standard deviation values, the calculated minimum number of subjects in each group is twelve. (A similar figure is achieved by analysing the effect on the other metrics).

```
c * m = sqrt(var / n)
```
where
- c is the critical significance value at 5%
- m is the smallest treatment effect that can be detected.
- v variance of the sample
- n number of subjects in the sample

Table 5.7 Calculation of minimum sample size

Taking all these issues into account the following points will need to be incorporated in the design of the experiments:

- Each experiment will be undertaken within a single cohort of students to ensure homogeneous control and treatment groups.
- Novices will be allocated to treatment and control groups randomly.
- Experiments will take place in a controlled environment
- Experiments will be extra to normal course of study for these students.
- At least 12 students will be allocated to each group to ensure that the statistical tests used are as powerful as that used in the previous work.

5.5.4 Potential Feedback.

This section will discuss the nature of the feedback to be tested in these proposed experiments. The aim is for the feedback to be targeted at the particular needs of the class of novice. There are three categories of novice identified in earlier work in this thesis (labelled novice category B, C, and D) and it seems sensible to target feedback at each.

- Novice group D students are defined as those who have difficulty in getting a working program and need help with the syntax and use of programming languages.
- Novice group C students are defined as those that develop their programs using a single stage. That is, they attempt to get the whole program working in one

increment. They are confident with compiling programs but lack expertise in defining the stages in an incremental PSDP. These students need help in problem solving.

- Novice group B students are defined as those that exhibit some skills in using an incremental PSDP. They are able to develop programs incrementally but need help to improve their problem solving skills.

Group D Feedback: Group D novices are characterised as these who have difficulty in getting a working program. The feedback that these novices require is to help them to compile their programs; considerations of strategic development are not to the forefront in their minds. In an ideal situation the feedback given to these novices would capture the intention of the novice. This is the idea behind the PROST project, Johnson (1988). To achieve this intention and to provide tailored feedback is a major research development. The feedback considered here, labelled feedback X, is much less ambitious but more practical. The aim of this feedback is to focus the novice on the immediate problem, provide better information regarding the error and remove extraneous (subsequent errors) information. Only the first compilation error will be presented to the novice to help them focus on the main issue. Additionally some extra explanation and potential causes for the error will be given to the novice. Table 5.8 gives an example of the output when the first error is a missing semi-colon.

```
Prog.pas unexpected ; at line 85

Error indicates that there was a missing semi-colon.
In Pascal there must be a semi-colon between statements.
This error often indicates that there is a semi-colon missing at the
end of line 84.

Check carefully in the lines before line 85 for ; missing from the
end of the line.
```

Table 5.8 Example feedback given to group D novice.

The feedback will be accomplished by enhancing the command used by the novice to compile and run their program, the `pc` command. In addition to the source code being the output will also be intercepted and processed to give more "friendly" feedback.

Group C Feedback: Group C novices are able to get a program working but they do not follow the incremental development model. Typically they attempt to build the whole program in one stage. These novices need help in problem solving and in particular in defining the incremental nature of the development. To help these students acquire this technique a preferred incremental development is defined and imposed on the novice. The program development is defined by a series of stages for the PSDP with.each stage consisting of one or more tests to be passed. Novices are required to develop their program in the prescribed order of the stages. The aim of the feedback is to help the novice keep to the sequential order of the stages. The proposed feedback to be tested on these novices, labelled feedback Y, delivers to the novice a table indicating which tests they have passed and some hint as to which area they should concentrate on next. An example of the feedback given to these novices is outlined in table 5.9a. and 5.9b.

```
Stages Complete:
Stage 1 Test 1 : Program compiles
Stage 2 Test 1 : Opens database for reading OK
Stage 2 Test 2 : Input of sales data complete

Incomplete Stages
Stage 3              : Processing of transactions
Stage 3 Test 1  OK   : read transaction
Stage 3 Test 2 Error : process first transaction

You need to concentrate on:
Stage 3 test 2 : process first transaction
Before going onto the next stage of development.
```

Table 5.9a Example feedback to novice group C-1

```
Stages Complete:
Stage 1 Test 1 : Program compiles
Stage 2 Test 1 : Opens database for reading OK
Stage 2 Test 2 : Input of sales data complete
Stage 3 Test 1 : Read transaction
Stage 3 Test 2 : Process first transaction

Congratulations you have completed a stage in the development.
You should add the code for the next stage in the development.
The aim of the next stage is:
Stage 4 : Calculate average values
Stage 4 Test 1 : Calculate average order value.
Stage 4 Test 2 : Calculate average orders per day.
```

Table 5.9b Example feedback to novice group C - 2

This feedback can be delivered to a novice by automatically testing the program, in the same way as carried out in previous experiments, and building into the "system" knowledge of the expected order of steps and stages in the incremental model. The feedback can be generated from the current state of the program and does not depend on any previous state within the development.

Group B Feedback: Group B novices are able to get programs to compile and undertake some form of incremental development. The feedback targeted at these students, labelled feedback Z, is aimed at improving their insight into the incremental development technique. Instead of forcing a specific incremental value the novices are given support in their choice of incremental development. The textual feedback given to these novices will be similar to that given to group C novices in that it informs the novice of the tests passed within each stage. Additionally a diagram is presented which tracks the progress through the stages in the development of the program. It indicates the stages completed, stages attempted and stages not yet started. The feedback for these novices will be based partially on historical data. Stages or tests that have previously been passed and are currently not passed are noted as incomplete stages. This information helps the novice visualise the

sequencing of the PSDP and assist in cultivating their judgement.

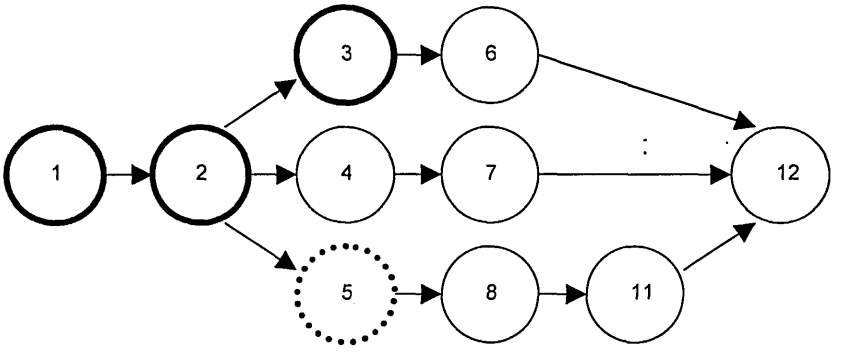

Figure 5.8 Diagrammatic output given to novice group B.

```
Stages Complete:
Stage 1 Test 1 : Program compiles
Stage 2 Test 1 : Opens database for reading OK
Stage 2 Test 2 : Input of sales data complete
Stage 3 Test 1 : Read transaction
Stage 3 Test 2 : Process first transaction

Incomplete Stages
Stage 5              : Calculation of Totals
Stage 5 Test 1  OK   : Order total
Stage 5 Test 2 Error : VAT total
Stage 5 Test 3 Error : Carriage Total
You need to concentrate on:
Stage 5 Test 2 Error : VAT total
Stage 5 Test 3 Error : Carriage Total
Before going onto the next stage of development.
```

Table 5.10 Example feedback to novice group B

This feedback needs historical data in order to detect where the novice previously passed a test and currently has not passed a test. This can be achieved by storing the tests passed in a log file that is appended to after each attempt at running the program. Access to this file can be obtained using the same mechanism as used in the previous experiments namely through a pre-written procedure.

5.5.5. Potential Experiments.

Three experiments are described here. The first attempts to see if there is a placebo effect due to feedback. The second looks at the effect of feedback tailored to the

individual student, and the third looks at the effect of applying all the feedback to all students.

Is there a placebo effect? The hypothesis to be tested here is whether the presence of any feedback in itself has a positive effect on the students unconnected to any significant semantic content. To investigate this a group of 24 students is needed split randomly between a control and a treatment group. Under controlled conditions each novice is asked to develop a program and at each stage the program is captured in the same way as reported earlier in this thesis. The treatment group is given feedback but this is not related to the state of their development. This feedback will be chosen from a bank of possibilities designed to offer a positive, encouraging and supportive atmosphere though unconnected with performance.

1. Informing the student that they are being paid for the experiment
2. Wishing the student good luck
3. Inform the student the time
4. Inform the student the time left during the experiment

After the experiment the process metrics are calculated from the data captured during the development. A comparison of metrics between the treatment and control groups can be carried out using two way analysis of variance according to Coolican (1999).

It is expected that there will be no significant effect on any of the metrics. Although it is possible that there is a small placebo effect it is expected to be less than the effect of the correctness feedback of section 5.4 and the experiment is not sensitive enough to pick up such small effects.

Tailored feedback. There are three types of feedback under investigation. For a full analysis giving the clearest possible view of the effects of the feedback, each type of feedback should to be tested on each category of novice and not just on the target category. The expected effects of the different feedback types are tabulated in table 5.11. By carrying out this experiment a great deal will be found out about the effect of feedback that may enhance our understanding of expertise in computer programming, see chapter 6. There are however a number of problems with this experimental design, from a practical point of view.

	Control Group	B Novices	C Novices	D Novices
Feedback X	required	none	none	improvement
Feedback Y	required	possible improvement	improvement	none
Feedback Z	required	improvement	possible improvement	none

Table 5.11 Showing the expected effect of the feedback on the groupings on novice.

As indicated in section 5.5.3, twelve novices are needed in a group to gain sufficiently sensitive data to detect a change due to the feedback of the same order to that detected in previous experiments. Hence for a full analysis of the effects of the three types of feedback there must be forty eight novices of each category and so one hundred and forty four novices in total.

The experiment as described above requires that the experimenter will know the category of novice prior to starting the experiment. This implies that some form of pre-test is required that will determine the category of a novice. This pre-test has advantages as it can be used to train the novices in the particular development environment to be used in the experiment bearing in mind that the experiment will

take place in a controlled environment. The output from the pre-test is that the experimenter must be able to identify forty eight novices within each category. From previous experience there are fewer students in category C, around 20% of all students. In order to ensure that there are forty eight students in category C then two hundred and forty students need to be invited for the pre-test. Even if the pre-test were not used the same number of students would be needed for the full experiment.

Two hundred and forty novices invited to a pre-test and one hundred and forty four asked to participate in the full experiment is a large investment in time and resources, a much larger scale experiment than used in the previous work. To justify such an experiment it is necessary to ensure that the results expected to come from the experiment are sufficiently worthwhile. In this case a large number of treatments are given to inappropriate novices, see table 5.11, for completeness, in the expectation that the feedback will have no effect. The information collected would be useful in building our knowledge concerning expertise in computer programming but add nothing to the practical task of improving the PSDP of novice programmers. A smaller and more manageable experiment is to test each treatment on it's target group. This would be more limited in the information the experiment would provide. It would not be possible to conclude anything about the effect of a feedback treatment on it's non target group. However an analysis of the main effects of the treatment would be measurable and more importantly the number of novices required is reduced to seventy two, from a pre-test population of one hundred and twenty subjects.

To embark on either of these experiments involves a large investment in resources.

On balance, the benefits of measuring the improvement due to the specific feedback on its target category of novice, do not seem likely to be sufficient to outweigh the costs of the experiments. Thus no experiments are recommended that will look at the effect of each of these feedback methods on every category of novice. Rather this should remain as potential experiments until such a time as the need or value of conducting such a thorough test is established.

Non Specific Feedback: The experiment described above was looking at the effect of specific feedback on it's target novice category in the expectation that targeted feedback would be better in some way than non targeted feedback. However it is easier to provide the same feedback to all novices irrespective of their classification of novice. The experiment described here is of a smaller scale than above. The experiment investigates the effectiveness of providing all types of feedback, X, Y and Z to all novices. It is expected that the results from this experiment will have an effect at least as big as the effect found in section 5.4 as this was the approach used there and the feedback is similar but more extensive nature.

The hypothesis to be tested here is whether novices can improve by picking the appropriate feedback from amongst that given to them. This is investigated using a similar experimental design to that used to investigate the placebo effect. The experiment requires a minimum of twenty four novices, although using more novices would make the experiment more powerful. The single group of students will be split equally between control and treatment groups. Under controlled conditions each group will be asked to develop a computer program. During the development of the program the novices in the treatment group will be offered feedback. All three types

of feedback will be available to each novice in the treatment group. On completion of the experiment process metrics will be calculated and the effectiveness of the treatment measures.

After the experiment it is intended to request that the novices fill out a questionnaire. The purpose of the questionnaire is to determine the novices views as to the usefulness of the feedback. Taking the post experiment questionnaire with the experimental results some pointers as to how the feedback can be tuned will be gained. It is at this stage when it may be necessary to revisit the previously described experiment to determine the effect of different feedback features on different categories of novice.

5.5.6 Summary of proposed experiments

- Each of the proposed experiments is to be undertaken within a single cohort of students, in a controlled environment.
- At least twelve students are needed within each treatment or control group to detect an effect that is a least as marked as that found in the experimental work of this chapter.
- The first experiment will investigate the placebo effect of giving feedback and will be used as a prototype for the experiment proposed later.
- An experiment to investigate the effect of targeted feedback on the target novice category is of a larger scale than has been used so far in this research.
- A second experiment will examine the effect of offering all three types of feedback to the novices as a whole. Whilst it is expected that providing non specific feedback is not as good as providing specific feedback it is practically

easier to deliver and more feasible to test and, the difference in effect may be slight.

5.6 Summary.

This chapter has discussed an experiment that has show the positive effect feedback has had on a group of students. The chapter has then discussed potential experiments that are able to examine the effect of different feedback mechanisms. The main points of this chapter are:

- An experiment was completed that gave feedback in the form of correctness to novice programmers.
- Analysis of the results showed that the PSDP improved in terms of the `rrpi` and `rtpi` metrics.
- The feedback did not improve the `cpi` metric as expected.
- A future experiment is described that is designed to show the effect of three further types of feedback aimed at improving the PSDP employed by novices.

6. The Process – Product Relationship.

6.1 Introduction.

The basis of all work on quality is the principle that improvements to the process lead to a better product. The goal for any engineering discipline is to build a product that meets various quality thresholds. In many engineering disciplines this is achieved by carefully controlling the processes and procedures used in the development of the product. This is indeed the basic premise behind the software engineering movement. In order to improve the quality of computer software, procedures were put into place that sought to improve the software development process and thus the quality of the final product. However this premise that a quality process inevitably leads to a quality product is often stated but not widely tested. The main reason that the premise has not been tested is that there has been a lack of reliable metrics that measure the quality of the software development process. This research has investigated the way novice programmers develop programs and has established a set of PSDP metrics for novices. Hence in this restricted area, that of novice programmers, there is the opportunity to test the hypothesis that a quality process leads to a quality product.

In the course of this research, the three elements are available to allow the correlation between the product and process to be investigated. Data has been collected from novice programmers during their development of the code. A set of metrics has been established that measures the quality of this development process. In reported work

such as Ceilidh (Benford et.al. 1993a) an established method of objectively measuring the quality of a novice computer program is available.. Hence the relationship between the process and product metrics for novice programmers can be investigated. This investigation has two phases. The initial phase is to draw scatter plots to visualise the relationship between the process and product metrics. A more objective view of the relationship between the process and product metrics is gained by calculating the correlation coefficients between them. The hypothesis being tested in this chapter is that a quality process leads to a quality product for the development of computer programs by novice programmers. This hypothesis is the application of the quality principle to novice computer programmers.

6.2 Product Metrics.

Much work has been carried out in producing workable criteria and metrics that are used in the automatic calculation of quality of a novice program, see section 2.2. The method and criteria used in this research is based on that used in the Ceilidh system (Benford et.al. 1993). This popular system is used in a number of Universities in the UK for the automatic the marking of student programs. Human marking of computer programs is subjective in nature and Ceilidh has introduced objective metrics as a proxy for these subjective features.

Table 6.1 below gives the five criteria, taken from the Ceilidh project, used as the basis of the measurement of a novice program.

Area	Symbol	Name	Description
Static Quality	pl	Program Layout	Measure of how well a program is presented on the page
	pd	Program Design	Measure of how well the code is written
	c	Complexity	Measure of the program complexity
Dynamic Quality	sp	Specification Proportion	Proportion of the specification that has been satisfied
	e	Efficiency	Efficiency of the program

Table 6.1 Criteria used to measure novice programs.

6.2.1 Program Layout.

The factors used to measure the style or layout of a program were taken from previous work of Rees (1982), and Meekings (1983) and incorporated into Ceilidh. The factors used in this research are given in table 6.2 below.

Program Layout			
Symbol	**Name**		**Notes**
l_indent	Proportion of indented lines	A	An indented line is one which is indented from the previous code line
l_blank	Proportion of blank lines	A	A blank line is one with no code nor comments
l_comm	Proportion of comment lines	A	A comment line is one where is no code.
l_chr	Average number of characters per line	A	Only counted within code lines.
l_spc	Average number of spaces per line	A	Only counts spaces within code lines.
l_id_len	Average length of identifiers	A	Reserved words and special symbols are ignored in this calculation
l_p_id	Proportion of identifiers with good length	A	Good length is defined as between 6 and 8 characters.

Table 6.2 Factors making up the program layout component of a novice program metric.

Note that in table 6.2 above and 6.3 later the third column indicates whether the

factor is absolute (A) and can be compared to some global value or, relative (R) where the value depends on the problem in question. The style factors shown above are all absolute values and do not depend on the problem under construction.

These factors above are converted into marks using a trapezium function according to the methods of Rees 1982 and described in figure 2.1 in section 2.2. The various marks are then averaged to generate an overall mark for the program layout.

6.2.2 Program Design.

The program design criteria attempt to measure how the program was constructed, whether the programmer used established rules of good programming practice. These factors must not be confused with program layout or style that is independently assessed. The program design is perhaps the most subjective area to measure. The objective factors used in Ceilidh are based on a series of counts put forward by Rees 1982. The design quality is measured by comparing the novice program with an "expert" or "model" program. The factors used are counts of various features of the program. For example, a count is made of the number of procedures in the program. The argument being that if this number matches the "expert" solution then this factor in the novice program is well designed, if it veers away from the expert solution then this factor of the design is of a lesser quality. Whilst many may question whether the amalgamation of these factors measures the design quality in the way they expect, the measurements are accessible and easy to calculate.

Program Design			
Symbol	**Name**		**Notes**
d_mod_len	Average module length	A	Measure in terms of code lines. A code line is any line with executable code on it.
d_p_mod	Proportion of modules with good length	A	
d_n_for	Number of fixed loops	R	Number of for loops
d_n_loop	Number of variable loops	R	Includes while and repeat loops
d_n_if	Number of if statements	R	
d_n_case	Number of case statements	R	
d_n_proc	Number of procedures	R	
d_n_func	Number of functions	R	
d_n_id	Number of identifiers	R	

Table 6.3 Factors making up the program design component of novice program quality.

The overall measure of program design quality is obtained using the trapezium function to compare each factor to a "model" value. These individual values are then averaged to obtain the overall metric.

6.2.3 Complexity.

The methods to measure program complexity have been discussed in section 2.2. The measurement of complexity used here is McCabe's 1976 cyclomatic complexity. It is clear from the literature that the validity of this and other complexity metrics is open to question. It is not the purpose of this research to generate a new complexity metric but to use established work. Ceilidh uses the same complexity measure.

The complexity metric for any program is related to the problem. In the experiments undertaken here the problems do not involve difficult algorithm designs and hence differences in complexity is not an important issue in defining the quality of the final program.

6.2.4 Specification Proportion.

The specification proportion measures the proportion of the specification that has correctly been completed. It is a measure of the correctness of a program. The specification proportion is based on a set of test data. A program is deemed to be 100% correct if it passes all the tests in the test data bank. The specification proportion (`sp`) is calculated from those tests that are correctly executed out of the complete set

`sp = np / nt` where
`np` = Number of tests completed satisfactorily
`nt` = Total number of tests in test data bank

6.2.5 Efficiency.

The efficiency of a program involves quality factors such as speed and space required for a program. The programs considered during this research are very fast to run and differences in speed are slight. Space requirements for all programs are similar. In the modern climate with fast processors and cheap memory this factor is less important decades ago when Rees carried out his original work. Ceilidh does include this factor but it is given small weighting when the single mark is calculated. For these reasons this criteria is not used in calculating the product metrics in this research.

The table below gives a summary of the product metrics used in the research reported here:

Name	Symbol	Definition
Program Layout	pl	Average of 7 factors
Program Design	pd	Average of 9 factors
Complexity	c	McCabe cyclomatic complexity
Specification Proportion	sp	sp = np / nt where np = Number of tests completed satisfactorily nt = Total number of tests
Efficiency	e	Not Used Here

Table 6.4 summary of product metrics used on novice programs.

6.3 Process-Product Correlation.

The initial step in investigating the process – product correlation is to visualise the data. To do this a single measurement of the program quality and a single measurement of the PSDP are used. The single measurement of the program quality is gained from the weighted average of the criteria discussed above.

```
program_quality = 0.25*pl + 0.25*pd + 0.2*c + 0.3*sp
```

where
pl – program layout
pd – program design
c - complexity
sp – specification proportion

This method of weighting the various aspects of the program quality mark is widely used in automatic marking schemes, for example Ceilidh (Benford et. al. 1993). The relative values of the weights can be used to bias the marking to a specific aspect of quality that the assessment is addressing. In this experiment an overall program quality mark is required and hence the near equal weighting to each attribute is given. Complexity is not an issue with the programs under investigation here and is

weighted less than the average, correctness is an issue and is weighted more than the average. A sensitivity analysis of the weights showed that the program quality mark is not sensitive to small (less than 0.2) changes in the weights.

The single measure of the process metric is that discussed in section 4.7 being derived from the weighted average of the individual process metrics.

```
process_quality =
  0.25*cpi + 0.25*rrpi + 0.125*sl' + 0.125*st' + 0.25*rtpi
```

where

`cpi` – compile progress index
`rrpi` – relative run performance index
`sl'` - amended stages in lines of code
`st'` – amended stages in testing
`rtpi` – relative total progress index

The values `sl'` and `st'` are calculated from `sl` and `st` respectively. The value of `st` for a novice is compared to that of an expert and the amended `st'` value generated using a trapezium function as described in section 4.7.

These values are calculated for all the subjects used in this research and a scatter plot drawn of process quality measurement against product quality measurement, shown in figure 6.1.

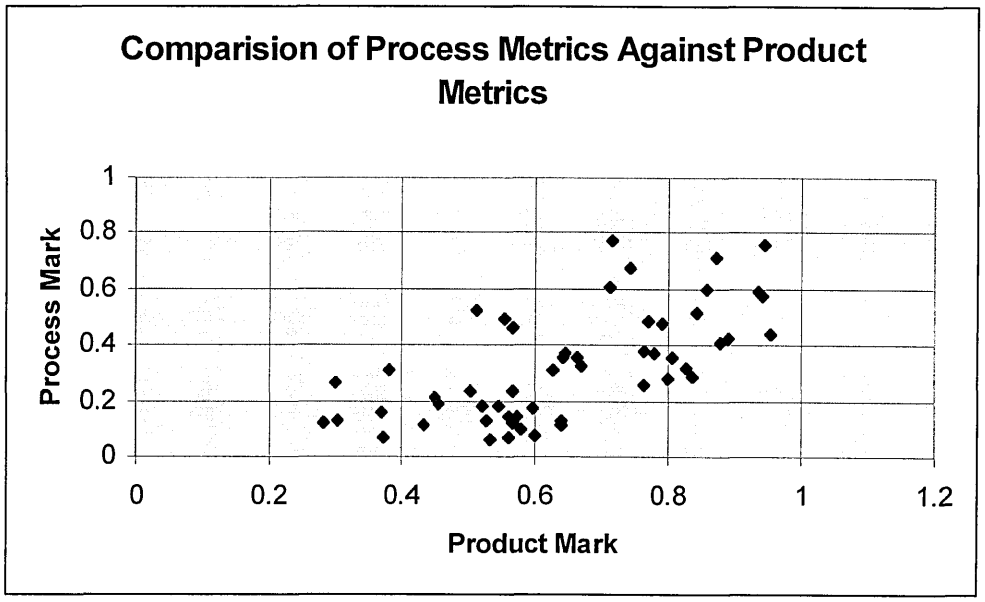

Figure 6.1 Scatter plot of process and product metrics.

The data in the scatter plot above is spread over a wide area of the plot indicating that the correlation between process and product is not very strong. However the points do appear to follow a (weak) trend from bottom left to top right. This indicates that there is some correlation between the values.

Correlation	Combined Process Metric
Combined Product Metric	0.623
Combined Product Metric without correctness	0.044

Table 6.5 Combined process and product correlation.

The process / product corelations are calculated and shown in table 6.5 above. It shows a correlation coefficient of 0.623 between the process mark and the product mark. As there are forty subjects a correlation figure above 0.267 would indicate a significant relationship between the factors. The correlation figure between product and process mark is well above this figure indicating a strong relationship between the two factors.

However this analysis is naive. The correctness (sp) is a component in the product mark and also used in the definition of three of the five process factors. Thus a correlation would be expected. If the correctness is taken out as a component of the product mark then the correlation values is re-calculated as 0.044 which is a low correlation indicating no correlation between the process and product metrics.

The next stage in the analysis is to look at the individual factors. There is a problem with the independence of some of the factors. In particular the definition of relative run performance index (rrpi) includes the specification proportion (sp) (rrpi = rpi * sp). Similarly the relative total progress index (rtpi) also includes the specification proportion in its definition (rtpi = tpi * sp). Hence in the search for correlation both rrpi and rpi, rtpi and tpi have been included. The correlation is calculated between each of the product and process factors. Table 6.6 below gives the correlation. Note again that with 40 subjects any correlation above 0.267 is significant at the 5% significance level.

		pl	pd	c	sp
Compiling process indicator	**cpi**	0.092	-0.076	0.039	0.471
Relative run progress indicator	**rrpi**	-0.093	-0.060	-0.020	0.608
Run progress indicator	**rpi**	0.270	-0.166	0.165	0.284
Stages in l.o.c	**sl**	-0.094	0.134	0.013	0.530
Stages in testing	**st**	-0.067	0.117	0.028	0.629
Relative total progress indicator	**rtpi**	-0.073	0.018	0.082	0.886
Total progress indicator	**tpi**	-0.129	0.005	0.043	0.195

Table 6.6 Process - product correlation. (bold figures indicate significant results)

These results show:

- the expected high correlation between the specification proportion (`sp`) and the two process metrics, relative run progress index (`rrpi`), and the relative total progress index (`rtpi`). This is because of the involvement of `sp` in the definition of the metrics.
- a significant correlation between `sp` and the compiling performance index (`cpi`) metric. This is interesting because these metrics are totally independent. The novices that show greater knowledge of how to achieve a correctly compiled program tend to achieve a more complete program. This is perhaps obvious when stated in the opposite way; novices that have low skill in compiling programs tend not to get correct programs at the end
- a significant correlation between `sp` and the stage metrics (`sl` and `st`). This shows that novices that show evidence of a staged development tend to get a more correct program than those who do not. This is an important result as it shows, perhaps for the first time, that novice programmers who have some strategy for solving the problem tend to get a better final result. This is intuitive but the evidence has now been given to support this view.
- a significant correlation between the run progress indicator and the specification proportion. This shows a link between this aspect of the process and the correctness of the final product. Those subjects that develop the program well tend to get a more complete program. However this relationship is not strong only just significant 0.286 as against a critical value of 0.267.
- There is no significant correlation between the total progress indicator and the specification proportion. This is a contrary result than that found above.
- there is only one correlation between any of the process metrics and the product

style (`rpi` and `pl`). This single correlation is marginal and may well be down to chance. The lack of correlation between the process and product metrics confirms the results of Parrish et.al. (1997). They used numbers of compiles and time taken as their process metrics and assessor based mark as their product metric. They also failed to find any correlation between their process and product metrics.

6.4 Intra Process Correlation.

The previous section examined the correlation between the process metrics and the product metrics. The experimental data also gives the opportunity to look at the relationship between the process metrics. Table 6.7 below gives the correlation between the individual process metrics. Again as there are 40 subjects under investigation any correlation above 0.267 is regarded as significant at the 5% level.

		cpi	rrpi	sl	st	rtpi
Compiling process indicator	cpi	1.0	0.278	0.280	0.376	0.468
Relative run progress indicator	rrpi		1.0	0.161	0.233	0.861
Stages in l.o.c	sl			1.0	0.972	0.468
Stages in testing	st				1.0	0.555
Relative total progress indicator	rtpi					1.0

Table 6.7 Process metric correlation.

The process metrics measure three different aspects of the PSDP.

- `cpi` measures the compiling aspect
- `rrpi` measures the "getting a program to run" aspect

- `sl` and `st` measure the planning aspect.

The metric `rtpi` combines the first and second aspect in some "combined" metric.

The table of results show:

- a significant correlation between `cpi` and the other metrics. One possible explanation of this is that those novices who have a good command of the compiling aspect of the programming development process tend to have a greater understanding of the other aspects of program development. Another way to look at this is that novices who have difficulty with the compiler need to concentrate on the compiler and language aspects of program development to the detriment of the other aspects of program development.
- a non significant relationship between the relative run performance indicator, `rrpi`, and the stages metrics (`sl` and `st`). An interpretation of this is these metrics measure different skills which are independent.
- a positive and significant correlation between `sl` and `st` against `rtpi`. This shows that those students who strategically manage their program development are also those who make the most progress at each step in the development. Those novices who do not plan the development well also find the intra-stage development difficult.

There are also a number of false correlations caused by the mutually dependent definitions of the metrics. (`st` and `sl`; `rrpi` and `rtpi`; `cpi` and `rtpi`)

These results show that mastery of the programming language, as measured by `cpi`, is a key skill needed prior to learning any sophisticated problem solving skills. If novice is not familiar with the programming language then they are not able to

develop programs in that language. However for those novices who have skill in the syntax of the programming language there are two independent problem solving skills to be learnt. The first is the ability to split the problem into easily manageable stages. The second is to be able to develop efficiently each of these stages. The acquisition of knowledge of these two aspects of programming appears to be independent.

These results provide some data that can be interpreted in terms of the knowledge needed to become an expert programmer. This will be further discussed in chapter 6.

6.5 Conclusions.

Three important results have been established in this chapter:

- A positive correlation has been shown between process metrics and the final product correctness.
- No correlation has been shown between the program style, design and complexity and the process metrics.
- The ability to manage the program language (as measured by `cpi`) appears to be key to the development of a program.

The positive correlation between the set of process metrics and the final product correctness confirms the hypothesis that a quality process leads to correct product. Previously the relationship between process and correctness has been assumed and yet this is the principle upon which software engineering is based. The results given here give one of the few confirmations of this relationship, albeit in relation to novice

programmers.

No relationship between other aspects of program quality and the quality of the process has been established. This result mirrors other results in this area. There is now a body of evidence building up to suggest that no relationship exists. This leads to the conclusion that novices must acquire skills in PSDP management as well writing program design. These two aspects of programmer knowledge are built up independently and must be addressed separately within programming courses.

The correlations between the different aspects of the programming process as measured by the metrics used in this research again show that there are a number of skills that are called upon when developing a computer program.

These conclusions, and especially the last two conclusions, provide information that can be interpreted in terms the knowledge needed to become an expert programmer. This will be further considered in the next chapter.

7. Learning to Program.

7.1 Introduction.

The overall purpose of this research was to investigate the role of the programming process within the context of novice programmers. This, it was hoped, would provide an insight that could be used to improve the quality of the work produced by novice programmers and hence improve the learning of novice programmers

The previous two chapters have produced three key results namely:

- Simple feedback can have a significant effect on the quality of the PSDP employed by novice programmers.
- A positive correlation has been shown between process metrics and the final product correctness indicating for the first time the often stated belief that how well a novice develops a program affects the final correctness.
- No correlation has been shown between the program style, design and complexity and the process metrics indicating the independence of these skills.

In this chapter a unified framework of programming expertise is proposed. This proposal is based on previous reported work in the literature and justified in the context of the findings above. It is proposed that this framework is used within the context of building a computer aided learning package; the characteristics of the novice can be assessed related to the model and then used to plan the next stage in the learning for that particular novice.

7.2 A Unified Framework of Programming Expertise.

A model of programming expertise has been developed over the past 25 years starting with the work of Schneiderman (1976) and Schneiderman & Mayer (1979). They proposed a two level model separating syntactic and semantic knowledge. Syntactic knowledge is the knowledge about the language statements and how to combine these. Semantic knowledge relates to the understanding of how the program statements are executed within the computer. Later work by Solway & Erlict (1984), Gilmore & Green (1988) and Rist (1989) developed the idea that there was a third dimension to expertise in programming, the ability to recognise and use blocks of code to solve a "sub problem". These were called programming plans and used as the basis of CAL packages such as bridge (Bonar & Cunningham 1988) and Proust (Johnston 1988).

Mayer (1997),added a fourth layer to the view of expertise, by proposing a strategic element to programming. He realising that programming is problem solving and encapsulated the ideas of general problem solving into a new framework of programming expertise. This is shown in table 7.1.

Mayer 1997	
syntactic	Language units and rules for combining language units
semantic	Mental models of the major locations, objects, and actions in the systems
schematic	Categories of routines based on function
strategic	Techniques for devising and monitoring plans

Table 7.1 Mayer (1997) Programming Knowledge Framework Summary

Anderson (1983) put forward a view, widely accepted in psychology, that knowledge can be categorised into declarative knowledge (knowledge about something) and

procedural knowledge (ability to use this knowledge). Davis (1991) who related this to schematic knowledge (programming plans) took up these two stages of knowledge acquisition. Davis showed that novices moved to some intermediate level by acquiring more programming plans and then onto becoming experts by learning how to use the programming plans. He established a two layer model with reference to schematic knowledge acquisition.

McGill & Volet (1997) built the idea of two levels of understanding into their framework of programming knowledge seen in table 7.2. Surprisingly they did not split their final category (strategic / conceptual) when Davis (1991) had earlier established the two levels in this category.

McGill & Volet 1997	**Declarative**	**Procedural**
Syntactic	Knowledge of syntactic facts related to a particular language	Ability to apply rules of syntax when programming
Conceptual	Understanding of, and ability to explain semantics of the actions that take place as a program executes	Ability to design solutions to programming problems
Strategic/Conceptual	The ability to design, code and test a program to solve a novel problem.	

Table 7.2 McGill & Volet 1997 Programming Knowledge Framework Summary

The experimental results collected as part of this research work has given an insight into the behaviour of novice programmers. In particular it has shown the importance of the programming process to the skill set of an expert programmer. This aspect of programming knowledge is not fully explained by either of these two frameworks. The Mayer (1997) framework addresses the idea that problem solving or programming process skills are separate skills and are included in the fourth level in his model. However he does not include the important distinction between

declarative and procedural knowledge in the model. McGill & Volet (1997) on the other hand do include the declarative / procedural distinction but have given scant regard to strategic knowledge. Indeed they do not consider a declarative / procedural split for their third level, the strategic / conceptual level.

A unified framework is proposed based on the four level model of Mayer (1997). This unified framework includes the declarative / procedural split at all levels as would be consistent with the work of Anderson (1983). This leads to four categories of programming knowledge each split into two, declarative and procedural. This unified framework is the obvious application of Anderson's work to Mayer's framework and was part used in the McGill & Volet model. The unified framework is shown in table 7.3.

Knowledge	**Declarative** 1	**Procedural** 2
Syntactic A	Knowledge of language rules A1	Ability to apply language rules A2
Semantic B	Mental model of the actions that take place as a program executes B1	Ability to design solutions to programming problems B2
Schematic C	Recognise chunks of code or programming plans used in program solutions C1	Ability to apply programming plans to program solutions C2
Strategic D	Knowledge of the personal software processes that are used to solve programming problems D1	Ability to use a personal software process that is used to solve programming problems D2

Table 7.3 Unified Framework of Programming Expertise

Whilst this framework has a theoretical simplicity it is necessary to justify the model from experimental evidence. The discussion in the next section seeks to justify the unified framework by taking evidence from the literature and from key experimental

results found in the course of this research.

7.3 Unified Framework of Programming Knowledge – Justification.

7.3.1 Syntactic Knowledge.

This category represents the knowledge of programming language syntax (declarative syntactic knowledge) and the ability to apply this knowledge (procedural syntactic knowledge). This is the knowledge introduced at the beginning of introductory text books. These books tend to introduce a number of new syntactic constructs to allow the novice to build up their declarative syntactic knowledge and then ask questions that require the novice to apply this knowledge to build up their procedural syntactic knowledge. Thus there is acknowledgement, albeit implicit by authors, that the syntactic knowledge consists of these two components, declarative and procedural syntactic knowledge.

7.3.2 Semantic Knowledge.

The semantic level is defined as *the "mental model of the actions that take place as a program executes*" and the application of this mental model. Novices with a superficial declarative level of knowledge may be able to read and understand programs but a greater level of expertise is required in order to apply this knowledge.

Weidenbeck, Fix and Scholtz (1993) claimed that expert semantic knowledge was "*well founded in the recognition of basic patterns*" and "*well connected internally*".

Through experimental work they showed that novices lacked the ability to recognise basic patterns or could not connect the pattern to program text. These novices lacked some element of semantic knowledge.

Other authors have sought to examine differences in understanding of programs (semantic knowledge) by looking at the "debugging" skills of novices and experts, (eg. Weindenbeck 1985, Schmidt 1986, Gugenty & Olsen 1986 and Jeffries 1987). They explain the differences between novices and experts by the lack of a sophisticated mental model of the program. Additionally the novices can not apply the knowledge they have as well as experts. Weidenbeck (1985) conclude that experts have automated their semantic debugging skills whereas novices find difficulty in recognising the problem.

To conclude, the experimental evidence would suggest that the McGill & Volet (1997) model of semantic (conceptual) in their work) knowledge that contains both declarative and procedural elements is a valid interpretation of this aspect of programming knowledge. This aspect has been directly included in the unified framework of programming knowledge.

7.3.3 Schematic Knowledge.

The third layer in this framework is schematic knowledge. Schematic knowledge is the ability to recognise "chunks" or patterns of code that carry out various sub tasks within a program. The procedural aspect of this knowledge is the ability to apply or use these chunks to build up a program. These chunks are more correctly called

programming plans that have been studied extensively. Solway and Ehricht (1984) put forward a simple model that expertise in programming was based on having available these plans. Thus the Solway & Ehrlicht (1984) model of programming expertise considered the level of expertise to be based on the number of programming plans available, i.e. the level of declarative schematic knowledge. This model was used as the basis for the Bridge intelligent tutor scheme (Bonar & Cunningham 1988). This CAL system was designed to increase the number of programming plans known to the novice programmers.

It was Davis (1991) who showed that one of the main differences between novices and experts was not just the number of programming plans but the way these plans are organised internally. Davis introduced to the discussion a third level of programmer, the intermediate programmer. The model proposed by Davies and supported by his experimental work was that the transition from novice to intermediate consisted of building a library of programming plans, the Solway & Ehricht (1984) model. This is the acquisition of schematic declarative programming expertise. Davies showed that intermediate and expert programmers had a similar range of programming plans available to them but the experts were able to utilise these plans better. These experts had acquired procedural schematic programming knowledge. This model of Davies fits well into the general framework of programming expertise as it mirrors the skill acquisition model of Anderson(1981)

The third level in the unified framework splits the schematic layer of Mayer (1997) into declarative and procedural knowledge. The main argument for this split is the work published by Davis that is built on the general knowledge acquisition model of

Anderson (1983).

7.3.4 Strategic Knowledge.

Mayer (1997) defined strategic knowledge as the techniques for devising and monitoring plans. This is the ability to devise and keep to a satisfactory Personal Software development Process (PSDP).

The experimental evidence discussed in chapter 3 showed that the experts and some of the novices were following a plan when developing their program. This was a staged or incremental plan that would lead the programmer to be able to solve the given problem. These experts and novices have some extra ability in "how to program" above that shown by other novices. The experts were showing a greater strategic knowledge than the novices. The emergence of strategic knowledge collaborates the work of Mayer (1997).

The strategic knowledge is the fourth and final level of the unified framework of programming knowledge. The issue to be addressed next is whether there is any evidence that this knowledge can be split into declarative and procedural knowledge as are the other levels in the unified framework of programming knowledge.

The work of Anderson (1983) proposed this declarative / procedural split for the acquisition of all knowledge so it seems reasonable to see if there is evidence that these two stages of knowledge apply to strategic programming knowledge. This explanation is again based on the concept of three major groups of programmers. Novices who are building up their declarative strategic knowledge, intermediates

who have declarative strategic knowledge and are building up their procedural strategic knowledge, and experts who have both declarative and procedural strategic knowledge.

In the work reported in chapter 3 three categories of novice are identified.

- One group of novices (group D in chapter 3) did not get their program to work. The graphs of lines of code, specification proportion, and compilation errors against version do not show any pattern or evidence of planned build up of code. Thus it can be argued that these students did not have the relevant strategic knowledge to be able to solve the given problem.
- A second group of novices (group C in chapter 3) did manage to successfully produce a working program but their methods lacked sophistication. The PSDP employed by these novices was to write the whole program and then "debug" the program through a number of stages to produce a working solution. This shows evidence that these novices do not have the ability to apply any strategic knowledge they have. However it can be argued that they do have some strategic knowledge, as they were able to produce a solution. These students have developed from novices and are moving towards becoming intermediate programmers.
- The third group (group B in chapter 3) exhibited behaviour that mirrored in some ways the behaviour of the experts. The novices developed their program in stages, as did the experts. The number of stages was not the same as the experts, some used fewer stages others more stages. These novices showed evidence that they had reached intermediate status and were building to expert status. They knew how to develop a program (had gained declarative strategic knowledge) but

were not able to execute this plan in the same way as an expert (had not fully developed procedural strategic knowledge).

The feedback experiment reported in chapter 4 gives further evidence for this view of strategic knowledge. The feedback had the effect of improving the way in which the treatment group of novices developed their programs. Given that the feedback was solely in regard to the correctness it could not have added any declarative strategic knowledge to the novices. One explanation of how this improvement came about is that the feedback in some way helped the novice in their ability to apply the knowledge they already had.

The experiential work carried out in this thesis can be explained in terms of the strategic knowledge of the novice programmer and there is evidence that this strategic knowledge has two dimensions, declarative and procedural knowledge.

7.3.5 Summary

This section has sought to justify the unified framework of programming knowledge. This justification has been based on both previously reported work and experimental work carried out during this research. The novel aspect of this model is the explanation of strategic programming knowledge in terms of declarative and procedural knowledge. The next section will attempt to further explain this level of expertise in terms of process plans.

7.4 Strategic Knowledge - Process Plans.

The unified model of programming expertise discussed above provides for declarative and procedural strategic knowledge. Novices build up a skill set of strategic knowledge and also learn how to apply this knowledge. This section will explore the idea of process plans as the mechanism for acquisition of expertise in strategic programming knowledge.

Davies (1991), and others, have used programming plans to model the acquisition of schematic knowledge by an expert programmer. The idea of plans or patterns of statements is more widely used than just computer programming. They are widely used in psychology to explain knowledge acquisition. For example, plans have also been used to explain expertise in natural language as well, Carberry & Pope (1993). This section attempts to explain strategic programming knowledge using a concept of process plans.

Strategic programming knowledge is the *"techniques for devising and monitoring plans"* (Mayer 1997). It encompasses the problem solving aspect of programming answering the *question "How do I go about developing a program to solve this problem?"* In his seminal work Polya (1957) gives four stages to problem solving namely:

- understand the problem
- devise a plan
- carry out the plan
- check the results

This general problem solving strategy is based around a plan. This plan, if followed, leads to a solution to the problem. The general problem solving stages of Polya can be re-phrased in the context of computer programs:

- understand the problem and classify it according to previous experience
- choose an appropriate plan as the program development strategy
- carry out the plan
- check the results

The plan or strategy for developing a program is a process plan. These process plans identify the sequence of events to be carried out to solve a given problem. Process plans are generic in that a particular process plan is adaptable to a number of problems. The problems studied in the course of this research are essentially data processing problems that easily fit the traditional input-process-output mode of solution. The incremental personal software development process (PSDP) was identified as the strategy or plan used to develop a solution to this type of problem. This process is illustrated in figure 3.2. Parts of the strategy involves how compiler errors are fixed (relating to syntactic or semantic knowledge) and how an individual program stage (or "chunk") can be corrected (relating to schematic knowledge). If these items are left out of the PSDP then a process plan for incremental development is left. This is shown in figure 7.1.

```
Understand the problem and identify a development plan
Repeat
    Add code for next stage in development
    Get this code to work
Until full spec complete
```

Figure 7.1 Iterative Process Plan.

For a smaller problem (for example a program to add up a list of numbers) that involves one programming plan then a simpler process plan can be identified, the all-or-nothing process plan, and is illustrated in figure 7.2.

```
Understand the problem
Write the code
Get this code to work
```

Figure 7.2 All-or-nothing Process Plan.

Figures 7.1 and 7.2 show two possible process plans. The experimental evidence of how novices and experts have developed their programs may be interpreted in terms of these process plans.

- Expert programmers have available a number of solution methods, process plans, from which they can choose an appropriate one to develop the particular program. As they have available a suitable process plan and the necessary procedural strategic knowledge to execute the plan they develop the program well.
- Advanced novices (group B novices of chapter 3) exhibit behaviour similar to the experts but may use a different number of stages in the development. This can be interpreted as these novices having available to them a sufficient range of process plans to enable them to choose a correct plan but they lack the procedural knowledge to execute the plan in the same way as do experts.
- Novices (group C in chapter 3) attempted to solve the problem in one attempt, using the simple process plan of figure 7.2. Thus they lacked the variety of process plan to enable them to choose and use a "good" plan.
- Poor novices (group D in chapter 3) did not produce a solution and showed no evidence on implementing any solution. One possible interpretation was that they lacked the variety of plans and could not implement their chosen plan with the

difficult problem facing them.

Process plans have been put forward as a way to explain strategic knowledge of programming. Novices need to build up a number of process plans and then learn how to apply these process plans in order for them to become expert programmers. The idea of process plans has been used to explain the strategic knowledge exhibited by novice programmers in the course of this research.

7.5 Applying the Unified Framework - Lessons for Learning .

This chapter has suggested a framework of programming knowledge by consolidating aspects from the literature. The idea of process plans has been put forward to explain strategic programming knowledge. This final section describes the application of these theoretical findings to the teaching of computer programming either by traditional lecture based courses or more particularly on-line courses.

Not only must the teaching of programming deal with language syntax, semantics and programming plans it must also deal with the programming process - how to create a program. Many on-line packages will take the program created by the student and give feedback by automated marking or style analysis. Some systems may allow the student to increase the marks by repeated submissions of a program to the automated analysis tool. Such feedback can at best inform the students' syntactic, semantic or schematic knowledge.

To give feedback on the strategic ability of the student an on-line package will need some extra features not currently available. To be able to comment or mark the strategic knowledge of a novice a history of the program development is needed. The experiments carried out during the course of this research show that it is feasible to collect versions of a program during development, convert each version to a series of counts that are used to calculate a number of process metrics. These metrics can be converted to a single process mark that is fed back to the novice in the same way as product marks are.

In order to build up the strategic knowledge of the novice programmer a wide range of problems with different "process plans" needs to be offered to the novice. In many currently available computing courses the author will give a wide variety of problems that show varied syntactic, semantic and schematic aspects of programming. The work carried out here implies that a wide variety of strategic problems needs to be offered.

In many CAL packages, after the novice has answered an assessment, it is marked and the next lesson given to the novice is dependent on that mark. This is often carried out simply by asking novices who have "failed" an assessment to repeat the lesson. More sophisticated packages will have a number of "next" lessons dependent on the mark awarded. One drawback of this method is that at best the final program is assessed and the decision as to which "next" lesson given on the basis of the final product. As a result of this research there is the potential to measure the process and assess how well the development was carried out. For the first time a judgement can be made regarding how well this solution was gained. The "next" lesson can be based

not only on the quality of the program written but also on the way it was written. If the novice struggled to write the solution then a simpler "next" lesson can be given but if the development process was good a harder or different "next" lesson can be given. The students' progress through the programming course can be altered not only on the quality of the programs produced but also on the quality of the program development process, thus assessing and building up their strategic knowledge.

.

The discussion above shows how the process can be evaluated at the end of the development and used to influence the "next" lesson. It would also be possible to use the data collected during the development to calculate the process metrics during the development of the program. Potentially this could be used to identify the characteristics of the novice and feedback given to the novice based on their ability. Further research is needed to evaluate the usefulness of such an approach to programming education.

The main finding of this research is the confirmation of the importance of the program development process in the acquisition of programming expertise. The development process and how a novice is to develop a program is the key to the education process and must be made explicit rather than left for the novices to learn implicitly.

7.5 Summary.

This chapter has derived and examined a unified framework of programming expertise incorporating previously reported results and theoretical models.

The main conclusions are

- a unified framework of programming expertise has been proposed with four categories, (syntactic, semantic, schematic and strategic) each split into declarative knowledge (knowledge about something) and procedural knowledge (ability to use the knowledge).
- the idea of process plans was proposed to explain strategic programming expertise. The experimental evidence can support this idea
- there are practical lessons to be learnt from this work with regard to the way computer programming is learnt.

8. Final Conclusions and Future Work.

8.1 Introduction.

If software engineering has taught the computing industry anything it is the importance of quality and the focus on the software development process rather than just the product. The control of quality may only be undertaken if there are some measurements and metrics available whereby different software developments may be compared. Despite the general acceptance of this in the software engineering industry little has permeated through to software engineering education and the need to address the issue of "quality of process" for novice programmers. This thesis has looked at the personal software development employed by novice programmers and identified issues, created suitable metrics, related this to final products and evaluated the importance of process for the novice programmer.

This chapter concludes the work by reviewing the objectives of the research set out in chapter one and summarises the findings of this research. The chapter goes on to discuss briefly how the findings of this work can be applied in a practical way i.e. the application of the results to teaching and learning of novice programmers. Finally this chapter and the thesis conclude with a discussion of future work in this area.

8.2 Review of Research Objectives.

The aim of this research was given in chapter one and is restated here as:

Overall Aim: To explore the personal software development process employed by novice programmers in order seek to improve the quality of computer programs.

This aim leads to a number of research objectives. A summary of the work carried out in pursuit of each research objectives is given.

Objective 1: *To analyse the personal software development process employed by novice and expert programmers*

Software was written that captures the novice's computer program after each compiling attempt thus a full program development history was saved. Graphs of lines of code, specification proportion (correctness) and compilation errors against version number (used as a proxy for time) were drawn and used to "visualise" the software development process. A similar activity was undertaken for a group of expert programmers.

An incremental model of software development drawn from the literature was used as the basis for the analysis of the personal software development process used by novices and experts. An analysis of the graphs showed that the expert programmers did indeed follow the incremental model justifying the model as a basis of the analysis.

The novices were categorised into three groups each showing a different approach to the personal software development process.

- One group of novices exhibited behaviour that followed the incremental model of program development. These novices showed that they had some strategic programming expertise. Their behaviour did not mirror exactly that of the experts thus hinting at metrics which could be used to measure the level of strategic knowledge of an individual.
- A second group developed the program in one increment (all at one go). These novices exhibit behaviour suggesting that they have syntactic, semantic and schematic knowledge but lack the strategic knowledge.
- A final group of novices showed chaotic behaviour, for example, adding large numbers of lines of code when the previous versions did not compile . Not only did this group lack strategic knowledge but they lacked other elements of programming knowledge, (syntactic, semantic or schematic).

A brief summary of the analysis of the personal software development process employed by novice and expert programmers is shown in table 8.1.

Classification	Description
Experts group A	Followed the PSDP model
Novices group B	Followed in outline the incremental PSP model but strategic knowledge is less advanced than with experts.
Novices group C	Only one increment in their PSDP indicating a lack of strategic knowledge
Novices group D	Could not develop a solution due to lack of sufficient strategic, schematic, semantic or syntactic knowledge.

Table 8.1 Classifications of experts and Novices.

Objective 2: *To establish a set of metrics that measure the quality of the personal software development process employed by novice programmers.*

In line with good practise the metrics developed within this research were based on an underlying model of software development. The model used here is the incremental model of the personal software development process from which a number of metrics were developed. The metrics were developed with reference to one set of data from a group of subjects and subsequently validated against a second set of data gained from a second group of subjects. Thus the metrics have some validity outside the dataset from which they were developed. Table 8.2 summarises the metrics developed in this research.

Compiling Progress Indicator	`cpi`
Relative Run Progress Indication	`rrpi`
Stages in l.o.c	`sl`
Stages in testing	`st`
Relative Total Progress Indication	`rtpi`

Table 8.2 Validated Metrics.

These metrics measure adherence to the incremental model of software development and it is argued, as this is a reasonable model, measure the quality of the personal software development process employed by novice programmers.

Objective 3: *To analyse the correlation between the personal software development process and the program product.*

It is an often stated, but not often proved, conjecture of software engineering that by improving the process, the product itself improves. This thesis analysed this

relationship with reference to novice programmers. The metrics outlined in table 8.2 were used to measure the personal software development process and the final software product measured by well established metrics. Previous work has failed to show any correlation. However this work was based on poor process metrics. The work here showed a strong correlation between the process metrics and correctness but not with any other product metrics. This indicates that those novices who use the incremental model of program development tend to get more correct programs than those who do not. This result indicates that efforts to improve the personal software development process of novices can have a positive effect on the correctness of the final product. In line with previous work, no correlation was found between the process metrics and the final product metrics. A picture is emerging that there is little or no relationship between the two. The implications of this are that the novice needs to explicitly learn these different skills and neither should be ignored in the education process.

Objective 4 : *To evaluate improvements in the personal software development process and software product using simple feedback mechanisms.*

The education of computer programmers focuses on the product being developed and not on the way this product was developed. This is contrary to the main premise in software engineering and quality in general, the process quality dictates the product quality. The experiment undertaken during this research showed that software process metrics were improved by providing simple feedback to the novices during their development of the program. Focussing some educational effort onto the process may make improvements in the personal software development process. This

result has implications for the education process namely that education and training courses must invest some effort in the development of the personal software development process in addition to the product as skills in the personal software development process are necessary in the education of novice programmers. There is potential for greater improvement if the feedback given to the novices reflects the current needs of the novice rather than the simple feedback given here.

Objective 5: *To synthesise current frameworks of programming expertise into a unified framework that includes a strategic knowledge facto, that is expertise in the personal software development process.*

A general framework of programming knowledge was proposed that included four layers of knowledge: syntactic, semantic, schematic and strategic. Each of these layers exhibits declarative knowledge, knowledge about a skill, and procedural knowledge, ability to use the skill. The framework was produced by expanding previous reported work and is supported by the evidence produced as a result of this research. The general unified framework of expertise in programming is shown in table 8.3

Knowledge	Declarative	Procedural
Syntactic	Knowledge of language rules	Ability to apply language rules
Semantic	Mental model of the actions that take place as a program executes	Ability to design solutions to programming problems
Schematic	Recognise chunks of code or programming plans used in program solutions	Ability to apply programming plans to program solutions
Strategic	Knowledge of the personal software processes that are used to solve programming problems	Ability to use a personal software process that is used to solve programming problems

Table 8.3 Unified Framework of Programming Expertise

The aspect of programming expertise related to the personal software development process, strategic knowledge, was not adequately covered in the literature and it was this aspect that has been expanded due to this research work.

In an attempt to explain strategic expertise it is useful to consider three classes of programmer; novice, intermediate and expert. The transition from novice to intermediate is the building up of the declarative strategic knowledge; the transition from intermediate to expert is the build up of procedural strategic knowledge. The model used for the acquisition of strategic knowledge is analogous to the acquisition of schematic knowledge again based on novice, intermediate and expert programmers and their declarative and procedural knowledge of programming plans (Davis 1995). The concept of process plans was put forward as the way to think about strategic programming knowledge. These plans are development strategies programmers employ to write programs. The development from novice to intermediate involves building up these process plans and the further development to expert involves being able to choose and use an appropriate process plan. The framework put forward can be used to explain the differences in program

development as visualised with reference to objective 1 in this research.

8.3 Application to the Teaching and Learning of Computer Programming.

The work described in this research may at times seem theoretical. It is not meant to be. The purpose of this research is to generate some results and to apply these results to the practical task of the teaching and learning of computer programming.

8.3.1 Automatic Assessment

Many systems are available, most notably Ceilidh (Benford et.al. 1997), that allow a computer program to be automatically assessed. This assessment deals exclusively with assessing the final product. This research has described methods that can be used to collect data showing the development of the process. This data can be used to measure and therefore assess the personal software development process. An immediate area for the practical application of this work is to include into the integrated development environment (IDE) a facility that stores the program version as part of the compilation process. Thus when the program is assessed not only can the final product be assessed but also the process metrics calculated from the development history. The latter can be used to produce a process assessment mark. Novices are therefore assessed in terms of their product and the process that generated the product.

8.3.2 Include the Process into the Learning.

This research has shown the importance of the personal development process to the

novice programmer. The strategic knowledge needed for a programmer is a different skill to any syntactic, semantic or schematic knowledge they have. Any learning method (be that lecture or CAL package) must include in it some development of the strategic skills in it. In many CAL packages at the end of a "lesson" some test is given and if "passed" the student progresses to the next stage. If the student "fails" the test they are required to repeat the "lesson". No account is taken about whether the student developed the solution well or struggled to get the answer. Hence a practical application of this work is to measure both the product and the process at the end of the "lesson". If the student passes both the product and the process assessment then they can go on to the next stage. A student who passes the product assessment but fails the process assessment needs to be directed to some re-mediation work regarding the process of software development rather than continuing on with the next stage. This idea of including process can be used with any teaching environment.

8.3.3 Automating Feedback.

During the course of the experimentation discussed in chapter four a system was developed whereby feedback was given to the students during the development of their programs. The feedback given in this case was simple: a statement of the correctness of the program. However simple the feedback, the analysis of the data has shown the effectiveness of the feedback. The idea of giving feedback during the development of a program can easily be built into the IDE or as part of a CAL package.

8.4 Future Work.

The work described in this thesis looked at an often neglected area of computer programming education, namely the personal software development process. The thesis has reported on some initial work in this area. There are a number of areas in which to apply and extend the findings for future work on this topic.

8.4.1 Online Learning.

Section 8.3 discussed three areas where the work reported here could be included into the teaching of novice programmers. However there is further potential in this method which has not been addressed during this research.

1. Metrics are calculated at the end of the process and used in the assessment of the process. However as the whole development history is available during the personal development process it is possible to calculate the process metrics (and product metrics) during the development of the program. Experimentation is needed to see whether feeding back the metrics during the development can enhance the development itself. The experiments in chapter 4 give one metric (correctness) as feedback. Is it true that the other metrics can also be used to improve the process?
2. During this research the metrics were calculated at the end of the development. A major area for further work is to investigate if information can be gained by calculating the metrics during the development and using these metrics to find the intention of the student and attempt to alter any incorrect behaviour. This is an extension of the ideas of Johnson (1990) who sought to identify the programmer's "intention" from their error and then to suggest corrections to the

error. Johnson restricted his ideas to the program itself. Further research is needed to extend these ideas to the personal software process.

3. What is "appropriate" feedback? The work reported in the experiment of chapter 4 uses some simple feedback to produce the significant improvement in the process metrics of the treatment group of novices. There is no claim that the feedback given is optimal in any sense. The treatment was chosen because it was simple to achieve. The suggestion above is to use other metrics in feedback. It is necessary to identify what is appropriate feedback. Is detailed feedback which is likely to be difficult to automatically generate useful or is a simple "pat on the back" just as effective?

8.4.2 How do Students Learn?

The model of programming knowledge presented in chapter 5 describes the knowledge that an expert programmer will have. Within that model is the idea of programming plans which describe how schematic knowledge is achieved and the new idea of process plans to describe how strategic knowledge is built up. These theories do not address the problem of how novices learn to program. What is the "learning curve" for a novice programmer? Another fruitful area of research is to investigate the development of novices over time to see how they build up the necessary knowledge to become expert programmers.

8.4.3 Professional Programmers.

The work carried out in this research has examined novice programmers. It would be interesting to carry out a similar investigation of professional programmers. It is

recorded that some professional programmers are orders of magnitude more productive than others. An investigation into the work patterns of professional programmers would seek to show the validity of this statement. The metrics used in this research are focussed on novices.These metrics would need to be validated with professional programmers. Experimentation along the lines carried out in this research can then be used to see if the differences in programmer output is because of the different personal software processes used by these people. If this is the case, training schemes along the lines of the online package described above could be implemented with the aim of improving the personal software development process of the weaker programmers.

8.5 Conclusion.

This work has examined in detail how novice programmers develop their programs. This process has been modelled and measured allowing a comparison to be made between personal software development processes. This aspect, the software development process, is often ignored by educators and this research has highlighted its importance. The strategic knowledge needed by a novice programmer has been added to current frameworks of programming expertise thus highlighting this aspect of programming.

As a result of the work carried out during this research the "importance of process" (Lund et al 1998) has been established.

References.

Adelson.B. 1981 Problem Solving and the Development of Abstract Categories in Programming Language. *Memory and Cognition.* Vol.9. pp422-433.

Adelson B 1984 When Novices surpass experts: The difficulty of a task may increase with expertise. *Journal of Experimental Psychology: Learning, Memory and Cognition.* Vol.9 pp.422-433

Allwood C.M. 1986 Novices on the Computer:a review of the literature. *International Journal of Man-Machine Studies.* Vol.25 pp.633-658.

Anderson, J.R. 1983. *The Architecture of Cognition.* Harvard University Press.Cambridge MA.

Basili V.R. and Turner A.J. Interactive Enhancement: A Practical Technique for Software Development. *I.E.E.E. Tran. On Software Engineering.* Vol. !, No. $. Pp.390-396.

Bayman, P. and Mayer, R.E. 1988. Using Conceptual Models to Teach BASIC Computer Programming. *Journal of Educational Psychology.* Vol.80. pp291-298.

Benford S., Burke E. and Foxley E. 1993a Learning to Construct Quality Software with the Ceilidh System. *Software Quality Journal.* Vol. 2. pp.177-197.

Benford E., Burke E.K, Foxley E., Gutteridge N. and Zin A.M. 1993b Experiences using the Ceilidh System. *Proceedings of the First All-Ireland Conference on Teaching Computing* Dublin.

Benford S.D, Burke E.K., Foxley E., Gutteridge N.H. and Gibbon C.A. 1994 Observations on the Impact of the Ceilidh System on the Teaching of Computer Programming. *Proceedings of the Second All-Ireland Conference on Teaching Computing* Dublin.

Bishop-Clark, C. 1992 Protocol Analysis of a Novice Programmer. *ACM SIGCSE Bulletin.* Vol 24. No.3. pp14-18.

Bonar, J. and Cunningham, R. 1988. Bridge: an Intelligent Tutor for Thinking about Programming. In. Self. J. Ed. *Artificial Intelligence and Human Learning.* Chapman-Hall.

Bornat R.1990 *A Core Craft Beyond the Statis.* Times Educational Supplement. 2nd. November 1990.

Brooks F.P. No Silver Bullet: Essence and Accidents of Software Engineering. *I.E.E.E. Computer* Vol.20. No.4. pp.10-19.

Carberry.S. and Pope.W.A. 1993. Plan Regognition Strategies for Language Understanding. *International Journal of Man-Machine Studies*. Vol.39. pp.529-577.

Chi, M.T.H., Glaser R. and Farr, M.J. 1988 *The Nature of Expertise.* Lawrence Erlbaum Associates. New Jersey.

Coolican, H. 1999. *Research methods & Statistics in Psychology.* London: Hodder & Stoughton.

Conway R. 1978 *A Primer on Discipline Programming.* Winthrop Publishers. Cambridge Mass.

Davies, S.P. 1989 Skill Levels and Strategic Differences in Plan Comprehension and Implementation in Programming. In Sutcliffe A. and Macaulay L. Ed. *People and Computers V.* Cambridge University Press.

Davies, S.P. 1990a Plans, goals and selection rules in the comprehension of computer programs. *Behaviour and Information Technology*. Vol.9 No.3. pp201-214.

Davies, S.P. 1990b The nature and development os programming plans. *International Journal of Man-Machine Studies*. Vol.32 pp461-481

Davies, S.P. 1991a Characterising the program design activity: neither strictly top-down nor globally opportunistic. *Behaviour and Information Technology*. Vol.10 No.3. pp173-190

Davies, S.P. 1991b The role of notion and knowledge representation in the determination of programming strategy: a framework for integrating models of programming behaviour. *Cognitive Science*. Vol.15 pp547-572

Davies, S.P. 1994 Knowledge restructuring and the acquisition of programming expertise. *International Journal of Human-Computer Studies*. Vol.40 pp703-726.

Davies, S.P. and Castell, A. 1992 Contextualizing design: narratives and rationalization in empirical studies of software design. *Design Studies.* Vol.13. No.4. pp.379-392.

de Marco, T. 1982. *Controlling Software projects: management, measurement and estimation.* Yourdon press.

Dugard, P. and Todman J. 1995. Analysis of Pre-test-Post-test Control Group Designs in Educational Research. Educational Psychology. Vol.15. No.2. pp.181-198.

Edmunds G., 1990 Experiences using CAAPE: Computer Assisted Assessment of Programming Exercises. *Computers & Education.* Vol. 15. No.1-3 pp.45-48.

Ejiogu. L.D. 1993 Five Principles for the Formal Validation of Models of Software Metrics. *ACM SIGPLAN Notices*. Vol.28 No.8 pp.67-76.

Fenton, N. 1992. When a software measure is not a measure. *Software Engineering Journal.* Vol. 7. No.9. pp.357-362.

Foxley E., Higgins C. and Gibbon C. 2000 *The Ceilidh Courseware System* [Online] Nottingham University. Available from http://www.cs.nott.ac.uk/~ceilidh/papers/overview.cal [Accessed 01/05/2000]

Genter, D. and Stevens, A.L. Eds. 1983. *Mental Models.* Hillsdale NJ.

Gilmore, D.J. and Green, T.R.G. 1988 Programming Plans and Programming Expertise. *Quarterly Journal of Experimental Psychology.* Vol.40a. pp423-442.

Goodwin, L. and Sanati, M. 1986. Learning Computer Programming through Dynamic Representation od Computer Functioning: Evaluation of a new Learning Package for Pascal. *Internaltional Journal of Man Machine Studies.* Vol.25. pp327-341.

Grove, R.F. 1999. Using the Personal Software Process to Motivate Programming Practices. *ACM SIGCSE Bulletin.* Vol 31. No.1. pp98-101.

Gugerty L. and Olsen G.M. 1986 Comprehension differences in debugging by skilled and novice programmers. In Soloway E. and Iyengar S. Eds. *Empirical Studies of Programmers.* Northwood New Jersey.

Halstead, M.H. 1977. *Elements of Software Science.* Elsevier North-Holland. NY.

Hannemyr. G. 1983 *Automatic Assessment Aids for Pascal Programs –Rats!* SIGPLAN Notices Vol.18. No.4. pp16-18.

Henry, S. and Kafura, D. 1984 The Evaluation of Software Systems' Staructure using Quantative Software Metrics *Software Practice and Experience.* Vol.14. No.6. pp.561-573.

Herbsleb, J. Zubrow, D. Goldenson, D. Hayes, W. and Paulli M. 1997. Software Quality and the Capability Maturity Model. *Communications of ACM.* Vol 40. No.6. pp.30-40.

Hilburn, T.B. and Towhidnejad, M.1998 Doing Quality Work: The Role of Software Process Definition in Computer Science Curriculum. *ACM SIGCSE Bulletin.* Vol 30. No.1. pp277-281.

Hoc, J-M. Green, T.R.G. Samurcay, R. and Gilmore, D.J. 1990 *Psychology of Programming.* Academic Press.

Hou, L. and Tomayko, J. 1999 Applying the Personal SoftwRe Process in CS!: An Experiment. *ACM SIGCSE Bulletin* Vol.31. No.1. pp.322-325.

Howatt. J.W. 1994. On Criteria for Grading Student Programs. ACM *SIGCSE Bulletin* Vol.26. No.3. pp3-7.

Huff, K.E. and Lesser, V.R. 1989 A Plan-Based Intelligent Assistant that Supports the Software Development Process. *Proceedings ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments. SIGPLAN Notices* Vol.24. No.2. pp97-106.

Humphrey, W.S. 1995. *A Discipline for Software Engineering.* Addison-Wesley Reading MA.

Humphrey, W.S. 1997. *An Introduction to the Personal Software Process.* Addison-Wesley Reading MA.

Hung, S-L., Kwok. L-F, and Chan R. 1993 Automatic Programming Assessment. *Computers Education.* Vol.20. No.2. pp.183-190.

Hutchens D.H. and Kutz E.E. 1996 Using Iterative Enhancement in Undergraduate Software Engineering Courses. A.C.M. SIGCSE No.2 pp266-270.

ISO9000, *Quality Systems – Model for Quality Assurance in design, Development, production, Installation and Servicing.* International Standards Organisation.

ISO9000-3, *Guidelines for the Application of ISO9000 to the Development, Supply and Maintenance of Software.* International Standards Organisation.

Jackson. D. 1991 Using Software Tools to Automate the Assessment of Student Programs. Computers & Education. Vol.17. No.2. pp.133-143.

Jeffries R. 1982 A Comparison of Debugging Behaviour of Novice and Expert Programmers. *American Educational research Association Annual Meeting.* New York.

Johnson, L. 1988. Modelling Programmers Intentions. In Self. Ed. *AI and Human Learning* Chapman-Hall.

Kernighan B.W. and Plauger P.J. 1974 *The Elements of Program Style.* McGraw-Hill New York.

Lake. A. and Cook. C. 1990 STYLE: An Automatic Program Style Analyser for Pascal. *SIGCSE Bulliten.* Vol.22. No.3. pp.29-33

Linn, M.C.1985 The Cognitive Consequences of Programming Instruction in Classrooms. *Educational Researcher.* Vol.14. No.5. pp.14-16,25-29.

Lovegrove. G.L. and Rees. M.J. 1984 Some Steps towards Automatic Teaching and Marking. *University Computing.* Vol.6. pp.6-23.

Lund, G.R. 1994. The Program Development Process. *In Innovations in Computing Teaching.* SEDA Paper 88.

Mayer, R.E. 1979. A Psychology of Learning BASIC Computer Programming: Transactions, prestatements and chunks. *Communications of ACM.* Vol.22. pp.589-593.

Mayer, R.E. 1997. From Novice to Expert. *Handbook of Human Computer Interaction 2nd.Edition.* Helander, M. Landauer, T.K. and Prabhu, P. Eds. Elsevier-Science B.V.

McAlpin D.A., 1992 Software for Teaching Computing. *Computers Education.* Vol.19. No.1/2. pp27-36.

McAlpin D.A., O'Docherty. B.A., O'Donoghue. P.G. and Teague. K.G.1995. Flexible Automatic Assessment of Clarity, Complexity and Style of Student's Modula2 Programs. *Proceedings of the Second All-Ireland Conference on Teaching Computing* Dublin.

McCabe, T.J. 1976. A Complexity Measure *IEEE Transactions Software Engineering.* Vol. 2. No. 4. pp. 308-320.

McGill, T.J. and Volet, S.E. 1996 An Investigation of the Relationship Between Student Algorithm Quality and Program Quality *ACM SIGCSE Bulletin.* Vol 27. No.2. pp44-48.

McGill, T.J. and Volet, S.E. 1997. A Conceptual Framework for Analysisng Students' Knowledge of Programming. *Journal of Research on Computers in Education. Vol.29. No.3. pp.276-297.*

Meekings. B.A.E. 1983. Style Analysis of Pascal Programs. *SIGPLAN Notices.* Vol.18. No.9. pp.45-54.

Oman. P.W. and Cook. C.R. 1989. A Paradigm for Programming Style Research. *SIGPLAN Notices.* Vol.23. No.12. pp.69-78.

Pardoe. J. and Vickers P. 1994 Using a Prototype Program Assessment Tool.. . *Proceedings of the Second All-Ireland Conference on Teaching Computing* Dublin.

Parrish, A. Lester, C. Cordes, D. and Moore, D. 1997. Assessing Computer Usage Patterns in a Software Development Course. *ACM SIGCSE Bulletin.* Vol 29. No.2. pp58-61.

Polya.G. 1957 *How to Solve It.* Pincetown University Press. New York.

Rambach, H. D. & Basili, V.R. 1987 A Quantitative Assessment of Software Maintenance. *Proceedings of the Conference on Software Maintenance.* Austin Texas. September 1987. pp.134-144.

Reddish. K.A. and Smyth. W.F. 1986 Program Style Analysis: A Natural By-Product of Program Compilation. *Communications ACM* Vol.29. No.1. pp.126-133.

Reddish. K.A. and Smyth. W.F. 1987 Evaluating Measures of Program Quality. *The Computer Journal* Vol.30. No.3. pp.228-232.

Rees. M.J. 1982. Automatic Assessment Aids for Pascal Programs. *SIGPLAN Notices* Vol.17. No.10. pp.33-42.

Rich C. and Walters . R,C,. 1988 A Research Overview. *Computer* Vol. 21., No.11, pp11-24.

Rimmer A., Pardoe. J. and Vickers P. 1995 Interactive Program Assessment Using SPROUT. *Proceedings of the Third All-Ireland Conference on Teaching Computing*

Rist, R.S. 1989. Schema Creation in Programming. *Cognitive Science.* Vol.13. pp.389-414.

Rist, R.S. 1991. Knowledge Creation and Retrieval in Program Design: A Comparison of Novice and Intermediate Student Programs. *Human Computer Interaction.* Vol.6. pp.1-46.

Rosenthal. D. 1983. Correspondence from the members. SIGPLAN Notices Vol.18. No.3. pp.4-5.

Rowe. G. and Gregor. P. 1999 A Computer Based Learning System for Teaching Computing : Implementation and Evaluation. *Computers and Education.* Vol.33. No.1. pp65-76.

Rowe.G. and Thorburn. G., 2000 VINCE – an Online Tutorial Tool for Teaching Introductory Programming. *British Journal of Ecuational technology.* Vol.31. No.4. pp.359-369.

Schmidt, A.L. 1986. Effects of Experience and Comprehension on Reading Time and Memory for Computer Programs. *International Journal of man Machine Studies* Vol.25. pp.399-409.

Schorch T. 1995 CAP: An Automated Self-Assessment Tool to Check Pascal Programs for Syntax Logic and Style Errors. *SIGCSE Bulletin.* Vol 27. No.1. pp.168-172.

Shepperd, M. 1988 An Evaluation of product metrics. *Information and Software Technology.* Vol.30. No.3. pp.177-188.

Shepperd M. 1992. Products, Processes and Metrics. *Information and Software Technology.* Vol.34. No.10. pp.674-680.

Shepperd, M. & Inc, D.C. 1994. A critique of Three Metrics. *Journal Systems Software.* Vol. 26. pp.197-210.

Shneiderman, B. 1976. Exploratory Experiments in Programmer Behaviour. *International Journal of Computer and Information Sciences.* Vol.5. pp.123-143.

Shneiderman, B. and Mayer R.E. 1979. Syntactic / Semantic Interactions in Programmer Behaviour: A Model and Experimental Results. *International Journal of Computer and Information Sciences.* Vol.8. pp.219-238.

Solway,E. and Ehrlich, K. 1984. Empirical Studies of Programming Knowledge. *I.E.E.E. transactions on Software Engineering.* Vol.10. pp595-609.

Sommerville, I. 1996. *Software Engineering.* 5th. Ed. Addison Wesley.

Sopher J.C. and Solway, E. 1986. Novice Mistakes: Are the Folk Wisdoms Correct? *Communications of A.C.M.* Vol.29. No.7. pp624-632.

Thorburn. G. and Rowe. G. 1999 PASS: An Automated System for Program Assessment. *Computers & Education.* Vol.29. No.4. pp.195-206.

Thorburn. G. and Rowe. G. 1996 PASS - An Automated System for Program Assessment. *4th All Ireland Conference on the Teachning of Computing. 27th-30th August* Dublin..

Webb, N.M. Ender P. and Lewis, S. 1986 Problem Solving Strategies and Group Learning Computer Programming. *American Educational Research Journal.* Vol.23. pp.243-261.

Wiedenbeck, S. 1985 Novice / Expert Differences in Programming Skills. *International Journal of Man Machine Studies.* Vol.23. pp.3883-390.

Appendix A.

Problems solved by the Novices

Problem 1 - Weather Station Problem

Problem 2 - River problem

Problem 3 - Traffic Problem

Problem 4 - Tennis Balls Problem

Problem 5 - Tyre Wear Problem.

Problem 1 Weather Station Problem

A new weather station is to be placed on the top of Cairn Aosta above the Glenshee skiing Developments. Readings are taken every 4 hours from 4.00 a.m. on Monday until midnight on Sunday. These weekly readings are collected and the data automatically transferred to the main computer at UAD. Software needs to be written to analyse the weekly data.

As will all good projects a pilot study is to be undertaken for the week commencing Monday 24th June. In the pilot study only 2 readings are taken by the weather station, the current temperature and the current wind speed. The number os readings will be extended in the full implementation. The analysis needed to be carried out is to calculate the minimum and maximum temperature and wind speeds for each day. The experiment is also looking at the variations during the day hence the average values are to be calculated for each 4 hour time. Finally the overall minimum, maximum, and average of both wind speed and temperature is required.

The test file as collected on the University computer is a series of lines representing a sat of data. Each line contains the time (24 hour clock) date (day number only) temperature (degrees C) and wind speed (m.p.h.). For example the following data is possible:

```
4       24     2.6     20.9
8       24     4.8     21.5
12      24     10.2    29.0
16      24     16.4    35.8
24      24     2.2     15.4
4       25     7.3     15.3
8       25     9.6     14.3
....
....
24      30     2.5     5.0
```

The data will reside in a file called week*n*.dat, where *n* is the week number. Your program should prompt the user for the week number and from that input the correct file. An example file `week01.dat` is included in the class library. The data capture equipment is reliable and the data file will always contain the correct amount of data.

Problem 2 River Problem

As part of an investigation into providing flood defences for Perth, Tayside Region Water Services are monitoring water levels along a section of the Tay. Daily readings are taken every 100m. along a 1Km. section of river. The readings are taken of water velocity and the depth. The data is stored in the monitoring station for subsequent transmission to the main computer at UAD. Software needs to be written to analyse the weekly data.

As will all good projects a pilot study is to be undertaken for the week commencing Monday 24th June. In the pilot study the water velocity and depth is recorded at the 10 sites along the river. The number of readings may be extended in the full implementation. The analysis needed to be carried out is to calculate the minimum and maximum velocity and depth on each day. The experiment is also looking for the average velocity and depth taken at each monitoring station. Finally the overall minimum, maximum, and average of both velocity and depth is required.

The test file as collected on the University computer is a series of lines representing a sat of data. Each line contains the station number, date (day number only) velocity (m/s) and depth (m.). For example the following data is possible:

```
1       24      2.6     20.9
2       24      4.8     21.5
...
10      24      2.9     14.6
1       25      2.2     15.4
2       25      3.3     15.3
...
...
9       30      3.4     7.0
10      30      2.5     5.0
```

The data will reside in a file called tay*n*.dat, where *n* is the week number. Your program should prompt the user for the week number and from that input the correct file. An example file `tay01.dat` is included in the class library. The data capture equipment is reliable and the data file will always contain the correct amount of data.

Problem 3 Traffic Problem

The traffic flows along the A90 Perth to Dundee Road are to be investigated by the Scottish Office. Automatic traffic counters count the traffic flowing both east and west along the road. The data is then onto a computer file on the main computer at UAD. Software needs to be written to analyse the data collected during the week.

As with all good projects a pilot study is to be undertaken for the week commencing Monday 23rd. June. In the pilot the traffic flows east and west are collected for throughout the week. For the purposes of this pilot study the day is split into 8 periods of interest. For each period on each day readings of easterly and westerly traffic flow are available. The number of readings, and periods, may be extended in the full implementation. The analysis to be carried out is to calculate the minimum and maximum traffic flows in each direction on each day. The experiment is also looking at daily traffic patterns. They require the average flow rates, both east and west, for each time period, averaging over the different days of the week. Finally the overall minimum, maximum and average of the east and west traffic flows is required.

The test file as collected on the VAX computer is a series of lines representing a set of data. Each line contains the period, date (day number only) easterly and westerly flow (veh/min). For example the following data is possible:

```
1 23 2.6  20.9
2 23 4.8  21.5
....
8 23 2.9  14.6
....
....
7 29 9.4   7.0
8 29 2.5   5.0
```

The data will reside in a file called `a90Rn.dat`, where *n* is the week number. Your program should prompt the user for the week number and from that input the correct file. An example file `A90R01.dat` is included in the class library. You may assume that the data is always correct and there are no missing data values.

Problem 4 Tennis Balls Problem

A Tennis ball manufacturer must test 15 balls from each batch. These balls are tested until they are "unusable" with the elapsed time recorded. The manufacturer needs to know the average time until "unusable".

Write a program that allows the user to input the data for 15 tennis balls, calculate the average and standard deviation for the batch. From this you can calculate the threshold which is the average elapsed time minus twice the standard deviation. Finally the number of balls above the threshold is calculated.

Problem 5 Tyre Wear Problem

A Tyre manufacturer must test 20 tyres from each batch. These tyres are tested until they are have "illegal" depth tread, with the number of miles recorded. The manufacturer needs to know the average time until "illegal".

Write a program that allows the user to input the data for 20 tyres, calculate the average and standard deviation for the batch. From this you can calculate the threshold which is the average mileage minus twice the standard deviation. Finally the number of tyres above the threshold is calculated.

Appendix B.

The Data generated during this research:

Subject Cohort A – Weather Problem Results

Subject Cohort B – River Problem Results

Subject Cohort B – Tennis Problem Results

Subject Cohort C – Traffic Problem Results

Subject Cohort C – Tyre Problem Results

Subject Cohort A - Weather Problem Results

	sp	cat	N	Nc	cr	cpi	asc	sloc	srn	sl	st	rpi	tpi	rtpi	rrpi	sl'	st'	Mark	sp * mark
AAR	0.063	D	53	841.3	0.019	0.471	#####	1	1	1	0	1.000	0.472	0.030	0.063	0	0	0.141	0.009
ACX	0.125	D	44	352	0.341	0.607	4.625	2	2	1	0	0.133	0.432	0.054	0.017	0	0	0.169	0.021
AH2	0.750	C	149	198.7	0.342	0.480	8.000	4	7	2	1	0.137	0.369	0.277	0.103	0	0	0.215	0.161
AKR	0.500	C	19	38	0.526	0.889	5.500	1	2	1	1	0.200	0.526	0.263	0.100	0	0	0.313	0.157
AM1	1.000	C	37	37	0.757	0.778	3.250	2	3	1	1	0.071	0.243	0.243	0.071	0	0	0.273	0.273
AM2	1.000	C	34	34	0.265	0.440	9.333	6	1	1	1	0.111	0.353	0.353	0.111	0	0	0.226	0.226
AWW	1.000	A	23	23	0.609	0.889	3.250	5	9	5	5	0.643	0.739	0.739	0.643	0.75	0.75	0.755	0.755
CCX	0.750	C	38	50.67	0.868	0.600	2.667	1	5	1	1	0.152	0.211	0.158	0.114	0	0	0.218	0.164
DEX	1.000	A	11	11	0.364	0.714	3.333	2	4	1	1	1.000	0.818	0.818	1.000	0	0	0.633	0.633
DHM	1.000	A	7	7	0.429	0.500	5.000	2	2	1	1	0.667	0.571	0.571	0.667	0	0	0.435	0.435
GCS	1.000	B	151	151	0.563	0.742	3.640	9	9	9	9	0.106	0.384	0.384	0.106	0.75	0.75	0.496	0.496
GDX	1.000	C	52	52	0.577	0.500	3.200	1	5	1	1	0.167	0.308	0.308	0.167	0	0	0.244	0.244
GRL	1.000	A	23	23	0.609	0.889	2.500	7	7	7	6	0.500	0.652	0.652	0.500	1	1	0.76	0.76
HMX	0.000	D	145	#####	0.000	0.368	#####	22	0	1	0	0.000	0.510	0.000	0.000	0	0	0.092	0
LDN	1.000	A	13	13	0.538	0.833	3.000	4	4	2	2	0.571	0.692	0.692	0.571	0	0	0.524	0.524
LKN	0.125	D	44	352	0.068	0.073	#####	2	2	1	0	0.667	0.273	0.034	0.083	0	0	0.048	0.006
MBX	1.000	B	81	81	0.680	0.852	2.625	7	5	5	3	0.091	0.333	0.333	0.091	0.75	0.25	0.444	0.444
MRT	0.000	D	24	#####	0.000	0.478	#####	6	0	1	0	0.000	0.458	0.000	0.000	0	0	0.12	0
MSG	0.188	D	16	85.33	0.875	1.000	2.000	1	2	1	1	0.143	0.250	0.047	0.027	0	0	0.268	0.05
NDG	0.875	C	21	24	0.762	0.800	2.667	1	3	1	1	0.188	0.333	0.291	0.165	0	0	0.314	0.275
NSG	1.000	B	49	49	0.694	0.800	2.875	6	7	6	6	0.206	0.388	0.388	0.206	1	1	0.599	0.599
RAH	1.000	B	23	23	0.609	0.667	4.000	3	2	3	3	0.214	0.391	0.391	0.214	0.25	0.25	0.381	0.381
SKX	1.000	C	27	27	0.704	0.750	3.667	2	3	2	1	0.158	0.333	0.333	0.158	0	0	0.31	0.31
SRJ	1.000	A	11	11	0.545	0.600	2.667	3	6	3	2	1.000	0.818	0.818	1.000	0.25	0	0.636	0.636

Subject Cohort B - River Problem Results

	sp	N	steps	Nc	cr	cpi	asc	sloc	srn	sl	st	rpi	tpi	rtpi	rrpi	sl'	st'	Mark	sp*mark
AGX	1	C	24	24	0.458	0.692	3.6	4	3	1	1	0.273	0.5	0.5	0.273	0	0	0.366	0.366
AH2	0	D	20	#####	0	0.632	21	3	0	1	0	0	0.6	0	0	0	0	0.158	0
AL1	0.188	D	29	154.7	0.069	0.462	28	3	1	4	3	0.5	0.483	0.091	0.094	0.25	0	0.193	0.036
APX	0	D	11	#####	0	0.2	12	1	0	1	0	0	0.273	0	0	0	0	0.05	0
AS2	0.125	D	35	280	0.143	0.552	8.5	4	1	1	0	0.2	0.486	0.061	0.025	0	0	0.159	0.02
AWM	0.5	C	85	170	0.424	0.469	6.4	5	5	5	4	0.139	0.376	0.188	0.069	0.5	0.25	0.275	0.138
BBX	0	D	41	#####	0	0.6	42	2	0	1	0	0	0.585	0	0	0	0	0.15	0
CCX	0	D	66	#####	0	0.169	67	2	0	1	0	0	0.303	0	0	0	0	0.042	0
CMX	0.688	B	66	96	0.47	0.771	4.182	2	4	2	2	0.129	0.47	0.323	0.089	0	0	0.296	0.203
CPX	0	D	78	#####	0	0.403	79	5	0	1	0	0	0.397	0	0	0	0	0.101	0
CTX	0.063	D	63	1008	0.127	0.333	14.75	9	1	1	0	0.125	0.302	0.019	0.008	0	0	0.09	0.006
DC1	0	D	34	#####	0	0.212	35	1	0	1	0	0	0.294	0	0	0	0	0.053	0
DTX	0	D	26	#####	0	0.44	27	2	0	1	0	0	0.423	0	0	0	0	0.11	0
JAC	0	D	46	#####	0	0.422	47	4	0	1	0	0	0.413	0	0	0	0	0.106	0
JAM	0	D	17	#####	0	0.5	18	1	0	1	0	0	0.471	0	0	0	0	0.125	0
JF2	1	B	116	116	0.569	0.64	4.125	6	9	5	5	0.136	0.353	0.353	0.136	0.5	0.5	0.407	0.407
JHX	1	B	50	50	0.26	0.622	8.4	6	3	6	5	0.23	0.52	0.52	0.23	0.75	0.5	0.499	0.499
JPK	0	D	19	#####	0	0.444	20	1	0	1	0	0	0.421	0	0	0	0	0.111	0
KDX	0.5	C	37	74	0.514	0.5	7	6	3	1	1	0.158	0.378	0.189	0.079	0	0	0.192	0.096
KLX	0.438	C	32	73.14	0.469	0.706	4.4	4	4	1	2	0.267	0.5	0.219	0.117	0	0	0.26	0.114
MH1	0.063	D	31	496	0.484	0.813	3.67	2	1	1	0	0.067	0.452	0.028	0.004	0	0	0.211	0.013
MNX	0.188	D	32	170.7	0.781	1	2.167	2	3	2	2	0.12	0.313	0.059	0.023	0	0	0.27	0.051
MP2	0.75	C	24	32	0.042	0.522	24	1	1	1	1	1	0.541	0.406	0.75	0	0	0.419	0.315
MWX	0.063	C	19	304	0.053	0.667	19	1	1	1	0	1	0.684	0.043	0.063	0	0	0.193	0.012
PDX	0.938	B	139	148.3	0.489	0.69	3.95	9	11	10	8	0.162	0.432	0.405	0.152	0.25	0.75	0.437	0.409
PLX	1	B	66	66	0.343	0.628	4.909	10	7	6	5	0.304	0.515	0.515	0.304	0.75	0.5	0.518	0.518
RHX	0	D	31	#####	0	0.4	32	2	0	1	0	0	0.387	0	0	0	0	0.1	0
SB1	0.125	D	130	1040	0.038	0.25	26	5	1	1	0	0.2	0.282	0.035	0.025	0	0	0.078	0.01
SH1	1	B	155	155	0.27	0.531	7.64	11	6	11	10	0.143	0.426	0.426	0.143	0	0.25	0.306	0.306

Subject Cohort B - Tennis Problem Results

	sp	N	steps	Nc	cr	cpi	asc	sloc	srn	sl	st	rpi	tpi	upi	rrpi
AGX	1		21		0.524	0.5	3.5	7	5	5	2	0.455	0.476	0.476	0.455
AH2														0	0
AL1	1		31		0.065	0.379	15.5	4	2	1	1	1	0.419	0.419	1
AWM	1		81		0.358	0.558	6.78	4	5	3	2	0.172	0.42	0.42	0.172
BBX	0.667		59		0.017	0.526	30	3	1	2	2	1	0.525	0.35	0.667
bmx	1		9		0.444	0.8	2.67	3	2	2	2	0.5	0.667	0.667	0.5
CMX	1		29		0.31	0.65	5	2	3	2	1	0.333	0.552	0.552	0.333
CPX	1		44		0.114	0.436	14	1	3	1	2	0.6	0.477	0.477	0.6
CTX	0.833		97		0.237	0.575	6.692	9	3	5	6	0.13	0.463	0.386	0.108
DC1	1		22		0.727	0.833	4	2	5	2	3	0.313	0.455	0.455	0.313
DTX	0.833		44		0.432	0.625	3.27	4	4	4	5	0.21	0.432	0.36	0.175
gfx	0		31		0	0.366	32	2	0	1	0	0	0.355	0	0
JAC	0		12		0	0.545	13	1	0	1	0	0	0.5	0	0
JAM	1		59		0.22	0.644	7.57	6	6	5	5	0.461	0.593	0.593	0.461
jf1	1		27		0.185	0.571	8.33	2	3	1	1	0.6	0.556	0.556	0.6
JHX	1		98		0.326	0.606	4.67	10	6	9	9	0.188	0.47	0.47	0.188
JPK	0.833		38		0.342	0.64	4.125	4	4	3	4	0.308	0.526	0.438	0.257
jsx	0.5		29		0.483	0.643	3.5	4	3	2	2	0.214	0.414	0.207	0.107
KDX	1		117		0.598	0.702	3.611	13	5	14	14	0.071	0.325	0.325	0.071
KLX	1		42		0.357	0.519	5.5	3	4	2	3	0.267	0.429	0.429	0.267
MNX	0		56		0	0.255	57	1	0	1	0	0	0.25	0	0
MP2	1		27		0.592	0.727	2.833	21	2	2	2	0.125	0.37	0.37	0.125
MWX	1		60		0.233	0.521	12.5	2	2	2	3	0.143	0.433	0.433	0.143
PDX	1		34		0.411	0.75	4.333	3	6	3	3	0.428	0.618	0.618	0.428
PLX	1		105		0.362	0.597	6.153	7	3	5	6	0.131	0.429	0.429	0.131
RHX	1		85		0.541	0.59	4.9	4	5	7	6	0.109	0.329	0.329	0.109
SB1	0.5		28		0.571	0.75	3	3	3	1	1	0.188	0.429	0.215	0.094
SH1	1		114		0.096	0.422	15.7	6	3	6	6	0.273	0.404	0.404	0.273
tbx	1		11		0.455	0.5	4	2	3	2	2	0.6	0.545	0.545	0.6

Subject Cohort C - Traffic Problem Results

	sp	N	steps	Nc	cr	cpi	asc	sloc	srn	sl	st	rpi	tpi	rtpi	rrpi
BHX	1	3	8	8	0.875	1	2	2	4	1	1	0.571	0.625	0.625	0.571
CHW	1	3	71	71	0.042	0.456	35	8	2	1	1	0.667	0.464	0.464	0.667
DAS	1	3	6	6	0.667	1	2	1	1	1	1	0.25	0.5	0.5	0.25
dhx															
DJN	0.063	4	52	832	0.173	0.429	9.6	3	1	1	0	0.111	0.365	0.023	0.007
DSX	1	2	128	128	0.625	0.583	4	6	8	7	6	0.1	0.281	0.281	0.1
fsx															
GTW	1	1	46	46	0.326	0.613	5.429	4	2	1	1	0.133	0.457	0.457	0.133
IRC	0	4	11	#####	0	0.4	12	1	0	1	0	0	0.364	0	0
JDG	1	3	42	42	0.405	0.56	13.5	1	5	1	1	0.294	0.5	0.5	0.294
JLM	1	3	71	71	0.056	0.462	68	6	3	1	1	0.75	0.479	0.479	0.75
JMM	1	3	40	40	0.175	0.485	34	7	4	1	1	0.571	0.5	0.5	0.571
MBX	1	2	66	66	0.212	0.481	18.33	9	3	2	1	0.214	0.439	0.439	0.214
MDX	0.063	4	12	192	0.167	0.222	11	1	1	1	0	0.5	0.25	0.016	0.031
MEM	1	2	34	34	0.059	0.5	33	7	1	2	1	0.5	0.5	0.5	0.5
MGP	1	3	37	37	0.541	0.824	5.25	3	4	1	1	0.2	0.486	0.486	0.2
NGS	0	4	21	#####	0	0.15	22	5	0	1	0	0	0.286	0	0
PGX	0	4	14	#####	0	0.385	15	3	0	1	0	0	0.429	0	0
RD4	1	2	43	43	0.186	0.257	36	4	3	2	1	0.375	0.349	0.349	0.375
RED	1	2	45	45	0.289	0.594	17	8	5	3	1	0.385	0.533	0.533	0.385
RTX	0.063	4	59	944	0.051	0.393	29	4	1	1	0	0.333	0.39	0.024	0.021
SHX	1	2	61	61	0.689	0.737	4.167	6	7	3	4	0.167	0.344	0.344	0.167
TDD	0	4	11	#####	0	0	12	2	0	1	0	0	0	0	0
TMA	1	3	52	52	0.712	0.8	2.5	1	3	1	1	0.081	0.288	0.288	0.081

Subject Cohort C - Tyre Problem Results

	sp	N	steps	Nc	cr	cpi	asc	sloc	srn	sl	st	rpi	tpi	rtpi	rrpi
BHX	1		28		0.607	0.636	4.667	3	4	4	4	0.294	0.428	0.428	0.294
CHW	0.875		42		0.429	0.583	5.8	3	3	4	4	0.167	0.404	0.354	0.146
DAS	0		29		0	0.321	30	1	0	1	0	0	0.31	0	0
dhx	0.25		20		0.45	0.4	6.5	1	2	1	0	0.222	0.3	0.075	0.056
DJN	0.25		37		0.054	0.542	18.5	3	1	1	0	0.5	0.541	0.135	0.125
DSX	1		20		0.45	0.727	6.5	3	3	3	1	0.333	0.55	0.556	0.333
fsx	1		10		0.6	0.75	3	2	2	2	2	0.5	0.6	0.556	0.5
GTW														0	0
IRC	0.875		32		0.313	0.727	12	1	4	1	1	0.4	0.625	0.547	0.35
JDG	0.875		70		0.371	0.704	5	4	3	5	4	0.115	0.486	0.425	0.101
JLM	1		65		0.507	0.688	5.571	5	6	5	5	0.182	0.431	0.431	0.182
JMM	1		49		0.429	0.714	3.8	5	6	5	6	0.286	0.531	0.531	0.286
MBX	0		46		0	0.266	47	2	0	1	0	0	0.261	0	0
MDX	1		28		0.321	0.631	4.8	5	3	4	3	0.333	0.536	0.536	0.333
MEM	0		12		0	0.364	13	1	0	1	0	0	0.333	0	0
MGP	1		21		0.762	1	2.25	2	2	2	2	0.125	0.333	0.333	0.125
NGS	0.25		87		0.287	0.387	6.636	4	1	1	0	0.04	0.287	0.072	0.01
PGX	0.125		36		0.111	0.548	11.67	2	1	1	0	0.25	0.5	0.063	0.031
RD4	0.75		154		0.37	0.525	5.042	8	5	5	5	0.088	0.363	0.272	0.066
RED	1		53		0.604	0.809	5.2	2	5	4	2	0.156	0.415	0.415	0.156
RTX														0	0
SHX	1		27		0.629	0.8	3	2	3	1	1	0.176	0.407	0.407	0.176
TDD	0.25		23		0.087	0.6	8	2	1	1	0	0.5	0.565	0.141	0.125
TMA														0	0

Appendix C.

Lund, G.R. 1995. The Program Development Process. *In Innovations in Computing Teaching.* SEDA Paper 88.

Lund, G.R. 1995 Controlling Plagarism in Student programs IN: *Proceedings of 3rd. Annual Conference on the Teaching of Computing.* 29th August – 1st September, Dublin, Ireland.

Lund, G.R., Elder, L., Natanson,L.D., and Miller, C.J., 1997 The Importance of Process In: *Proceedings of 5th. Annual Conference on the Teaching of Computing.* 6th – 29th August Dublin, Ireland.

Title: *The program development process*

Authors: *Geoff Lund*

Institution: *University of Abertay Dundee*

General	
Computer Science & Programming	✓
Information Systems	
Computer Literacy	

Characteristics of the Teaching & Learning Strategy		The Problems Tackled		Transferable Skills and Competences Developed	
Project		Increasing group size		Self-responsibility	
Laboratory		Different Student ability levels		Independent learning	
Work experience/ field work		Non-specialist subject		Groupwork/teamwork	
Feedback to student	✓	Motivation/variety	✓	Personal development	✓
Student-centred learning	✓	Relevance/realism		Time Management	
Ownership of learning		Other learning issues		Written communication	
CAL		Teaching issues	✓	Oral presentation	
Peer assessment		Peer assessment		Using information sources	
Self assessment		Self-assessment	✓	Managing tasks and solving problems	
Group teamwork		Curriculum issues		Numeracy	
Written presentation		Other assessment issues	✓	Information Technology	
Oral presentation		Financial resources			
Video presentation		Human resources			
External links/industry		Physical resources			
Credit acc. & transfer		Quality of contact time			
Profiling/portfolios		Staff development			

11 The program development process

Geoff Lund
University of Abertay Dundee

Background

At universities throughout the country there is a large number of introductory programming courses aimed at students studying BSc Computing or for a similar qualification. The aims of these courses are written in course documents and many are similar to those at the author's university. These are:

> *'To provide students with an understanding and practical experience of the production and maintenance of software implemented in a high-level language to meet the user requirements'.*

To achieve these aims we present programming courses in a variety of ways. However, many of these courses are similar in the manner in which they are presented, with a lecture course plus practical time allocated to enable the student to write programs. A number of programs will be submitted for assessment throughout the course. We expect these programs produced by our students will:

- meet user requirements
- be developed using good programming practice
- exemplify good programming style.

Traditionally, lecturers assess course work by assessing the quality of the final code and how well it meets the specification given. Students hand in their code with some output and this is used as the basis of our assessment. We do not usually assess the method of achieving the final product. Perhaps there is an assumption that a good program has evolved from a good programming method. Today there is ever more emphasis on quality in all organisations. One aspect of quality is based on the premise that to produce a quality product one needs to set up a quality process. We, as university teachers, assess the quality of the product and assume that the quality of the process to achieve the product was as good. This is not an acceptable solution. Thus there is a need to teach and assess the programming process.

In this paper the author will describe attempts to examine and assess the programming process.

The programming process

Faced with a problem, how do we go about writing a program to solve it? As an illustration consider the following example taken from a first year programming course.

Problem:

Write a program that processes rainfall records throughout Scotland. There are 30 sites used to measure the rainfall every day for a four week period. The program should calculate the average rainfall at each site, average rainfall each day and the overall average. The data is to be read from a file. Each input line consists of:

day number site number rainfall amount

There are a number of sub problems contained in the main problem. The student has to:

Read the file
Calculate the average of each site
Calculate the average for each day
Calculate the overall average
Output the results

One way to write the program is to use pseudocode to express the solution to the problem in a top-down manner. The student will work at getting a fully worked design before attempting any coding. The design will then be coded and tested. Any test failures will result in the student altering the code until success is achieved. This is a traditional design first method of working which is suggested by many text books.A second way to solve this problem is to think about the sub problems and, design and code them in turn. Hence the student may initially concentrate on reading the file successfully, then design and write code to allow the average for each site to be calculated. On success, attention will be focused on averaging the daily rainfall. The subprograms are tackled one by one until the whole problem is solved. This incremental development method has a number of advantages over the first method namely:

1. Students often cannot be bothered with designing programs using pencil and paper and equate programming with typing into the computer. With incremental development they can legitimately type in earlier in the development.

2. By adding to a solution, the position in the code where an error occurs is narrowed down to the new code added rather than anywhere in the program.

3. The student can focus on smaller sub problems which are intellectually easier to manage.

Software engineering is different to most other engineering disciplines, civil or mechanical, in that you do not need a fully working design prior to starting to build the product. It is very useful to be able to part build a program prior to designing the rest of the program - something not available to

the rest of the engineering community. It is this second method of incremental development that is used as the model for program development in this paper.

As an aside, focusing on incremental development in this way may produce a different program than if a fully worked design is carried out prior to any coding. In the sample problem above, using incremental development would inevitably lead to storing the data in a two dimensional array, whereas the traditional design first method could lead to a program where the raw data is not stored.

Measuring the programming process

Students are not going to log how they went about tackling the program accurately unless there is something in it for them. Moreover if they know they are being assessed on how they achieve the final result they will tend to record their progress selectively. The lecturer could record how well the student is progressing but lecturer time is only finite and developments outside class time will go unrecorded. Hence we are left with trying to record the progress of the student automatically. To do this we need to interact with the programming process. At the author's university and a number of others, programming is taught using TURBO PASCAL, which provides an integrated editing, compiling and run system which is much better than the separate editor and compiler available on many mainframe computers. It is not realistic to add items to the TURBO PASCAL system itself and it would be stupid to try to write our own integrated environment. To monitor the programming process we must build some software that the students will use regularly throughout the development of the program. The software must give the students something extra that TURBO PASCAL does not. A test monitor has been built that allows student to run their programs through a set of test data. The students now have an easy way of testing their programs. This monitor provides the student with a report indicating which tests have passed and which have failed. At the same time as testing the program for the students, the program copies the student's program for later analysis by the lecturer.

The lecturer now has available a number of versions of the students program. If the student uses a good programming method then the program will grow in size and function as the student puts in more detail until the final program passes all the tests. The size of the program can easily be measured by counting the number of lines of code (ignoring blank lines and comments). The function of the program can be measured by counting the number of tests passed. A weak student, one that does not develop the program systematically, would have a more erratic development method. The code may not increase with time but may reduce in size at certain time indicating an attempt at a solution that did not work and was discarded. Weaker students may also fail tests which were previously passed indicating problems with the program development.

Once the programs have been submitted for assessment an analysis is made from the different versions of the programs. The analysis shows a series of number pairs indicating the number of lines of code and number of tests passed. From this the marker can give mark the process carried out by the student. The programs submitted for assessment have three components to their marks, one assesses the correctness, the second assesses the quality of the final product and the new component assesses the development method.

The author has used this method in a number of experiments with one group of first year students. The results are useful in showing which students develop their program well and which do not. However there are a number of problems which are discussed below.

Discussion of problems with examining programming process

There are a number of problems highlighted by this method:

Picking suitable examples. When a program is tested there must be an easy way to find out if the test has been successful or not. The best examples for this type of analysis are those that calculate a distinctive number or numbers. To find if the program is correct is simply a matter of searching through the output for the distinctive string. The example given above falls into this category; for example, data can be chosen to give different averages for each calculation. Programs where layout (reports) or order (sorting programs) are important are not so amenable to analysis.

Measuring the process. At present the measures of the process are crude, especially counting the number of tests passed. Even good students make mistakes and sometimes fail tests that have previously been passed thus causing their progress to look erratic. Focusing on the number of tests passed can mask some bad practice. A student may pass the test they are trying to achieve but in the process fail a previously passed test. This is masked when the number of tests is taken as the measure. An alternative could be to use the easiest test failed but this causes problems of how to order the tests.

Choosing the tests. The lecturer must be very precise in the description of the problem to ensure no misunderstanding in the students' mind. If there is doubt about the structure of the test data then students will be penalised for bad production process when in fact they misunderstood the problem. The lecturer must also provide a set of test data to test the programs to be written by students. The provision of such a set of data is not easy. The lecturer has to pick the tests from their knowledge of the problem rather than knowledge of the program under test. It is relatively easy to pick tests that would test the full working system but the lecturer also has to provide tests to test part of the program without making assumptions concerning the order of program development for the student. The author tried to order the tests from easy to hard and found that the students' ordering of the tests would not have been the same. Different solution methods will find different parts of the problem easier to pass. A full test suite testing every statement in the solution cannot be provided, but indicative tests are chosen that ensure a reasonable coverage of the problem space. Students can 'fix' the results if they know what data is being input hence some variation in the actual numbers in the test data through the development process is needed.

Looking at the whole process. The method described in this paper allows the lecturer to find information about how the student produced the program. The student will however have made many attempts at compiling the program and perhaps run the program a few times on their own data prior to trying to pass the lecturer's tests. Hence the data collected only looks at the final stages in the development of the program. Investigations are being made into how data can be collected throughout the whole development process.

Objective marking needed. The results from the analysis of the different versions of a student's program are examined subjectively by the lecturer and translated into a mark. It would be much better if an objective mark, an 'extent of development', could be gained from the data. This could then easily be translated into a grade.

Correctness measure achieved for free. Using this method of working has a couple of advantages not related to the measurement of process. Firstly the lecturer has the test results of all programs in a standard way thus allowing him to mark how correct the program is. Secondly as there is a full history of the program as developed, there is less opportunity for a student to copy another's work. The lecturer can be sure that the work done is the students' own.

Conclusions

Traditionally, when marking student programs the assessment is based on the quality of the final code and the correctness of the product. This ignores an important aspect of programming, namely the programming process. How did the student achieve the final result? This paper discussed a way in which the development process can be examined. When students submits their final program for marking, three components of the mark are given: one marking the programs correctness, one marking the quality of the final product and one marking the development process. The development process for a program must be examined to allow a lecturer to assess how a student achieves the final program. To do this in a cost-effective way means that the data collection must be done automatically. This paper describes an experiment in which the students have access to a test monitor that tests their programs, reporting on the results. As part of its task the test monitor takes a copy of the program under test for later analysis. Thus on submission the lecturer has a full version history of the program.

Each version of the student program is measured for size, lines of code, and correctness, and how many tests were passed. These measures are transformed into a mark indicating how well the program was developed. There are a number of limitations in the method described in this paper. However, the overall aim of assessing the program development method used by a student is an attractive idea.

Controlling Plagiarism in Student Programs

G.R.Lund.
Department of Mathematical and Computer Sciences.
University of Abertay Dundee
Bell St.
Dundee
DD1 1HG

Correspondence should be addresses to:
Mr. Geoffrey Lund


Abstract

In recent years there has been a move away from conventional exams as a means of assessing programming courses towards coursework assessment. The rational being that we need to assess the students "doing" the programming rather than just talking about it. The exam system has one large advantage over coursework, in that the lecturer can be assured of the authorship of the work. Unfortunately we find that some of our students are not the sole author of the work they submit to be assessed. A small amount of help from a fellow student may be a good thing. However submitting another student's work as their own is not permitted. Dealing with students who have copied work is not a pleasant task for the lecturer so we tend to avoid dealing with this important subject. This paper will discuss ways of controlling plagiarism in student programs.

Plagiarism can be controlled in two ways. Assessments can be designed in such a way that the lecturer can ensure the student did the work. Alternatively the lecturer can put in place a mechanism to detect plagiarism.

Lecturers pride themselves in being able to detect pieces of copied work. Programs have a fingerprint that a lecturer can detect by eye. There is usually some style anomaly or peculiar method used that gives it away. When confronted with the lecturer's suspicions of plagiarism many students confess to copying. However some do not and it is incumbent on the lecturer to prove their case. Proving plagiarism needs hard evidence, especially if it is to be brought before an appeals board. Work in the late 1980s showed how software could be used to detect plagiarism in programs. The software is not sensitive enough. It can detect plagiarism in more obvious cases but can easily be confused by extra pieces of code.

A better method is perhaps to develop courseworks that avoid the problem totally. One way is to have some aspect of the coursework carried out under exam conditions. This is easier to do for small programs and not suitable for large programs. An alternative method is to chart the development of the program. By examining the coursework over a number of versions some of which occur during class time the lecturer can be more convinced that the work is the students own. These and other methods will be discussed in the paper.

Plagiarism is not a topic that lecturers wish to dwell on. By taking steps to combat plagiarism we are in some way thinking ill of the students. However we all know that plagiarism exists and any professional lecturer must consider the issue and put in place steps to control it.

1. Introduction

A decade or so ago students were primarily assessed for their degree by examination. More recently there has been a move towards coursework playing a more important part in student assessment. The rational being that we need to see the students "doing" rather than just talking about it. The importance placed on coursework differs between universities with the "new" universities usually placing more importance on coursework than the "traditional" universities. Also the proportion of coursework used in the final examination tends to decrease throughout the course. However the recent quality assessment of computing departments in UK universities showed at least one university where coursework formed 60% of the honours degree mark, (Clare 1995).

One advantage of an examination over coursework is that the lecturer can guarantee that the work done by the student is their own. Unfortunately we find that some of our students are not the sole author of the work they submit to be assessed. A professional lecturer must ensure that the coursework mark given to a student reflects their effort and ability and not the effort and ability of a "friend". This is often hard for the lecturer. Most good lecturers try to build up a relationship between themselves and their students. The act of questioning the authorship of a piece of work can be an unpleasant business and can destroy the trust between lecturer and student. For this reason we should perhaps look for ways of avoiding plagiarism rather than detecting it.

Most plagiarism is easy to detect. Students hand in direct copies of their friend's work. These students do not understand the work and have no idea how to hide their naivety. More sophisticated students may alter or add to another's work which makes the detection task harder. When the work looks like a copy the lecturer must prove the copying. In the absence of an admission from the students this evidence must be able to stand up in an appeal procedure. Hence we must have sound plagiarism detection systems if we are to ever combat the plagiarism problem.

A second method is to devise student assessments that avoid plagiarism. These may be individual tasks or controlled tasks where the lecturer can be sure of the authorship of the piece of work.

This paper starts by discussing different levels of copying from direct copies of a friend's work to asking for help on a piece of work. A categorisation of copying is needed so that we as lecturers can clearly state what is and what is not allowed. Once we understand what plagiarism is a lecturer needs a clear policy regarding what happens if plagiarism occurs. This is discussed next in the paper. The paper then deals with the different methods of plagiarism detection and finally goes onto discuss assignment methods that avoid plagiarism.

The paper concentrates on plagiarism in programming courseworks although some of the

ideas may be useful in other areas.

2. Levels of copying.

Not all copying is bad. Two or more students may work together on a problem and produce similar work to which they have all cooperated. Most lecturers would agree that this is an acceptable way of working for students who cannot solve the problem themselves. At the other end of the scale, a student getting someone else to do the work would not be acceptable to most lecturers. Below is presented a scale of copying:

1. **Identical copy:** This can be the work of another student that has been stolen and retyped with only a change of name, or another student may do the work for some reward (financial or otherwise). Solutions may also be bought from more experienced programmers.

2. **Duplicate work**: The work is altered using the editor, perhaps the layout is altered, or some variable names, but essentially the structure of the program is unaltered.

3. **Additions:** A program is taken, perhaps not fully working, and altered by adding to the program. This could be extra procedures, different report layouts but in some way adding to the program but keeping to the overall structure.

4. **Rebuilding**. This is where a student takes another's program and alters it significantly so that the structure of the program is altered.

5. **Collaboration.** This is where a group of students work together on a program producing similar programs with each student participating.

6. **Help**. This is where a student may ask another student for some help on a small part of the program but has devised the overall structure themselves.

7. **Solo**: The student has done all the work themselves.

What is fair and what is not? The author suggests that level 1 and 2 are not acceptable as the student is trying to get credit for someone else's work. Level 3, 4 and 5 would be acceptable is the student acknowledged the help from their colleagues. Level 6 and 7 is acceptable. Another way to look at the classification is to note the difference between level 3 and level 4. Level 3 and lower levels of copying indicate that the student did not produce the overall design of the program themselves. They have copied the structure from another program. If

we were to draw structure charts of the two programs they would be similar in nature. In level 4 and above the structure charts of the two programs would be different. This indicates that the student has given some thought to the overall structure and has obtained help in the detail which is arguably better.

3. Plagiarism Policies.

The classification above allows a lecturer to present to a student a policy regarding the authorship of work. The need for such a policy was identified by Shaw(1980). This policy should cover:

1. What is acceptable working together
2. What is unacceptable
3. How plagiarism is to be detected
4. What action is to be taken following detection of unacceptable plagiarism.

The author uses the following rules which are similar to those distributed with Ceilidh (Benford et. al. 1993):

1. Handing in a piece of work with your name on is an indication that the work is solely the work of the author. If this proves to be incorrect disciplinary action will take place.

2. Plagiarism includes copying a program or part of a program from a book, or another student with or without their knowledge. If work has been copied the original author must be acknowledged and the mark will be adjusted accordingly. No further action is taken if you acknowledge where the original came from.

3. Working in groups to create a single program that is submitted as your own is unacceptable.

4. Students can discuss the problems with their peers. Discussions about the algorithm used or data structure used are acceptable and indeed encouraged. However you must not copy the structure of your program from another. Asking for help on some small detail of the program is again acceptable.

5. If a lecturer suspects group working he will mark the work once and divide this by the number of students in the group. These students will be asked to meet the lecturer to discuss the work where a fairer distribution of marks may take place.

6. Outright copying will result in a mark of 0% being entered and disciplinary

measures taken in accordance with the examination regulations.

Crucial to the effective working of a plagiarism policy is a mechanism for the lecturer to detect plagiarism when it occurs. The majority of students will work fairly within these rules. There will however be some students who try to "cheat" and gain credit for work that is not their own. To be fair to the majority as well as keep up the university's standards we must be able to detect or control plagiarism. This is discussed below.

4. Controlling Plagiarism.

Most lecturers would wish to initiate methods that limit plagiarism. There are two possibilities. plagiarism can be detected when it occurs or avoided by using assessment techniques that rule out plagiarism. The latter, avoidance, keeps in tact the relationship between the lecturer and the student. The marking is simpler in that the lecturer is confident they are marking the students own work. however setting of assessment is harder and more limited in nature.

If no avoidance methods are used the marking must check the similarity between programs. various detection methods have been discussed in the literature, (Donaldson et. al. 1981, Grier 1991, Jankowitz 1988, Whale 1990, Wise 1992) and are discussed below. However these methods are not fool proof and errors occur. If a lecturer accuses a student of copying and they have not the student loses faith with the lecturer, course and university and in extreme circumstances may resort to law. If plagiarism is not detected then students will have marks awarded to which they are not entitled.

Plagiarism exists and must be tackled. However lecturers are often busy hence methods of detecting or avoiding plagiarism cannot be too time consuming in staff time. The next sections discuss various practical methods of plagiarism detection and avoidance.

5. Detection of plagiarism.

Methods for detection of plagiarism are based on software metric work. Simple systems are based on counting significant features of the program and the use of Halstead's metrics (Halstead 1977). These techniques do not pick up and more than level 1 and 2 plagiarism, hence if a student adds bits to a program they will not be found out. Recent work by Leach(1995) highlights a method of using Halstead's metrics, complexity measures and counts of coupling types which he claims to work well. Work by Jankowitz(1988) aimed to find plagiarism at higher levels by building a static execution tree for programs and comparing these. Earlier in this paper the author suggested that changes in structure of a program were key to deciding whether plagiarism has occurred or not. Techniques such as discussed by

Jankowitz are needed to allow this discrimination to be made.

The plagiarism detection system should be a two stage process. Stage one is to use the software to detect plagiarism as discussed above. The second stage in the detection system would be for the lecturer to instigate a discussion with the students to see if they know how the program works, why certain features were used etc. In the authors experience these discussions usually show who has done the work if it was a group effort or who copied if the programs are direct copies. The evidence gained in these two stages can be used to redistribute marks between students who worked in a group or initiate any disciplinary procedures necessary.

Any plagiarism detection system may show up plagiarism when it does not occur and not show plagiarism when it does occur. Minimising these errors is an essential part of the detection system. If a class of 100 students are all given the same piece of work to do then it is likely that independently a number will use the same algorithms and produce similar code. The second stage of the detection process is designed to show which students have worked independently even though they may gain a similar final program.

Students who are not found out by the detection method may feel a sense of relief at being allocated a mark when little effort was put in. These students are often known to their colleagues who feel resentment at getting perhaps a lower mark despite putting in a lot of effort. It is for these students that any detection system must be extremely accurate. It is argued by Leach (1995) that just the knowledge that there is a plagiarism detection system is sufficient to cut down dramatically the amount of plagiarism occurring.

It can be argued that if a student conceals the copying so well that the lecturer cannot detect the plagiarism they must have put some work into the program. To do this they must have understood the program they were altering and perhaps deserve to pass!

6. Avoidance of Plagiarism

Whereas detection of plagiarism involves some automatic analysis of the finished product, avoidance techniques must be included in the assessment at the beginning. There are two distinct avoidance techniques. One is to conduct the assessment in some controlled manner similar to an examination. In this way the lecturer is totally convinced of the authorship of the work but the assessments are limited in scope and may lead to examination fear in some students. The other avoidance technique is to give different work to each student thus illuminating any option of copying but not of another person doing the work for the student.

The simplest avoidance technique is an examination. However as was stated earlier in this

paper, examinations and coursework assess different features and the focus here is how do we assess a student writing a program. Various different "examinations" are possible:

Class test: Students are asked to write a program in a 3 hour controlled environment. This can really only be used with beginners where the program are small.

Whole Day Test: A program is given to the students at the start of the day and they are given all day (6-8 hours) to write a program to do the task. Programs cannot be too large as there is a time limit of perhaps 8 hours. The technique may work better if the task is to alter a program rather than write one from scratch but this is testing a different skill. This technique is often used in engineering disciplines to test design concepts.

Prepared tests: A coursework is given out two to three weeks prior to the class test. The students are asked to write a program to solve some problem. They bring their code along to the test where they are presented with another problem similar in nature to the original, or an addition to the original and asked to write a program to solve this new problem. This technique is similar to that discussed by Turner(1995).

The problem with examinations is that they are time limited. This causes stress to the students. The examinations as discussed above rely on technology which can fail thus invalidating the examination. Three further techniques can be used that are not time restricted:

Different coursework: Each student is given a different piece of coursework. This is too difficult in large classes so perhaps a smaller number of pieces of work are allocated amongst the class at random. The lecturer must ensure that all pieces of work are equal in difficulty and length.

Super problem: Here a large problem is developed with a number of alternatives From these alternatives an individual piece of coursework can be generated. The advantages are that the students are working on the same super problem and can discuss their work this gaining from group contact but no two students are writing the same program and hence plagiarism is limited. This method is reported in Smith and Tomlinson(1995).

Version Control: In this technique all the versions that a student goes through to come to the final product are kept and analysed. An analysis of the versions can tell how well the program was constructed which is an often neglected area of assessment, Lund(1995). A side effect of this is that the lecturer can spot plagiarism. For example if version 1 is identical to another students work, or, only

1 version exists, or a totally different program is submitted from that which was developed.

These techniques do not stop a student getting a friend to do the work for them. There is no control that the student typing in code is the student whose name is on the piece of work. This would have to be found from some other security system.

7. Conclusions.

This paper has discussed plagiarism and methods of avoiding or detecting plagiarism. the issue of plagiarism must not be ignored if we are to reward students for their efforts and not some one else's effort. Several methods of plagiarism avoidance or detection have been discussed. The author considers avoidance methods are better as they minimise the nastiness that can ensue when a lecturer accuses a student of cheating. Avoidance techniques usually involve more work in setting up the coursework.

The following steps should be taken by lecturers to deal with the plagiarism question:

1. Tell the students the acceptable and unacceptable levels of working together.
2. Inform the students of the disciplinary procedures that may occur if copying is found
3. Plan the assessments so that plagiarism can be avoided and the lecturer is confident the they are marking the students work OR
4. Put in place automatic plagiarism detection methods and a method to deal with the results of the detection system.

If students know that the issue is being dealt with seriously they will respond positively.

7. References.

Benford, S., Burke, E. and Foxley, E. (1993) 'Learning to construct quality software with the Ceilidh system', *Software Quality Journal* **2,** pp.177-197.

Clare, J. (1995) 'Universities fail to make the grade', *Daily Telegraph* March 22. 1995 p19.

Donaldson, J. L., Lancaster, A. M. and Sposato, P. (1981) 'A Plagiarism Detection System' *SIGCSE Bulletin* **13,** 1, pp21-25.

Grier, S. (1981) 'A Plagiarism Detection System' *SIGCSE Bulletin* **13,** 1, pp15-20.

Halstead, M. (1977) *Elements of Software Science,* Elsevier North-Holland, New York.

Jankowitz, H.T. (1988) 'Detecting Plagiarism in Student Pascal Programs' *The Computer Journal* **31,** 1, pp1-7.

Leach, R.J. (1995) 'Using Metrics To Evaluate Student Programs' *SIGCSE Bulletin* **27,** 2, pp41-48.

Lund, G. (1995) 'The Program Development Process' in Hart. J. (ed.) *Innovations in Computing Teaching* SEDA Paper 88 pp. 89-93.

Shaw, M. (1980) 'Cheating Policy in a Computer Science Department' *SIGCSE Bulletin* **12,** 1, pp72-76.

Smith, M. and Tomlinson, A. (1995) 'Copying and programming' in Hart. J. (ed.) *Innovations in Computing Teaching* SEDA Paper 88 pp. 105-109.

Turner, J. (1995) 'Post-assessment testing in computing' in Hart. J. (ed.) *Innovations in Computing Teaching* SEDA Paper 88 pp. 129-136.

Whale, G. (1990) 'Identification of Program Similarity In Large Populations' *The Computer Journal* **33,** 2, pp140-146.

Wise, M. J. (1992) 'Detection of Similarities in Student Programs : YAP'ing may be preferable to PLAGUE'ing' *SIGCSE Bulletin* **24,** 1, pp268-271.

The Importance of Process.

G R Lund, L Elder, C J Miller and L D Natanson

School of Informatics, University of Abertay Dundee, Bell Street, Dundee, DD1 1HG

[redacted]

ABSTRACT: *This paper investigates how students develop computer programs. It is argued that the development process is equally as important as the final product. Data has been collected during the development of computer programs and this is used to view the development process of novices and experts. A number of metrics are proposed that seek to track the evolution of computer programs. These metrics are also used as the basis for classifying students in terms of their relative skill.*

KEY WORDS: *Programming Process, Program Assessment, Novice Programmers.*

---

1. Introduction.

The teaching and assessment of computer programming is an important feature of most if not all Computing and Software Engineering degree courses. Current educational practice in the assessment of computer programs is based on appraising the final version of a program. Features such as programming style, correct use of constructs, complexity, efficiency and correctness are assessed and weighted to give an overall grade, as in Ceilidh (Benford et. al. 1993)

By contrast the software industry has in recent years focused on the software process, i.e. how the software is developed. Indeed the whole software engineering movement aims to improve the software development process and by inference improve the final product. In introductory programming education the process of actually producing software is given little emphasis; the emphasis is primarily on the final product alone. A survey of a number of standard programming text books indicates that little of each book deals with how the student is to create the program.

This paper is based on the premise that the process involved in producing a computer program is a key feature in influencing the final product, as discussed in Lund (1994). The first step in considering the program development process is to capture suitable data throughout the development cycle. This paper describes such a system. The system has been used to collect data from students and experts (staff) during the development of a computer program. Important characteristics of both the students and the expert program development processes are discussed here. The paper goes on to propose a number of objective measurements that can be used to characterise features of the program development process. These measurements clearly differentiate between classes of novice programmers.

2. Data Collection.

To investigate how a program has evolved a system must be put in place to capture data throughout the development of the program. The simplest way is to capture the state of the program at various times throughout its construction. A system has been written that captures the program every time it is submitted to the compiler. The students involved in this study were programming in Pascal using VAX/VMS. Instead of using the standard compile command the subjects were given an enhanced version that compiled, linked and ran their program and, invisibly to the students, copied programs to a safe place for later analysis. This method of data capture did not interfere with or delay the development process and the authors could be confident that the data was not corrupted by the capture method. Indeed the data capture method gave the students a better interface to the VMS compiler which meant they would use the system.

Once all versions of the program were captured some analysis was carried out. A number of counts were made from each program. These include the number of lines of code, comment lines and counts of various Pascal key words. Each version of the program was compiled to find the number of compilation errors. For those programs which compiled successfully a measure of what proportion of the specification achieved was calculated by detailing a number of tests that covered the whole specification and then counting how many of these were passed successfully.

At the end of this procedure each version of the program was reduced to a number of counts and the change in these counts over the development period represented the program construction process.

3. So how do students develop programs?

The data as collected and analysed were inspected to see if any useful information could be gained. Graphs of number of lines of code against version number show the development of programs in terms of the code added. A second set of graphs plots the number of tests passed against version number. These graphs illustrate the progress made towards the final goal over time.

The data indicates that many students had a large number, sometimes a very large number, of versions. This suggests a wide variation in the programming technique. It also illustrates that the students have commitment and energy which may not be correctly focused.

The strategy used by a student falls into one of three categories. One strategy used by students was to build up programs in stages: adding code to build part of the specification and focusing on this until completed before moving onto the next stage. A second strategy was to enter most of the code prior to the program passing any tests. Those students following this strategy did not or were not able to split the problem up but aimed at building towards a correct solution by correcting the whole program. A few of students exhibited no strategy and made little if any progress towards a fully correct solution. They were unable to correct their compilation errors showing a lack of understanding of the programming language.

4. How do experts program?

The same program was developed by a group of expert programmers (staff). The results were analysed using the same method and graphs drawn. Observation of the development profile for these programmers shows

- The number of versions of the program created by experts was less than students indicating an expertise in the development of programs.
- Experts would build up their program in stages. Each stage building on the previous stage, adding more code to satisfy more of the specification.
- Evidence of incremental development shows the ability to partition the problem into sub-problems.
- The number of versions that compile was high and those that do not compile had fewer errors and were corrected in fewer attempts. This shows expertise in the use of the language.

These observations suggest that experts follow an incremental policy for developing programs. This policy can be described using pseudo code as:

```
Understand the problem and identify a development plan
Repeat
    Add code for next stage in development
    Loop
        Loop
            Compile
            Exit if no compiler errors
            Correct compiler errors
        End Loop
        Run program
        Exit if code satisfies stage of development
        Update code to correct run errors
    End Loop
Until full specification complete
```

This is the way experts program and it should be the aim for our students. The next stage in this study is to find ways to identify deviation from this model and provide feedback to the students to help them get back on track.

5. Measurement of Process.

The authors wish to find measurements taken from the development process that give a mark of quality for the process undertaken.

- The time or effort put into the development cannot easily be measured. The *number of versions* taken to produce the final program will be used to measure the length of development.
- Proficiency at using the language and correcting errors is measured by the *compiling ratio*, the proportion of versions that compiled, and the *compiling progress indicator* which is the proportion of versions that had fewer compiler errors than previously.
- Proficiency at solving the problem is measured by the *run progress indicator* which is the proportion of versions that satisfy more of the specification than previously.
- A measurement of how much of the development made progress towards the goal can be given by calculating the *total progress indicator*. A version makes progress over a previous version

if there were fewer compilation errors or it satisfied more of the specification.

These metrics have been calculated for each subject in the study. Here, the metrics are validated by subjectively categorising the subjects in the study and examining how well the metric discriminates between the categories. Each subject is categorised as an expert (experienced programmer who exhibits decomposition skills), novice (learner programmer who is capable of producing a correct program) and beginner (learner who failed to write a correct program).

- The *number of versions* is capable of discriminating between experts and novices. Beginners sometimes gave up early and are confused with experts.
- The *compiling ratio* distinguishes novices from beginners. Experts can be confused with beginners as they tend to have fewer versions overall and fewer attempts at getting a stage of the development correct.
- The *compilation progress indication* fails to discriminate categories of subject.
- The *run progress indicator* discriminates between categories of subject except where novices have few versions that compiled.
- The *total progress indicator* discriminates between the three categories of subject.

This shows that the four metrics, *number of versions*, *compiling ratio* , *run progress indicator*, *total progress indicator*, can discriminate between various groups of programmer. These metrics provide the lecturer and student with evidence of how well the student develops programs and feedback can be tailored accordingly. At the outset of this paper the authors put forward the view that the way a student builds a program is an important aspect of programming skill. A combination of these metrics suitably scaled as in Rees(1984), is capable of gauging how well the student can build programs and hence used as a grade for assessment.

6. Summary

This paper has introduced the idea of assessing the program development process as well as the final program. This is in line with current thinking on quality. A number of metrics have been defined and calculated for a group of experts and students. From the experimental data there is evidence that these metrics do indeed measure the program development process and might be useful to compute objective gradings for individual students.

Further work that is currently being carried out includes:

- investigating the relationship between program development process and the final program produced,.
- exploiting this relationship between process and product by introducing teaching methods aimed at improving the students' program development process and thus their programs.
- using the metrics to provide feedback to the students during the development of their program.

Reference

Benford, S., Burke, E. and Foxley, E. 1993 Learning to Construct Quality Software with the Ceilidh System. *Software Quality Journal.* **2**, pp. 177-197.

Lund, G.R., 1994 The Programming Process. In: J.S. Hart and M. Smith, editors. *Innovations in the teaching of Computing.* Staff and Educational Development Agency (SEDA) Paper 88.

Rees, M.J. 19984. Automatic Assessment Aids for Pascal Programs. *ACM SIGPLAN Notices.* **17**(10) pp. 33-42.

Appendix D. – Programs and Scripts

Programs used in the data collection and analysis.

`PC.VMS`	VMS script to capture the versions of the programs.
`CntLn2.cpp`	C++ program to analyse a single program.
`PRIV.VMS`	VMS Script to control the analysis of a whole PSDP.
`Metric.cpp`	C++ program to calculate the potential process metrics for a single program development

```
$! To copy files for further analysis and to improve the River
$! Pascal Compile and Run system.
$! Written By G.R.L. March 1995
$! Amended for River G.R.L. Nov 1995.
$!
$!
$!
$! Check that a parameter exists
$ if p1 .nes. "" then goto continue1
$ write sys$output "Parameter missing"
$ exit
$!
$!
$!
$ continue1:
$! To get the important bit of the username in name
$ username = f$user()
$ x = F$LOCATE (",",username)
$ y = F$LOCATE("]",username)
$ len = y - x - 4
$ name = F$EXTRACT(x+4,len,username)
$!
$!
$!
$! Find the project name
$ x = f$locate(".",p1)
$ y = f$length(p1)
$ if x .eq. y
$    then
$      project = p1
$    else
$      project = f$extract(0,x,p1)
$ endif
$!
$!
$!
$! Check source file exists
$ write sys$output "Checking source file exists"
$ sourcename = project+".pas"
$ fullname = f$search(sourcename)
$ if fullname .nes. "" then goto continue2
$ write sys$output "Source does not exist"
$ exit
$!
$!
$!
$ continue2:
$! Copy source file across
$ x = f$locate("]",fullname)
$ y = f$locate(";",fullname)
$ genno = f$extract(y+1,50,fullname)
$ finalname = "ins:[c1.gl.safe]"+name+project+genno+".pas"
$ xfile = f$search(finalname)
$ if xfile .NES. "" then goto continue3
$ copy 'fullname' 'finalname'
```

```
$ set prot=(w:rd) 'finalname'
$!
$!
$!
$ continue3:
$!Compile source file
$ write sys$output "Compiling program"
$ set noon
$ define/user_mode sys$output qsvwdvefv.rgv
$ pascal 'sourcename' /list
$ set on
$ errorfile = f$search("qsvwdvefv.rgv")
$ if errorfile .nes."" then goto continue4
$ link 'project',ins:[c1.library]river,ins:[c1.library]weather,
ins:[c1.library]display
$ write sys$output "Running program"
$ define/user_mode sys$input sys$command
$ run 'project'
$!
$!
$!
$!Tidy up
$ xfile = project + ".obj;*"
$ delete 'xfile'
$ xfile = project + ".exe"
$ pu 'xfile'
$ xfile = project + ".lis"
$ pu 'xfile'
$ exit
$!
$!
$!
$!Compilation errors
$ continue4:
$ type qsvwdvefv.rgv /p
$ delete qsvwdvefv.rgv;*
$ exit
```

```
// FILE: CountLn.h
struct a_count {
        int seq_num ;
        int gen_num ;
        int comp_err ;
        int tests ;
      int line_count ;
      int comment_count ;
      int blank_count ;
        int code_line ;
      int procedure_count ;
      int function_count ;
      int while_count ;
      int repeat_count ;
      int until_count ;
      int for_count ;
      int if_count ;
      int case_count ;
      } ;

void Open_File (int argc, char* argv[],ifstream& ins, int& error) ;
void Close_File (ifstream& ins) ;
void Read_Line (ifstream& ins, char line[]) ;
void Output_Results( a_count count) ;
void Process_Line( char line[], a_count& count, int& status) ;
void Initialise_Count (a_count& ) ;
int Blank_Line(char line[]) ;
int Comment_Line(char line[],int& in_comment) ;
void Process_Code_Line(char line[],a_count& count,int& status) ;
int Next_Char(char line[],int ind);
void Get_Word(char line[],int i,char word[]) ;
void Update_Count(char word[],a_count& count) ;
void Upper_Case(char word1[], char word2[]) ;
int Generation_Number(char name[]) ;
int Sequence_Number(char name[]) ;

const int l_size = 256 ;
// const int FALSE = 0 ;
// const int TRUE = 1 ;
```

```
//  FILE: CntLn2.cpp
//  Counts the number of lines and reserver words in a PASCAL program
//  Written By GRL March 1996

#include <stdlib.h>
#include <fstream.h>
#include <iostream.h>
#include <iomanip.h>
#include <ctype.h>
#include <string.h>

#include "CNTLN2.H"



int main(int argc, char* argv[] )
//  Main program to count lines of Pascal code plus the reserved
words.
//  GRL March 1996
{
   char line[l_size] ;
   a_count count ;
   int status ;
   ifstream ins;
   int error ;

   Open_File(argc,argv,ins,error) ;
   if ( error != FALSE)
   {
      return error ;
   }
   Initialise_Count(count) ;
   status = FALSE ;
   Read_Line (ins, line) ;
   while (!ins.eof ())
   {
      Process_Line( line, count, status) ;
      Read_Line (ins, line);
   }
   count.seq_num = Sequence_Number(argv[2]) ;
   count.gen_num = Generation_Number(argv[1]) ;
   cerr <<"Enter error count > "<<endl ;
   cin >> count.comp_err ;
   if (count.comp_err == 0 )
   {
      cerr << "Enter number of tests passed > " <<endl ;
      count.tests = 3333 ;
      cin >> count.tests ;
   }
   else
   {
     count.tests = 0 ;
   }
   Output_Results(count) ;
   Close_File (ins) ;
   return 0;
}
```

```
void Open_File (int argc, char* argv[],ifstream& ins, int& error)
//  Procedure to open the specified file
//  GRL March 1996
//  Parameters
//  in argc - number of parameters in the command line
//  in argv - array of the command words
//  out ins - pointer to the openned input stream
//  out error - error indicator
{
  error = 0 ;
  if (argc != 3)
  {
    cerr << "Need to provide filename and sequence number on command
line " ;
    cerr << endl ;
    error = 1 ;
  }
  else
  {
    cerr << "File " << argv[1] << " being processed " << endl ;
    ins.open (argv[1]);
    if (ins.fail ())
    {
      cerr << "*** ERROR: Cannot open " << argv[1]
         << " for input." << endl;
      error = 2 ;;
    }
  }
  return ;
}


int Generation_Number (char name[] )
// to get the generation number from the file name
{
  int x ;
  char y[4] ;
  y[0] = name[5] ;
  y[1] = name[6] ;
  y[2] = name[7] ;
  y[3] = '\0';
  x = atoi(y) ;
  return x ;
}


int Sequence_Number(char name[])
// to get sequence number from argument
{
  int x = 0 ;

   x = atoi(name) ;
  return x ;
}



void Close_File(ifstream& ins)
//  Procedure to close the specified file
//  GRL March 1996
//  Parameters
//  in/out ins - pointer to the openned input stream
{
  ins.close();
```

```
  //cout << "file closed " << endl;
  return ;
}



void Output_Results( a_count count)
//  Procedure to output the counts calculated
//  GRL March 1996
//  Parameters
//  in count - structure of the counts
{
   int code_lines ;

   code_lines = count.line_count - count.comment_count-
count.blank_count ;
   cout << setw(5) << count.seq_num ;
   cout << setw(5) << count.gen_num ;
   cout << setw(5) << count.comp_err ;
   cout << setw(5) << count.tests ;
   cout << setw(5) << count.line_count;
   cout << setw(5)<< count.comment_count;
   cout << setw(5)<< count.blank_count ;
   cout << setw(5)<< code_lines ;
   cout << setw(5)<< count.procedure_count ;
   cout << setw(5)<< count.function_count ;
   cout << setw(5)<< count.while_count ;
   cout << setw(5)<< count.repeat_count ;
   cout << setw(5)<< count.until_count ;
   cout << setw(5)<< count.for_count ;
   cout << setw(5)<< count.if_count;
   cout << setw(5)<< count.case_count ;
   cout << endl ;
   return ;
}



void Read_Line (ifstream& ins, char line[])
//  Procedure to read a line of the input file and place line read
//  into array line terminated with /0
//  Written by G.R.L. December 94
//  Parameters
//  in/out ins - pointer to the openned input stream
//  out line - line input from file
{
   char next_ch;
   int n=-1 ;

   ins.get (next_ch);
   while ( (next_ch != '\n') && !ins.eof() && ( n < l_size) )
   {
      n++ ;
      line[n] = next_ch;
      ins.get (next_ch);
   }
   n++ ;
   line[n] = '\0' ;
   while ((next_ch != '\n') && !ins.eof())
   {
```

```
        ins.get (next_ch) ;
    }
}



void Process_Line( char line[], a_count& count, int& status)
//  Procedure to process the line input
//  GRL March 1996
//  Parameters
//  in line - line of text
//  in/out count - structure of all the counts
//  in/out in_comment - indicator to say whether in a comment or not
{
     count.line_count ++ ;
     if (Blank_Line(line) )
     {
       count.blank_count ++ ;
     }
     else
     {
       if (Comment_Line(line,status))
       {
           count.comment_count ++ ;
       }
       else
       {
           Process_Code_Line(line,count,status) ;
       }
     }
     return ;
}



void Initialise_Count ( a_count& count)
//  Procedure to initialise the struct count
//  Written by G.R.L. December 94
//  Parameter
//  out count - count values
{
  count.line_count = 0;
  count.comment_count = 0 ;
  count.blank_count = 0 ;
  count.procedure_count = 0 ;
  count.function_count = 0 ;
  count.while_count = 0 ;
  count.repeat_count = 0 ;
  count.until_count = 0 ;
  count.for_count = 0 ;
  count.if_count = 0 ;
  count.case_count = 0 ;
  return ;
}



int Blank_Line(char line[])
```

```
//  Function to indicate if a line is blank
//  GRL March 1996
//  Paremeters
//  in line - line under test
//  result - indicates if line is blank either TRUE or FALSE
{
    int i = 0 ;

    i = Next_Char(line,0) ;
    if (line[i] == '\0')
    {
      return TRUE ;
    }
    else
    {
      return FALSE ;
    }
}




int Next_Char(char line[],int ind)
//  Function to find the next non blank character on line starting at
ind
//  GRL March 1996
//  Paremeters
//  in line - line under test
//  in ind  - starting character
//  result - indicates if line is blank either TRUE or FALSE
{
    int i ;

    i = ind ;
    while(isspace(line[i]) && (line[i] != '\0'))
    {
      i++ ;
    }
    return i ;
}



int Comment_Line(char line[],int& in_comment)
//  Function to indicate if a line is a line with only a comment on
//  GRL March 1996
//  Paremeters
//  in line - line under test
//  in/out in_comment - status of comment indicator
//  result - indicates if line is a comment line either TRUE or FALSE
{
    int stat ;
    int code_found = FALSE ;
    int i ;

    stat = in_comment ;
    i = Next_Char(line,0) ;
    while ((line[i] != '\0') && (!code_found) )
    {
      if (stat == TRUE)
      {
          if( line[i]=='}')
```

```
            {
              stat = FALSE ;
            }
        }
        else
        {
            if (line[i] == '{')
            {
              stat = TRUE ;
            }
            else
            {
              code_found = TRUE ;
            }
        }
        i = Next_Char(line,i+1) ;
    }
    if (code_found == TRUE)
    {
      return FALSE ;
    }
    else
    {
      in_comment = stat ;
      return TRUE ;
    }
}



void Process_Code_Line(char line[],a_count& count,int& status)
//  Procedure to process a line of code
//  GRL March 1996
//  Paremeters
//  in line - line under test
//  in/out count - counts to be updated
//  in/out status - status to be updated
{
    int i  ;
    char word[l_size] ;

    i = Next_Char(line,0) ;
    while ( line[i] != '\0' )
    {
      if (status == TRUE)
      {
            if (line[i] == '}')
            {
                  status = FALSE ;
              }
      }
      else
      {
            if (line[i] == '{')
            {
                  status = TRUE ;
            }
            else
            {
              Get_Word(line,i,word) ;
              Update_Count(word,count) ;
                  // i points at last character considered
                  i = i + strlen(word) -1 ;
```

```
            }
        }
        i = Next_Char(line,i+1) ;
    }
    return ;
}



void Get_Word(char line[],int i,char word[])
//  Procedure to get the next word in line starting at i
//  GRL March 1996
//  Parameters
//  in line - line from whcih to get word
//  in i - start position in line
//  out word - fist word in line starting at i
{
    int ind1 = i ;
    int ind2 = 0 ;
    if (isalpha(line[ind1]) )
    {
      while ( isalpha(line[ind1]))
      {
      word[ind2] = line[ind1] ;
      ind2 ++ ;
      ind1 ++ ;
      }
    }
    else
    {
      while ( !(isspace(line[ind1])) &&
            !(line[i] == '\0') && !(isalpha(line[ind1])))
      {
      word[ind2] = line[ind1] ;
      ind2 ++ ;
        ind1 ++ ;
      }
    }
    word[ind2] = '\0' ;
    return ;
}



void Update_Count(char word[],a_count& count)
//  Procedure to update the counts fusing word
//  GRL March 1996
//  Parameters
//  in word - word from which counts are updated
//  in/out count - counts that are updated
{
    char new_word[l_size] ;

    Upper_Case(word,new_word) ;
    if (strcmp(new_word,"PROCEDURE") == 0)
    {
      count.procedure_count ++ ;
    }
    if (strcmp(new_word,"FUNCTION") == 0)
    {
```

```
      count.function_count ++ ;
    }
    if (strcmp(new_word,"WHILE") == 0)
    {
      count.while_count ++ ;
    }
    if (strcmp(new_word,"REPEAT") == 0)
    {
      count.repeat_count ++ ;
    }
    if (strcmp(new_word,"UNTIL")==0)
    {
      count.until_count ++ ;
    }
    if (strcmp(new_word,"FOR")==0)
    {
      count.for_count ++ ;
    }
    if (strcmp(new_word,"IF")==0)
    {
      count.if_count ++ ;
    }
    if (strcmp(new_word,"CASE") == 0)
    {
      count.case_count ++ ;
    }
    return ;
}



void Upper_Case(char word1[], char word2[])
//  Procedure to convert a word to uppercase
//  GRL March 1996
//  Parameters
//  in word1 - input word
//  out word2 - as word1 but any letters made upper case
{
    int ind1 = 0 ;
    int ind2 = 0 ;

    while (word1[ind1] != '\0' )
    {
      word2[ind2] = toupper(word1[ind1]) ;
      ind1 ++ ;
      ind2 ++ ;
    }
    word2[ind2] = '\0' ;
    return ;
}
```

```
$! To analyse all river files in a directory
$! Creates a .new file from all the .pas files
$! Written By G.R.L.
$!
$!
$!  Define names
$   cnn_exe := $mct:[gl.programs]cntln2.exe
$   outfile = "RIVERDAT.NEW"
$   create 'outfile'
$   seq = 1
$   name= "mct:[gl.safe96."+p1+"]river*.pas"
$   write sys$output "Name "+name
$
$
$ loop_top:
$   sourcefile = f$search(name,1)
$   write sys$output sourcefile
$   if sourcefile .eqs. "" then goto loop_end
$!  analyse the program
$   write sys$output sourcefile
$   write sys$output seq
$   x = f$locate("]",sourcefile)
$   shortname = f$extract(x+1,12,sourcefile)
$   set noon
$   define /user_mode sys$output x.x
$   define /user_mode sys$input sys$command
$   cnn_exe 'shortname' 'seq'
$   set on
$   append x.x 'outfile
$   delete x.x.
$!
$!  prepare for next file
$   seq = seq + 1
$   goto loop_top
$ loop_end:
$ exit
```

```
// FILE: metric.cpp
//  Program to calculate metrics from .new files.
//  Written By G.R.L. December 1997.


#include <iostream.h>
#include <fstream.h>

// Constants
//const int FALSE = 0 ;
//const int TRUE = 1 ;
const int max_test = 16 ;


// Type definitions
typedef struct a_step {
  int step ;
  int genno ;
  int n_comp_err ;
  int n_test ;
  int loc ;
  int n_com_l ;
  int n_blk_l ;
  int n_cod_l ;
  int n_proc ;
  int n_func ;
  int n_while ;
  int n_rpt ;
  int n_until ;
  int n_for ;
  int n_if ;
  int n_case ; } ;

typedef a_step a_process[300] ;

// function prototypes
void read_file(char * n , a_process &p , int &m, int &error) ;
void output_comp_metric(float cr, float cpi, float asc ) ;
void output_stage_metric(int slc, int spf, int srn) ;
void output_sn(int s1, int s2, int s3) ;
void output_progress_metric(float rpi, float tpi) ;
void output_sp(float sp) ;
float calc_cr(a_process p,int n);
float calc_cpi(a_process p , int n ) ;
float calc_cas(a_process p, int n) ;
int calc_slc(a_process p, int n) ;
int calc_spf(a_process p, int n) ;
int calc_srn(a_process p, int n) ;
int calc_s1(a_process p, int n ) ;
int calc_s2(a_process p, int n ) ;
int calc_s3(a_process p, int n ) ;
float calc_rpi(a_process p, int n) ;
```

```
float calc_tpi(a_process p, int n) ;
float calc_sp(a_process p, int n) ;
int maximum ( int a , int b ) ;
int minimum ( int a , int b) ;

int main(int argc, char * argv[] )
{
  a_process process ;
  int n_step ;
  int error ;
  float cr ;
  float cpi ;
  float asc ;
  int slc ;
  int spf ;
  int srn ;
  int s1 ;
  int s2 ;
  int s3 ;
  float rpi ;
  float tpi ;
  float sp ;

  read_file( argv[1], process, n_step, error) ;
  cout << n_step << endl ;
  if ( error == FALSE )
  {
    cr = calc_cr(process,n_step);
    cpi = calc_cpi(process,n_step) ;
    asc = calc_cas(process,n_step) ;
    output_comp_metric(cr,cpi,asc ) ;
    slc = calc_slc(process,n_step) ;
    spf = calc_spf(process,n_step) ;
    srn = calc_srn(process,n_step) ;
    output_stage_metric(slc,spf,srn) ;
    s1 = calc_s1(process, n_step ) ;
    s2 = calc_s2(process, n_step ) ;
    s3 = calc_s3(process, n_step) ;
    output_sn(s1, s2, s3) ;
    rpi = calc_rpi(process, n_step) ;
    tpi = calc_tpi(process, n_step) ;
    output_progress_metric(rpi,tpi) ;
    sp = calc_sp(process, n_step) ;
    output_sp(sp) ;
  }
  return 0 ;
}



// read the .new file
void read_file(char * n , a_process &p , int &m, int &error)
```

```
{
  char line[80] ;
  ifstream ins ;

  cout << "File " << n << " being processes" << endl ;
  ins.open(n) ;
  if (ins.fail())
  {
        cout << " *** ERROR Cannot open file ***" << n << endl
 ;
        error = TRUE ;
  }
  else
  {
        m = 0 ;
        while (!ins.eof())
        {
              ins >> p[m].step >> p[m].genno >> p[m].n_comp_e
rr ;
              ins >> p[m].n_test >> p[m].loc >> p[m].n_com_l
;
              ins >> p[m].n_blk_l >> p[m].n_cod_l >> p[m].n_p
roc ;
              ins >> p[m].n_func >> p[m].n_while >> p[m].n_rp
t ;
              ins >> p[m].n_until >> p[m].n_for >> p[m].n_if
;
              ins >> p[m].n_case ;
              ins.getline(line,80) ;
              m++ ;
        }
        m -- ;
        error = FALSE ;
  }
}


void output_comp_metric(float cr, float cpi, float asc )
{
  cout << "Compiling ratio          " << cr << endl;
  cout << "CPI                      " << cpi << endl ;
  cout << "Average steps to compile " << asc << endl ;
}


// calculate compilation ratio, proportion of steps that compil
e
float calc_cr(a_process p,int n)
{
  int c = 0 ;
  float cr ;
  for (int i = 0 ; i < n ; i++)
```

```
  {
        if (p[i].n_comp_err == 0 )
        {
              c ++  ;
        }
  }
  cr = float (c) / float (n) ;
  return cr ;
}


//  calculate compilation performance index
float calc_cpi(a_process p , int n )
{
  int n_c ;
  int step_p = 0 ;
  int step_o = 0 ;
  float cpi ;

  n_c = minimum(30, p[0].n_comp_err) ;
  for (int i = 1 ; i < n ; i++)
  {
    if (n_c > 0 )
        {
              step_o ++ ;
              if ( n_c > minimum( 30 , p[i].n_comp_err))
              {
                step_p ++ ;
              }
    }
        n_c = minimum(30,p[i].n_comp_err) ;
  }
  if (step_o > 0)
  {
        cpi = float(step_p) / float(step_o) ;
  }
  else
  {
        cpi = 0.0 ;
  }
  return cpi ;
}



//  calculate average number of steps to successful compile
float calc_cas(a_process p , int n)
{
  float cas ;
  int n_comp_stage = 0 ;
  int n_step = 0 ;
```

```
  int n_c =0 ;

  for (int i = 0; i <= n ; i++)
  {
    if (p[i].n_comp_err == 0)
    {
      if (n_c == 0)
      {
         // do nothing
      }
      else
      {
        n_step ++ ;
        n_comp_stage ++ ;
      }
    }
    else
    {
      n_step ++ ;
    }
    n_c = p[i].n_comp_err ;
  }
  if (n_c > 0)
  {
    n_comp_stage ++ ;
  }
  cas = float(n_step) / float (n_comp_stage) ;
  return cas ;
}


//  Function to calculate the stages (lines of code) metric
int calc_slc(a_process p, int n)
{
  int slc = 1 ;
  int n_loc ;

  n_loc = p[0].loc ;
  for (int i = 1 ; i < n ; i++)
  {
        if (( p[i].loc - n_loc ) >= 12 )
        {
              slc ++ ;
        }
        n_loc = p[i].loc ;
  }
  return slc ;
}


//  Function to calculate the stages metric s1
int calc_s1(a_process p, int n)
```

```
{
  int i = 0 ;
  int s1 = 1 ;
  int n_loc ;

  while (( p[i].n_test <= 2 ) && (i < n))
  {
    i++ ;
  }
  if ( i == n )
  {
    return s1 ;
  }
  else
  {
    n_loc = p[i].loc ;
    for (int j = i ; j < n ; j++)
    {
      if (( p[j].loc - n_loc ) >= 10 )
      {
        s1 ++ ;
      }
      n_loc = p[j].loc ;
    }
    return s1 ;
  }
}


//  Function to calculate the stages (procedure/function) metri
c
int calc_spf(a_process p , int n)
{
  int spf ;
  int n_pf ;

  if ((p[0].n_proc + p[0].n_func) == 0 )
  {
    spf = 0 ;
  }
  else
  {
    spf = 1 ;
  }
  n_pf = p[0].n_proc + p[0].n_func ;
  for (int i = 1 ; i < n ; i++ )
  {
    if ((p[i].n_proc + p[i].n_func) > n_pf)
    {
      spf ++ ;
    }
    n_pf = p[i].n_proc + p[i].n_func ;
```

```
  }
  return spf ;
}


//  Function to calculate the stages metric s2 - procedures/fun
ctions
int calc_s2(a_process p, int n)
{
  int i = 0 ;
  int s2 = 0  ;
  int n_pf ;

  while (( p[i].n_test <= 2 ) && (i < n))
  {
    i++ ;
  }
  if ( i == n )
  {
    return s2 ;
  }
  else
  {
    if ((p[i].n_proc + p[i].n_func) == 0 )
      s2 = 0 ;
    else
      s2 = 1 ;
    n_pf = p[i].n_proc + p[i].n_func ;
    for (int j = i ; j < n ; j++)
    {
      if (( p[j].n_proc + p[j].n_func ) > n_pf )
      {
        s2 ++ ;
      }
    n_pf = p[j].n_proc + p[j].n_func ;
    }
    return s2 ;
  }
}


//  Function to calculate the stages (run) metric
int calc_srn(a_process p , int n)
{
  int srn = 0;
  int n_rn = 0;
  for (int i = 1 ; i < n ; i++)
  {
    if ( p[i].n_comp_err == 0 )
    {
      if ( p[i].n_test > n_rn )
      {
```

```
        srn++ ;
        n_rn = p[i].n_test ;
      }
    }
  }
  return srn ;
}


//  Function to calculate the stages metric s3 - run - tests
int calc_s3(a_process p, int n)
{
  int i = 0 ;
  int s3 = 0 ;
  int n_rn ;

  while (( p[i].n_test <= 2 ) && (i < n))
  {
    i++ ;
  }
  if ( i == n )
  {
    return s3 ;
  }
  else
  {
    s3 = 1 ;
    n_rn = p[i].n_test ;
    for (int j = i ; j < n ; j++)
    {
      if ( p[j].n_test > n_rn )
      {
        s3 ++ ;
        n_rn = p[j].n_test ;
      }
    }
    return s3 ;
  }
}


//  Function to output the stage metrics
void output_stage_metric(int slc, int spf, int srn )
{
  cout << "Stage - lines of code       " << slc << endl ;
  cout << "Stage - procedure/function " << spf << endl ;
  cout << "Stage - run progress        " << srn << endl ;
}


//  Function to output s metrics
void output_sn(int s1, int s2, int s3)
```

```
{
  cout << "Stage - S1                    " << s1 << endl ;
  cout << "Stage - S2                    " << s2 << endl ;
  cout << "Stage - S3                    " << s3 << endl ;
}


//  Function to calculate the run progress indicator
float calc_rpi(a_process p , int n)
{
  int n_run = 0 ;
  int n_prg = 0 ;
  int n_ttt = 0 ;
  float rpi ;

  for (int i = 0 ; i < n ; i++)
  {
    if (p[i].n_comp_err == 0)
    {
      n_run++ ;
      if ( p[i].n_test > n_ttt )
      {
        n_prg ++ ;
        n_ttt = p[i].n_test ;
      }
    }
  }
  if (n_run == 0 )
  {
    rpi = 0 ;
  }
  else
  {
    rpi = float(n_prg) / float(n_run) ;
  }
  return rpi ;
}


//  Function to calculate the total progress indicator
float calc_tpi(a_process p , int n)
{
  int n_ttt = 0 ;
  int n_ccc = 0 ;
  int n_tpi = 0 ;
  float tpi ;

  for (int i = 0 ; i < n ; i++)
  {
    if (p[i].n_comp_err < n_ccc)
    {
```

```
      n_tpi ++ ;
    }
    if (p[i].n_test > n_ttt)
    {
      n_tpi ++ ;
      n_ttt = p[i].n_test ;
    }
    n_ccc = p[i].n_comp_err ;
  }
  tpi = float(n_tpi) / float(n) ;
  return tpi ;
}


//  FUnction to output the progress metrics
void output_progress_metric(float rpi, float tpi)
{
  cout << "Run Progress Indicator   " << rpi << endl ;
  cout << "Total Progress Indicator " << tpi << endl ;
}


//  Function to calculate the specification proportion
float calc_sp(a_process p, int n)
{
  int n_t = 0 ;
  float sp ;

  for ( int i = 0 ; i < n ; i++)
  {
    if (n_t < p[i].n_test)
    {
      n_t = p[i].n_test ;
    }
  }
  sp = n_t / float(max_test) ;
  return sp ;
}


//  Function to output specificatio proportion
void output_sp ( float sp)
{
  cout << "Specification Proportion   > " << sp << endl ;
}


//  Function to find the maximum of 2 integers
int maximum(int a , int b )
{
  if (a >b)
    return a;
```

```
  else
   return b ;
}


//  Function to find the mimimum of 2 integers
int minimum ( int a , int b)
{
  if (a < b)
    return a ;
  else
    return b ;
}
```