UNIVERSITY OF SOUTHAMPTON

FACULTY OF LAW, ARTS AND SOCIAL SCIENCES

SCHOOL OF EDUCATION

An Extended Case Study on the Introductory Teaching of Programming

By

Michael William Jones

Thesis for the degree of Doctor of Education

September 2010

Contents

University of Southampton

Abstract

Faculty of Law, Arts and Social Sciences

School of Education

Doctor of Education

An Extended Case Study on the Introductory Teaching of Programming

By Michael William Jones

Learning to program is a complex and arduous process undertaken by thousands of undergraduates in the UK each year. This study examined the progress of transforming the pedagogical paradigm of an introductory programming unit from a highly controlled, reductionist 'cipher' orientation to one in which students have more freedom to explore aspects of programming more creatively. To facilitate this, certain programming concepts were introduced much earlier that had previously been the case. This was supported by an analysis of the semiotics and symbology of programming languages that showed that there was no intrinsic support for the traditional sequence of introducing programming concepts. A second dimension to the transformation involved doubling the number of assessments to emphasise the benefits of continual engagement with programming. The pedagogical transformation was to have been phased over four successive cohorts, although the fourth phase had to be delayed due to a revalidation that amalgamated three programmes into a framework.

The study was planned during the second phase of the transformation. To ensure that the study did not disrupt the students' learning experience the main focus of the

research was on quantitative analyses of the work submitted by the students as part of the coursework for the unit. This work included programming portfolios and tests. In all, the work of more than 400 students completing more than a thousand portfolios and a thousand tests were analysed, providing a holistic view of waypoints in the learning process.

The analyses showed that the second and third cohorts responded positively to the greater level of freedom, creating more sophisticated applications utilising a wider range of programming constructs. In the latter part of the fourth cohort a more traditional, constrained approach was used by another tutor that resulted in a narrowing of the range of programming concepts developed.

The quantitative instruments were augmented by questionnaires used to gauge the students' previous experience, and initial views. Analyses of these returns showed that there appeared to be a limited relationship between a student's previous experience and the likelihood that he or she would succeed in the unit and be eligible to continue to the next stage of the undergraduate programme.

The original plan was for qualitative instruments to be introduced in the final two cohorts. The re-organisation alluded to earlier restricted qualitative methods to short, semi-structured interviews during the third cohort.

Within the study, certain aspects of the pedagogical transformation were considered in more depth: the development and use of a code generator and criterion-referenced assessment. These innovations were part of another dimension of the transformation of the unit, emphasising comprehension and modification equally with construction. This dimension reflects the changing nature of programming, incorporating existing code wherever possible. The analyses showed that comprehension skills developed to a greater extent within the unit compared with modification and construction.

The main conclusions of the study were that the pedagogical changes had a beneficial effect on the learning of all students, including those with considerable previous experience, and those who had never written a program before.

Contents

Figures

Tables

Declaration of Authorship

I, **Michael William Jones** declare that the thesis entitled "**An Extended Case Study on the Introductory Teaching of Programming**" and the work presented in the thesis are both my own, and have been generated by me as the result of my own original research. I confirm that:

- this work was done wholly or mainly in candidature for a research degree at this University;
- where any part of this thesis has previously been submitted for a degree or other qualification at this University or another institution, this has been clearly stated;
- where I have consulted the published work of others, this is always clearly attributed;
- where I have quoted from the work of others, the source is always given;
- with the exception of such quotations, this thesis is entirely my own work;
- I have acknowledged all the main sources of help;
- where the thesis is based on the work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.;
- none of this work has been published before submission.

Signed: *Michael William Jones*

Date: 18th March 2010

Acknowledgements

First and foremost I want to thank my lovely wife, Paula, for her unstinting support and encouragement, and for her technical knowledge. Without her, this work could not have been completed. Secondly, I thank my supervisor, Ian Bryant, who gave me sound advice and guidance at every stage. This was very useful throughout the research, particularly at the outset and in the latter stages. The other tutors on the Doctor of Education programme in the School of Education provided valuable learning opportunities – I am only sorry that I was unable to explore these in greater detail. Next, many thanks are due to the Bournemouth University students for the effort they put in during their first year Programming unit. I also owe a debt of gratitude to Sally Fincher, of the University of Kent, who selected me to participate in the Disciplinary Commons in Introductory Teaching of Programming. My fellow 'Commoners' helped to make this an enjoyable and stimulating experience, and they provided feedback when this was requested. Various colleagues at Bournemouth provided technical assistance and information at crucial points. Finally, I thank my employers, Bournemouth University, for providing the funds.

Glossary

BTEC — Business and Technology Education Council. A UK education body more generally associated with vocational qualifications.

Cohort — Single intake of undergraduate students. A cohort may include students who are repeating due to previous failure.

CWFDP — Cumulative Weighted Frequency Difference as a Percentage. This is a measure derived for use in this research for comparing two populations. The frequency of the intervals in each are calculated as percentages. The difference between the corresponding interval in each population is then calculated and weighted according to the value of the interval. These weighted figures are then accumulated. The measure highlights the nature and location of shifts between the two populations, such as between the results obtained on two successive assignments by the same cohort of students.

DBMS — DataBase Management System. Software that provides services and controls the access to and integrity of a database. In modern computing, this is synonymous with RDBMS (Relational DataBase Management System).

HEA — Higher Education Academy. Formerly known as the Learning and Teaching Support Network (LTSN).

HTML — HyperText Markup Language. Document markup language used widely in the World Wide Web (WWW). Specified by Tim Berners-Lee in 1989.

ICS — Information and Computer Sciences – a group within the Higher Education Academy (HEA)

ICT — Information and Communications Technology. Wider definition of IT.

IT — Information Technology. A more general term used for ICT.

ITP	Introductory (or Initial) Teaching of Programming. In the context of this research the term is used in connection with first year undergraduates.
Java	Programming language widely used in higher education. Developed by Sun Microsystems.
JCQ	The Joint Council for (General) Qualifications. The UK body responsible for coordinating pre-University educational awards.
K-12	Designation used in the United States and Australia for pre-University education (first to twelfth grades).
LMS	Learning Management System. A repository for learning materials. A VLE is an LMS augmented with assessment components and mechanisms.
Lickert	Psychometric scale often used in questionnaires. Developed in the 1930s by Rensis Lickert.
LTSN	Learning and Teaching Support Network. Renamed the Higher Education Academy.
OO	Object-orientation. A software paradigm where the attributes and operations associated with a noun are encapsulated in a class. Each object is an instance of a class.
PAL	Peer Assisted Learning. A system in which more senior students (PAL Leaders) teach more junior students in formal lectures or seminars. These sessions are additional – they do not replace other sessions. The tutors provide the materials are provided by the tutors, and also provide guidance and tutoring to the PAL leaders.
Python	Programming language. Name (allegedly) inspired by Monty Python's Flying Circus.

URL	Uniform Resource Locator. Used to identify a resource (document) on the World Wide Web.
VLE	Virtual Learning Environment. Term related to Learning Management System.

Appendices

1. The Introductory Programming Landscape

1.1. Introduction: software in the community and economy

The widespread adoption of 'e' business practices and technologies throughout the world has contributed to economic growth around the planet. Computing and IT is estimated to be the second largest sector in the UK economy (after banking), and all estimates indicate that demand for computing and IT skills will continue to outstrip supply. A report conducted by Microsoft UK and conducted by Lancaster University Management School and the British Computer Society (Lancaster University Management School, 2006) painted a stark picture of the problems facing the UK software industry. An average of 20,000 computing and IT (Information Technology) graduates emerge from UK universities with a combination of foundation, undergraduate, and postgraduate degrees each year. This figure is insufficient to satisfy the estimated 141,000 computing and IT jobs likely to be required in each of the next five years. This picture was corroborated by one of the e-skills quarterly reports for 2005 (e-skills, 2005). This skills gap is currently being filled by a combination of graduates moving into IT from other disciplines and by recruitment from overseas. Since the global recession, demand for IT skills has dipped by around a third (Office of National Statistics, 2010), but there is still a skills shortage in this area. It is also the case that thousands of computing professionals retire each year. The UK Government recently announced the establishment of an IT Academy (due to open in 2010) as part of a wider drive to encourage more UK citizens into this industry.

Social networking using mobile phones and websites such as MySpace, FaceBook, LinkedIn and SecondLife is growing so rapidly that it is estimated that more than 70% of all UK citizens visit social networking sites for 4 or more hours per month (comScore, 2009). Partly, this is facilitated by cost: purchasing and using a computer is now within the reach of almost all UK citizens. As technology has advanced, so networking costs have declined. Many governments around the world are investing heavily in IT, keen to avoid being on the wrong side of the 'digital divide'.

The importance of IT has long been recognised within the UK education system. IT

has been a feature of the National Curriculum for many years, with around 24,000 students annually being awarded grade C or above in GCSE ICT (Information and Communications Technology) (JCQ 2009a). At A level, both Computing and ICT are available, with the numbers of students being 4,710 and 11,948 respectively in 2009 (JCQ 2009b). Males account for approximately 90% of the Computing figures and around two thirds of those for ICT.

One might imagine that the clear growth in the use of computing and IT within almost all industries and forms of entertainment, and the availability of well-paid employment would result in an increased demand for computing and IT education. In fact, the reverse appears, at least superficially, to be the case. There has been a steady decline in the numbers of students in these areas over the past few years. For instance, the figures for Computing and ICT A level students in 2006 for the UK were 5,629 and 14,208 respectively (JCQ 2006), some 15% higher than the corresponding figures in 2009. Similarly, demand for Computer Science at undergraduate level has declined in recent years both in Europe and North America, and prompted a number of initiatives, for example the K-12 programmes in the United States (Penuel, 2006). Many of these are aimed at encouraging more young people to become engaged in computing, rather than remain content to remain within the IT environment. The difference between computing and IT can be envisaged in terms of the nature of participation: computing professionals are engaged in the construction of the software artefacts (designs and programs), whereas IT professionals tend to select, configure and exploit existing artefacts.

On closer inspection, however, the nature of the decline in computing in higher education is rather more complex. A number of 'creative' disciplines in computer games and animation and digital media development have seen significant growth in recent years with many universities now offering undergraduate programmes in these disciplines.

The tasks that each piece of software performs are linked to a context. That context might be relevant to many users, but some will want to modify or extend the capability of the software for specific purposes. Most software is therefore configurable. Simple configuration involves selecting components to include or

exclude, or set certain values. When a configuration file includes logic (decision making and repetition) it becomes a program. Programming therefore lies at the heart of a software system, being used both to construct the system, and facilitate complicated configuration. An example is a web browser: website developers embed programs or program segments within HTML (HyperText Markup Language) documents, with the browser rendering the HTML, and executing the programs.

1.2. Barriers to Programming

The barriers to participation counterbalance the large number of career opportunities offered by the software industry. Software is highly contextualised in terms of technologies – heavily influenced by the operating system, the hardware, the programming language, and the database management system (DBMS). Modern software systems may also incorporate more than one programming language (as in the case of a website), and each of the components will have multi-faceted relationships with the other components. Moreover, whilst software is undoubtedly creative, the resulting artefact has to fulfil an exact set of requirements. The low cost of computing equipment means that there is the no significant technology cost barrier to adding more capability. By contrast, writing software is very labour intensive: if existing software can be extended or re-configured (instead of commissioning new software) much of these costs can be mitigated. One consequence is that the comprehension of software (to establish its suitability) has become an essential skill for a software developer. It has been recognised, however, that understanding software is a highly complex cognitive task (Misra and Akman, 2008).

A programming language is essentially an algebra and, as such, was initially designed for a specific purpose. As with many other algebras, extensive use is often made of symbols, producing a visual appearance for a program unlike that of a snippet of any natural language. Most programming languages began as personal or limited projects that have then been adopted and extended by a wider community. The design decisions inherent in the limited scope of the original purpose of the language may not be observed in any continued development, resulting in idiosyncrasies that the programmer must accommodate. The nature of these

‘features’ is generally such that programmers cannot reason effectively – they just need to know the specific details. This necessary accumulation of minutiae requires significant continuous investment of time, and tends to attract certain personality types, and repel others. It is not, perhaps, a surprise that software development is seen (Grandin & Duffy, 2008) as one potential area of employment for those suffering from Asperger’s Syndrome, a particular and relatively mild form of autism that often manifests itself in obsessive attention to detail.

The grammar of a programming language is one layer of the complexity of learning to program. Each language also has an associated set of 'norms', which programmers using that language are expected to observe, in addition to those dictated by the syntax of the language. These norms are intended to convey meaning above and beyond that which is strictly necessary in terms of the algebra. For instance, it is conventional in the Java programming language to begin variable names with a lowercase letter; in C# it is conventional to use uppercase letters to start variable names. In both cases, the choice is not arbitrary, as it is linked to a wider set of styling practices that aim to shorten the time taken by a professional to comprehend the code. The 'comprehension time' is critical in modern software development, as incorporating as much existing software as possible is economically prudent.

Another layer of complexity is represented by the ubiquitous nature of software: every component of a computer system contains some software, and a computing professional may need to have some understanding of this. It could be that the time delay involved in processing data is problematic in specific situations; likewise there may be an impact on the choice of hardware device or networking capability. Many fonts include snippets of program code to handle kerning as text is resized. That all components of a computer system can be programmed is therefore both a blessing and a curse. Software offers almost total flexibility, at the cost of complexity in every dimension.

1.3. The Pedagogical Landscape

A number of reasons are put forward to explain the inconsistent nature of computing and ICT education prior to university (Reynolds *et al.*, 2003). The demand for IT

skills in industry means that few teachers have experience or relevant qualifications. A typical software developer with five years experience could expect to earn considerably more than a teacher with comparable length of experience, notwithstanding the fact that the teacher may have to have spent longer in the education system.

In a world of school league tables and considerable autonomy in curriculum implementation within schools in the UK, the computing and ICT syllabus is subjected to a number of interpretations. Where a student is required to write a program to fulfil coursework requirements, the teacher will often have considerable flexibility in the choice of examination board, programming language, and in the complexity of the algorithm to be constructed. Coursework is generally marked by the school, and moderated externally. The combined effect is that university undergraduate courses cannot assume a given level of understanding of programming (or other elements of computing) even where students have completed apparently related preparatory courses.

Another salient issue for the design of the programme for an introductory programming course is that the desired endpoint is rarely clear. The complexity of software development precludes any likelihood that emerging graduates will be competent software developers, unless they have considerable prior experience, or they are prepared to invest considerable time in addition to that required as part of the study programme. What, then, is an acceptable endpoint, and how should that be reflected in the goals for the initial teaching of programming? This leads to an exploration of the depth versus breadth debate, and to an examination of what constitutes depth.

When people learn to drive, the general assumption is that they will become non-professional drivers – if a person wishes to become a racing driver or a taxi driver, then it is generally accepted that additional study and tutoring will be required. As a consequence, ‘non-professional’ pupils are given basic skills in controlling the car, observing the law, and being courteous to other drivers. There is little coverage of brake horse-power, or the technology of the camshaft. In the context of the IT industry, any individual may need to develop or configure software, or interact with

software developers, so it seems reasonable to insist (as do all computing and IT undergraduate programmes in the UK) that all computing and ICT students study one or more units or modules in programming.

The issue remains: what are the fundamentals of programming? Should the theory of programming language grammar be explored, to facilitate an easier transition between programming languages, given that most professional software developers need to be proficient in more than one language? On the other hand, if the relationships between the programming language and the database management system, the operating system, the graphical user interface or the network are not explored, there is the possibility that this will limit the acquisition of transferable skills in these (more practical) dimensions.

The Association of Computing Machinery (ACM), the main professional body for computing in the United States, has set out an example curriculum for undergraduate programmes in computer science (ACM, 2008). The programming elements are linked not only to the programming language grammars, but are also heavily influenced by mathematics. Mathematics is the root discipline of computer science: all computers are realisations of a Universal Turing Machine (Turing, 1936), which developed out of research into computability theory. Each programming language has a deterministic grammar, which precisely spells out not only what is acceptable, but also the consequences of the execution of a given instruction. On closer inspection, the tacit assumption that computing is a specialisation of mathematics is not so clear. Mathematics is essentially a modelling technology, which has been adapted and developed to model many aspects of the world, from physics to economics. Attempts have been made to model the complexity of modern software systems: many companies included a software metrics element in their software development operations throughout most of the 1990's. Software metrics remains an active research area, but most companies are more likely to use metrics derived from project management methodologies when estimating the time and costs involved in developing a software system.

The power and affordability of modern computers have allowed software to penetrate many more facets of the operations of most businesses. This has taken

place in a very short timespan. The rapid change in the nature and scope of software is one factor that confounds mathematics: the other is the necessity to include (or not to exclude) the other components of the software system. If I wish to study gravity by dropping something from a window, it matters little which window, what is dropped, by whom, and on what day or time of day. The model can ignore many aspects of the context of this experiment without compromising the results. With software, one does not have that luxury: how long an application will take to execute, or even whether it will fail or not, may be heavily dependent on the hardware, the operating system, and what else is executing concurrently. For the vast majority of software systems, mathematical modelling is not feasible, and it can be seen that to suggest that it might be (through the design of the curriculum) might be seen as counter-productive.

The second argument for including mathematics in introductory programming is that of transferable skills: constructing a mathematical argument consists of selecting and sequencing mathematical components, relevant to the algebra in question. This is analogous to writing software, but this analogy is not limited to mathematics. Most disciplines involve similar intellectual and cognitive activities, but all disciplines can be approached in a variety of ways. Despite many years of research, no reliable link between programming and any other discipline has been found. For instance, Dehnadi (2006) proposed a simple test that he claimed indicated whether someone would be likely or not to be able to program. Caspersen *et al.* (2007) failed to replicate Dehnadi's findings.

The conclusion is that neither the tutor nor the student can reliably predict how difficult the student will find programming. It is therefore not surprising that a wide range of student motivation and capability in programming has been observed many times (e.g., Bergin and Reilly, 2005; Jenkins, 2001). This disparity need not disappear over time. In industry, it is not unknown for software developers with similar educational and experiential profiles to differ in productivity by one or more orders of magnitude. The more productive developers seem to make better choices and are more consistent, therefore making many fewer mistakes. The more capable appear to have more viable mental models (Ma *et al.*, 2007*).* Any differences in programmer capability can be magnified, as testing (and the consequential correcting

of errors) occupies a high proportion of a developer's time. Many undergraduate computing and IT cohorts exhibit a similar wide range of student application and ability. This complicates the notions of achievement thresholds, and the design of activities and support. It is difficult to avoid being more favourable either to the stronger and most committed students, or to those more in danger of not progressing.

1.4. The Research Landscape

The scope of the research was the first year students on a programming unit reading computing and IT honours programmes within a single UK university. The students were (and are) mainly UK citizens who have just completed A level or BTEC Higher National programmes in Computing or ICT. The cohorts are predominantly male. Widening the scope of the study to include postgraduate and foundation degree students was rejected at an early stage. The postgraduate students at the higher education institution involved generally have significant prior experience. The foundation degree students are based at various external locations, and the nature of the programmes at each location is not directly comparable.

The focus for the study was the students' perception and progress of four successive cohorts through a single 20 credit first year unit – Programming. This unit runs throughout the year, and is one of six equally weighted units that comprise the first year. The format of delivery is a weekly one-hour lecture and guided, supervised workshops. Additional support sessions are also available.

The detail of the research approach was investigated and formulated based on primary and secondary research, although certain boundaries were identified. As the unit in question contributes directly to students' progression through an undergraduate programme, controlled studies were not considered ethically acceptable, neither were they feasible, given that the study necessarily had a significant time dimension.

The underlying theme of pedagogic research is to improve the student experience. In that sense, there is a significant element that aims to make positive statements about which aspects of the approach are more successful, and which are not. Given that the

nature of learning is highly complex, it is also likely that any theory that emerges will need to be grounded in analytical data. The personal aspect of learning will also dictate that an element of the data gathering may well be qualitative, alongside more quantitative measures of student achievement. The philosophical aspects are considered in detail later. The intention was to rely heavily on grounded data, and to remain close to the evidence when planning and implementing modifications. The rationale for this was that there is a great temptation toward abstraction, in order to simplify the message and hence provide a more tractable student experience. The acknowledged steep initial learning curve for programming encourages this tendency. However, the complexities of the inter-dependency of many technologies and techniques mean that abstraction can hide (or gloss over) significant factors.

Within a student cohort there will be a wide range of capability and expectations. With the importance (in programming) of many small pieces of very specific information, this range can become magnified, and potentially unmanageable. One avenue that could be explored is to focus on one of a number of subgroups (females, mature students, those with particular aspirations or previous experience). A decision was taken early on to evaluate and support the learning of all types of students: those who enjoy programming, those who are largely neutral, and those for whom the unit is a necessary evil. To enable this to be measured, it is essential that all types of students are clearly engaging with the unit content, rather than either setting their own agenda, or 'borrowing' from other students. Plagiarism is a major concern, as ready-made solutions (or ones commissioned by students) are readily available (Clarke and Lancaster, 2006).

Measuring success is not a trivial exercise, given that parallel studies are not possible, and that the makeup of successive cohorts cannot be controlled. External factors, such as the development of a new course elsewhere in the university, or at a rival institution, may affect the nature of the recruitment. One approach is to compare the unit with the others being presented to the same students. Those units will focus on databases, web technologies, networking, systems design, and the business environment. Apart from the latter two, all units involve practical elements and assessments. Each student is asked to complete a questionnaire on each unit towards the end of the academic year, which is administered centrally by the

university. There are quantitative measures of achievements in assessments, and student use of online materials can be monitored. Every effort is made by nominated individuals (not directly linked to the programme) to interview any students who withdraw; those interviews may yield important information, but were not considered relevant in the context of this study. These interviews are voluntary and the reports compiled do not provide reliable or verifiable data.

The research was linked to changes in the curriculum and the programme. The programme had been revalidated two years before the main part of the study began, and a number of significant modifications had been made. The Programming unit was in transition from a very traditional, constrained scheme, towards one that allowed students more scope for creativity.

The aim of the research was to monitor, measure, understand and guide this transition, and thus help to achieve a positive profile of student achievement and motivation in the Programming unit.

The transition was planned to take place over four successive cohorts. This dictated the length of the study. As the changes were manifestations of a deep-seated modification in the ethos of the unit (from transcription to creation) it was necessary to investigate as many aspects of the delivery of the unit as possible, to ensure that any negative consequences would be highlighted.

The main focus for the study was the ephemera produced as a consequence of the learning and assessment processes: the coursework (tests and programs), and examinations. Of these, the programs are the most directly linked to programming, so the study concentrated on the applications submitted by the students for assessment.

If there were limited ranges of size and complexity in the programs submitted, then this would indicate that at least some of the students were being constrained, provided the variety in prior experience and expectations existed. Analyses of survey responses established that the cohorts did include students with considerable prior experience and ones with limited motivation to succeed at programming.

One would expect, therefore, to see a widening of the spectrum of achievement as the unit progressed. This may be a significant issue in a learning environment. Is the purpose of the unit to facilitate learning in all students, or is it to enable all students to achieve a given level of understanding and knowledge, implying that those already at that level should not, in effect, receive support? The changes to the pedagogy associated with the Programming unit reflected a transition from the 'competence level' approach to one where all are encouraged to stretch themselves.

One of the features of the redesigned delivery programme for introductory programming was an increased use of online and software resources, particularly ones that facilitate student interaction and assessment. None of these were employed in the previous delivery regime. The researcher was successful in bidding for some funds to develop software systems to assist in both learning and assessment. These were added to systems already built by the researcher to facilitate online submission of fully validated applications and multiple-choice online tests (both used previously in the delivery of other units).

By analysing the size and complexity of programs, when they were started, and the students' understanding of programs, it was hoped that a rich picture would be painted of the learning experiences of all students.

In the event, the research showed that all students gained understanding and all felt able to express themselves adequately. Each of the transitions in pedagogy were successful, in that they facilitated the writing of larger and more complicated programs at an earlier stage in the unit by all students.

1.5. The Research Issues

The logical starting point for any research is to define the nature of the task being studied. In this case, an exploration of the cognitive complexity of programming would highlight the key constraints for the design of the programming course itself, and frame the analytical research associated with it. This complexity would need to embrace the multi-faceted technological issues identified already, but also necessarily involved examination of motivation, problem solving and language

acquisition. Two examples will suffice. A group of around 200 first year undergraduate computing students were supplied with a program that printed an outline of the life of Alexander the Great. Instead of indicating that he conquered a number of countries, the tutor used 'visited' as a euphemism, including the single quotes in the print instruction. The students were then encouraged to substitute aspects of their lives, as an initial exercise. Some time later, students had to submit programs for assessment. 25% (N=47) out of 197 included programs which, whilst detailing aspects of their lives (and clearly not in a euphemistic sense), still retained the single quotes around the verb. This provides an informal measure of how passive students of programming can be, willing to substitute, but not delete. Secondly, almost all programming languages use English words, although the semantics are often quite different. In C, Java, C++ and C#, the word 'static' has a precise meaning, which is at variance with all the common uses of the word. Likewise, punctuation generally plays a more important role in most programming languages than in written or spoken language. This linguistic 'distance' between the perceived semantics and the specific use of words and symbols, may make a further contribution to the sense of dissociation many students feel. Sternberg (1977) suggested that analogical reasoning could be used to foster a sense of connection between two domains. Halasz and Moran (1982), however, argued that analogical reasoning could be detrimental as it depends on the assumption that the inferred and projected domains share fundamental characteristics in the specific areas where the analogy is being applied.

The examination of the linguistic, technological and pedagogic complexities of learning to program were followed by an examination of more of the literature associated with learning in general, and learning to program in particular. Much research has been devoted to this topic over many years, and by many researchers and research teams. A good deal of this research is experiential, with most articles detailing the student achievements and responses. Themes were derived from an analysis of this material both in terms of specific learning points, and at a more abstract level regarding the nature of how introductory programming is perceived. One study in particular, the Disciplinary Commons in the Introductory Teaching of Programming (Fincher, 2005) will be examined in detail, as it brought together a number of tutors (including the author) over a period of time, each reflecting on

selected issues by drawing upon his or her own experience (and that of the students).

The experience of students in programming on this programme had been patchy, with (often) large failure rates and a markedly bi-modal distribution of performance. Another tutor had introduced consistency in the delivery and assessment that had rectified the problems of completion, but there had been no modifications to the pedagogy.

The initial design involved modifications to the delivery and assessment, as well as to the adoption of Java as the programming language. Java, developed and maintained by Sun Microsystems, is widely used in the computing industry, and is closely linked to a number of other popular, professional languages, such as C# (used by Microsoft in its .NET framework). Java had been tried unsuccessfully before, over a period of three years. Rather than modify the pedagogic approach, the language itself was blamed for the problems. There was therefore some apprehension associated with reverting to Java.

A programme of change was designed and implemented over a series of cohorts. These incremental changes in content, presentation and assessment were monitored within each cohort, with the analysis influencing the nature and rate of change for successive cohorts. In the event the circumstances changed unexpectedly. During the third delivery of the revised unit, the university announced a major redevelopment of all undergraduate programmes, leading to larger cohort sizes through amalgamation of existing programmes. The fourth presentation of the material (again modified based on the evaluation) was to over two hundred students, two and a half times larger than the previous cohorts. The 'additional' students came from programmes that were more focused on IT than computing. The opportunity to evaluate the extent to which the revised course was applicable to different students was seized upon, and considerable data gathered and analysed. Nonetheless, the increased number caused a number of logistical issues, which potentially complicated the evaluation.

2. The Inspiration for the Research

There were five main sources of inspiration for this research:

1. An interest in the semiotics of programming.
2. An interest in the research relating to problem solving.
3. Developments and innovations in the Initial Teaching of Programming (ITP).
4. The Disciplinary Commons in ITP.
5. The Redesign of the Programming Unit.

2.1. The Semiotics of Programming

Every computer programming language uses syntax and sentence structures that are at least unusual, and often incomprehensible, to the layperson. Whatever the intention behind the design of the language, the general norm is that the meaning attached to a symbol or word when used in a computer program will run contrary to the meaning when that same symbol or word is used in natural language. For instance, loops in many commonly used programming languages are announced by the use of ‘for’. In English, ‘for’ is a preposition and does not imply action, let alone repetition. This dissonance between natural language and programming languages extends to the use of symbols, especially brackets and punctuation. The opportunities for novices to use transferable or analogical reasoning in program comprehension are therefore significantly reduced. Even apparent similarities can be traps for the unwary. Some students talk in terms of ‘if loops’, apparently conflating two concepts: the ‘if’ decision statement (that does not involve repetition), and the loop, which does. Closer inspection illustrates the complexity of the semiotics of computer programming languages *vis-à-vis* natural languages. In a natural language setting one might say: (when inserting coins into a parking meter) ‘if the total has not reached the required level, insert another coin’. In this context, it appears that ‘if’ controls the repetition, but it is the implied loop associated with the ‘insert coin’ action that suggests that the test should be repeated. Many individuals may not be aware of this semantic subtlety, which is analogous to young children calling an apple a ‘napple’ by falsely assuming that the indefinite article is always ‘a’. In programming one cannot replicate this sophistication, with the result that the phrasing would have to be

‘until the total is reached, or ‘while the total has not been reached. This ’false’ reasoning can be difficult to overcome.

Another main area of difficulty for students of programming is the semantics of (grammatical) symbols. In natural language these are essentially delimiters – segregating the text into more comprehensible elements. When to use a particular delimiter can be a matter of debate. As Kurt Vonnegut remarked of semi-colons: “they are transvestite hermaphrodites representing absolutely nothing. All they do is show you've been to college” (Vonnegut, 2003). In programming, each delimiter has a precise meaning and a defined set of circumstances in which it can be used. Appendix F contains a more detailed exposition of these issues, demonstrating the extent of the linguistic leap that novices need to make as one of the elements in learning to write computer programs.

2.2. Problem Solving

A solution to a problem is a network of components that, when traversed in connection with a given problem, produces the desired set or sequence of results. Problem solving is therefore the process of selecting, constructing, and sequencing solution components into a suitable network. A network (or directed graph) is required in the case of complex problems, as there is the implication that certain paths will be traversed more than once, if not many times. This is another of the elements that makes computer-based problem solving so difficult for the novice. There is a significant intellectual overhead in ‘walking through’ a potential solution component several times, whist maintaining an accurate mental model of the state of associated data. Programmers evolve their own ‘style’ of programming specifically in order to help manage this intellectual complexity. Using names and indentation in particular ways can provide additional cues that help to compartmentalise the traversal process, thus reducing the intellectual load. Style is a child of necessity not idiosyncrasy. Novices have no access to such intellectual shortcuts.

The selection of the solution components in programming may seem an intellectual process, but is heavily (and increasingly) a knowledge-based activity. There is a plethora of existing solutions and solution components readily available to software

developers via libraries and frameworks. Multiple potential solutions can usually be constructed, leading to the need both to know what is available, and to evaluate the qualities of the various alternatives. Using existing solution components is generally desirable as these significantly reduce the extent of testing and the likelihood of logic errors. Testing is the most expensive part of the software development process.

2.2.1. Problem Solving and Planning

Computer programs are often not merely the solutions to problems: they need to be manifestations of a particular type of planning system. In planning, the goals and operators that define the solution space are not immutable, as they are in problem solving. The goals associated with a computer application will often change, based on users' requirements, legal considerations, and errors. A computer program is, therefore, (in the artificial intelligence sense) a planning system where the goals and operators are subject to change, but where those goals and operators are fixed at any given moment. This has similarities with Mayer *et al.* (1986) who explored the connection between thinking and programming, suggesting (pp.608-9) that a focus on cognitive elements pertaining to programming is more productive than the development of more general intellectual skills.

2.2.2. Do Programmers need to be Geeks or Nerds?

The terms 'geek', and 'nerd' are all used (often pejoratively) to encapsulate the personality traits associated with IT in general, and programming in particular.

'Geek' (dictionary.com, 2009a) originally referred to fools or simpletons, or circus performers with bizarre 'acts', sometimes including biting the heads off chickens. It was another association, that of the ability to concentrate on detail (particularly detail meaningless to others), that led to its re-emergence in common parlance in the 1970's. Nerd (dictionary.com, 2009b) is a more recent addition to the language, first being mentioned in Dr Seuss, and gaining popularity in the 1950's as a slang term for a 'drip' or 'square', especially in Detroit, Michigan.

Do programmers need to be geeks or nerds? And, if so, is that a problem?

The limitations of current software development tools mean that programmers need

to be obsessed with detail. The intellectual complexity of comprehending and evaluating potential solutions also implies an immersion in the process that can lead to dissociation with colleagues. Yet it is also true that modern software development cannot be an individual activity. Software projects require a multi-disciplinary team that includes designers and users as well as project managers and testers, in addition to other developers. A professional software developer must therefore be able to relate to the potential solution (i.e., system) at multiple levels in order to communicate effectively with the other team members..

2.2.3. Summary

There are echoes of some of the elements of programming in other disciplines. Choreography, for instance, involves an arcane language and the construction of a 'solution' involving many components. The scale of the two disciplines is not so readily comparable. A twenty-minute dance will involve hundreds of steps. Most modern software projects have tens of thousands of instructions. The human form and capabilities constrain the choreographer. Software has almost no boundaries. The final difference is that dance steps are built from fundamental movements: in programming all of the concepts form a network, where each is dependent on others. Identifying a starting position and a direction of travel are not trivial problems.

2.3. ITP Research

The challenges inherent in learning to program have long been recognised. The psychological dimensions have also been explored over much the same time period. Gerald Weinberg (1998), Elliot Soloway (1985, 1986) and Thomas Green (Green, 1989; Petre and Green, 1993) (among many) have written extensively on the psychology of programming, and there are conferences and interest groups dedicated to a greater understanding of this area. The Psychology of Programming Interest Group publishes articles and holds an annual workshop. At a recent workshop Dehnadi *et al.* (2009) argued the psychological benefits of consistency in introductory programming.

The problem solving facets of constructing programs is an aspect that has received considerable attention. Papert (Solomon and Papert, 1976) with his invention of

Logo, attempted to create a more readily accessible programming system. The ‘turtle graphics’ within Logo have been translated for use in other programming languages, including Pascal and Java. In turtle graphics the programmer guides a writing object (the turtle) around a two dimensional space. The anthropomorphising of the graphical processes has itself become of interest to the research community, notable in the development of the Alice project (Cooper *et al*, 2000; Kelleher and Pausch, 2007) and Scratch (http://scratch.mit.edu). Scratch is one of many learning projects based on Squeak (http://www.squeak.org).

One interesting difference between Logo and these environments is the extent to which recognisable symbols and entities are used. In Alice, the programmer can create three-dimensional worlds and populate them with characters, such as animals and people. Although there are obvious physical differences between these virtual worlds and the real one, much of the movement and physical attributes are plausible. In Logo, the turtle is imaginary, only that part of its path when it was writing is visible. The intention was to assist the student in creating an abstracted mental model of the algorithm creation that was not necessarily based on common psychological processes. Recursion is a powerful problem solving technique that can only be realised within a computer. Papert and other advocates of languages like Logo (e.g., Scheme) based on Lisp aimed to help students learn to understand and use this alien problem solving technique. This can be demonstrated in the use of Logo in highly recursive algorithms such as fractals and Sierpinski curves (Ross, 1983).

Recursion depends on the creation and manipulation of multiple concurrent states of the solution space, so the use of characters in Alice prevents the consideration of recursion. Alice therefore emphasises the ‘translation’ paradigm of programming, where the individual seeks to represent his or her own (non-recursive) solution in terms of a program. It is the case that recursion is most effective within a highly regular solution space containing relatively few operators. Few real-world problems exhibit these characteristics, so the priority to cover recursion has reduced as programming has permeated more aspects of life.

Syntax-directed editors and specialised development tools are approaches that aim to lower the slope of the initial learning curve. A syntax-directed editor such as Xcode

(Apple, 2008), allows the programmer to create a program by filling the required slots, with the editor supplying the constant elements. The author used such an editor (based on the work by Morris and Schwartz, 1981) in an introductory course some years ago. Only one student (of 12) found it useful. Almost all the others formed a positive aversion to using it. BlueJ (Barnes and Kolling, 2002) and jGrasp (Cross and Hendrix, 2007) are two examples of visual development tools that espouse the same approach, extending the notion to include methods and classes.

These types of systems have benefits, but there are risks. A learning curve is not one-dimensional: attempts to lower the slope in one direction may cause an increase another facet of the slope. The design of Java is an example: every Java application must include the same text – text that only relates to the operation of the Java system, not the program under consideration. Forcing the students to type this text in each time they write a program is likely to lead to errors and increase the slope of the learning curve. Wrapping up this standard text, or providing some means by which it is automatically generated eliminates that problem at the risk of preventing the students from understanding the boundary between the language and the system supplying the text.

Mechanisms such as ‘advance organisers’ (originally suggested by Ausubel, 1960) have been tried (Mayer, 1981) in an attempt to improve students’ comprehension of program code. The early experiment by Ausubel showed that a suitable advance organiser could assist the learner in retaining recognition and recall of symbols, but this was challenged in a later paper (Clawson and Barnes, 1973). That many struggle to memorise and retain the memories of unusual symbols was not disputed. The use of pseudo-code could be seen as one application of advance organisers. At first glance it would seem sensible to provide students with a view of creating programs that is not concerned with the detailed syntax. This would be the case were it not for the influence the syntax has in many programming languages on the solution (program) that should result. For instance, the range and type of data structures available in a particular language often dictates key elements of the solution. The process of translating the pseudo-code into a working program is therefore rarely a trivial process, and could be so complex that the use of pseudo-code could be counter-productive. Programming books rarely include sections on

pseudo-code.

The commitment required to reliably assimilate the syntax, structure and conventions of a particular programming language and programming would suggest that the range of achievement within a cohort of students is likely to be quite large, even where all students have comparable prior experience. Commitment itself cannot be considered to be a fundamental factor. Learning styles and prior experience may have a more profound influence on the learning process and hence on the level of commitment. Thomas *et al.* (2002) examined the effect of learning styles on achievement in ITP, and found a relationship between learning style and achievement. Whether this was due to an underlying influence of learning style or to bias linked to the delivery programme could not be established. Byrne and Lyons (2001) and Jenkins (2002) have also commented on the influence of learning style, without coming to any firm conclusions.

An ever-present danger is that the existence of any (even notional) commitment 'threshold' will polarise the student cohort into (generally) three categories. There might be those that are prepared to make the requisite commitment, and those who are not. There may be a number in the middle willing to make a conditional commitment, such as to achieve a pass in an assessment component. Jenkins and Davy (2000) and Davis *et al.* (2001) have observed this 'tri-partite' phenomenon. The danger is that students might select an approach to learning that is more connected with a perceived desire to project a particular social orientation than with their own learning needs. The consequence would also be a bi-modal achievement profile, with the committed students achieving high grades and the individuals in the other groups being awarded much lower marks, as there is little difference between intermittent commitment and low commitment where neither reaches the required threshold. Members of these two lower-achieving groups might be tempted to plagiarise, and this has become a major concern in teaching programming (Culwin and Lancaster 2000). Plagiarism in programming is itself a major research area, and many resources are available (HEA-ICS, 2008), and conference and journal articles appear regularly (e.g., Joy and Luck, 1999; Chen *et al.*, 2004; Jadalla and Elnagar, 2008).

The sequencing of the introduction of programming concepts has also received attention. Which statements to introduce and in which order, when and where to introduce data structures, can lead to numerous permutations of delivery scheme. When the paradigm (process-oriented, data-oriented, object-oriented) is overlaid, further alternatives are available. Even within object-orientation, the point at which new classes are defined can vary.

The programming language and system are also considerations that can complicate the issue of conceptual selection and sequencing. Different programming languages affect this design in different ways. A language like Java (widely used in higher education) does not offer the same flexibility as (say) Python with regard to building simple programs. Once the initial hurdles are overcome, Java includes features that assist the learner by providing more feedback and control. As the choice of programming language has such a pervasive influence on the delivery scheme and the learning experience, it is not surprising that much energy has been expended in support of particular languages. It is often said that the easiest way to start an argument between computing people is to suggest that a given language should be used to teach programming.

2.4. Disciplinary Commons in ITP

This was an initiative started by Sally Fincher and Josh Tenenberg. A National Teaching Fellowship was awarded to Professor Fincher in 2004, and part of the funding connected with this award was used to fund the UK activity. Tenenberg led the US initiative, where the remit was widened to include all of Computer Science, but confined to a limited geographical area. The idea for the Commons was to bring together lecturers with considerable experience of teaching programming to pool their ideas in the form of portfolios. More information is available (Fincher, 2005). The intention was to create portfolios of experience and reflection, not to create a definitive learning regime or to decide (for instance) on the most appropriate programming language.

The notion of a portfolio as a research instrument is associated with researchers such as Pat Hutchings and Lee Shulman (Hutchings, 1998; Hutchings and Shulman,

1999). Interested individuals respond to an invitation to participate, and participants are selected on the basis of perceived expertise. Within the study, participants engage in a range of activities, including directed reading, discussions, and the selection of relevant artefacts. The participants are then asked to reflect on the activities. This is a specialisation of case study research, as the researcher selects the aspects under consideration, and the participants are active elements in the generation of data, but are not involved in other facets of the research process. It is generally the case that the researcher analyses the reflection and the selections, but does not pass judgement on the artefacts selected or the selection process.

There is a displacement in time, where participants are asked to reflect on past events. The nature of the reflection is the focus for the analysis: the tenor and content, the volume of positive, negative, and descriptive elements, the level of introspection, and the extent of the use of corrobative data can all provide insights into the nature of the intellectual engagement of the participant with different aspects of the activities.

Criticism of portfolios is not difficult to understand. The researcher cannot know for how long the participant reflected, nor can it be established whether the reflection truly reflects the participant's views – either at the point of reflection or during the activity. There will be an element of *post hoc* rationalisation, as always in the case of time-displaced studies. The reflection need not be supported with verifiable data; therefore the veracity of the statements may not be dependable.

The key element is the motivation of the participant. The reflection should be seen as a positive process for the individual, both in terms of the immediate response of writing the reflection, in terms of sharing with others and of experiencing the reflections of others. If this is the case, it is reasonable to assume that the extent of misleading rationalisation may be limited. If the researcher is seen as judgemental of the portfolio or applying some level of comparison between portfolios, or (for instance) one or more participants does not perceive himself or herself as one among equals, then it may easily be the case that the participants will be defensive, and the reflection coloured beyond the point of usefulness.

The ‘Commons’ was conducted during 2005-06, and involved 20 academics with considerable experience in ITP being selected from those who expressed an interest. Plenary monthly meetings were held in London South Bank University. Professor Fincher decided the theme and agenda for the meetings, each of which lasted around six hours. The attendees were asked to contribute by participating in the discussions within the meetings, and writing a reflective piece on the topic under discussion (either before or after the meeting). These pieces were then collected by the individuals into their respective portfolios and submitted to Ms Fincher for analysis, and with the intention of their being made available at a later date to the wider academic audience. The experience has been disseminated in articles (Fincher, 2006b) and a website (Fincher, 2006). Since then, several other computing Commons studies have taken place.

Prior to the first meeting, the author expected that the function of the Commons was to be positivist – to collate and evaluate experience and to decide on the most appropriate delivery and/or assessment scheme. The author soon realised his mistake and initially found the lack of focus on producing definitive answers to be somewhat disconcerting. As the diversity of approaches being employed by the various attendees became more apparent, the rationale for the Commons (a group of equals) became clearer. The intention was to expose the participants to richer and more reflective views of a variety of approaches to the initial teaching of programming than could be achieved via journal papers or conference presentations.

The next consequence was to enthuse the attendees to extend the level of their reflection, as the environment created was so positive and supportive. The time taken to travel back and forth from and to the meetings provided additional opportunities for reflection, the rail system notwithstanding. Attendees were provided (at the meetings) with ‘train reading’ to provide an impetus to the reflection for the next meeting.

A request for participation was made early in the summer of 2005, with the selection being confirmed some two months prior to the first meeting. Two participants had to withdraw due to work pressures, and not all those who persevered to the end submitted complete portfolios. The actual number of submissions was 12.

Dissemination of the results took the form of conference papers and journal articles, written by various subgroups of the participants. A Commons in the teaching of user interfaces took place during 2007-08 and others have taken place since.

As the Commons progressed, it became clear that the approach to teaching in general, and programming in particular, had received far less research focus in the author's School than was the case for the other institutions represented. This was evident in the learning materials being used, and in the nature of the learning and assessment processes being employed. This consideration of innovation resonated with the author, and provided enhanced motivation to understand in more detail (through research) the processes relevant to learning to program.

The author found the process of portfolio creation within the Disciplinary Commons to be cathartic and illuminating. Thinking more deeply about aspects of the activity highlighted the extent to which components of the delivery and assessment scheme had become axiomatic simply through custom and practice. The timescale of the Commons (some 9 months) allowed a gradual evaluation of the precepts underpinning the design of the delivery and assessment. These deliberations were a significant factor in the redesign of the introductory programming unit. It was not surprising to discover that the symbol used in the Commons logo was derived from Frank Lloyd Wright's experiment with a form of architecture commons that he conducted in his home (Taliesin) in the early part of the twentieth century. The conduct of the Commons also has echoes of Donald Schön's practicum (Schön, 1984).

The views of the other participants were of more incidental benefit to the author. Each was working in a different environment of institution and course; some were mainly teaching, whilst others were more focused on (generally non-pedagogical) research. There was a great sense of camaraderie and collective participation, at least from the author's perspective that helped to engender a sense of responsibility for the completion of the portfolio. It was also the case that Sally Fincher organised the meetings meticulously, so the author found each of the meetings stimulating, despite their length.

2.5. The Redesign of the (Introductory) Programming Unit

A number of conclusions can be drawn from the literature regarding the necessary and desirable components that should comprise the delivery and assessment scheme for an introductory programming unit. These were encapsulated as a set of eighteen principles, which are listed in Appendix G. These principles suggest that individualising the learning experience is central to a notion of creative programming, and as a direct result of the nature of programming and the processes involved in the learning of programming. There is also recognition of the complexity of programming that should influence any aspirations the tutor may have for the level of student achievement that is possible in an introductory programming unit.

2.5.1. Initial Application Genre

Although all programs provide a mapping between an input dataset and an output dataset, there are many forms of mapping. Each form can be considered to represent a separate genre of program.

The following is an indicative list of some of the most common programming genres.

1. Data Processing.
2. Graphical User Interface (GUI).
3. Mobile.
4. Game.

The differences become apparent in non-trivial programs – programs having multiple manipulations. Historically, data processing applications emanated from the processing of business data, typically financial transactions. In a data processing application most manipulations are in some critical manner different from other manipulations. In a timetabling system, for instance, scheduling will probably involve logic specific to the type of room (e.g., lecture theatres, meeting rooms, laboratories).

In GUI applications the user is presented with a number of components on a form where each component is capable of responding to a variety of user actions. The critical element is that, when the user interacts with a visible component, this user action is generally linked to the manipulation of other components.

Mobile applications are similar to GUI applications in terms of managing components and the consequences of user actions, but there is the addition of elements specific to the deployment of the application. This brings in the need to understand the underlying systems software, something that is also true of web applications.

The term ‘game’ here is used to embrace those applications where the elements of the solution space are well defined. It is also the case that there are often regularities to the solution space that can be exploited. Solitaire would be an extreme example, where only one type of move is allowed. Game applications often involve creating solutions by exploring networks of potential strategies or actions.

Elegance

‘Elegance’ is a term often applied to solutions in general. It is interpreted here to mean ‘apposite’ or ‘parsimonious’ – the use of sufficient elements to solve the problem and no more. Each genre of application involves a different interpretation of elegance. In game applications with their exploration of highly regular solution spaces, elegance tends to manifest itself in terms of recursive, compact algorithms. An elegant solution for solitaire might involve less than 20 instructions.

Elegance is not simply an aesthetic aspiration. One tends to find that inelegant solutions are very difficult to comprehend and therefore to modify reliably. A long-winded solitaire program will generally include extraneous elements that may well render the program incapable of being modified without introducing secondary errors.

In GUI, mobile, and web applications the ‘two worlds’ of the user interface and the data manipulation need to be separated when represented in the program. If this is not followed, then complex dependencies will be generated that will prove difficult

to modify. The model-view-controller (XXX) observer design pattern was created to help programmers realise this separateness. The ‘view’ corresponds to the user interface, and the ‘model’ to the data and its manipulation. The ‘controller’ facilitates communication between the two in such a manner that dependencies are minimised.

Data processing applications can be considered to correspond with the ‘model’ element of the MVC design pattern. Many ‘real-world’ applications have little or no user interface. Examples would include applications that process invoices, and programs that download and apply updates to operating systems or applications software. Data processing applications are characterised by being aggregations of different manipulations. There will often be insufficient regularity in these manipulations to use the same sequence of statements to perform more than one manipulation. In attempting to create a single algorithm that can be applied to multiple manipulations, the programmer may impose constraints that will make it more difficult to test or modify the program. In data processing applications it will generally be the case that the size of an application will tend to be proportional to the number of manipulations, and vice-versa.

Within the programming units across the programme, the main genres explored were (and are): data processing, GUI and web genres. As ‘data processing’ is common to all these genres it was decided that this would be the genre to which the students would be first introduced. It follows that application size would be an appropriate element in the measurement of student progress..

2.5.2. A Different Programming Language

The programme (which includes the first year Programming unit) was substantially remodelled during 2004 and 2005, in terms of the unit structure and content. The first year (level C as it is designated) Programming unit remained, with modifications to the assessment (with the re-introduction of an end of unit examination) and to the programming language (Java replacing C). There were also changes to the teaching team, with the author becoming the unit leader.

The impetus for these alterations came largely from two sources: to rectify reductions in recruitment, and to update a very traditional content and pedagogy. The

Programming unit typified the traditional nature of the approach taken. The C programming language was developed in the late 1960's (Kernighan and Ritchie, 1988) originally as a high-level assembly language, and remains a language primarily used to write systems software. The programming assignments were highly prescriptive, echoing the 'cipher' or 'translator' role of a developer within the software engineering discipline. In a strict interpretation of software engineering, the user is responsible for the requirements of a new software system, and the software engineer's responsibility is to gather, analyse, then transcribe these faithfully into software. Incidentally, this partitioning of responsibilities is generally inappropriate in a modern commercial marketplace where both user and developer are encouraged to be creative in the application of information technology.

2.5.3. Operational Considerations

All learning in a formal environment is subject to a number of strategic, tactical, and operational considerations. The strategic and tactical elements can be considered to be reasonably static, as programming will form part of every computing undergraduate degree for the foreseeable future, and the main objectives of such a unit will remain much as they have been.

Higher education is subject to cost constraints, in common with every area of government activity. These constraints are partially linked to limits on income, and there are cost pressures associated with advances in learning technology. These technological developments would extend beyond the learning management software, to include networking of lecture theatres and laboratories to capture learning activities and facilitate access to the learning system and to other learning resources. The learning management system (also termed a virtual learning environment) can offer mechanisms to simplify and automate submission of assignments and provide access to feedback, marks and grades. These latter two may also be integrated with the student records system, to simplify the production of results for consideration by examination boards.

Staff costs associated with technical support is also a consideration.

The technology helps to individualise the learning experience by allowing students

to access materials at a time and in a place that may be more suitable and convenient. More materials may also be available, especially if one factors in the plethora of websites related to almost every area of education. One consequence is that supervised workshops will gradually morph into learning studios, where members of staff are available to support and guide, rather than direct learning. A reduction in the interaction between tutors and students would then be expected that, in turn, leads to pressures to increase the number of workstations in laboratories, and the creation of larger learning environments or 'studios'.

A parallel development, aimed at increasing student choice (as well as reducing costs), is the introduction of frameworks, which accommodate multiple programmes, each of which may include a number of pathways, or awards. A framework consists of a number of units (or modules). A programme is a particular subset of units, and an award (or pathway) will consist of a specific subset of units within a given programme.

Common units in a framework can then be shared, reducing costs. Greater incidence of commonality in the first year allows students to delay the choice of their specialism, which many students find appealing. Common units thus tend to have very large numbers of students, and need to cater for a wide range of student expectation, motivation, and prior experience. There may be additional learning constraints on such units, as they will need to provide the underpinning for each of the successive units in the various programmes.

The framework model was adopted prior to the first cohort. Four pathways were included, three of which (Software Engineering, Software Engineering Management, and Software Product Design) were focused on specific areas within computing. There was also a Computing award where students could select any of the units from the focused awards, which then catered for those students wishing to have a more individual learning programme.

Subsequently, this Computing framework was amalgamated with two others (in Business Information Technology and Networking) to produce the Software Systems framework. Over the four-year period of this study, Cohorts A, B and C were part of

the Computing framework and Cohort D was the first of the Software Systems cohorts. Programming was a compulsory unit in all three of the frameworks that were amalgamated into the Software Systems framework, although the delivery and assessment schemes were different. All three frameworks also included a third year industrial placement, but a minority of students undertook placements that involved producing software.

The learning management system (or virtual learning environment - VLE) was gradually phased in over a few years, starting after the introduction of the Computing framework. Consequently, most of the facilities offered by a VLE had to be replicated by the author. These included online tests, assignment submissions and access to assignment feedback.

2.5.4. A Redefinition of Axioms

The adoption of the following axioms underpinned the redesign of the unit:

1. Universe transitions impede learning
2. Learning to program benefits from continuous engagement.

Many introductory texts recognise the complexity inherent in learning to program, and attempt to simplify the process with examples that do not require either loops or collections (e.g., arrays). This approach requires at least three fundamental ‘universe transitions’ before students can tackle realistic problems: the use of loops, collections, and classes. A ‘universe transition’ is here taken to mean the modification (as opposed to the expansion) of the scope and nature of the problems being examined, and the operators available to construct solutions.

An early adoption of loops and collections (e.g., arrays) reduces the number of universe transitions. The semiotic overhead of the syntax of loops and arrays was considered to be negligible, given the high semiotic complexity of even the simplest Java program. The need for multiple transitions also increases the chances of students developing problem solving strategies that are too closely tied to the current universe, thus producing resistance to the transition above and beyond that which might be expected. See Appendix H for a more in-depth analysis.

More detail on the redesign of the Programming unit is available (Jones, 2007).

2.5.5. A Phased Approach

It was decided to introduce a more modern pedagogical approach gradually, phasing the process over four successive cohorts, to facilitate a high level of control over the introduction and the measurements of the effects of the changes. The level of success of students in obtaining industrial placements and employment remained very high, indicating that many aspects of the curriculum and approach were deemed appropriate. Student feedback was also largely positive.

The component of the transition were elucidated as:

1. *Introduction of portfolio assignments*. A portfolio more closely links the weekly workshops with the assignment, and also provides the student with more choice in the selection of work to be submitted.

2. *Increasing the number of assessed elements*. The intention was to emphasise the benefits of a continuous approach to learning to program.

3. *Increasing the profile of program comprehension*. Modern software construction is a fusion of original algorithm design and implementation, and integration with existing software components.

4. *Increasing the self-reflective element in assessment.* The objective was to increase the participation of the students in the design of requirements, and to raise the students' awareness of the programming constructs.

5. *Placing more emphasis on a wider range of programming constructs.* The constructs that traditionally receive little attention in introductory units are those dealing with error handling.

6. *Early introduction of a realistic problem universe.* This implies that students would be introduced to the design of classes at a very early stage.

As each component represents a separate (albeit evolutionary) step in the change

process, it was decided to synchronise the introduction of components with successive cohorts. Component 1 was introduced in Cohort A, components 2 and 3 in Cohort B, and 4 and 5 in Cohort C, with component 6 being delayed until Cohort D. In the event, the amalgamation of the three programmes into the Software Systems framework created a significant challenge, so the sixth component was changed to:

6. *Promulgation of the delivery scheme to a larger and more diverse student population.* The numbers of students trebled, the learning environment changed, as did the staffing.

2.5.6. The Central Role of the Portfolio

The successful introduction of a portfolio assignment was the crucial first step in the planned pedagogical evolution. To ensure success, the portfolio would need to satisfy meta-requirements, and so be more than an arbitrary collection of applications. Instead of writing software to meet specific, functional requirements, each student would need to select a group of applications that together demonstrated specified programming techniques. For computing to be seen as a creative, rather than a translation, discipline, the focus needs to be placed equally on the programming techniques and on the requirements of the commissioning user. The applications in a portfolio can thus be seen as analogous to studies in an artist's portfolio.

2.5.7. More Assessment Elements

The prior assessment regime included an assignment submitted towards the end of each of the first two terms. The time between assignment deadlines was therefore in the region of three months. The workshops continued, and so there were opportunities for students to receive support and informal guidance, throughout the academic year. No formative assessment was provided. For novices, such a time span between summative assignments is considerable, and that is not the only potential source of anxiety. Where there are two assignments in an assessment regime, both need to be represented in the calculation of the final mark.

An increase in assessed elements can ameliorate both sources of concern. With (say)

four programming assignments, students would more readily be aware of their progress, and there would be the potential for the weakest mark to be omitted (from the final calculation). Such an increase would double the marking load, so action was required to manage this. The placing of the first programming assignment was also a consideration. An early assignment presents opportunities and threats. Provided (almost) all students submit a passable piece of work, general anxiety within the cohort would be reduced; the reverse would be true if even a substantial minority were to fail (or fail to submit).

Purely automatic marking was discounted, as this inevitably drives one towards a purely mechanistic assignment, and there is little scope for innovation (on the part of the student) to be rewarded. Elements of the assessment process therefore became the focus for automation. These were identified as:

a. *Online tests.* The use of online tests can minimise marking time. Making a subset of the questions available during practice sessions can provide formative assessment and augment the learning process.

b. *Online assignment submission*. This would substantially reduce the effort involved in collecting and processing CDs or floppy disks, and minimise errors, provided the submission system rigorously validated the submitted files.

c. *Generation of feedback.* This involved the use of standard phrases linked to document generation software, and helped to minimise the time taken to generate feedback once the marks had been decided.

d. *Use of a supplied library.* This provided a range of features to create new projects and simplify input/output. This was more than ‘wrapping up’ complexity for convenience – the features were based on sound programming principles.

e. *Generation of sample code.* The intention was to facilitate more emphasis on code layout and variable naming, as well as enable those students with minimal confidence to create applications suitable for submission.

f. *Identification of concept realisation.* Every program contains realisations of programming concepts via constructs. This software was intended to enable students to identify the relationships between the concepts and their corresponding realisation(s).

g. *Identification of programming constructs.* Software which can list the constructs used in an application can be useful in providing formative feedback, as well as assist in the marking processes for summative assignments.

At this time, the HEI (Higher Education Institution) in question was in the process of evaluating Virtual Learning Environments (VLEs). Fortunately, the author had been involved in the development of a learning management system that included an online question component (Jones *et al.*, 2003). This was easily modified to provide the necessary functionality. Similarly, the author had constructed a rudimentary online submission system that was enhanced, particularly with regard to the validation of the submitted files. As the intention was to create an integrated system, it was not possible to include any of the many software products and systems available to fulfil the other requirements. This necessitated the production of the software, some of which was done in projects commissioned under the 'Releasing Potential' initiative launched at the HEI in 2006. In all, three projects were awarded to the author within this programme.

2.5.8. Student Activities

The revised delivery programme included one of the innovations introduced previously: that of multiple exercises in a given workshop. The use of a long list (e.g. 12) of exercises allowed students to select certain activities, as it would not be possible for a novice to complete all the exercises between the weekly scheduled sessions. An example of a revised workshop session is included in Appendix L.

Appendix L also includes an example of the first programming assignment. This was distributed at the outset of the delivery of the unit, to be completed by week 6. The assignment allows for student choice in terms of the applications to be submitted and the number of applications. Two of the options allow the student to include code

generated by a supplied software tool (JCodeGen) that is described later as one of the pedagogical innovations. Where generated code is included the student would be required to correct any styling errors, replace any unsuitable variable names with more appropriate ones, and add relevant comments. This emphasises the equal importance given to comprehension in the early stages of the delivery of the unit.

2.5.9. Fostering Creativity

Creativity can be viewed as the reconciliation of multiple goals (some of which may not be fully understood) into an artefact that has the power to astonish. There is a sense in a creative discipline of some element of discontinuity, or of nonlinearity – redefining problems and/or realigning solution components in novel ways.

Inherent in the notion of creativity is a considerable element of freedom. As Albert Einstein is quoted as saying “You can never solve a problem on the level on which it was created”. The danger this poses in an educational environment is that the consequence may be a susceptibility to the charge of ‘relativism’ – many potential solutions could be seen as impervious to criticism.

The solution is a focus on meta-requirements: on the underlying techniques and structures. Many arts subject use this approach, and it can easily be adapted for programming.

Accordingly, the requirements for the portfolios were expressed in terms of programming concepts, with students being free to create and mould their own problems around a specified set of programming constructs.

The corollary of this approach is that the assessment needs to examine the artefact (in this case program code) in order to check that the concepts have been used. Software was developed to assist in this process. The experience of this style of assessment is described later as one of the innovations introduced in the delivery and assessment of the revised unit.

2.5.10. Program Comprehension

Examining existing artefacts and techniques is vital in all learning. There may be an

initial ‘assimilation shock’ with the learner feeling unable to take in all the ramifications, but this can be ameliorated through explanation and exploration. With the semiotic and conceptual overhead in programming there is a danger that this shock will manifest itself in an ‘all-or-nothing’ strategy whereby the student blindly accepts the program and uses it ‘as-is’, or simply fails to engage with the artefact at all.

To reduce the chance of assimilation shock it was decided to get the students to focus on aspects of the presented code by rectifying errors and limitations in styling or naming. As there are few rules involved, it was hoped that students would spot problems fairly easily, whilst exposing them to the code over an extended period of time.

A program generator was written to produce endless variations of small programs, thus fostering some sense of identification within the students, and creating an environment where students could collaborate freely. Two further versions were created, based on student feedback.

2.5.11. Framework for the Modifications to the Delivery and Assessment Schemes

In response to the conclusions, the delivery and assessment schemes were modified. One consideration was that the changes should be seen as evolutionary, rather than revolutionary. The previous regime had satisfactory results and students expressed general approval of the unit. A second element was also crucial in this respect. The Peer Assisted Learning (PAL) system used successfully in the Computing framework involved second year students tutoring first year students on aspects of the curriculum (Fleming, 2004). It was therefore important that the second year tutors should understand that the changes would add to the learning experience.

2.5.12. Modifications to the Delivery Scheme

Appendix A includes an outline of the changes to the delivery and assessment schemes that were implemented for Cohort A. Eight changes were made to the delivery scheme, and twelve to the nature and operational arrangements for the assessment scheme. These changes emanated from an analysis of the eighteen

principles outlined in Appendix G. A key element in these modifications is the increase in the visibility and immediacy of the delivery. By demonstrating the writing of code in front of students, and by getting them to demonstrate their code to a tutor, there is the sense of connection between the individuals and the artefacts that can lead to more timely informal and formal feedback. A more detailed analysis can be found in Jones (2007).

2.5.13. Summary

Many small, incremental changes were required to the delivery and assessment schemes in order to implement the range of changes required. The increased use of automated components (via the supplied library and the various online and marking support systems) meant that the level of staff effort would decline (once the systems were produced). The systems associated with assignment submissions, demonstrations and marking were also designed to minimise the time taken to mark and return student work.

The production time for the support systems interfered with the time to produce the learning materials, with the result that in the first term of Cohort A more use was made of manual processes than originally envisaged. This had a negative impact on the time taken to mark (and turnaround) the two assignments.

Incidentally, for purposes of convenience, all programming assignments were designed as ‘portfolios’. Of the 14 assignments analysed, three were constrained assignments, where students were required to write a single program. The constrained assignments were the final assignment for Cohort A, and the two final assignments for Cohort D. It should also be noted that another tutor was responsible for the delivery of the latter part of the unit for Cohort D. This was done, in part, to smooth the amalgamation of the three programmes into the Software Systems Framework.

3. Research Approach

Where resources are limited (which is the case with almost all research studies), compromises need to be made in terms of the three main dimensions: time, width, and depth. The ‘width’ of a study refers to the number and/or profile of the participants and the number of facets to be investigated. The ‘depth’ is the degree to which the selected facets are to be investigated. Time can be used either to focus on one group over a period of time, or to study a number of groups each over a period of time. The scope of the study should be driven by the aims and objectives of the research, although logistical and resource implications need to be considered.

3.1. Aims

The overall aim of the research was to obtain a better understanding of some of the key elements in the process of learning to program in the context of an academic course that is linked to an academic group which focuses on applied research in software engineering. To gain some insight into a range of factors, considerable data needed to be gathered and analysed. Whilst no group of students is homogeneous in terms of its computing background and ambition and motivation, the variety in each of these dimensions is not linked to gender, age, or ethnic origin. It therefore follows that data needed to be gathered from all the students in a given cohort. The use of multiple locations would provide a richer picture of the processes involved, but pedagogical incompatibilities and logistical problems limited the geographical scope to a single programme in a single institution.

As one aim of the research was to examine the management of change, it was clear from an early stage that more than one cohort would need to be considered. In the event four consecutive cohorts were included in the study. The first two relate to the time immediately prior to the start of the study, and were included to provide insight into the progress of the transition, a more general view of the facets of learning to program, to facilitate comparison with later cohorts, and to obtain some estimate of the research value of the data gathered as an inherent consequence of unit delivery. In parallel with the planning of the research study (and the delivery of the unit to Cohort C), the nature of the undergraduate computing provision was being redefined,

and three programmes (Computing, Business Information Technology, and Multimedia and Networking) were amalgamated into one (Software Systems) framework with a common first year. It was decided to include the first delivery of the unit to the first intake of the revised framework, even though pedagogical compromises were necessary (as a consequence of the amalgamation), and there would be logistical issues emanating from the larger numbers and changes in staffing. On balance it was felt that the aims and learning objectives of the revised unit were sufficiently similar to those of the preceding unit to enable a reasonable degree of comparative analysis.

3.2. The Cohorts

The research study therefore covered the first full-time academic years of four consecutive cohorts, identified by the letters A to D. As mentioned previously, the first Cohort (A) was completed prior to the commencement of the consideration of the topic for the research. The delivery of the unit to Cohort B coincided with the development of the research proposal. The additional mechanisms used to support concurrent gathering and analysis of research data were then first available for Cohort C. These were refined for the research in association with the delivery of the revised unit to Cohort D.

Although the cohorts had many similarities, there were certain characteristics pertinent to each cohort. Cohort A was the first intake following a revalidation of the Computing programme in 2005. This revalidation included significant changes to the structure of the programme and units. An examination was re-introduced for the first year Programming unit, and the Java programming language was adopted. There were also changes to the teaching team for the Programming unit. There were therefore considerable risks associated with the delivery of this unit, which resulted in a more conservative approach to introducing changes to the delivery and assessment regimes.

The following cohort (Cohort B) saw the introduction of the major changes to the learning and assessment regime. These were: the online submission of assignments, the generation of feedback, the use of online practice and assessed tests, an increase

in the number of assignments, and the introduction of a code generator.

Some minor innovations were made for Cohort C. A library of useful classes and applications was distributed to all students, and portfolio assignments needed to include a file containing the claims of concept realisation. In addition, the code generator was revised, as was the feedback generation system.

The presentation of the Programming unit for Cohort D (the first intake for the combined Software Systems Framework) was to in excess of 200 students, considerably more than the 70-85 students involved in the three previous cohorts. The pedagogical compromises necessitated by the amalgamation resulted in the adoption of the Computing approach for the first term, migrating to the approach previously taken by another tutor in the other programmes for the second term.

3.3. Research Methodology

The various elements pertinent to the research were summarised and encapsulated as a series of research questions.

3.3.1. Review of the Purpose of the Research

The research study aimed to monitor and measure the progress made towards transforming an introductory programming unit in an undergraduate computing programme from a reductionist to a creative paradigm. Instead of a developing a toolkit of solution components then applying these in solving defined problems, the intention of the learning programme would become one in which the students would select their own activities. As almost all forms of programming include the same concepts, the conceptual area covered is likely to be the same, but it was hoped that students would become more confident of their own capabilities, and explore aspects of programming beyond the basic curriculum.

This transformation could not be achieved in a single step: the pedagogical perspectives of the other tutors, the other units in the programme, and the majority of views expressed in the prevailing literature were (and largely are) reductionist. Additionally, there remained a number of unknowns regarding the pedagogy that

would facilitate more creativity, and the nature of the intervening points along the transformational path. The progress along the path had to be controlled, to ensure that the direction of movement was always positive. If not, the scepticism of the other stakeholders may have been reinforced. More importantly, risk-taking would be unethical, as programming is a compulsory part of both the first and second years of the programme that was the focus for the study. It is (and was) also the case that few students have prior experience of programming, which means that a wide range of capability and motivation will exist within a single cohort of students. It was noted that the (then) current pedagogy had not produced results that required immediate and drastic modification: the motivation for change was pedagogical enhancement, not pedagogical recovery.

3.3.2. The Research Questions

The main objective of the research was to monitor and measure the effects of implementing a paradigm shift in the pedagogy of an introductory programming unit, where that shift was to be implemented incrementally over four successive cohorts. The main research question ‘can a paradigm shift be implemented’ needed to be elucidated in *offensive* and *defensive* terms. Offensively, the shift should improve students’ capabilities in programming. Defensively, no student should have his or her learning impaired by the changes.

As the pedagogical modifications were to be phased over a period of four academic years, these offensive and defensive aspects were to be realised both within each cohort and between cohorts.

Within the delivery to each cohort the following questions need to be answered:

1. *Is there evidence of learning in all students?* All students should demonstrate measurable learning throughout the unit delivery. On average, student performance should improve, and there should not be an increase in failures or withdrawals. In terms of programming, each portfolio assignment should be more sophisticated than the previous one, in terms of the numbers of instructions, and the breadth of programming concepts realised.

2. *Does the profile of student performance follow a uni-modal Normal distribution?* All aspects of student behaviour and achievement should be characterised by a broadly Normal curve that should extend over a reasonable range. One might expect a certain amount of skew where (for example) there are a number of students with similar profiles and expectations towards one end of the spectrum. A rapid decline in participation or discontinuities in the achievement curve would indicate that at least one section of the cohort felt disenfranchised and had become dissociated from the unit.

3. *Is there an internal consistency in the data gathered?* The continuation and participation rates should broadly align. Both are inter-linked with the level of achievement in coursework and examination. One would always wish there to be a reasonably high correlation between coursework and examination achievement, even where (as in this case) the activities are quite different. If such a correlation exists, it would indicate that a similar level of intellectual capability and involvement permeates the delivery and assessment. One would also expect high positive correlations between coursework assessment components.

4. *Are performance and progression (largely) independent of prior experience*? (This follows from the recruitment policy for the programme, rather than the Programming unit itself). The nature of computing and information technology education at primary and secondary level is such that no assumptions can be made regarding students' prior experience or expertise. Students with minimal or no prior experience of computing should still be able to progress. Ideally, progression of those with no experience should match those students with some, or even perhaps considerable prior computing experience.

5. *Is there a good level of student achievement*? It is not enough that students' progress – they must feel able and confident to express themselves and to explore their interests within the subject. In terms of programming, this can be seen in the use of more advanced concepts, and in self-authored applications.

Between cohorts one would wish to have the following questions answered:

1. *Is there evidence of evolution?* No subject remains stationary, which applies equally to the pedagogy and the domain. There should be developments in the approach taken in delivery and assessment, and in the topics being covered. These changes should recognisably flow between cohorts wherever possible, so that each builds on a firm foundation.

2. *Is there evidence of an improvement in student engagement?* Each cohort can be assumed to be independent from the previous one, but the learning environment should be enriched by the experiences of those previous cohorts. Where the learning environment (including staffing and resources) remains relatively stable, one would expect to see small improvements year on year.

3. *Is there consistency between cohorts?* This is more focused on defensive aspects. A similar level of effort, application and capability shown by a student should lead to a comparable reward irrespective of the cohort. The idea here is that, if the learning environment has improved, then it is reasonable to expect a similar increase in the level of student attainment.

The next issue impinging on the design of the research method was the extent to which these questions could be answered. There were two significant areas demanding the attention of the author/researcher: supporting pedagogical change and conducting the research. As the two were inter-dependent, it was decided that the ongoing data analysis would be conducted to a level that could be used to monitor and inform the pedagogical changes. Moreover, the analyses would be segregated according to the facets of the study. The aggregation of these individual views would provide a comprehensive statement on the success or otherwise of the paradigm shift in the pedagogy.

3.3.3. Overview of Research Methods

There are many facets to a research study: the research methodology; ethical considerations; resource availability; the skills available to the research team; the availability of participants; other research in the field; and the characteristics of the

field itself. These factors form a network of influence, with the hegemony lying with the intent of the research, embodied in the beliefs of the research team (Maxwell, 1998:88). The result is that the design of the research programme will necessarily be an iterative process. This can continue after the start of the research, as opportunities and issues manifest themselves. As resources were limited (especially with regard to the size of the research team), the data collection regime was considered to be the dominant component.

At face value, including the beliefs of the research team would appear to lay the research open to accusations of bias; that the research was more concerned with supporting the beliefs of the researchers, as opposed to adding to the body of knowledge in a rigorous manner. The results of the research may be considered to be unreliable or invalid, in that another group of researchers might not be able to produce similar results, and the relationship between the data captured and data representative of the domain might be too tenuous to convince the wider academic public.

Where that which is being studied has also been designed, delivered and assessed by the researcher(s), the potential for bias is considerably increased. In the context of this study, it was not possible to commission the design of the research programme, neither was it considered desirable to utilise the existing pedagogical regime. The design of the research programme had to be cognisant of the extent of potential bias, and to make strenuous efforts to at least limit, if not mitigate the effects wherever possible.

One important consideration was the extent to which the research (as distinct from the pedagogy) should impact upon the students' experience. The temptation was to focus mainly on gathering quantitative data arising naturally from the learning process: using assessments and online activities as the main source of primary data. Such an approach could be seen as non-invasive. The main limitation would be that this would restrict the level of data associated with the experience, rather than the effects (phenomena).

3.3.4. Philosophical Influences

"Education implies that something worthwhile is being or has been intentionally transmitted in a morally acceptable manner" (Peters, 1966:25). This is quite a broad definition, especially if one delves into the various meanings of 'intention'. This is borne out by Ryle, quoted in Peters (1967:1) who asserts, "the logical geography of concepts in the act of education have not been mapped". One might imagine one approach to defining education is to look through instances of education and attempt to abstract general principles from them. Or to "formulate a definition of 'education' and to see whether this would fit all examples of it" (Peters, 1966:23). Either approach, Peters argues, would run counter to that proposed by Wittgenstein (1953, cited in Peters, 1966:24) in his later writings where he writes that games are a "family of words united by a complicated network of similarities overlapping and criss-crossing; sometimes overall similarities, sometimes of detail". If one substitutes 'education' for 'games', one can see that Wittgenstein suggests that there is no one characteristic or group of characteristics which are shared by all examples of education.

Peters' definition does not imply that all worthwhile things are necessarily education, but it does separate learning from education, in that the moral and intentional dimensions are not logically necessary for learning. Education is a special case of what Ryle termed an 'achievement' word, although Scheffler (1960:11) widens the concept by adding 'trying' as a part of the educational process.

If research results are to have any validity beyond the immediate study, there has to be some concept of the separation of the 'local' and the 'general'. These two elements have been characterised as 'subjective' and 'objective', the implication being that only the objective can be translated or projected into a different scenario. Many scientists strive for objectivity in the design of their experimental frameworks, harking back to the scientific method associated with the Age of Enlightenment. This positivism (as characterised by Tiryakian, 1978:23) is based on the belief of the existence of a world that is mapped by a set of invariant, objective relationships, which can be represented in a precise manner. Logical Positivism suggests that the precision can be realised using mathematics.

Karl Popper challenged logical positivism, suggesting that verification through formal argument was not enough to ensure that an endeavour was 'scientific'. He introduced the principle of falsification (Magee, 1973:35-55) that contends that only that which can be falsified can be considered science. And falsification depends on the existence of objective knowledge. However, because Popper recognised the possibility of errors in methodology, it follows that there are no verified scientific theories, just ones that have been “corroborated”. Popper did not see 'science' as exclusive, urging researchers to be ambitious in seeking to apply falsification to a variety of theories. There was also a social dimension to his work: seeing the objective knowledge as an agent of democracy, in a manner that echoed the views of Dewey.

Thomas Kuhn was a critic of Popper who suggested that an equally important concern was the nature of scientific progress, which he characterised as being largely evolutionary, but occasionally revolutionary; the latter of which he saw as shifts in the paradigm. Later theories explain phenomena more fully than previous ones, but that does not mean that theories can be directly compared. This concept of 'uncommensurability' opened up Kuhn to accusations of relativism, which he rejected on the grounds that each theory was being measured. It is interesting to note that Stickney (2006:327-9) suggests that Kuhn's views (which he terms relativist) are being used to justify changes in educational organisation and funding. The idea being that one cannot directly compare approaches, one judges each against the phenomena of (presumably) grades and rates of completion.

3.3.5. Subjectivism

Phenomenology, hermeneutics and Critical Theory are branches of philosophy that are generally seen as being subjective. Phenomenology is "the intuitive exploration and faithful description of the phenomena within the context of the world of lived experience (*Lebeswelt*), anxious to avoid reductionist oversimplification and overcomplications by preconceived theoretical patterns" (Speigelberg, 1967:257). Edmund Husserl is generally acknowledged to be the "center" of phenomenology (Ricoeur, 1967:3), but Husserl himself stated that it was Brentano's development of "intentionality into a descriptive concept of psychology" (Husserl, 1931, quoted in Chisholm, 1967:1) that began the philosophical movement of phenomenology. This

movement can be described as more of a method than a doctrine, as it includes the "pure" phenomenology of Husserl's early writings, and the existential phenomenology of (among others) Heidegger, Sartre, and Merleau-Ponty.

Hegel considered phenomenology to be a "thorough inspection of all the varieties of human experience … [including] ethical, political, religious, aesthetic and everyday experience" (Ricoeur, 1967:3). This is potentially introspective and self-referential, which led to the American Pragmatists rejecting Hegel's views and approach. Husserl took a different approach (Husserl, 1973), focusing (in the earlier works) on the pursuit of “pure” phenomena, with the idea of then being able to aggregate these objective components into 'real' scientific theories and principles. He separated aspects of phenomena into *noesis* (the use as intended in a given situation) and *noema* (all intended uses of the object) (Gurwitsch, 1967:47), partly to resolve difficulties with locating “pure” phenomena.

Philosophers have often accused the social sciences of lack of (not only) rigour, but the intention to be rigorous (e.g., Popper's encouragement to social scientists to embrace falsification). Merleau-Ponty, a prominent phenomenologist, is an exception, being described by Tiryakian (1978:21) as being the “critical bridge between existential philosophy and the social structure of intersubjectivity. Edith Stein (a student of Husserl) outlined the notion of empathy as central to intersubjectivity: the experiencing of another as an individual, not as an object within other objects (Mooney and Moran, 2002).

Critical Theory grew out of the ill-defined critical theory espoused by the Frankfurt School, and has a "cognitive dependence on a 'pre-theoretical instance', an existing social interest in emancipation which it seeks to articulate" (Honneth, 1999:320). Critical Theory is characterised as being "empirical without being reducible to empirical and analytical science; it is philosophical but in the sense of critique and not first of philosophy; it is historical without being historicist; and it is practical, not in the sense of possessing a technological potential, but in the sense of being oriented to enlightenment and emancipation" (McCarthy, 1978:126).

The postmodern era has seen two variations of critical theory, according to

Brunkhorst (2009:94): fundamentalist, as epitomised by followers of Hedeigger, and anti-fundamentalist, epitomised by Rorty. Adorno, one of the leading lights in the pre-war Frankfurt School, is suggested as aligning more with the anti-fundamentalists, in the sense that he presaged sentiments which Rorty has expressed, such as "the freedom of the plural is not endangered by 'science and naturalistic philosophy' but by 'scarcity of food … and the secret police'" (Rorty, PMN, 1978:389). Such sentiments do accord with the Left Hegelian and Marxist view of the emancipatory aspiration of people. Where there is a difference is that Adorno saw this emancipatory aspiration as applying only to the proletariat; an aspiration that could only be facilitated by collective opposition to capital. This has been redefined by Jürgen Habermas to focus on the paradigm of communication in his Theory of Communicative Action (Russill, 2005:285), rather than the paradigm of production (Honneth, 1999:327). This re-drafting of Critical Theory has become the dominant philosophy in modern Germany, although Habermas is not without his critics (Heinrich, 1999).

3.3.6. Application to the Study

The 'situational' and 'cultural' aspects of the intended research would tend to suggest links with the pragmatism of John Dewey. His philosophical roots lay in the American Pragmatism of William James and Charles (C.S.) Pierce, sharing with them the Hegelian rejection of Cartesian dualism. Unlike James and Pierce, and their followers, including C.I. Lewis, Dewey developed a social and biological dimension to his view of pragmatism (Peters, 1977:104). Education as an instrument of 'democracy' is a theme of much of his writing. Dewey went as far as to suggest (1916, republished 1966:328) “philosophy may be defined as 'the general theory of education'”.

Dewey's views heavily influenced the design of the American education system. In the 1920s he proposed a different agenda from the prevailing individualistic view of education favoured by other pragmatists. His 'constructivism' had five elements, each relating to the mutually inter-twined concepts of society, culture, democracy, and education. For Dewey, a key aspect of democracy is fluidity: the absence of social, political, or educational barriers.

One accusation levelled at pragmatic research is that the acceptance of the integral nature of the situational and cultural factors necessarily limits the extent to which conclusions can be applied in differing environments.

It may seem that subjectivity may not be relevant where a large volume of quantitative data is to be analysed, as in this research study. The relevance of a consideration of subjectivism in the context of this study is to explore the notions that the quantitative measures may be subjective, and that the inferences drawn from the analysis almost certainly will be subjective. Suppose a student selects the correct option in a multiple-choice test, or correctly identifies the specific lines of a program that realise a given programming concept. The quantitative measures, whilst precise, will tell only part of the story. Treating them as definitive or complete requires a subjective judgement, as does viewing the researcher is an independent observer.

One alternative considered was to use a variation of action research, where teams of individuals collaborate to solve problems. This can have a strongly social context, but this is not necessary. The essential element is continuous data gathering and analysis, rather than external sampling by an individual researcher or researchers, or purely self-reflection. An important consideration in action research is the notion of a solution. Who decides what a solution is, how can it be measured; is it static or can it change?

In this study the researcher is totally immersed in the scenario, designing the delivery and assessment, and being mainly responsible for both. There was continuing contact with the participants, most of which was concerned with their learning. The researcher was not to be in the role of participant, neither would the participants be overtly involved in solving the problem of designing an improved unit. The latter was considered unethical, as it may have deflected the students from their main purpose, that of learning about programming and hence passing the unit.

Given those caveats, it was still possible to consider the research as following the action research paradigm. Data would be continuously available and analysed frequently, via the assessments and actions of the students. Minor modifications to the delivery and assessment could be made in response to particular circumstances,

with more significant changes being applied to the regimes for the following cohort. The students' input to the problem solving process would necessarily be indirect, as their engagement and achievement (or lack thereof) would be significant. This would not deviate from the tenets of action research, provided the students were made aware of the overall enterprise. It would also be the case that within the learning process, there may be time for the students to provide more direct input into realising the objective of a more effective introduction to programming.

The participants in this study would not be (directly) helping to solve the problem of facilitating a paradigm shift in pedagogical style. This was an important consideration, and one that indicated that the research study could not be characterised as classic action research. Also, although the participants could be viewed as equals, and they would be constructing a portfolio of work during the academic year, they would not be asked to reflect on this portfolio: deflecting the participants' efforts away from studying programming would not be ethical in the context of an environment where the participants' performance was being evaluated.

In the context of the population under consideration, it was concluded that the most suitable technique would be a sequence of case studies each one focused around all of the activities of a given cohort. This conclusion was rationalised in terms of:

1. It was not relevant, practical, desirable or ethical to select any group within a cohort. This would apply to the following potential groupings: prior experience, gender, ethnicity, and age. Selecting (say) those with no prior experience would have been a haphazard process, given the lack of definitive and verifiable data. There would also have been ethical considerations arising from such a choice, given that the researcher was also responsible for the assessment and marking.

2. The planned phased evolution of the curriculum necessarily involved multiple cohorts.

3. Multiple cohorts would provide opportunities to evaluate the selection, application and calibration of instruments.

4. Within practical limits, it was not considered valid to select a subset of activities for inclusion in the case study. Focusing on one aspect may lead to ignoring positive or negative effects on other facets of the unit.

Case study research is a well established, although sometimes controversial group of research techniques. Each technique involves the identification of a delineated set of activities involving a discrete group of participating individuals. These activities are observed using instruments and various data points recorded. The controversy often focuses on whether such an approach can be scientific and manifests itself in terms of one of more of the following accusations:

a. *Practical knowledge is seen to be of less value compared with theoretical knowledge*. The highly contextualised nature of a case study implies that any theoretical data is compromised by contextual data, rendering the results meaningless.

b. *Case studies are useful for generating hypotheses, rather than rigorously testing them*. Case studies involve observation (ideally general rather than selective) that lends itself to analyses that yield patterns that can be used to generate theories. This grounding of the theory in the data is a common form of case study research.

c. *It is difficult to generalise from a single case study*. Again the unresolved dichotomy between specific case-related data and more applicable domain data may compromise any extrapolation to, or comparison with another case.

d. *There is a danger of unrecorded and unverifiable bias in the data gathered*. No case study can record all data points, and those that are recorded may be subject to error or bias. Other researchers (and the researchers themselves) may be unable to spot or assess bias in selection and recording. This would impact the validity of the data.

e. *Summarising a case study may be problematic*. Summarising implies the application selection and abstraction, raising the import of certain data above

others, and grouping data around and into a higher-level conceptual framework.

Underpinning these concerns is that there must be an element of indiscrimination in the observation: the researcher wishes to capture the richness of the activities and so cannot be too selective in gathering the data. It may be that instruments need to be modified (or even introduced) as the case study evolves. Furthermore, as stated above, no data gathering can be comprehensive, and it may not be possible for a researcher (often immersed in the case) to be aware of the subtleties of bias. Alternatively, the researcher may embrace the bias, or neglect to consider it, in order not to alienate himself/herself from the participants. The idea being that the participants will behave more naturally over the extended period of the case study if the researcher is seen as part of the case study itself.

Where the researcher moves between observer and agent there are additional challenges to the notion of case study as a rigorous research approach. Such participant action research intentionally uses the bias of the researcher to influence and/or enact change within the activities under consideration.

Flyvbjerg (2006) has systematically refuted each of the accusations above, and pointed out that examples are an essential element in social science research. The key is a set of good examples, well executed.

Having established that case study research was a potentially acceptable research approach, consideration moved to the selection of specific technique and the generation of a rigorous research method. The research was planned during the delivery of Cohort B. The retention of the programming assignments (portfolios) from Cohort A meant that quantitative analyses of all portfolios for all students across the four cohorts would be possible. Given the volume and number of this data, it was decided that the resources required to produce the analyses would limit the number and diversity of other instruments.

3.3.7. Case Study Research

Robert Yin has published widely on case study research, covering research methods

and sample cases (Yin, 1998; Yin, 2003; Yin, 2004). From an analysis of this and other literature (e.g., Stake, 1995; Cavaye, 1996; Huet *et al*, 2004), three of the factors that needed to be elucidated were identified (Kennedy, 1979:664-6): attributes (those aspects to be included), the functions (the actions involved), and the outcomes (ephemera). In terms of this research study, the attributes were the tests, surveys, and programs; the functions were the analyses to be performed; and the outcomes should answer the research questions. These aspects of the research should be applied (where possible) to each of the cohorts. As Kennedy (1979:674) indicates, one can only draw tentative conclusions when generalising over a number of case studies, as there may be a number of differences between the attributes, functions and outcomes in each case study. Whilst one may assume that the makeup of each cohort will be similar, this cannot be controlled, and one must assume a certain level of variety. Even where a unit has been delivered a number of times, there may be changes in aspects of the delivery, such as staffing, or the use of different assignments and test questions. The outcomes should bear a level of comparison, but one can readily see that the more detailed the analyses, the greater the likelihood of false positives or false negatives – the association of non-causal factors, or the dissociation of causal influences.

It was therefore decided to limit the depth of the statistical analyses to the common measures of population and distribution, with a small amount of regression analysis where appropriate.

It was decided that analysing all the test and survey responses, and every line of submitted code would provide a reasonable basis on which to compare each of the cohorts.

3.3.8. The Selection of the Research Method

The main objective of the research was to monitor the progress towards a change in pedagogical orientation over a number of cohorts. The students are crucial components in this progress, but the nature of their involvement had to be defined. All students had to be willing to actively engage with the pedagogy otherwise learning could not take place, or would be piecemeal. The question then remained as to whether (and to what extent) the students could influence the pedagogy either

within the presentation itself, or in the design of the pedagogy for the following cohort.

The pedagogy allowed for (and encouraged) students to create their own applications, thus one dimension of active involvement was already catered for. Extending this into influencing the pedagogical design was considered, but rejected, mainly on ethical grounds. If a group of students suggested a change of pedagogy, it would be highly likely that this may have a negative impact on other students, and there would be the chance that there would be other unforeseen consequences.

There are characteristics of the research that have elements of grounded theory, action research, and case study. The intention was that the recommendations should be grounded in data; the students would need to be actively involved in the pedagogy; and each cohort could be considered to be a case study.

Whilst one goal was for the recommendations to be grounded in analytical data, it was felt that the lack of control over the recruitment, staffing and other resources would limit the extent to which any derived or inferred theory could be considered reliable or capable of projection into the future, or into other HEIs..

Participant action theory was not considered appropriate for this study, as the students' participation was necessarily restricted to engaging in the learning process and commenting on it. In classic participative action the participants influence the activity, whereas here the development of the pedagogy was determined prior to the commencement of the study, as were the main mechanisms of pedagogical change and refinement. The research was intended to observe and monitor the changing pedagogical landscape, not shape it.

The attitudes of the students to their learning experience would play an important role in shaping any modifications for future presentations of the unit. There would therefore be an element of action research. It was felt that, although modifications to enhance the experience or correct deficiencies would be applied, these would take the form of exceptions: the intention would be that the progression towards a more creative orientation would continue (unless compelling reasons appeared). Therefore

the research method could not be classified as action research.

Any decisions regarding the nature of the progression or the modifications would be grounded in analytical data – mainly quantitative given the desire to minimise any invasion of the pedagogy. There are elements of grounded theory in the research, but the end result was not intended to be a new theory of the teaching of programming. The justification was that the study, although extending over four cohorts, applied only to one programme in one HEI. The limited timeframe, influences of other units and the ethos of the overall framework combined to limit the extent to which the inferences drawn from the data could be termed a theory.

The pre-eminence of observation tended to suggest a case study approach. One aspect of the observation would be linked to the success (or otherwise) of the proposed learning experience. There would be positivist elements of this observation. By analysing all ephemera, another facet would be explored: that of observing anything of interest. At the outset of the research the existence of this 'other' category could not be verified, neither could its nature be ascertained. This combination of positivist and 'neutral' observation suggested that the research would (most appropriately) be termed a 'case study'.

Two options presented themselves – separate case studies (one per cohort), or an extended case study.

Each cohort naturally forms a case study: a single group of individuals presented with the same learning experience and assessment regime. Moreover, each cohort would need to be analysed in order to judge progress towards the eventual goal and to identify issues that would have to be addressed.

The overall intention of the research was to measure the progress towards a more creative learning environment. Hence a more appropriate classification of the research method was to consider is as a single (extended) case study comprising four elements, each of which could be considered to be a case study constrained by the considerations of the overall, extended case study.

3.3.9. Implications of the Selected Research Method

Where multiple elements comprise an extended case study there is a tension between the desires to be consistent between the elements (cohorts), yet be flexible enough to recognise the need to contextualise the research to some degree.

The decision was made to use a selection of measures that would be applied consistently in the analysis of each cohort and other measures that would be specific to particular cohorts. For instance, all cohorts will be involved in writing programs, whereas only a subset may use a particular software tool.

The most important consideration was the establishment of a rigorous, comprehensive, non-invasive quantitative evaluation framework to facilitate analysis of student learning, at least in terms of ephemera, such as programs, tests and examinations. Accordingly, software was written to enable capture of relevant data, although this was not fully functional for the tests until the second cohort. All programs submitted for assessment were captured as a normal part of the assessment process, as were the questions and answers to all tests (after Cohort A) and examination results for all the first year units.

As the researcher also assessed the students, and was therefore highly influential in their progression to the next year of their studies, it was considered that this evaluation framework would comprise the main set of instruments to be used. This use of quantitative data somewhat separates this case study from many of the others, particularly those not associated with education.

The use of qualitative techniques was limited by further considerations: of practicality and veracity. The central role of the researcher would potentially influence the participants' responses to qualitative methods. It was not considered practical to (say) interview all the students. Nonetheless, a decision was made to experiment with some qualitative instruments, both to gain relevant data, and to evaluate their efficacy, suitability and practicality.

The quantitative data associated with the programs (portfolios) and tests, and the smaller quantity of qualitative data would provide differing views of the context of

the study. It was decided to augment these with questionnaire-based surveys before and during the delivery of the unit. Terminal surveys on all units were conducted by the central administration.

Gathering data from a large number of participants in a short space of time provides a challenge for the researcher. The time-honoured method has been to use a questionnaire, providing either Lickert scale responses or categories of responses. Online or paper-based delivery methods could be used, with completion either by the participant or a researcher. With questionnaires, one cannot know the motivation of the participant, so the quality of the data collected may be quite suspect. Participants may try to confuse or become defensive, or they may simply provide random answers.

Despite these misgivings regarding questionnaires, it was decided that these provided the most suitable means of gathering some quantitative data (especially relating to prior experience) and a limited amount of more qualitative data relating to attitude and motivation. It was felt that by mixing up the question types (1 of many, many of many, limited Lickert and multiple-choice) and providing some questions with many potential answers, students would find the survey interesting enough to take seriously. With online surveys it was possible to measure the times involved, and these seemed to bear out this interpretation.

The initial survey attempted to collect two types of data: the prior experience of programming and the initial motivation towards are programming unit. Much of the information to be gathered concerned the former, but it was recognised that particular care would be needed with the subjective questions. It was decided early on that the overall time for completion should not be longer than 15 minutes, thus the number of questions should not be more than 25, as each answer would require some level of thought.

Two dimensions of prior experience were the focus for most of the questions: the width of experience, and the depth. Width could be expressed in terms of the numbers of different technologies (e.g., programming languages) a student had used. The complexity of constructed applications equated to a notion of depth.

It was decided to keep the motivation and expectations questions to a minimum, and to offer simple alternatives, as the intention was to collect a sense of immediacy in the students' reactions to these questions: it is not implausible to suggest that new undergraduates may not have considered individual units to any significant degree.

By extending the study to include four successive cohorts it was hoped that the evaluation framework and the surveys would provide comparably reliable and complete data for each cohort. The extent of this comparability was not considered sufficient to warrant or justify any adjustment of the data for any cohort, therefore constraining the comparison to the main elements of the secondary data. Further analysis beyond the standard measures was not considered appropriate.

3.3.10. Ethical Considerations

All research should be ethical. There are complexities inherent in the semantics and interpretation of 'ethics' that necessitate codification, to assist researchers and assure both participants and commentators. These codes of ethics, typically drawn up by professional bodies, must exist harmoniously with civil and criminal law. A typical code of ethics (Christians, 2005:144-5) will include acquiring informed consent, avoidance of deception, observing privacy and confidentiality, and the need for accuracy. This list is subtly different from that advocated by the Royal College of Nursing (Hale, 1998), which also includes beneficence (trying to do good), non-maleficence (avoidance of doing harm), and justice (fairness to participants and others), in addition to informed consent and confidentiality. Reaves (1992:52-55) also points out the need to avoid coercion, deception, and the fabrication of data. The British Psychological Society (2005) also includes professional competence in its list, recognising the moral and legal obligation shared by all professionals.

These general principles are often complex to implement. One of the difficulties is that the relationship between researchers and participants is inherently paternalistic, as the researcher has the balance of the power and the knowledge (Hale, 1998:4). There may be skills that the researcher possesses that cannot be understood by the participants (for example, in medical research). This imbalance will continue through the research activity, so the principle of informed consent, as with the other principles, is continuous in nature, not discrete. Participants must be free to leave the

research programme at any time. Another factor, identified by Punch (1994, cited in Christians, 2005:144) is that informing participants may "kill the project stone dead". Ethical researchers must accept the abandonment (or modification) of the research in such cases.

There is a delicate balance between the assumption and the reality of researchers doing ‘good’ (Hale, 1998:7-8). Research may not produce a positive result, or it may be difficult to translate the theory into practice. Additionally, the intent of the research may be to educate the researchers, or simply explore a complex or novel area. Participation in research may induce unwarranted optimism in participants, and the feeling of exclusion on the part of non-participants. This might be exacerbated if the research violates the principle of fairness, by providing participants with preferential access to resources. A typical example is clinical trials for new drugs. Apart from the access to the new drugs (which might be a questionable benefit), the research timeframe may necessitate participants avoiding queues for routine procedures and tests.

In the context of higher education, this power relationship between researcher/lecturer and participant/student can be more problematic. As Blevins-Knabe (1992:153) indicates:

> *“The power comes from two sources; the professor both evaluates the student and is the authority in the subject matter”*.

Seven questions and four criteria are identified later (Blevins-Knabe, 1992:154-5) which, it is suggested, should be answered and applied when a tutor conducts research with students as participants. As with other ethical frameworks, these questions and criteria assist the researcher in considering all stakeholders and the manner in which they might be affected. These stakeholders include other faculty members, other students, and the wider academic community.

These, and other issues raised by the relevant codes of ethics need to be considered and resolved into an ethical plan for the data collection framework. This plan will normally be part of the research proposal, and may need to be submitted to a

professional panel (informed where appropriate by legal counsel) that will adjudicate on the plan before the research can begin. Part of the plan will identify the procedures that will be used to deal with unexpected events, to assure the panel that ethical considerations will play an active part in the ongoing research.

When designing an ethical plan for research into the decision making in novice programmers one needs to identify the intended benefits (to those learning to program), and the possible causes of physical and emotional harm. Where intrusive measurements are involved (for example, in the use of GSR - Galvanic Skin Response), the ethical plan must identify appropriate procedures for counselling participants, and ensure adequate monitoring for any negative reactions.

Emotional side effects of any research are inevitable (Hale, 1998:8): participating in any activity must have some impact on an individual. It is often difficult to judge these effects, as differences between the psychological characteristics of individuals are generally significant. A particular problem arises from the keenness of participants to provide the researcher with good data. Unless participants are carefully counselled prior to participation, this may induce anxiety if the participant's performance varies from that which he or she feels the researcher requires. Anxiety can also follow from the participant wishing to perform in a way that reflects well on them. For instance, in the research under consideration, a participant may feel that he or she may be considered to be an example of a weaker programmer, so may feel that the research is a kind of examination, rather than an investigation. Again, this must be acknowledged in the prior counselling. It has to be noted, however, that, as is the case in many situations, participants may not be particularly motivated to listen to this counselling.

The need for counselling may not be restricted to the participants - three other groups may be involved. Where more volunteer to participate in the research than the study requires, those rejected need to be made aware of the grounds for their exclusion. In order to prepare the ground for this eventuality, any selection criteria will need to be made clear when the call for volunteers is made. Nonetheless, researchers need to be aware that humans will rationalise, and may assign negative reasons for the exclusion, especially where the published criteria are unclear. The

third group includes those who did not volunteer, but who are aware of the research programme. These individuals need to be counselled as to the likely effects (if any) on their situation. For instance, the programme may require the exclusive use of some facilities at certain times. This must form part of the call for volunteers, but the researchers need to understand that those who do not volunteer may not even have read the information that has been made available. This can have an effect on the programme, if affected individuals complain to higher authorities, the fourth group affected by the research. Unless a suitable level of cooperation and the appropriate permissions are obtained, those in positions of responsibility may renege on agreements, citing inadequate briefing.

The ethical considerations of this study were examined, and the following conclusions reached:

1. The majority of data to be gathered would be the normal ephemera associated with the learning programme.

2. The analyses performed on the gathered data would not be used to influence the level of attainment or the likelihood of progression.

Furthermore, it was decided that:

1. None of the participants would become cognisant of the submissions of others.

2. A participant could voluntarily withdraw from any research activity (i.e., activity designed as part of the research study rather than the learning and assessment regimes).

3. Each participant would be associated with a randomly selected identification number, thus ensuring anonymity for the participants.

After detailed consideration, it was decided that the research method and instruments did not contravene any ethical, moral, or legal guidelines or frameworks.

3.3.11. The Quantitative Measures

One conclusion from the ethical analysis was that non-invasive techniques would be preferable, in order to minimise any potential conflict between the research and the pedagogy. This would indicate that one of the main elements of the research would be quantitative analysis of the ephemera (tests, programming assignments and examinations).

A key decision was whether to use descriptive or inferential statistical measures. As inferences between and within cohorts was an aim of the research, it would seem that inferential statistical measures would be more appropriate. Closer evaluation produced a more complex picture. Whilst there would, undoubtedly, be similar activities within each cohort, there would be differences emanating from the progressive shift towards a more creative learning environment. There might also be changes in student profile, in the mechanisms of delivery (e.g., development environment) and in the delivery team. There would also be the need to be flexible, to allow for each cohort to be treated as a separate entity not completely bound by the overall enterprise.

The basis for direct comparison was therefore deemed insufficient to justify the use of inferential statistical comparison between the cohorts. A nature of the use of descriptive measures was then considered, including mean, median, mode, minimum, maximum, standard deviation and skew.

The rationale for including (or excluding) a measure was based primarily on two factors: appropriateness and reliability. The mode might be an appropriate measure in many situations, but, in the case of the size of student portfolios, the mode may produce a misleading result if a precise measurement of size is used. Rounding of portfolio size may produce a more meaningful and reliable mode, but there may be consequences for the accuracy of the mean and median.

Another consideration was scope: should all data points be included? Each cohort will generally include some students with considerable prior experience of programming. Similarly, a few students may be demotivated at an early stage. Should these outliers be removed? If the intention of the research were to consider

the progress of (say) the non-outliers then the answer would have been ‘yes’. As it was, the intention was to include all students, so no removal of outliers took place.

The mean and median were included as general measures, and as indicators of the shape of the distribution. Ideally, the mean and median would be coincident, indicating a Normal distribution. A difference between the mean and median would indicate some level skew, and the skew value would provide a figure for the difference. The standard deviation would provide some measure of the spread of the distribution, as would the minimum and maximum.

The next consideration was the nature of the comparison between the single population measures. The methods selected were tabular and graphical. As before, more rigorous statistical measures were not used, in case this implied that the nature of the cohorts were of sufficient consistency to convey reliability to such measures. An increase in the mean between cohorts would suggest that the students in the second cohort had responded more enthusiastically to their learning regime. The difference between the figures could not be used to indicate the scale of the change.

3.3.12. The Primary Components to be Measured

Given the number and variety of data emanating from an introductory unit, consideration was given as to what to use as the primary measures of student progress. The application size (in bytes) is a very crude measure of the complexity of an application. Many factors can confound any inference that a larger program will always incorporate more, or more complex, programming concepts. Longer (or shorter) variable names, more or less indentation, more or fewer comments, are obvious factors. Also, one algorithm may be compact but appropriate, whereas another algorithm may contain unnecessary elements.

The decision to use the size of portfolios as a comparative measure was based on the notion of aggregation. It is reasonable to suggest that aggregating the results from a large cohort will tend to diminish the effect of any confounding elements in individual cases. It was decided to use three measures of size: gross size, net size, and the size of comments. The gross size was simply the total of the sizes of the source files in the submitted portfolio. Removing all styling and commenting from

those source files produced the net size. Separating the comments would facilitate a judgement as to the origin of any change in portfolio size.

The number of programming concepts used within a program is also a good indicator of the learner's confidence. Software was used to count the instances of different programming constructs, and descriptive statistical measures could be applied, as in the case of the three variants of application size.

It was decided that these two classes of component (portfolio size and construct frequency) would provide a reasonable basis on which to quantitatively compare within and between cohorts.

3.3.13. Normalising Data

Within a cohort, more direct comparisons might be considered to be more viable than when applied between cohorts. If one wished to compare (say) two successive programming assignments, then it might appear that statistical measures used to compare two populations might be appropriate. The problem is then one of the basis of comparison. The student population might be considered to be comparable, for obvious reasons. However, not all the students submitting the first assignment might submit the second. Also, the nature of the second assignment might vary. For instance, the latter assignment may allow the students more choice, or the required number of assignments might be different.

The desire was to demonstrate change within the profile of student performance; in particular to identify where, within the capability range, any shifts were taking place. Accordingly, a measure, called the CWFDP (Cumulative Weighted Frequency Difference as Percentage) was derived. This is described in more detail in a later section. In summary, only students submitting both assignments were included. The differences between the sizes of portfolios for each student were calculated and frequency distributions were created. These were then expressed as percentages of the total number of included students. Weighting and accumulating these figures was intended to highlight the location of shifts within the student profile.

3.3.14. The Design of the Data Collection Framework

The objective of any data collection is to gather reliable and valid data (Moore *et al.*, 1973:5). Reliability can be measured over time and space: to what extent does data gathered at a different time by another set of researchers differ from the data gathered in this research exercise by this team of researchers? As it may be difficult to control the environment in which the data is gathered, measurements of reliability are relative, rather than absolute. Data collection is considered valid if (and only if) the data collected is what is required. Failure to collect relevant data, or contaminating the data gathered with irrelevant or incorrect data, reduce the validity of the data collected.

Quantitative approaches seek to ensure reliability by placing the researcher as an impartial observer outside the experimental environment. The researcher will transcribe the data as readings from carefully calibrated equipment. In quantitative research the results are validated by modifying specific independent variables (causal factors) in a controlled fashion, and by measuring the effects of these changes on dependent variables (observed phenomena), whilst eliminating the influence of other independent variables through rigorous control of the experimental environment. Reliability is more difficult to achieve in a field experiment, and less so in a field study. Where the data has strong contextual influences, it will be the case that reliability can only be achieved at the expense of validity.

In the context of this study, the variables being manipulated are associated with the delivery and assessment schemes, in which the achievement and progression depend.

As the 'quantitative' researcher designs the experimental framework, decides upon the dependent and independent variables, calibrates the equipment, and controls the nature of the changes in the values of the variables, it is clear that, in no sense, is he or she an impartial observer. It is also the case that all human (and animal) participants operate in a single, continuous state. This means that any action or activity will produce changes in behaviour, making it impossible to replicate an experiment precisely or reliably.

Qualitative researchers use two forms of justification. They may believe that

examination of these phenomena is valid because they are causally linked to physical phenomena (i.e., some level of the resolution of a dualist paradox). Alternatively, they believe that the study of the phenomena has validity in and of itself.

Modern qualitative research can be categorised as interpretive, critical, or post-positivist (Guba and Lincoln, 1994:106). The choice of category will have a crucial effect on the data collection techniques employed, as will the nature of the data being collected. Nominal data, the simplest form of data, severely limits the range of data analysis which can be applied, yet this is the most obvious form gathered from many qualitative forms of data collection. Action research, participant observation, focus groups, dialogue journals, archival research, and interviews tend to lead to phenomena from participants that are significantly different in superficial structure and content. Only transformation (Meulman *et al.*, 2004:50-51) and detailed analysis yields data that can be used in stating conclusions. The nature of this analysis must adhere to the best practice within the selected methodology, if the resulting data is to be considered reliable and valid.

There is, then, an aspect of qualitative research that is opportunistic. This is not entirely absent in quantitative approaches, either. Most quantitative datasets will have some level of contamination – i.e., some data that was captured simply as a result of the opportunity presented by the study. Opportunism impinges on a vital aspect of research: that of ethics.

3.3.15. Quantitative Techniques

It is the aim of all research to obtain data that is reliable and valid, and the more controlled the environment, the greater the chances that these qualities will exist in the data collected. It is also desirable that the data can be used to generate meaningful findings, which implies gathering data that is quantifiable, and (hence) capable of manipulation.

A common quantitative technique is the survey questionnaire. Like all methods of data collection, many varieties exist. Alternative answers may be presented in addition to the questions, thus limiting the choice of the participant, and making the data analysis simpler. Many surveys use an ordinal scale (e.g., 1 to 10), or a 5 or 7

point Lickert scale to convert the responses into an ordinal scale. A justification for the Lickert scale is that most people are more comfortable with nominal, rather than ordinal, values. Variations of this include additional answers, including: don't know, or not applicable. These additional answers complicate the data analysis.

The main strength of a questionnaire is that the questions provide answers to specific questions (Fowler 1997:344). This is also one of the weaknesses: the questions may be specific, but are they representative? Also, the wording of the questions may introduce bias, despite the use of a pilot survey.

A key aim of the survey questionnaire is that the researcher should not influence the answers. This is easier when the questionnaire is online, when one has the additional advantage of being able to time how long the participant took to respond.

When considering the realm of the initial teaching of programming, two forms of survey questionnaire seem appropriate: skills, and attitude. The skills questionnaire is effectively a test of the participant’s knowledge and understanding which will provide valuable reference points for the data collected using qualitative techniques.

3.3.16. Qualitative Approaches

There are a number of practical considerations that were relevant to the evaluation of the suitability of using these techniques within this study. The environment of the delivery and assessment of the unit included lectures and workshops. The students had other scheduled sessions throughout the week. The student numbers and the diversity of student timetables prevented the extension of the scheduled times to accommodate evaluative elements pertinent to this study.

Accordingly, it was decided that qualitative data would be confined to short, semi-structured interviews, and surveys linked to attitude and motivation. These would be used for the last two cohorts as part of an exploration of the process of learning to program.

3.3.17. Implications for this Study

The research study aimed to take a pragmatic, interpretive approach to reflect the

progress of the programming unit from a reductionist to a creative perspective. Programming lends itself to the use of quantitative analysis, as much of the activity revolves around the manipulation of electronic data. As such, the programs themselves can be analysed by (yet more) programs.

A substantial volume of quantitative data had been gathered relating to the first two cohorts as a natural consequence of the assessment processes. All students were required to submit their assignments electronically and these had been archived. The continued use of the same submission processes would lead to comparable datasets being gathered from the latter two cohorts. These four datasets, gathered from all the students on each cohort would necessarily form the basis for much of the study. Related data, such as the students' prior academic achievements and their performances on other units, are also available via the institution's own archives and information systems.

One dimension not captured by these measurements is the attitude of the students, particularly at the outset. Students with no prior programming experience may (naturally) be anxious but (hopefully) open-minded. Others with some experience that may have not been entirely positive, may be concerned for different reasons. Finally, those with a great deal of programming experience may be concerned that the unit will not offer sufficient challenge.

A questionnaire had been developed and distributed to the students of Cohort B during their one-week induction programme. Whilst providing only a limited view of student attitudes, more qualitative alternatives were not feasible. There was insufficient time to conduct interviews or focus groups within the time constraints of this one-week induction programme.

The applications submitted by the students and the tests undertaken can be considered to be secondary data, notwithstanding the fact that the author designed, delivered, and assessed the students' performance. The conduct and content of the unit were designed to benefit the students' learning experiences – no requirements of the research study were taken into account. This was true of the research project option in Cohort C. Students were free to opt to be assessed on the design,

implementation and analysis of a research project instead of being assessed via the submission of programs. The same learning outcomes were being assessed, but via another means. The ethical considerations of the research project option are considered in more detail in Appendix C, which also contains more information on the student interviews (which were also confined to Cohort C). The motivation behind the interviews was to gain some insight into the development of individuals as they grappled with programming with a view to improving their learning experiences later on in the delivery of the unit.

No qualitative instruments were used in connection with Cohort D: the amalgamation, the revalidation, the changes in the teaching team, and the increased numbers all combined to imbue a greater focus on the design, delivery and assessment at the expense of the research considerations.

3.3.17.1 The Longitudinal Components of the Study

The main focus for this part of the study was student performance of each of the four cohorts in coursework and examination, and the related factors, such as entry qualifications, performance in other units and continuation statistics.

The intention was that the quantitative data linked to student engagement and performance in assessed work would help to illuminate and measure the efforts being put in place to improve the continuation rate within the cohort. It was also important to demonstrate that the change in underlying pedagogy was feasible, and quantitative data would assist in that respect.

All incoming students undergo a one-week induction programme. Interviewing all the students would not be feasible given the time constraints of the programme. Similarly, representative focus groups could not be identified at such an early stage. A questionnaire was designed and completed by the students as close to the start of the unit as possible.

One potential source of data was considered but not investigated in depth: the results of the institutional student survey. Each year the University conducts a survey that attempts to measure student reaction to their learning experiences. The results are

used both internally and externally. There were two problems with using this data in the context of this study. The format of the survey changed in each of the four years under consideration. Two were paper-based, of which the School conducted one and the other was handled centrally. The two online surveys (used for the first and final cohort) used different questions and delivered results in contrasting forms. In addition, there were changes in the questions asked (and the responses available) each year. In terms of crude figures (the only comparable elements), the satisfaction of the students for the unit rose from 3.2 (out of 5) for the first cohort, to 4.3 for the second, and 4.1 for the third. No single figure was available for the final cohort.

3.3.17.2 The First Cohort: 2005-06 (Cohort A)

Essentially, this cohort (the first with the revised programme structure validated in Summer 2005) experienced a delivery and assessment differing from the previous five cohorts, with the use of Java as the programming language in place of C, the introduction of a portfolio assignment, and the (re)introduction of a terminal, unseen examination.

The instrument specific to this cohort was the performance of the students in the portfolio assignment compared with the performance of the same students in the second, more prescribed assignment.

3.3.17.3 The Second Cohort: 2006-07 (Cohort B)

One pedagogical change introduced for this cohort was an increase in the number of coursework assignments, and the extension of the notion of choice by the students in the composition of the elements (programs) that were to be assessed. The second change was the introduction of a code generator to provide the students with practice with the styling of programs.

A number of changes were also made to the mechanisms associated with the coursework assignments. Online tests replaced in-class written tests (in three of four cases) and students used an online system to submit their four portfolio assignments. Practice online tests were made available, although it was decided not to capture data associated with these. The students were notified of this. The rationale was to reduce

any sense that the students were under scrutiny.

The rubric of the coursework component of the assessment was also modified to enable students to select the best three portfolio marks, in addition to the four test scores. This would also mean that students might not need to complete the final portfolio assignment. The numbers of students electing to submit the final portfolio despite having already achieved a pass grade would shed light on the nature of the continuation of student motivation. Comparisons could not be drawn with the views expressed in the initial questionnaire, as the association between the students and the completed questionnaires was not retained.

3.3.17.4 The Third Cohort: 2007-08 (Cohort C)

The learning and assessment regime for this cohort (and the associated research instruments) remained largely the same as for the previous cohort. The additional element was a pedagogical enhancement to the portfolio assignments: concept realisation. The students were required to explicitly identify where certain concepts were realised within their submitted applications. The code generator was revamped, as the first version was not well received.

Two additional instruments were used, one of which was a modification. The initial survey was no longer anonymous. Students were asked to supply their name, and the programme (within the framework) that they intended to follow.

A qualitative element was added in the form of short interviews with students. These took place in a separate room, within the timeframe set aside for programming workshops. Each week a number of students (who had previously expressed their willingness to participate) were asked to engage in a semi-structured interview. The rationale was that conducting the interviews within the scheduled sessions minimised the extent to which the students needed to reflect on past events. The same questions were used for all the interviews in a particular week.

In the event, the one area that required attention was the code generator. The second iteration had not been any more successful than the first, so that software was completely rewritten.

3.3.17.5 The Fourth Cohort: 2008-09 (Cohort D)

The first part of the delivery and assessment of the revalidated unit mirrored that of the previous two cohorts. The format and content of the examination were also comparable. Another tutor who favoured specified assignments, rather than portfolios managed the latter part of the delivery and assessment. It was decided to include the data from those assignments to facilitate comparisons with the two portfolio assignments.

An online questionnaire was made available to students following the submission of the second portfolio. A technical glitch limited the numbers of responses, but there were sufficient to enable an analysis of student attitudes midway through the unit.

3.4. Measures

A longitudinal study presents an essentially two-dimensional problem: there will be an analysis of a number of elements of each component (cohort) in the study, and there will be multiple components. A decision was needed as to which would become the major dimension and which the minor. It was decided to focus first on each component (cohort), analysing (where appropriate) the same elements. The result was four analyses (appendices A to D). Then each of the elements could be compared across each of the cohorts. That comparative analyses follows in the next section.

The next decision concerned the depth of the analysis of the elements of each cohort. One of the main objectives of the study was to examine the cohorts as a whole. Another objective related to gaining a holistic perspective of each cohort by examining a number of aspects of the delivery and assessment. It was decided that the data gathered would be aggregated for a cohort, and that the conventional measures of mean, median, maximum, minimum, standard deviation and skew would be derived. Visually, frequency histograms would provide the necessary confirmation or denial of the presence or absence of bi-modal distributions.

3.4.1. Comparing the Portfolios

Within a programming unit, it is generally assumed that students will focus more on the assignments than on other learning activities. The particular approach taken within this unit (with the control over the structure of the submissions) facilitated a number of analyses on the submissions well beyond the raw sizes of submissions, to include an examination of the use of programming constructs and styling, as well as gaining an estimated view of the time period involved.

There are essentially three elements to the size of a computer program: the comments, the styling and the instructions. The styling data was more effectively analysed using the results from the application of the additional software. The three measures applied to each portfolio were: total size, net size (i.e., the instructions), and the size of comments.

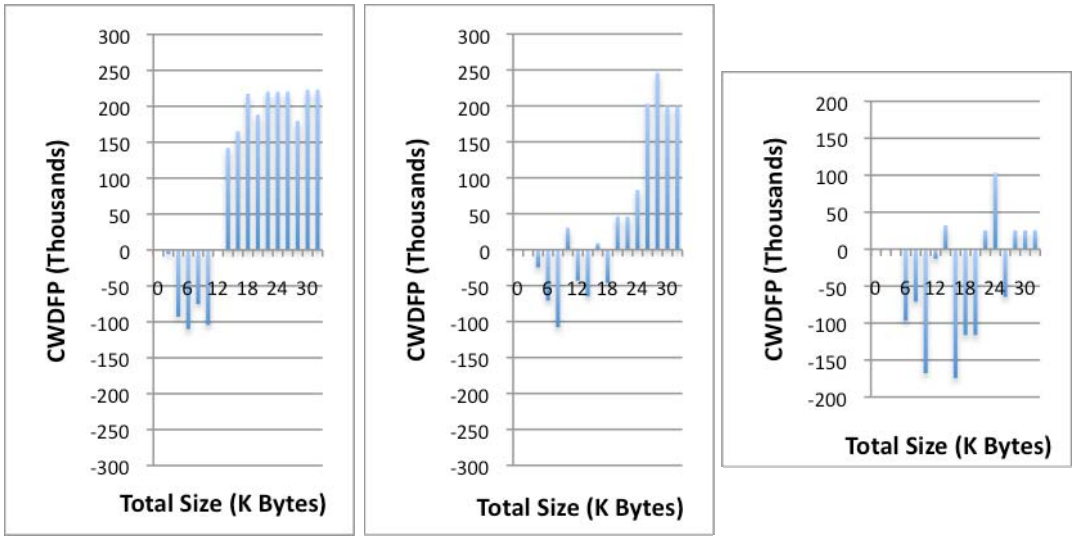
To facilitate the comparison between cohorts it was decided to use ‘frequency as percentage’ rather than the raw frequencies. These would be insufficient to illustrate the changes within a cohort, which might be quite subtle, especially in the case of the portfolio assignments. There were four such assignments in three of the cohorts, spaced only a matter of five or so weeks apart. Although one would expect some progression, the degree of change might be quite small. A measure of comparison was therefore required that highlighted relatively small changes.

The measure identified for this purpose was ‘cumulative weighted frequency difference as a percentage’ (CWFDP). Frequency distributions were created for each assignment, and the differences at each point were derived and then expressed as a percentage of the number of students in the cohort. Weighting each point by the value of that point (i.e., the size of the data point) enables a perspective to be gained as to where changes were most pronounced. Accumulating these weighted values provides an impression both of the degree and nature of the changes that have occurred.

When comparing consecutive portfolios one has an additional consideration as to whether to include the submissions of all students, or only those who submitted both. As an objective was to examine the progression of students, it was decided to

confine the analysis to those students who submitted both assignments.

To illustrate the CWFDP measure, consider the figure below. The distributions relate to the comparisons of consecutive portfolios. On the left, one can see that the cumulative figure starts below zero and ends well above zero, indicating a shift towards larger applications in the second of the two portfolios being examined. Moreover, the lack of change in the area of larger submissions demonstrates that this increase in the sizes of submissions is quite general within the cohort.



Comparison of 1 and 2 *Comparison of 2 and 3* *Comparison of 3 and 4*

Figure 1 – Sample Comparisons of Consecutive Portfolio Assignments

The second distribution illustrates the situation where the increase in focused in relatively few students' work. In the final case, the portfolios are broadly similar in size, with a slight overall increase, but this is concentrated in a few very large applications. This can be inferred because the number and extent of smaller applications towards the origin is balanced only when the portfolios get quite large (24,000 characters equates to around 600 lines of code).

The relative smoothness of the distributions is also indicative that the underlying population is not bi-modal.

3.5. The Evaluation Framework

The main aim of this study is to evaluate the effects of the modifications to the pedagogical approach applied to four successive undergraduate cohorts. Those modifications aimed to increase the extent of student learning by removing potential barriers and providing more feedback. Measuring the success (or otherwise) of these changes therefore centres upon the progression of learning of all students within each cohort. As the modifications extended over the cohorts, one would also anticipate some element of progression between the cohorts.

A framework was created to facilitate this holistic intra- and inter-cohort evaluation. The main components of the framework were:

1. The *modifications* to the unit structure, content, delivery and assessment. Any special (particularly unexpected) events or occurrences were also included.

2. The *entry routes* by which students arrived as new undergraduates. The various routes were grouped into three main pathways, two of which were 'A' and 'AS' levels; BTEC diploma and certificates. The various other routes, including mature and overseas students formed the third group. The level of achievement was not considered relevant to this study because the interest lay in whether there were differences in the patterns of achievement between the three groups.

3. The *student experience* as expressed by the students in the context of the unit. The instruments deemed most appropriate were surveys and interviews.

4. *Continuation statistics* were analysed to evaluate the level of student achievement within the Programming unit, and how this compared with other units in the programme.

5. The *portfolios* submitted by the students as part of the coursework assessment formed the most significant ephemera produced by the students in the context of the unit. These portfolios were subjected to rigorous analysis to gain a detailed picture of student engagement in the construction of programs.

6. The remaining elements of coursework assessment were *online and written tests*. Unlike the portfolio assignments, the elements test results are more amenable to taxonomic analysis.

7. The main elements of the cohort experience were *summarised* and *conclusions* drawn which then formed potential actions relating to the delivery for the succeeding cohort.

4. The Results

The analyses of the four cohorts are contained in appendices A to D respectively. The following is a selection of aspects that cut across the cohorts. The sequence of aspects starts with the entry routes then examines continuation and the student experience, before analysing the portfolios and test results.

In order to facilitate inter-cohort comparisons, the figures used are percentages (of the relevant cohort), except where stated.

4.1. Entry Routes and Initial Surveys

These were analysed to gain a perspective of the academic background, computing experience, and aspirations of the students. The pattern of entry routes by which students arrived on the programme fluctuated through the four cohorts.

	Cohort A	**Cohort B**	**Cohort C**	**Cohort D**
A and AS	51.6%	40.5%	36.5%	45.4%
BTEC	21.9%	20.3%	35.3%	5.1%
Other	23.4%	11.4%	12.9%	33.3%
Unknown	0.0%	10.1%	0.0%	0.5%
Not recorded	3.1%	17.7%	15.3%	15.7%

Table 1 – Summary of Entry Routes for All Cohorts

This can be seen graphically in the following figure:

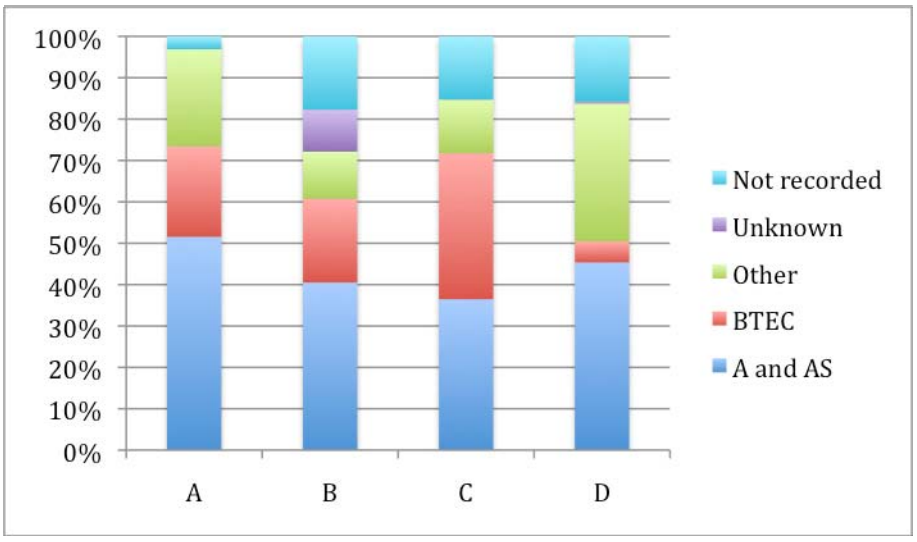


Figure 2 – Summary of Entry Routes for All Cohorts

There were changes in the methods of collecting and collating the entry route statistics, none of which explain the high percentage of students for whom statistics were not recorded. The figures show that the programme has an appeal beyond the traditional 'A' level route into higher education. Those following a BTEC programme will generally have focused more on coursework than their 'A' level counterparts.

Initial surveys were conducted with Cohorts B, C, and D. A large proportion of students completed the surveys. Slightly fewer completed the survey in Cohort B, which was conducted online, in contrast to the other two cohorts, where a paper questionnaire was used.

	Cohort B	**Cohort C**	**Cohort D**
Number	61	76	185
Percentage	77.2%	89.4%	85.6%

Table 2 – Cohorts B, C, D: Summary of Completion Rates for the Initial Survey

Each of the answers to the questions was given a score. With the many-from-many questions, students received a point for each answer. The responses were then

associated with particular characteristics, and the totals for each characteristic were calculated. The prior programming experience was judged against the most experienced in the cohort, in order to facilitate comparison between the cohorts, as well as within a given cohort. This assumed that the students with the most experience in each cohort were comparable, that would appear to be a reasonable assumption.

It is the case that on a computing undergraduate programme, there will be a few students in every cohort who have a level of programming expertise in excess of that derived from formal computing education, generally as a result of personal interest. If one assumes that the most experienced student in each cohort is approximately as experienced as in the other cohorts, it is possible to compare between as well as within cohorts, provided one expresses the experience of each student as a percentage of the most experienced (of that cohort). The table below indicates that each cohort contained many students with limited prior programming experience (relative to the most experienced). One can see that the mode lies around 30% in the case of each cohort.

	Cohort B	**Cohort C**	**Cohort D**
Mean	35.5%	34.3%	32.5%
Median	33.9%	32.6%	31.7%
Std. Dev.	22.2	23.3	19.6

Table 3 – Summary of Professed Prior Programming Experience for the Initial Survey for Cohorts B, C and D

The proportions of students achieving 70% relative to the most experienced individual in that cohort in each cohort were: 6.6%, 7.9%, and 4.3% respectively. Even at the 50% level, the figures were: 27.9% for Cohort B, and 16.1% and 18.9% for Cohorts C and D. One would expect Cohorts C and D to have lower figures, as they included students following IT programmes as well as those following Computing. The percentages for those students following Computing for Cohorts C and D were 22.4% and 31.3% respectively.

4.1.1. Comparing Groups

The initial survey in Cohort C embraced non-Computing students, and Cohort D comprised four groups (Forensic Computing and Security was added to the existing three programmes when the Software Systems Framework was formed). The four cohorts thus included seven groups. The table below compares the prior programming experience of the seven groups using conventional measures of average and distribution. (Note that precise comparisons between the cohorts are not viable, due to the changes in the questions and in the method and timing of the distribution of the survey.)

Group	Mean	Median	Standard Deviation	Skew
Cohort B	35.5%	33.9%	22.24	0.49
Cohort C (Computing)	34.3%	32.6%	23.34	0.47
Cohort C (Others)	27.2%	26.2%	22.02	1.10
Cohort D (Computing)	40.4%	38.6%	20.03	0.49
Cohort D (BIT)	37.2%	37.5%	23.60	0.40
Cohort D (NSM)	49.1%	46.4%	27.93	0.27
Cohort D (FCS)	27.9%	25.0%	22.82	1.94

Table 4 – All Cohorts: Comparison of Prior Programming Experience (Scaled)

The similarity of the median and mean figures, together with the low positive skew, indicates that the modal figure would be slightly lower than the median in each case, emphasising the limited prior programming experience of the cohorts.

The three Computing cohorts exhibit quite similar profiles, with the Network Systems Management (NSM) the only other group having as high a level of prior programming experience. The other groups are noticeably weaker in this respect. It

should be noted that the ‘other’ group in Cohort C included both BIT (Business Information Technology) and NSM students.

The figures for prior web development experience tell a similar story:

Group	Mean	Median	Standard Deviation	Skew
Cohort B	30.3%	27.3%	23.10	0.61
Cohort C (Computing)	43.2%	40.0%	20.99	0.40
Cohort C (Others)	28.0%	28.6%	21.15	0.99
Cohort D (Computing)	34.3%	30.0%	23.28	0.49
Cohort D (BIT)	26.0%	22.2%	23.07	1.42
Cohort D (NSM)	39.2%	28.6%	29.27	0.61
Cohort D (FCS)	27.9%	20.0%	38.73	1.35

Table 5 – All Cohorts: Prior Web Programming Experience (Scaled)

As these figures are relative to the student with the most professed experience in each cohort, and the focus is more on variety of experience than depth, none of these figures could be considered ‘high’, indicating that the students generally have limited experience of programming of any type.

These figures bear out the view that most students entering a computing and IT undergraduate programme have limited prior experience of programming.

4.2. Continuation

The pattern of progression remained relatively stable. Table 6 illustrates the percentages of students considered at the Examination Board and those eventually progressing.

Cohort	Number of Students	Percentage Considered at Examination Board	Progression Percentage
A	64	96.9%	73.4%
B	79	82.3%	70.9%
C	85	87.1%	72.9%
D	216	85.2%	76.9%

Table 6 – Summary of Progression Statistics for All Cohorts

It should be noted that this table does not coincide exactly with the official statistics. The number of students in a cohort is officially derived from those who have paid their fees and remain on the course on a specified date in the academic year in question. The number in the above table derives from those students for whom at least one mark has been recorded in a programming assessment. The official progression rates were consequently higher than those indicated above.

The figures show that around a quarter of those entering the programme fail to progress to the next academic year. 10 to 15 percent of the cohort failed at the examination stage (including the resit diet), which is a cause for concern for the programme team. It should be noted that those students who had manifestly ceased to engage, but who had not officially withdrawn, were included in those considered at the Examination Board for Cohort A, but withdrawn prior to the Examination Board for the subsequent cohorts.

When linking performance to entry route, the measures selected were the overall averages for all students on all units, and the percentage of students who were successful at the first attempt – i.e., who achieved a pass in each of the 11 assessment components (coursework and examination for 5 units, plus the unit assessed entirely by coursework). No clear pattern emerges, as can be seen from the following tables:

	Cohort A	Cohort B	Cohort C	Cohort D
A and AS	55.6%	61.0%	58.8%	60.3%
BTEC	52.9%	57.0%	58.2%	59.3%
Other	49.6%	44.5%	57.6%	60.0%
Unknown	n/a	n/a	n/a	n/a
Not recorded	n/a	n/a	58.8%	56.6%

Table 7 – Summary of Overall Averages by Entry Category for All Cohorts

	Cohort A	Cohort B	Cohort C	Cohort D
A and AS	63.6%	59.4%	64.5%	66.3%
BTEC	42.9%	50.0%	73.3%	54.5%
Other	33.3%	55.6%	45.5%	54.2%
Unknown	0.0%	0.0%	0.0%	0.0%
Not recorded	0.0%	0.0%	53.8%	35.5%

Table 8 – Summary of Students Passing First Time by Entry Category for All Cohorts

The high percentage of students for whom entry qualifications were not recorded in Cohorts C and D means that comparisons between the groups are not reliable. Cohorts A and B show a level of separation between the entry categories, but this disappears for the latter two cohorts. The lack of a clear pattern of differences between entry categories, plus the rather high level of students for whom entry qualifications were not recorded, led to the conclusion that further analysis relating performance to entry route would not be beneficial.

The results for the Programming unit were compared with the other units. The mean

and standard deviation of the unit scores for each cohort were identified, and ranked against the other units.

	Cohort A	**Cohort B**	**Cohort C**	**Cohort D**
Mean	55.5%	61.4%	60.8%	60.5%
Mean Rank	2	1	2	3
Correlation with Overall	0.92	0.86	0.92	0.94
Correlation with Overall Rank	2	4	2	1
Standard Deviation	19.69	16.27	17.33	19.14
Standard Deviation Rank	1	3	2	1

Table 9 – Summary of Programming Results Compared with Overall Performance for All Cohorts

The mean remains relatively consistent across all the cohorts, with student performance in the Programming unit generally being higher than for most other units. As the overall average for a student was calculated as the simple average of the scores for each unit, one would expect a high correlation of all units with the overall average, particularly for those units assessed by both coursework and examination. The high correlation of the unit marks in Programming with the overall marks indicates that the performance of a given student in Programming is similar to his or her performance in the other units. The ranking of the means and the correlations show that the Programming marks were broadly in line with those of the other units. Whilst the standard deviation is generally higher for Programming than for other units, one would expect that of a more technical unit. (A similar pattern emerges for the Databases unit).

The distributions of coursework, examination and unit marks are contained in the

relevant appendices. Each is uni-modal, with minor ‘blips’ in a couple of the examinations. The correlations between the coursework and examinations is given below:

Cohort A	Cohort B	Cohort C	Cohort D
0.695	0.736	0.635	0.620

Table 10 – Summary of Correlations Between Coursework and Examination Marks in Programming

These figures are broadly in line with each other, and indicate that there is a reasonably positive relationship between the marks achieved by individuals in coursework and those same individuals in the examination, despite the very different nature of both types of assessment. One can infer that there was an overlap between the concepts covered in both types of assessment. The conclusion is that Programming can be considered a typical unit within the programme for each of the cohorts.

4.3. The Student Experience

One-to-one semi-structured interviews were conducted with 36.5% (N=31) students in Cohort C. In Cohort D, an online survey was conducted halfway through the unit, and 57.9% (N=125) of the students responded, although technical problems prevented another 40 or so students from registering their responses. It was decided not to include the official student survey in the analysis, as the mechanisms and questions were overhauled entirely for each of the cohorts.

The circumstances in which the interviews were conducted were less than satisfactory, as insufficient staff were available to provide students with assistance whilst the interviews were taking place. This meant that the interviews that were conducted necessarily needed to be short, averaging less than 5 minutes each. More details are available in Appendix C.

The most useful conclusions drawn from the interviews were related to the beneficial

effects of working in one's own time, the continued motivation for most of the students beyond gaining a pass in the coursework, and the negative reactions to the Obfuscator (as the JCodeGen code generator was originally called). The values given to the student responses were highly subjective, and only used to gain some relative perspective on the answers provided. The correlation of 0.76 between the extent of working beyond the scheduled class hours with eventual performance can therefore be seen only as a guide, but the figure confirms the benefits of additional study.

Most students expressed a keenness to continue working on their programming beyond the point where they had amassed sufficient marks for a pass in the coursework. All the students went on to complete one of the two remaining portfolios, although less than half of those interviewed subsequently submitted the final portfolio. This shows that there was an element of strategy in some students thinking, as the rubric for the coursework (of selecting the best 3 portfolios) meant that marks gained in the third portfolio would improve one's mark, but there may only be a marginal effect to be gained from submitting the final assignment.

This is in line with expectations: some students enjoy programming, whilst others are more focused on securing a good grade.

Various factors limited the benefits to be gained from interviewing students. The small numbers was one, but the main one was the ethical dilemma of leaving students without staff support. It was decided that the interviews would be discontinued and a survey used instead. The large numbers of Cohort D were a secondary consideration. More details are available in Cohort D.

In Cohort D, an online survey was associated with the submission and assessment of the second portfolio. In this intermediate survey, most students felt that their motivation and confidence had increased over the first part of the delivery of the unit. Similarly, students felt that their ability to understand program code had increased, although their ability to construct programs was largely unchanged. That construction is the least affected is in line with expectations: the introductory phases will offer few chances for construction – the emphasis will necessarily be on fundamental techniques. The figures below summarise the findings:

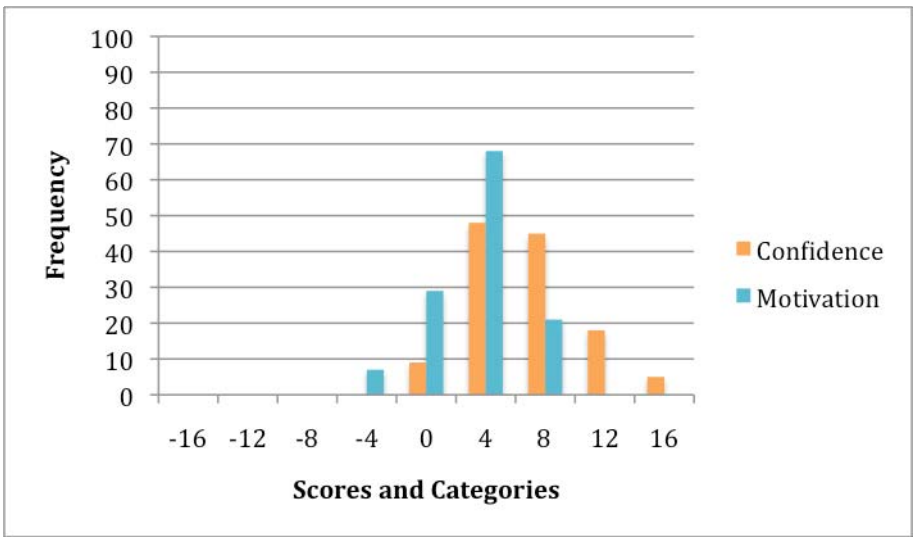


Figure 3 – Cohort D: Expressed Level of Motivation and Confidence

Most of the questions used a 5-point Lickert scale, and the scores represent the aggregation of those scores over all the relevant questions. A score of zero therefore indicates that the students' attitudes have neither become more positive nor more negative during that part of the delivery of the unit. Most students gave answers approximating to 'agree' when asked if their attitude towards an aspect of the unit had improved with regard to confidence and motivation.

A similar pattern emerges for the comprehension of program code, a main objective in the early part of the delivery of the unit. Modifying code requires a higher level of understanding, so one might expect that to remain relatively stationary in this initial period.

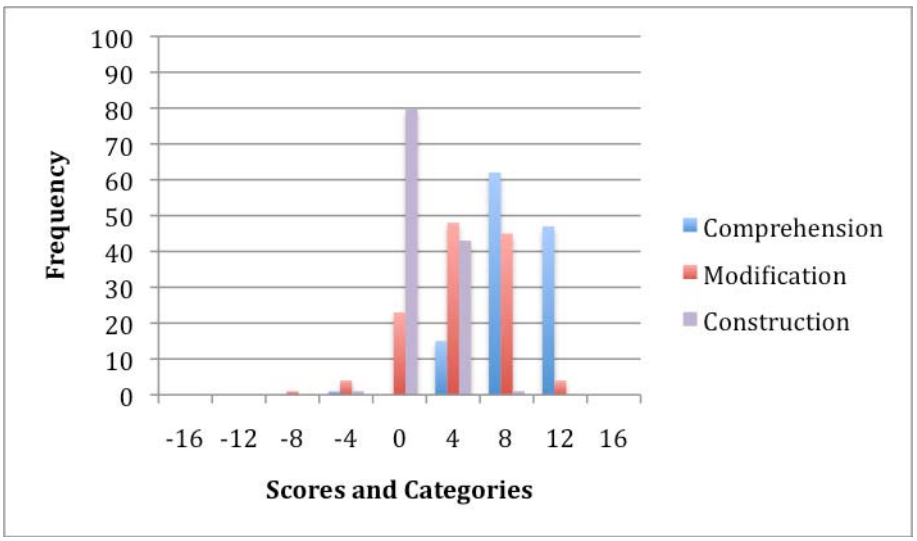


Figure 4 – Cohort D: Expressed Level of Comprehension, Modification, and Construction

The general increase in the level of confidence in modifying code is therefore positive, as is the slight increase in the ability to construct software for some students.

Many of the participants also added comments where the opportunity was presented. 34.4% (N=43) of the students made comments, almost all of which were focused on their own learning. This showed that student's considered their progress as being linked to their own capabilities and efforts, rather than on the tutor or other aspects of the unit delivery, such as computing equipment or support staff.

4.4. Portfolios

4,555 applications submitted in 1,405 portfolios over 14 assignments were analysed. Almost 100 data points were derived from each of the source files in every application (normally one per application). These data points included the use of various programming constructs, types of comments, and the use of the code generator. Together the analysis created a comprehensive view of the work that was submitted and provided some insight into the timeframe within which the work was produced.

A 'real' application is a single program where the logic is divided into a number of

source files, each containing multiple methods. Growing confidence in real programming techniques is therefore demonstrated by both a decline in the average number of applications per portfolio, and an increase in the number of source files per portfolio. The programming constructs analysis (in Figures 7, 9, and 10) shows that the number of methods used increased for each cohort.

Figures 5 and 6 below confirm the growing confidence of students in Cohorts B and C in constructing 'real' applications.

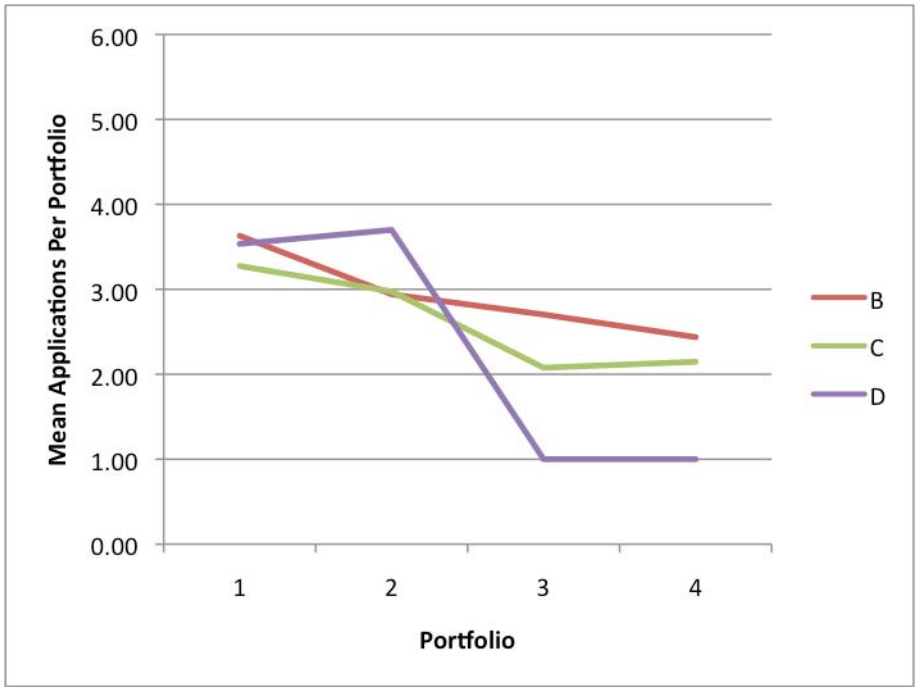


Figure 5 – Cohorts B, C, D: Average Number of Applications per Portfolio

The effects of the constrained nature of the last two assignments in Cohort D are clear in both figures – not only is the number of applications limited to 1 (as one would expect) but most of the submissions contained a single source file. By contrast, Cohorts B and C averaged close to 3 source files per application in Portfolio 4.

For reference, the means for the two portfolios in Cohort A were 5.05 and 1.00 (for applications per portfolio) and 9.56 and 2.71 (for source files per portfolio). This last figure shows that Cohort A managed to sub-divide the logic for the single application, something the students in Cohort D felt unable to do.

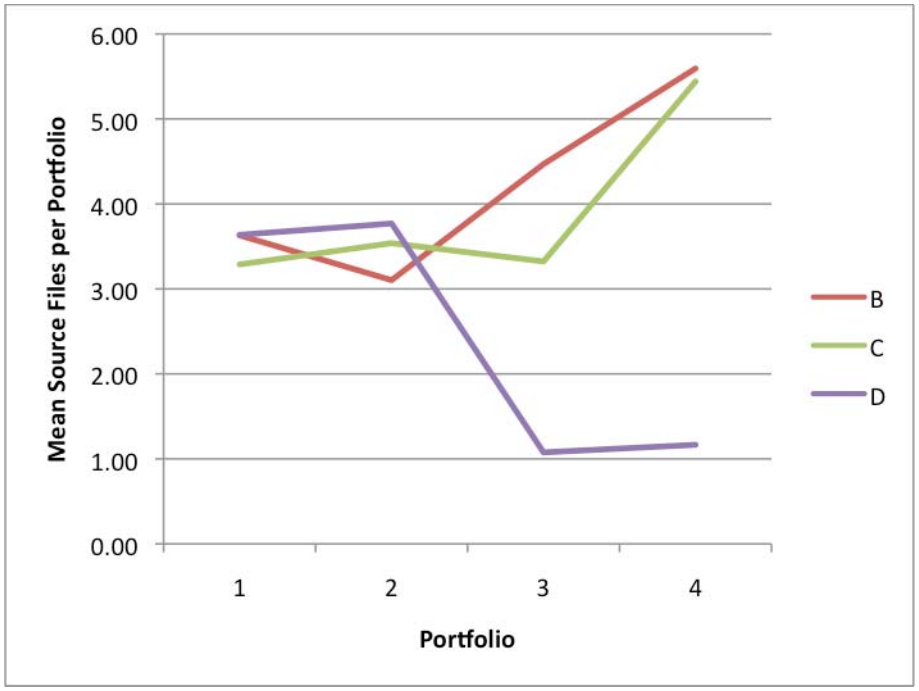


Figure 6 – Cohorts B, C, D: Average Number of Source Files per Portfolio

The level and complexity of the student work increased throughout each cohort, more markedly in cohorts B and C than in the other cohorts. In the latter stages of Cohorts A and D there was a greater focus on functional specification, rather than students creating their own applications around a conceptual specification. In assignments where function is closely defined, the means by which the end result is achieved is given less prominence. In an assignment where a conceptual specification is used, the students need to demonstrate the realisation (use) of a given set of concepts, but may well be free to select the nature of the application or applications to be built.

For illustration, the following figure shows the progression in the use of programming concepts within Cohort C.

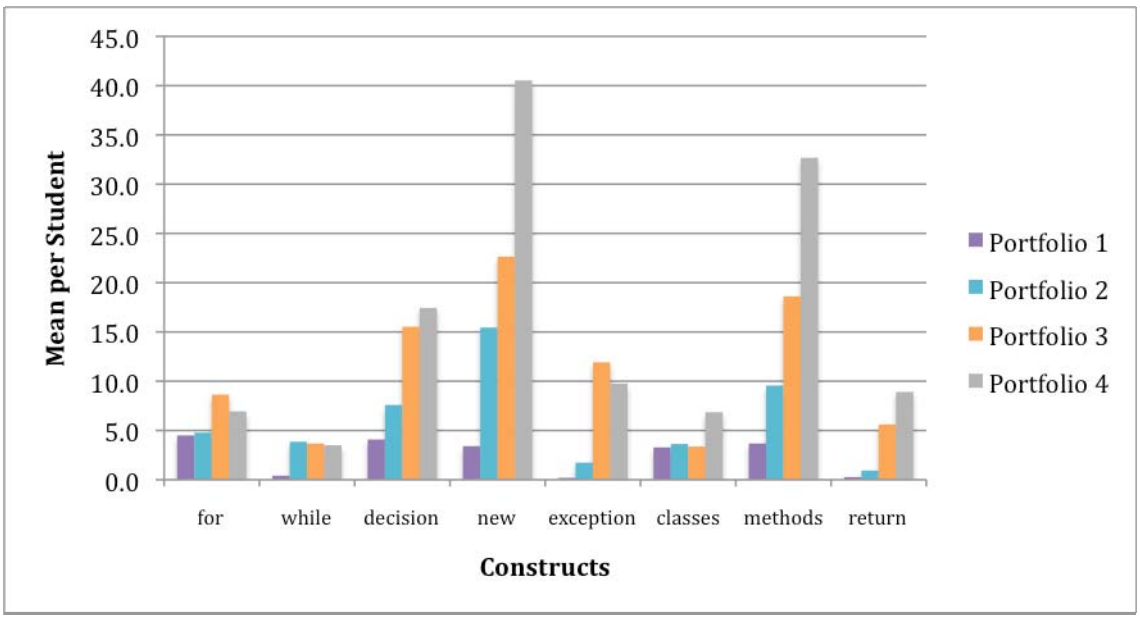


Figure 7 – Cohort C: Use of Programming Constructs in Portfolios

In most cases, the usage steadily grows for each portfolio, most noticeably with the use of 'new' (used to create new objects) and in the number of methods per application. The increase in the use of methods is particularly significant. Partitioning logic into separate elements is an advanced concept, as it requires understanding of the mechanisms of communicating (data) between the partitioned elements, as well as comprehension of the existence of such elements. The use of 'new' indicates that students felt confident to read the online documentation relating to Java and therefore learn of (and then utilise) a wider range of classes.

As applications become larger, the complexity tends to increase inside the loops, with the number of loop constructs increasing to a lesser extent. Similarly, exception (error) handling is mostly connected with input and output, as files may not be present, or permission for a given action may not be forthcoming from the operating system. This explains the increase in the use of exceptions between Portfolios 2 and 3. The figure for Portfolio 4 is very close to that for Portfolio 3, indicating that the focus for development in the latter stages of the unit delivery was an increase in program logic (i.e., other programming constructs).

The other measure of progress is the size of the portfolios submitted. Larger does not necessarily imply more sophistication, but where novices are involved, larger would normally be taken to indicate an increasing confidence and level of understanding.

Three measures of size are appropriate: the overall size of the portfolio as measured by the total size of the source files submitted; the net size (once comments and styling have been removed) and the extent of the use of comments. One also needs to compare successive portfolios in such a manner as to facilitate inter-cohort comparisons. The measure chosen to facilitate this comparison was the cumulative weighted differences in frequencies (in successive portfolios) as a percentage (CWDFP). Frequencies related to the size measures were created, and then expressed as a percentage of the number of students. Differences between the same frequencies in successive portfolios were then calculated, and then multiplied by the relative frequency. By accumulating these figures, it is possible to demonstrate where (within the frequency range) the differences are greatest.

In Cohort A all three measures indicate that students generally submitted smaller portfolios for the second assignment, as compared with the first. This is shown in the following figure:

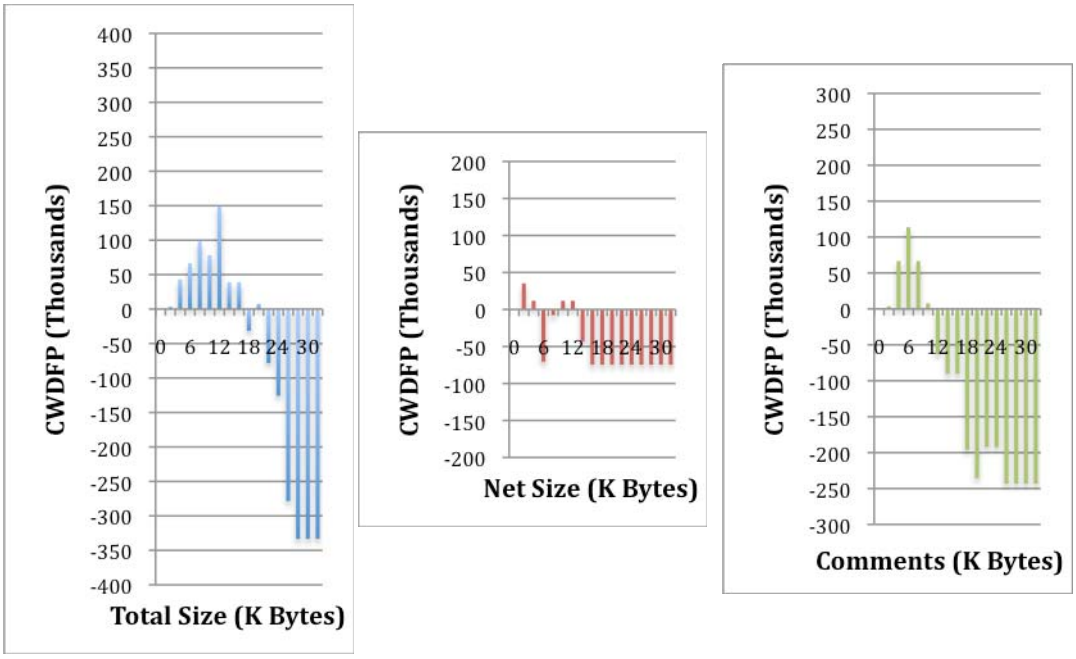


Figure 8 – Cohort A: Cumulative Comparison of Portfolios 1 and 2

The main reason is the much lower use of comments in the second assignment. This may be expected as programmers grow in confidence, as they tend to place more of the meaning in the names of variables, methods, and classes. Examining the net size

graph, one can see that the latter half forms a straight line, indicating that most of the difference (i.e., reduction) occurs before that, in the smaller applications. It is difficult to avoid the conclusion that student progress in this cohort was far from clear. This is borne out by the programming constructs:

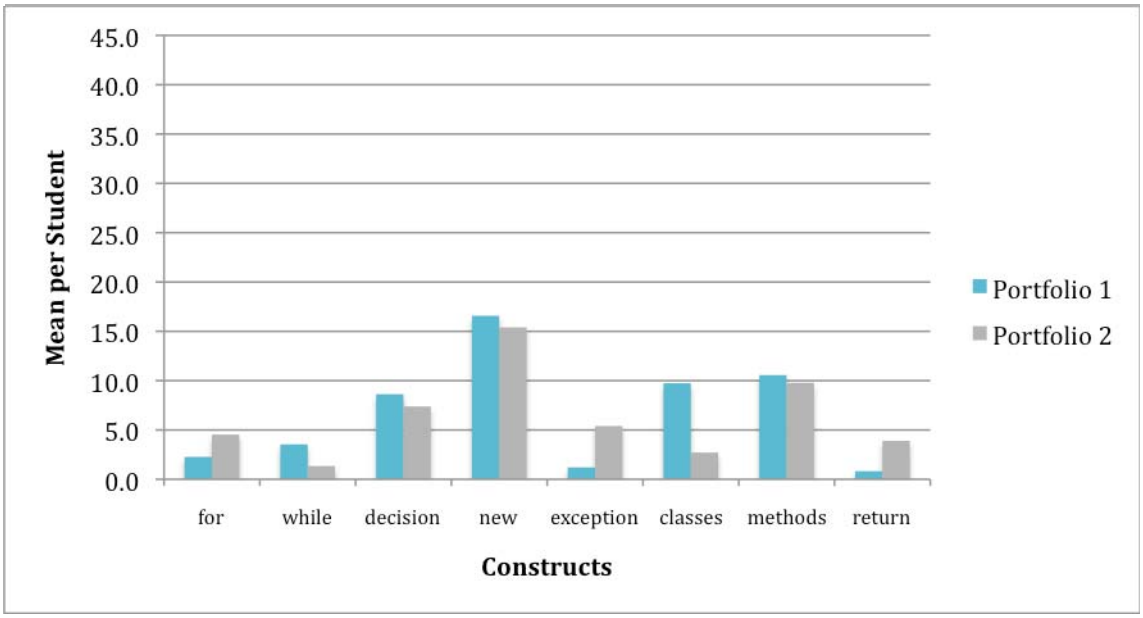


*Figure 9 – Cohort A: Use of Programming Constructs in Portfolios**

(* Note that the colours used coincide with portfolios 2 and 4 in the other cohorts. This is because the submission deadline for portfolios 1 and 2 in Cohort A roughly coincided with those of portfolios 2 and 4 in the other cohorts).

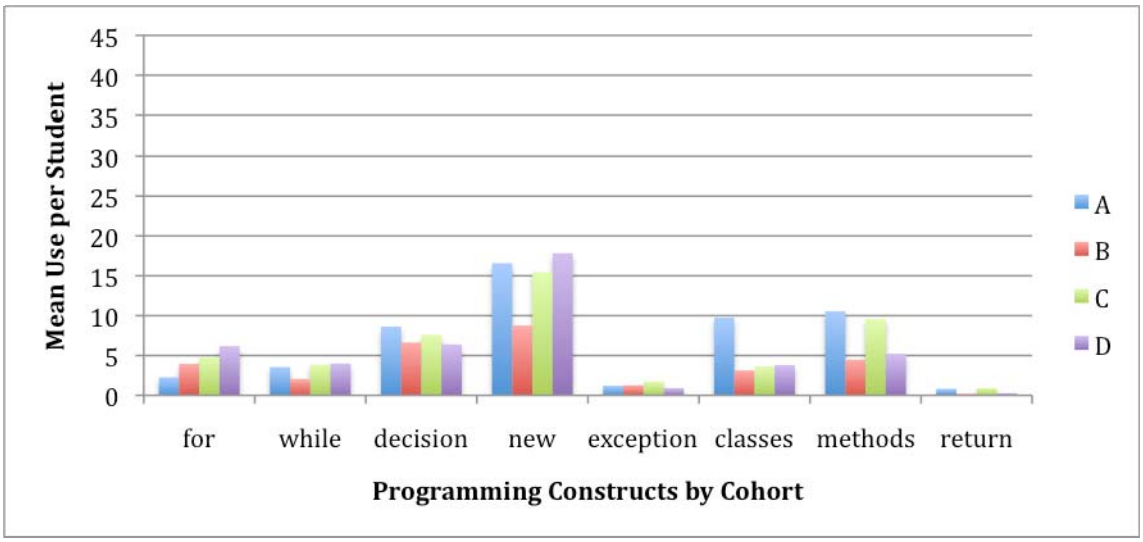


*Figure 10 – All Cohorts: Use of Programming Constructs Portfolio 2**

(* Portfolio1 is included for Cohort A, as its submission deadline aligned with the second portfolio for other cohorts).

The apparently promising figures for new, classes and methods for Cohort A is somewhat misleading. The students were introduced to a different format for applications – one which included a minimum of 2 classes, and hence a minimum of 2 methods and one 'new' declaration per application. The adjusted figure follows:

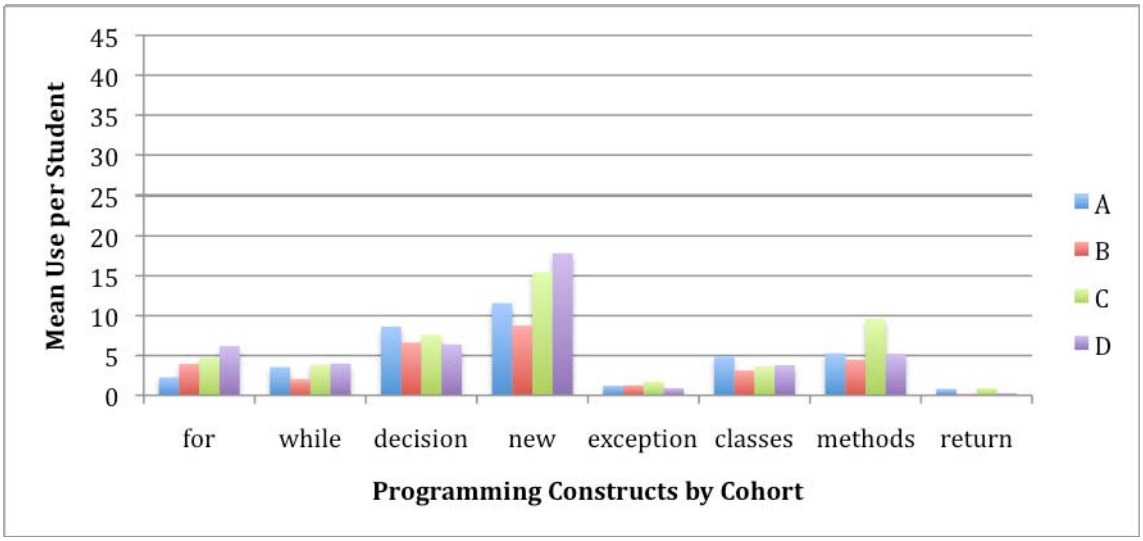


*Figure 11 – All Cohorts: Adjusted Use of Programming Constructs for Portfolio 2**

(* Portfolio 1 is included for Cohort A, as its submission deadline aligned with the second portfolio for the other cohorts).

These figures show that the progress in the early stages of each cohort is roughly the same. That successive cohorts gain more confidence can be seen by the figure below, comparing the final portfolio for each cohort.

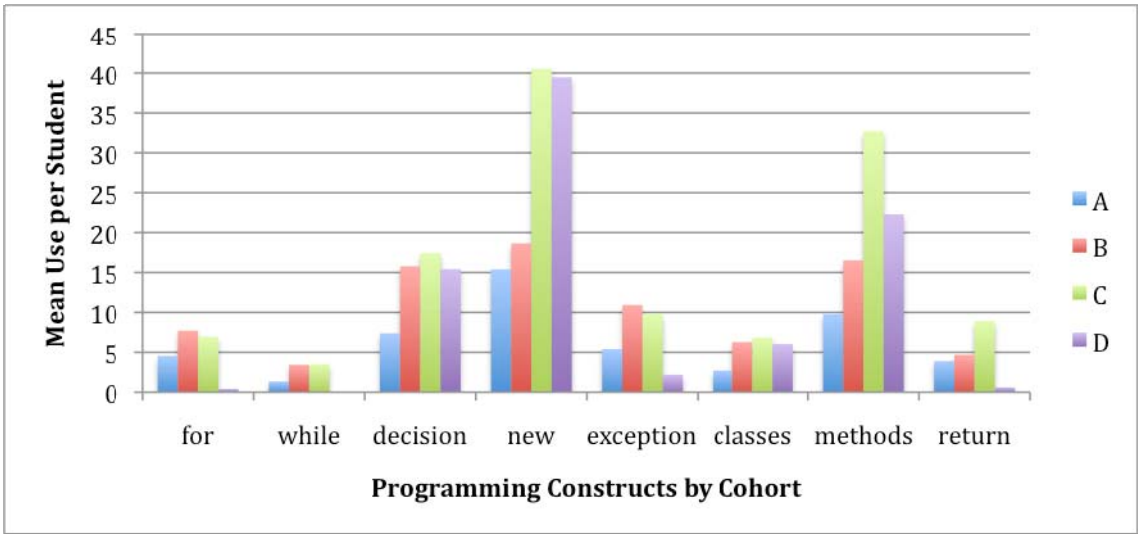


*Figure 12 – All Cohorts: Use of Programming Constructs for Portfolio 4**

(* Portfolio 2 is included for Cohort A, as its submission deadline aligned with the last portfolio for the other cohorts).

Again the comparison is not entirely straightforward. In Cohorts B and C, the final assignment was optional (provided one had already accumulated enough marks to pass the coursework), whereas in Cohort D (which used the same rubric), this was not made clear to the students. Also, the structure of other units was changed, which encouraged students to attempt the final assignment, as it involved creating a graphical user interface that would be related to a later assignment in another unit. That the use of constructs remained high for the final assignment of Cohort D indicates that the students were motivated by the assignment.

The sizes of portfolios for cohorts B and C provide more solid evidence of progress within the delivery of the unit. Below are tables showing the progression between successive portfolio assignments in those cohorts.

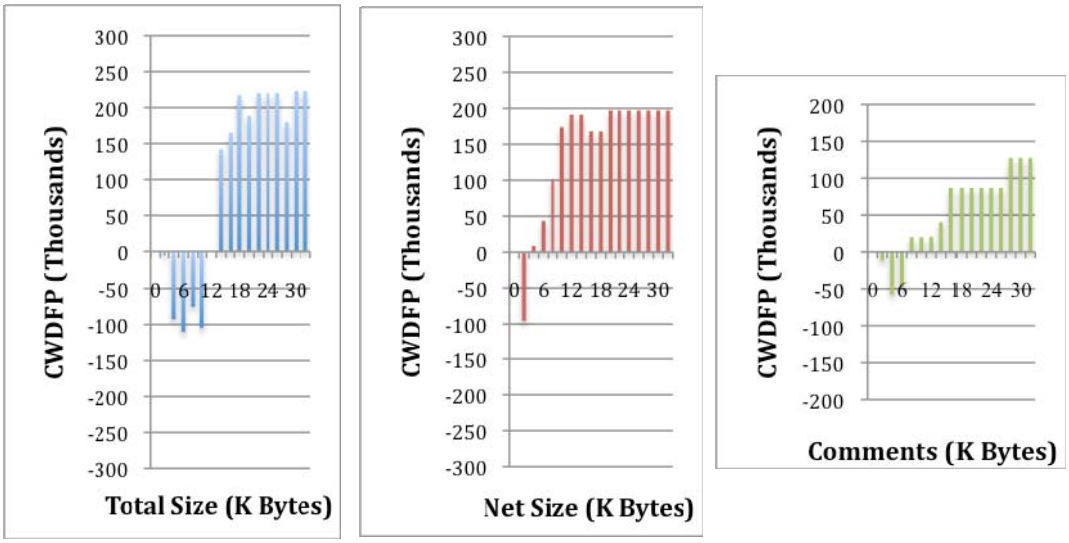


Figure 13 – Cohort B: Comparison of Sizes for Portfolios 1 and 2

The profile of the net size of portfolios indicates that there were fewer smaller portfolios submitted for Portfolio 2, as compared with Portfolio 1. At the larger end there was little change, indicating that the majority of students submitted a larger portfolio second time around.

The increase in each measure indicates a growing confidence within the student body as a whole. This demonstrates that the students assimilated the pedagogical change (of introducing collections at the outset) as easily as their predecessors assimilated writing applications without the use of collections. This supports the notion that programming concepts form a network rather than a hierarchy.

This pattern (of increasing confidence in each size measure) is repeated for the first two portfolios in Cohorts C and D. The transition between portfolios 2 and 3 follows a different path. In Portfolio 3, more students are beginning to experiment with writing their own applications. With more 'single application' portfolios, one would expect to see a reduction in overall size, not least because of the inherent overhead required for every class in Java. One would also expect to see fewer comments employed, as more meaningful names are chosen. The transition between Portfolios 2 and 3 in Cohort C illustrates this:

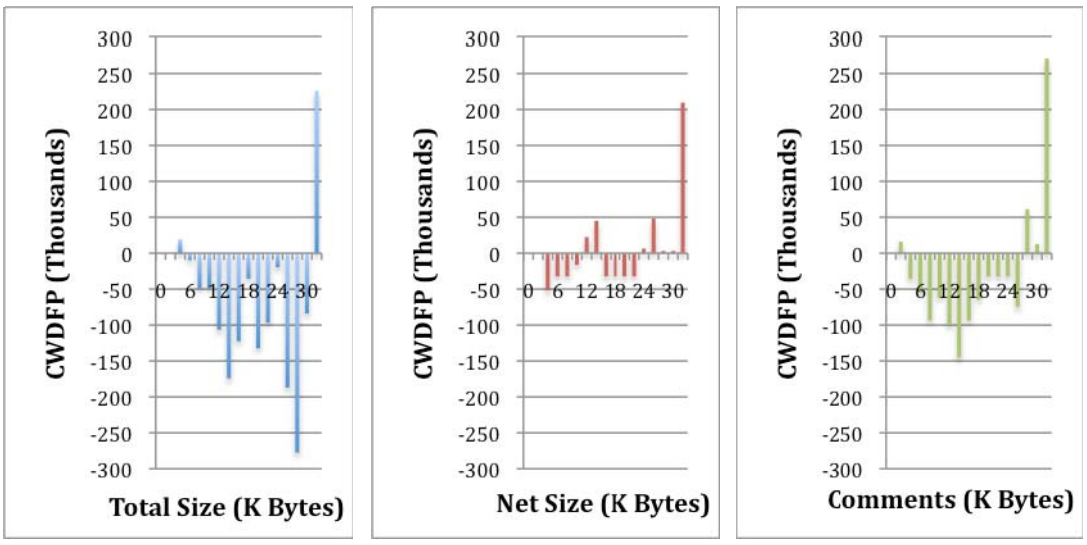


Figure 14 – Cohort C: Comparison of Sizes for Portfolios 2 and 3

The cumulative net sizes of the portfolios are almost identical, apart from a few very large portfolios linked to the keenest students. Within the majority of the cohort there is a marginal shift towards larger applications, but there is also a small reduction in very large portfolios. It should be noted that four of the students in this cohort went on to win the UK leg of the Microsoft Imagine Cup in their second year, finishing third overall.

The apparent plateau in progress between Portfolios 2 and 3 masks another dimension in the growing confidence of the majority of the student cohort. In Cohorts B and C, the specification for Portfolio 3 allowed students to design their own application. The average number of applications per portfolio reduced between Portfolios 2 and 3. The relative sophistication of each application therefore continued to rise.

This emphasises the benefits of a ‘collections early’ approach to the teaching of introductory programming. By introducing collections from the start of the unit, most students had gained enough experience and expertise to begin to explore their own interests by the start of Portfolio 3..

4.4.1. Summary

The analyses of the portfolios submitted by the students demonstrate progression within each cohort and Cohorts B and C showed more confidence and motivation

than those in the previous cohort. The distribution of confidence and motivation for each portfolio was continuous and uni-modal, with (as one would expect) a negative skew, caused by a small number of highly motivated and able students.

As there is progression between the cohorts, one can infer that the pedagogical changes and modifications to the conceptual approach were favourably received. Students were able to assimilate what might have been termed ‘advanced concepts’ without detriment. In particular, the introduction of collections of data at an early point did not lead to a lowering of achievement or motivation – indeed it could be reasonably suggested that the reverse was the case.

4.5. Comparisons of Applications Size by Programme

One of the questions in the study concerned the variation in student performance between cohorts and programmes. The following seven programmes were analysed: Cohorts B, C, and D (Computing); Cohort D (Business Information Technology - BIT); Cohort D (Network Systems Management – NSM); Cohort D (Forensic Computing & Security – FCS); and Cohort D (Unknown). The four Computing cohorts (A, B, C, and D (Computing)) were analysed together, as were the five groups within the final cohort (D). For more information see Appendix E.

4.5.1. Changes in Average

Figure 15 shows the changes in the means of applications submitted for all portfolios for the Computing cohorts. Each of the programmes shows an increase throughout the academic year, except where constrained assignments were set. These were: the final assignment for Cohort A, and portfolios 3 and 4 for Cohort D.

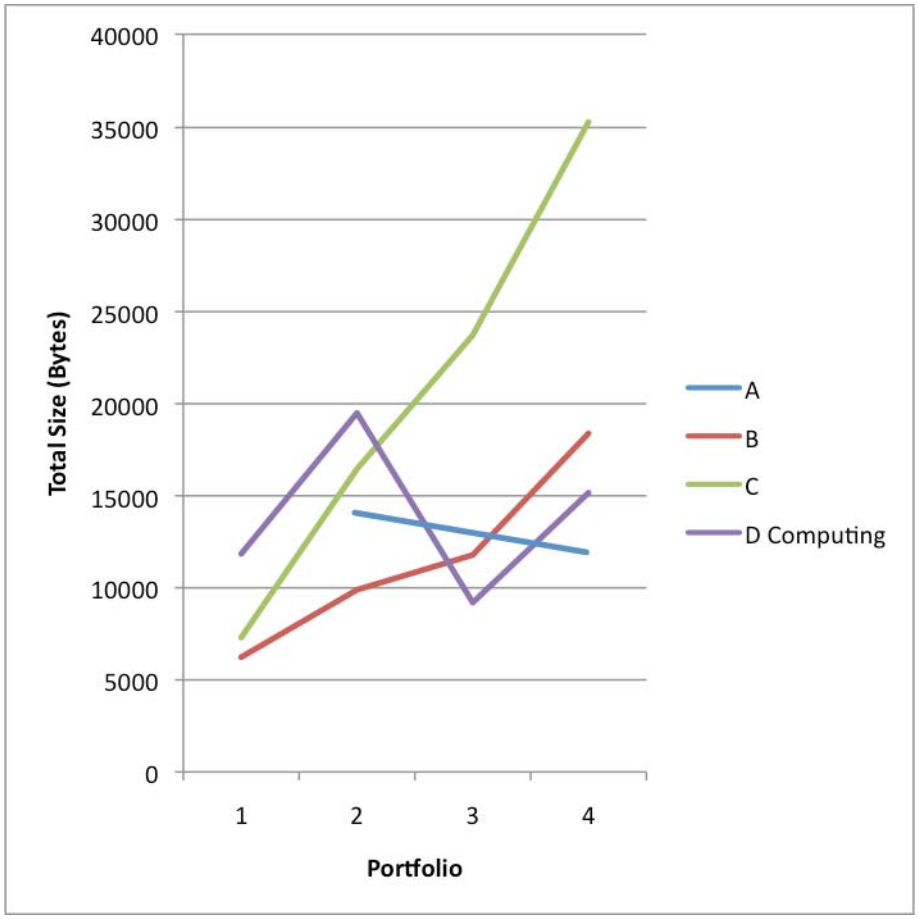


Figure 15 – Computing Cohorts: Mean Total Sizes of Portfolio

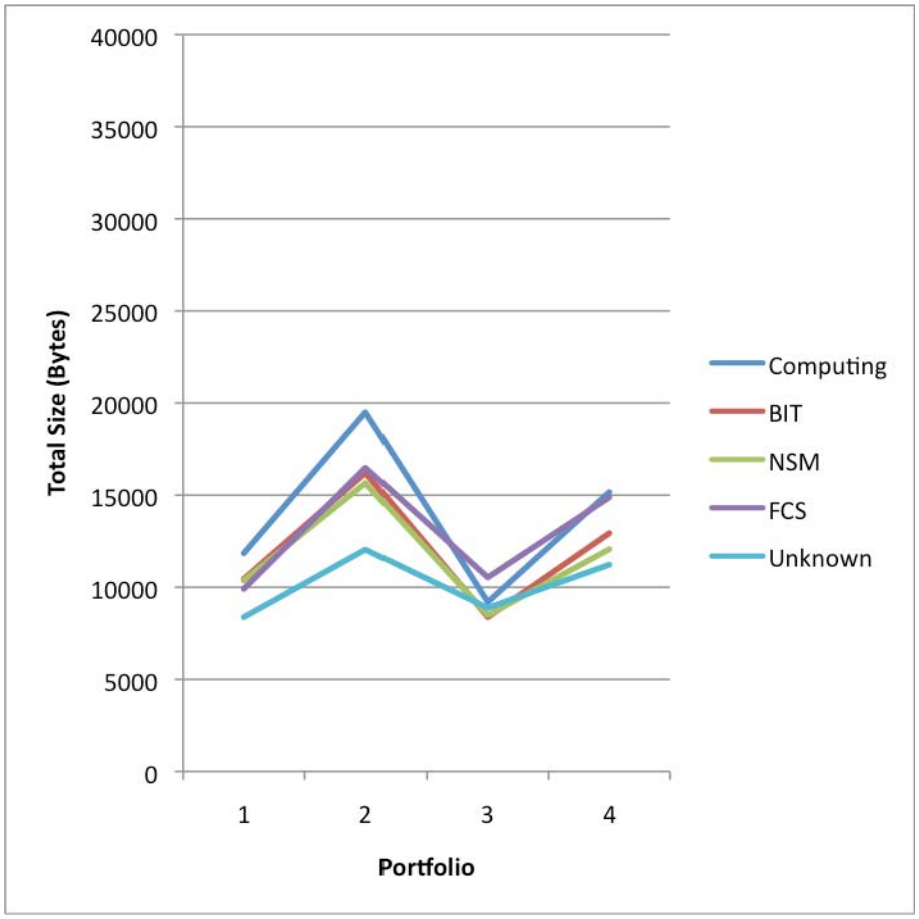


Figure 16 – Cohort D: Mean Total Sizes of Portfolio by Group

It can be seen that the constrained assignments limit the range of the sizes of applications, as one might expect.

Similar patterns were observed in the net sizes of portfolios and across the first and third quartile of the student population. The skew within each of the portfolios was, as expected, generally quite large and positive. The pattern was not necessarily consistent, as the figure below indicates.

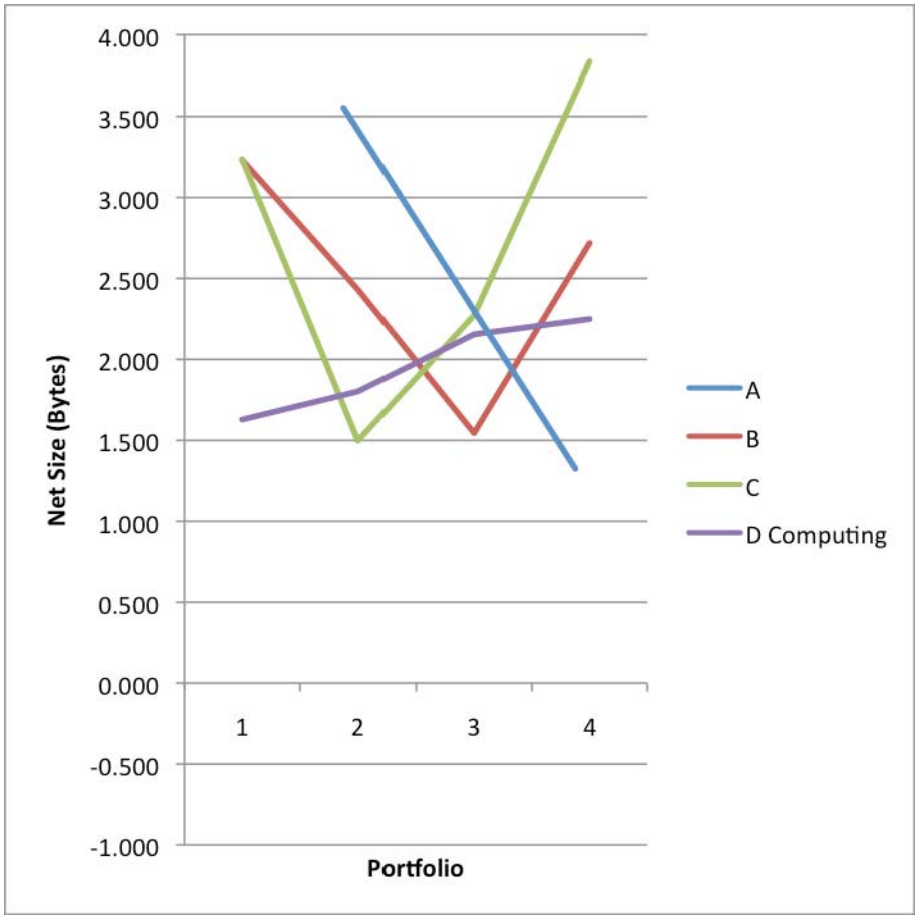


Figure 17 – Computing Cohorts: Skew in Net Sizes of Portfolio

The size of applications is one dimension of measurement that might indicate development of confidence, application and capability with a cohort. Another would be the breadth and depth of the concepts (programming constructs) being used. If an increase in size were due to the same group of constructs being used more often, then that might indicate that the students had reached a point where they felt lacking in the confidence to expand their horizons. Accordingly, the portfolios were analysed for their coverage of eight categories of construct (more details in Appendix E).

The progression through the academic year can be seen in the two figures below.

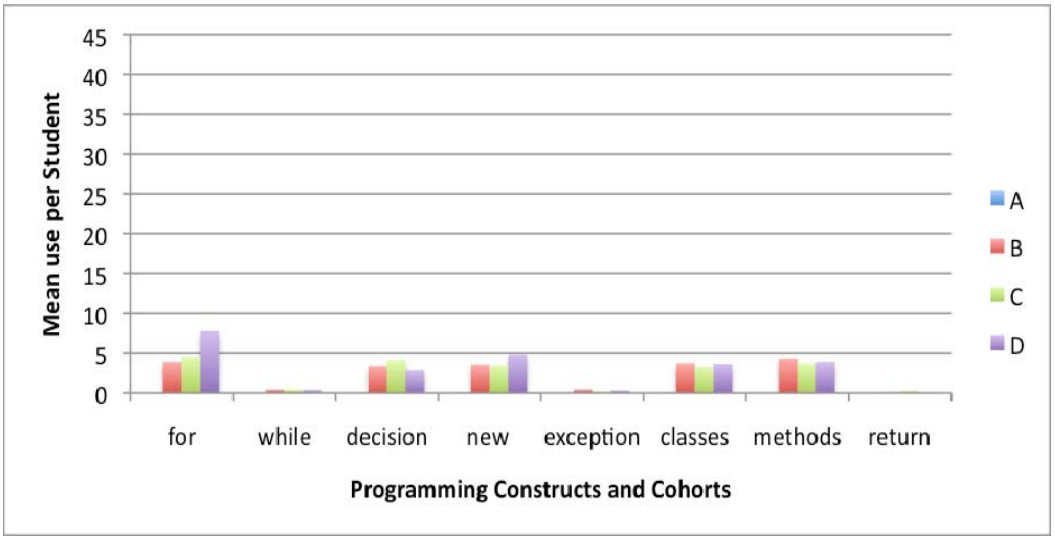


*Figure 18 – All Cohorts: Coverage of Programming Constructs in Portfolio 1**

(* There was no equivalent to Portfolio 1 in Cohort A.)

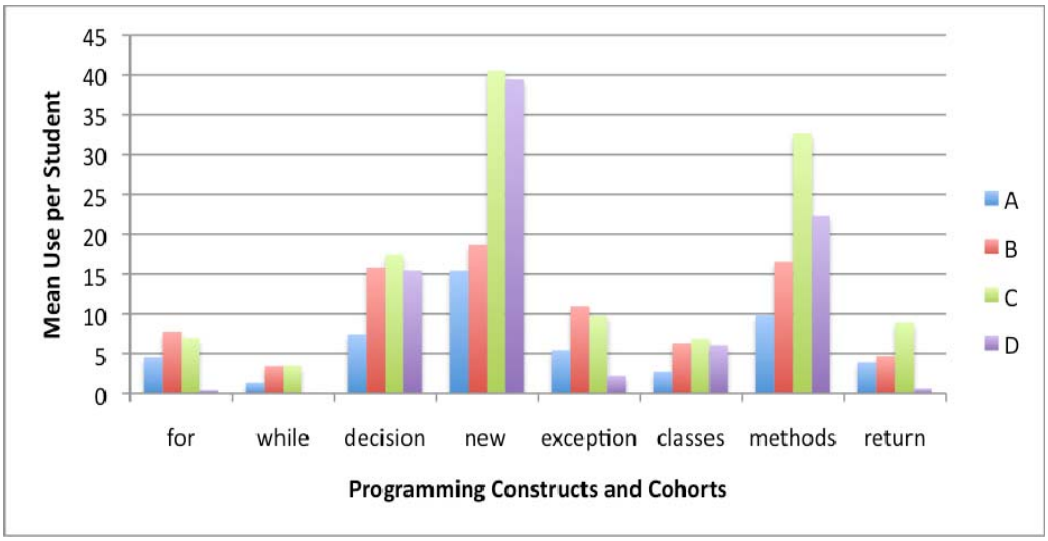


*Figure 19 – All Cohorts: Coverage of Programming Constructs in Portfolio 4**

(* Portfolio 2 was used in Cohort A, as this coincided with the deadline for the final portfolios in the other cohorts.)

The number and variety of constructs used grew throughout the academic year, demonstrating that students felt confident to assimilate new concepts as the academic year progressed.

A further analysis was conducted to attempt to establish whether the development

was broadly incremental or discrete. Discrete development is where students are introduced to a new, discrete set of concepts at different points in the delivery of the unit. To achieve this, a weighting was applied to the usage of each category, based on the distance of the usage of that category from the mean for all the categories. This is a rather crude measure, as it implies that each category should be roughly equally represented in an application. The justification was that the same measure was applied to each of the cohorts, enabling a general level of comparison. The figure below shows the pattern of weighted usage.

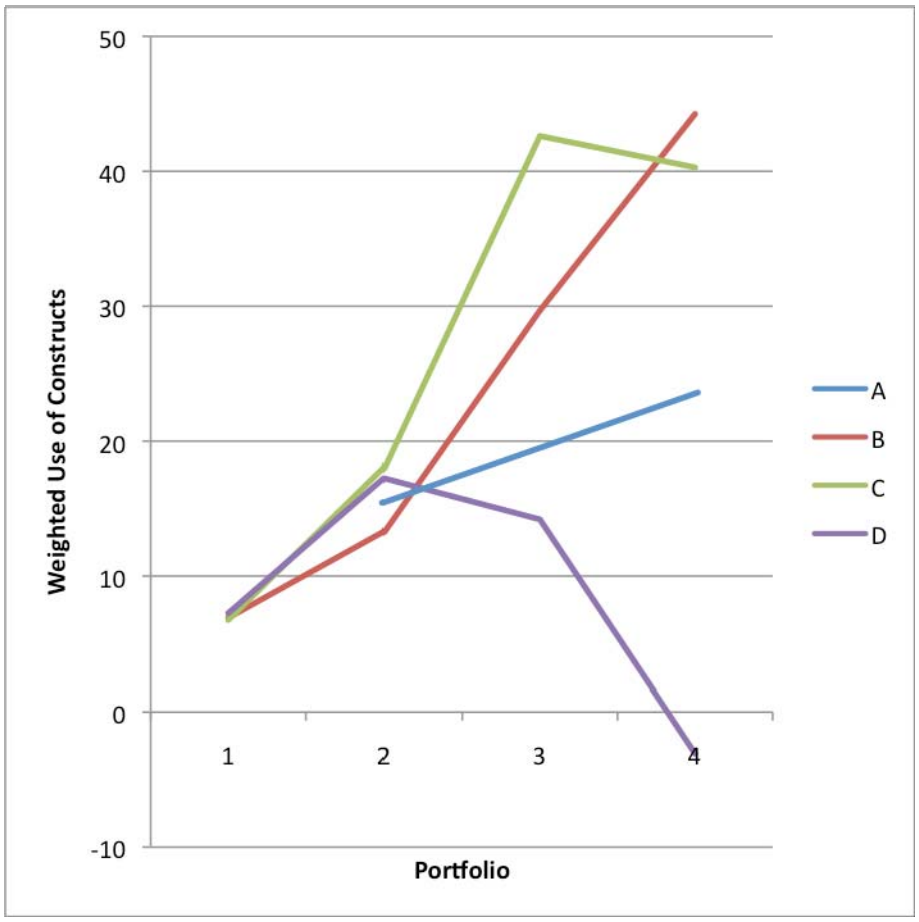


Figure 20 – All Cohorts: Weighted Use of Programming Constructs

The progression is evident in the first three cohorts, and in the early part of Cohort D. The last two values for Cohort D indicate that a more discrete approach was being taken, as usage was focused in specific areas. Another lecturer was responsible for the latter part of the delivery for Cohort D.

4.6. Tests

The analysis of the tests was confined to Cohorts B, C and D, as the tests conducted in Cohort A were written, and returned to the students, and no longer available for analysis.

Two classes of in-class test were used: online multiple choice, and written. The written test was omitted from the assessment regime in Cohort D, as was one of the online tests.

Taxonomically, there was a shift within the online tests, with more questions addressing higher levels of learning objectives in the later tests. One would therefore expect to see changes in overall marks, as full marks become more challenging. Likewise, the time taken to answer questions should increase, as students need to consider their answers more carefully.

4.7. Online Tests

The initial decision was taken to replace the in-class written tests with online tests to enable the students to receive some level of immediate feedback, whilst reducing the marking load. Certain logistical and pedagogical considerations meant that this transition should incorporate a level of transformation.

The first modification implemented was to allow the students to practice answering similar questions in preparation for the assessed tests. This mechanism is used widely in many computing-related certification programmes.

Pedagogically, it was decided that the tests should combine seen and unseen questions, partly as an incentive for students to use the practice (lesson) tests to increase their score. The other pedagogical justification for using online tests was to enable the students to gain experience of a key element of programming, that of accumulating specific pieces of knowledge. These snippets of specific knowledge need to be fused with more conceptual understanding in the production of a software artefact.

Operationally, there were a number of issues that had to be overcome. At the time when these tests were first introduced, the University had not adopted a virtual learning environment (VLE). Fortunately, the author had been engaged in the development of an eLearning system (Jones *et al*, 2003), which included an interactive multiple-choice sub-system. This was developed to facilitate the inclusion of practice questions into the assessed test, and to store the responses. The sub-system already had the capabilities to display the questions and alternatives in random order. The questions are stored in plain text, and can be created and edited using a conventional text editor.

The second operational issue was that there were considerably more students than workstations or supervisory staff, resulting in the presentation of the test being phased over the greater part of a day. Conventional systems allow students to take the test within a given time period on presentation of their credentials (typically user name and password). This enables students to take the test in an unsupervised location, with the potential for collusion. The software was extended to force the student to supply a second password that was given to the students verbally at the start of the test. A different password was set for each session. There was still the possibility that students would be able to communicate the verbally provided password to colleagues in different locations. Checking of attendance could have obviated this possibility, but it was felt that the verbal password would be sufficient.

4.7.1. The Two Modes – Practice and Assessment

The software operates differently in practice or assessed mode. Students receive immediate feedback and can continue to make selections in the practice mode. Neither of these features is available in the assessed mode. At the end of the test, the score is presented (in either mode). A detailed feedback page is produced later for the assessed mode. This ensures that students are not tempted or able to communicate the answers to specific questions to others (unless they answered all the questions correctly).

A key decision was taken with regard to the construction of assessed tests. In addition to unseen questions (which form part of each student's test), the software selects an equal number from a specified number of practice tests. It is therefore

unlikely that any two students will sit exactly the same test. This would appear to contravene the principle of fairness within a cohort, especially given that the practice questions were selected at random for each student. As questions covered the lower levels of Bloom's taxonomy (Bloom *et al.*, 1956; Bloom, 1965), it was likely that there will be slight variety in the profiles of the questions to be answered within the cohort. The equality of treatment instead refers to the equal opportunity that students have to answer those questions via the lesson tests. This minimised any perceived difference in the cognitive complexity of different questions. The online tests accounted for 20% of the coursework for the unit, and 10% of the assessment for the unit.

4.7.2. The Number of Alternative Answers and Marking Approach

The classic number of alternatives for multiple-choice questions is four: one correct, two plausible but incorrect, and one implausible (and incorrect). It was decided to vary the number of alternatives based on the nature of the question. The allowed the range of possible numbers of alternatives to vary from two to five or six. Within the 222 questions across the practice (lesson) tests and assessed tests, the distribution of alternatives was:

Number of alternatives	**Frequency**
2	15
3	68
4	120
5	16
6	3

Table 11 – Frequency of Numbers of Alternatives for Questions used in Multiple-Choice Tests

This gives an average of 3.66 alternatives per question.

With multiple-choice questions, there are several approaches to marking, some of which aim to penalise students guessing the answers by deducting marks for incorrect answers. As the purpose of the tests was largely to be formative, it was felt that penalising students would communicate a more summative aura. Given the varying number of alternatives, it is the case that the average expected score of students guessing every answer in the assessed tests would be between five and six out of 20. A specific figure is inappropriate, given the use of random selection of the practice questions. As the 'pass score' is notionally 40%, there is approximately a 12.1% chance that someone guessing the answers to all of his or her questions will get a score of 8 or more.

4.7.2.1 Categorisation of Questions

In line with the accessible and formative intention behind the tests, the questions were intended to cover snippets of useful information and simple concepts. Each question corresponded to one of the four lower levels of Bloom's taxonomy (Bloom, 1956): knowledge, comprehension, application and analysis. The table below shows the distribution of questions across taxonomic categories:

Category	**Number**
Knowledge	102
Comprehension	72
Application	39
Analysis	9

Table 12 – Distribution of Test Questions in Taxonomic Categories

This distribution echoes the intention that the tests should focus on specific knowledge, rather than examine the more advanced aspects of the students' conceptual development.

4.7.3. Grading and Positioning of Tests

The presentation dates of the online tests interleaved those of the submission dates for the portfolio assignments. This was intended to reinforce the continual nature of learning to program. In some cases the presentation was one week before a portfolio submission, in others the online tests preceded the portfolio submission deadline by two weeks. The reasons for the variation were often logistical, with tests being delayed due to the demonstrations for the previous portfolio assignment continuing to take place beyond the designated date, due to student absence due to illness or other personal reasons.

Each successive test contained a smaller proportion of questions with the lowest taxonomic category than the one before. It was anticipated that it would be easier to obtain a higher mark on the first test compared with the third. It was hoped that student progress would offset the increased challenge of the questions, resulting in a similar mark profile for each test.

4.7.4. Fairness

As there was a random selection of practice questions within the assessed tests, and the selection took no account of the taxonomic category nor the number of alternatives of those questions, there was a possibility that some students would be disadvantaged by being allocated more questions from higher categories than other students. Two measures of fairness were calculated: a correlation and a linear regression of the scores of students in the 'practice' questions against the sum of the taxonomic category and the number of alternatives. The correlation and the slope of the linear regression line were taken as the measures of fairness. A test would be considered ‘fair’ if the correlation and the slope of the regression line were close to zero.

Cohort	Online Test	Correlation	Linear Regression Slope
B	1	0.0179	0.0032
B	2	-0.1970	-0.0350
B	3	-0.0898	-0.0148
C	1	-0.0702	-0.0093
C	2	-0.0209	-0.0363
C	3	-0.1590	-0.0021
D	1	-0.1679	-0.0193
D	2	-0.1331	-0.0119

Table 13 – Measures of Fairness in Online Tests

One would expect, as the number of alternatives and question category increases, the student scores reduce. This is the case for each test except for Cohort B Test 1. Using the slope as a guide, it will require between 28 practice questions (in Cohort B Test 2) and 100 questions (in Cohort C Test 1) before the effect will amount to a single mark. It is therefore reasonable to assume that the use of a random selection that takes no account of question category or the number of alternatives does not affect the fairness of any of the tests.

4.7.5. Difficulty of Tests

Several factors affect the measurement of the challenge of an online test. The familiarity (or otherwise) of the test environment, the importance of the test, the category of questions, the numbers of alternatives available, and whether the students are familiar with the questions.

In the case of these tests, the first factor had negligible effect, as the practice environment was very similar to that in which the assessed tests took place. The physical location was the same, and the same software was used in both situations. The software did respond differently in the assessed environment, providing no feedback for individual answers. In both scenarios the students were allowed to change their answer to a question before proceeding, as the student had to click on a 'next question' button to signal that the selection made was the final one. Each test was important, in the sense that the score counted towards the final coursework (and unit) mark, although the weighting was low – 5% for each test. This level was judged sufficient to encourage the students to attend, without causing undue stress. Attendance levels at each of the tests exceeded or equalled the numbers of students submitting portfolio assignments.

The main varying factors were the numbers of alternative answers and the taxonomic category for each question.

Cohort	Online Test	Unseen Questions	Mean of Means of Practice Questions	Minimum Average of Practice Questions	Maximum Average of Practice Questions
B	1	1.20	1.25	1.00	1.50
B	2	2.30	1.71	1.30	2.20
B	3	2.30	2.39	1.90	2.90
C	1	1.30	1.53	1.10	2.10
C	2	2.30	1.70	1.30	2.40
C	3	2.30	2.39	1.90	2.90
D	1	1.20	1.59	1.20	2.20
D	2	1.90	1.51	1.20	1.90

Table 14 – Challenge of Online Tests linked to Taxonomic Categories

Cohort	Online Test	Unseen Questions	Mean of Means of Practice Questions	Minimum Average of Practice Questions	Maximum Average of Practice Questions
B	1	3.00	3.22	3.00	3.50
B	2	3.30	3.71	3.30	4.00
B	3	3.70	4.00	3.70	4.40
C	1	3.00	3.32	3.00	3.60
C	2	3.30	3.76	3.30	4.30
C	3	3.70	3.96	3.70	4.30
D	1	3.00	3.37	3.00	3.70
D	2	3.20	3.65	3.20	4.00

Table 15 – Challenge of Online Tests linked to Numbers of Alternatives

4.7.6. Student Performance

The analysis of student performance was limited to those students who took the test on the designated day. A different set of questions was used for students taking the test at a later date (due to mitigating circumstances).

The basic data for all the tests were:

Cohort	Online Test	Number of students	Mean Score	Standard Deviation	Skew	Number of fails (less than 8)
B	1	71	16.92	2.324	1.393	0
B	2	67	13.79	2.904	1.642	2
B	3	53	11.03	2.920	1.781	6
C	1	74	15.45	2.652	1.192	1
C	2	73	12.01	2.732	1.274	4
C	3	72	11.13	2.494	1.924	9
D	1	193	13.44	2.962	1.393	10
D	2	143*	12.8	3.109	1.332	9

Table 16 – Measures of Online Tests Scores

(* There were operational problems that affected 36 students on the designated test day.)

It can be seen from the results that student progress did not counterbalance the increased difficulty of the questions. Both Cohort B and Cohort C had very similar profiles, although the numbers attending the test on the designated day in Cohort C were considerably greater than for the previous cohort. Cohort D, with its wider student profile showed a consequently more varied performance, with more students gaining results in Test 1 rather in line with those which could be obtained by guesswork. The profiles of results for each cohort in test 2 were quite similar.

4.7.7. The Unseen and Practice Questions

The means and standard deviations of the eight tests are shown in Table 17.

Cohort	Test	Unseen Mean	Unseen Standard Deviation	Chance of Guesses in Unseen	Practice Mean	Practice Standard Deviation
B	1	7.662	1.812	0.5%	9.254	1.010
B	2	5.179	1.906	9.0%	8.612	1.595
B	3	3.396	1.633	22.3%	7.642	2.288
C	1	6.149	2.078	4.9%	9.297	1.459
C	2	3.699	1.721	19.6%	8.315	1.747
C	3	2.792	1.278	31.1%	8.333	1.993
D	1	4.762	2.093	11.7%	8.679	1.693
D	2	4.573	1.973	12.9%	8.231	1.890

Table 17 – Scores on Unseen and Practice Questions in Online Tests

The mean score on unseen questions reduces for each test within each cohort. This effect is far less marked in the practice tests, leading to one of two conclusions: either the practice questions were, in some senses, easier, or the students were taking advantage of the opportunities to practice using the lesson tests. In each test the standard deviation remains relatively stable, with three tests showing a standard deviation outside the range 1.4 to 2.1.

No data was gathered for the use of the lesson tests for Cohorts B and C. The disparity between the students' performance in unseen and practice questions in Cohorts B and C prompted a partial modification of this policy for Cohort D, with student requests for the lesson tests being monitored, but their performance in those lesson tests remaining unobserved.

The means for scores in the unseen questions in Test 3 in both Cohorts B and C are quite close to the figure which would be expected from students guessing each answer: 10 / 3.7 = 2.703. The mean for Cohort C in particular (2.972) suggests that

most students were unsure of most of the topics (or had acquired flawed understanding). If one assumes the curve follows the Normal Distribution, then there is a 22.3% chance that the mean for Cohort B (3.396) was also linked to guesswork, as the guesswork figure of 2.702 lies 0.425 standard deviations below the mean achieved.

An analysis of the success rates of student responses in the tests reveals the complexity of the relationship between taxonomic category and difficulty. The difficulty of each question was calculated as a function of the taxonomic category and the number of alternatives. An allowance was made where questions were unseen. When this anticipated difficulty is compared with the actual success rates, then a number of large discrepancies appear. Two examples will suffice – the unseen question the students found most challenging (compared with the projected difficulty) and the practice question that the students found the least difficult, when compared with the anticipated success rate.

4.7.8. Taxonomic Categories

An analysis was conducted to establish which questions caused the students the most (and least) difficulty. The measure selected was the number of incorrect or correct answers. These were then analysed in terms of the question categories, to see how the student performance varied depending on the taxonomic category.

The analysis of the answers to one of the questions exemplifies the dichotomy between the taxonomic category and semiotics of the underlying concept. The third most difficult question (Question 2 – see Appendix I) asked the students to identify which keyword indicates that a method was a class method, as opposed to an instance method. The answer is ‘static’. As this requires the students to recall a specific piece of information, the appropriate taxonomic category is ‘knowledge’. Lexicographically there is no interpretation of ‘static’ that could be interpreted as applying to the concept of a class method (i.e., a method which is available to all instances of the class as well as the class instance). Similarly, the reverse is true: ‘class method’ has little intrinsic meaning, which would not point anyone in the direction of ‘static’. The responses of the students to this question suggest that the perceived difficulty of a question is also positively related to the semiotic complexity

of the terms being used.

Question 6 asks the students to identify which of a list of alternatives does not figure on the mandatory part of a method specification. A knowledgeable programmer should opt for the scope, but this element of the method specification was never omitted in the examples shown to the students, and it is therefore not surprising that 6.9% (N=5) students selected that option. One of the other alternatives (that of the parameter list) was chosen by 68.1% (N=49) of the students to whom this question was presented. This would appear the most plausible of the alternatives, as methods may have no parameters. This confuses ‘parameters’ with ‘parameter list’, as (formally) an absence of parameters is considered to be an empty parameter list, but a parameter list nonetheless.

Question 58 should have been much more difficult than the test results indicate. In the event, only 2.1% (N=1) of the 47 students (of whom this questions was asked) answered incorrectly. The question offers alternatives for a method specification and the student is asked to select the one that matches a description. If one understands how to identify an array, and one knows that its type must precede each parameter, then selecting the correct alternative is not difficult. As the students had been introduced to arrays at an early stage, one would hope that the question would prove less challenging than the taxonomic category would suggest.

The explanation for these effects lies in the nature of the semiotics of each question. Whilst Question 6 appears to require recall, in fact it requires a deep understanding of the meaning of each of the terms. Conversely, although several pieces of knowledge apparently need to be employed to decode Question 58, knowing that arrays are announced by the name of a type followed by a pair of square brackets is sufficient to identify the correct alternative.

4.7.9. How Seriously Did the Students Consider the Tests?

In addition to availing themselves of the opportunities to practice answering some of the questions that might figure in the assessed test, another measure of how seriously students considered the tests would be the response times for each answer. Given that one can imagine that it will take around two seconds for a new question to be

displayed and superficially read by a student, response times close to that figure would indicate that the students were treating the test trivially (if guessing) or finding it trivial (if the answers were mostly correct).

The figure below shows the distribution of average response times to test questions;

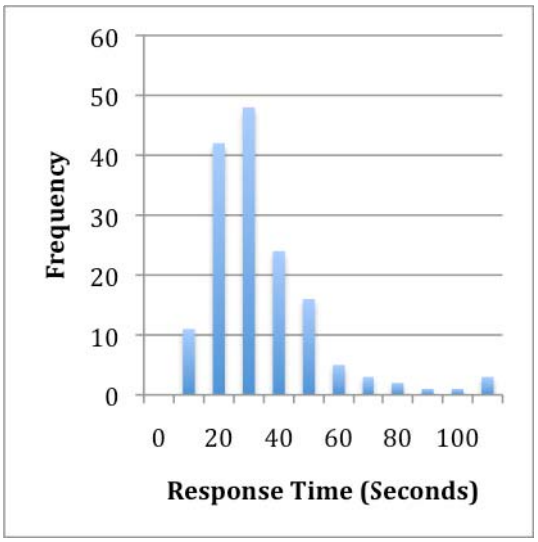


Figure 21 – Distribution of Response Times to Test Questions

The mean figure of these averages is 29.2 seconds and the median 25.3 seconds, both of which indicate that students considered their answers to the questions.

4.7.10. Operational Considerations

As the students took the test online, there are a number of opportunities for collusion or cheating. In common with examinations, students could potentially communicate with each other verbally, or via hand-written notes. The additional communication mechanisms are email and instant messaging. As the students also continued to have access to the Virtual Learning Environment (VLE), there was the additional possibility that students could consult the relevant lesson tests.

No specific measures were taken to counter the use of email, instant messaging, or consulting the VLE. Partially that was logistical, as it would require considerable resources to create an environment that allowed access to the tests, but denied the use of other applications and prevent access to other URLs. The construction of the test for an individual (with the random selection of questions from the lesson tests), together with the random presentation order of questions and alternatives were

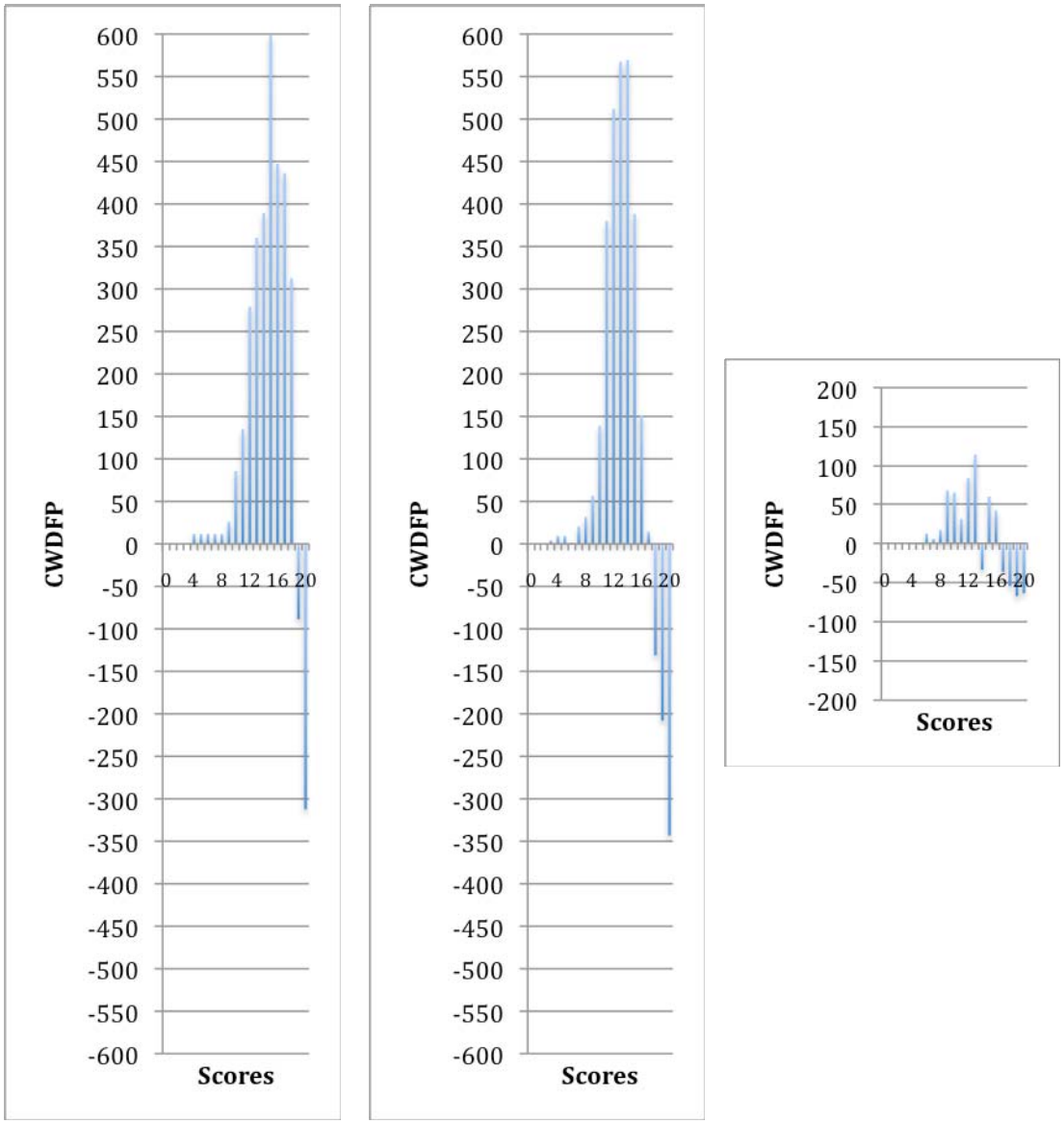
designed to minimise the opportunities for all forms of communication. The tests were also supervised, and the use of a verbally notified additional password was used to reduce the opportunities for students to take the test at an unsupervised location.

4.7.11. Conclusions

The analyses of a number of parameters of the online tests suggests that students prepared for the assessed tests using the lesson tests, and spent an appropriate amount of time considering their answer to each question. It is therefore reasonable to assume that the scores obtained are representative of the students' knowledge and understanding of the topics covered in the tests.

The results indicate that, in general, students were able to recall reasonably accurately the answers to the practice questions, but often found difficulty relating that knowledge and understanding to similar questions that formed the unseen part of the assessed tests. This is particularly marked in the third tests in each of Cohorts B and C, where the profile of student performance in the unseen questions closely resembles that which would emanate from pure guesswork. It is therefore highly likely that, for many students, there was limited understanding, and that which passed for knowledge and understanding was significantly biased by the effect of recalling practice questions and their answers. The random presentation of alternatives obviates the possibility that this recall was positional, but the disparity in the performance in answering practice and unseen questions suggests that the students were relying (to a considerable extent) on textual recall.

Conceptual understanding progressed slowly during the delivery of the unit. As the profile of the taxonomic categories covered by the questions changed with the introduction of 'higher level' questions, so scores dipped, despite the students growing experience of programming. This supports the notion that students found most of the concepts associated with the unit very difficult, given that the students generally had a positive perspective of the unit (as identified via the independently organised end of unit survey).



Cohort B *Cohort C* *Cohort D*

Figure 22 – Cohorts B, C, D: Comparisons of Scores in Tests 1 and 2

The pattern in cohorts B and C are very similar, with a general reduction, also indicated by a lowering of the mean score.

4.8. Did Students Work Continuously?

The number of concepts required to write even a simple program implies that the optimum approach to learning to program is to work continuously, thus providing maximum opportunities to absorb, understand and reflect on the realisation of the concepts.

The first cohort saw the first step towards the use of student-selection within assignments, and away from assignments that reward a compartmentalised, focus-on-the-next-assignment approach. The first assignment was a portfolio where students selected applications to submit. The second assignment continued the previous practice of writing an application to satisfy a set of criteria, although there were three options of varying programming complexity.

The student performance in the second programming assignment was markedly down on that in the portfolio assignment in two regards: the average dipped from 64.3% to 57.6%, and the numbers submitting fell from 63 to 52. That the submission date was immediately after the Easter vacation (thus affecting students' opportunities to confer) may have been a factor in these reductions.

Whilst it was clear that the students exercised choice in the construction of their portfolios, the nature of the construction process was impossible to ascertain from the submitted files, as only the last modified time is recorded when a file is included in a zip file.

The migration towards the use of a formal project directory structure in Cohort B was prompted by a desire to foster a more professional approach. This was facilitated by the use of a template structure contained in a zip file. This template included (batch) files that would be extremely unlikely to change. Any opportunity to use this data within the study was thwarted as the time and date associated with a file is copied from its entry in the zip file, when the file is extracted from the zip file.

A library that generated a project directory replaced the template zip file. This produced a more reliable start date and time for a project, provided at least one of the generated files was not subsequently changed. It would be highly unlikely for the compile and execute batch files to be tampered with for any reason, hence it is reasonable to assume that the data collected linked to batch file creation times and dates is a reliable indicator of when the student started that project.

4.8.1. Measuring Portfolio Creation

It became possible to gain some insight into how students went about constructing

applications when the use of the template zip file was replaced by the use of a command entered by the student. This ‘project builder’ application was contained in a library supplied and constructed by the author. As the files and directories were created by this command, the associated dates and times would be a reasonably reliable guide as to when the application was begun, provided at least one file (or directory) remained unchanged. All the directories would be modified during the editing, compiling, or executing phases, leaving the batch files as suitable candidates. There would be no reason for a student to modify either the build or the run batch file, so it is reasonable to suggest that the vast majority of these files remained in their original state throughout the development process.

The caveats are that a student might copy a project directory (and all its contents), and then modify certain elements for the new project. There is some evidence of this, as the dates and times of the ‘build’ batch files of different applications are coincident in a number of cases.

Figures 23 to 25 illustrate the evolving nature of the distribution of project construction approaches within the four cohorts. The times and dates of the batch files within a portfolio were compared, and categorised according to how many matched the others: all, some, or none, where none indicates that all had different times. Coincident times and dates would occur (for instance) if a student copied and modified a previous project (or used the template zip file), rather than use the project builder.

Portfolios containing a single program were placed in a separate category.

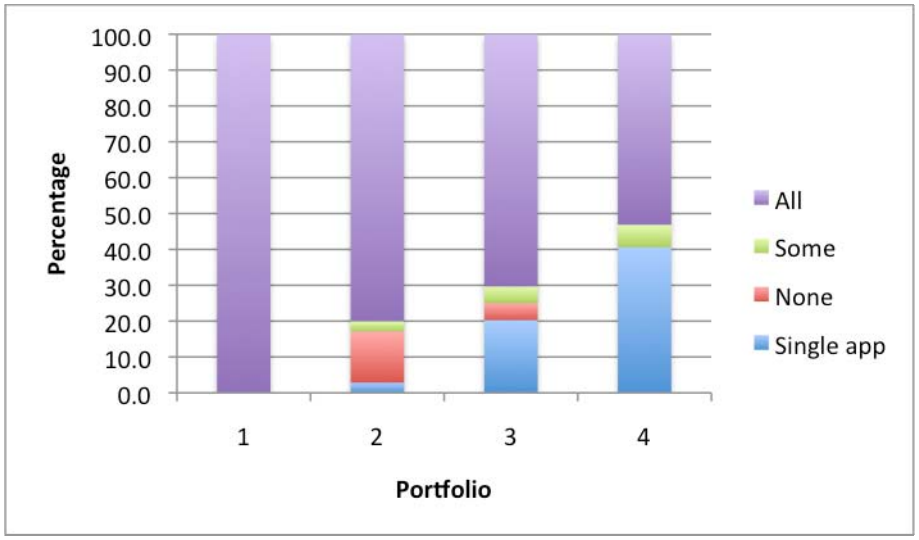


Figure 23 – Cohort B: Distributions of Build Batch Files Dates and Times

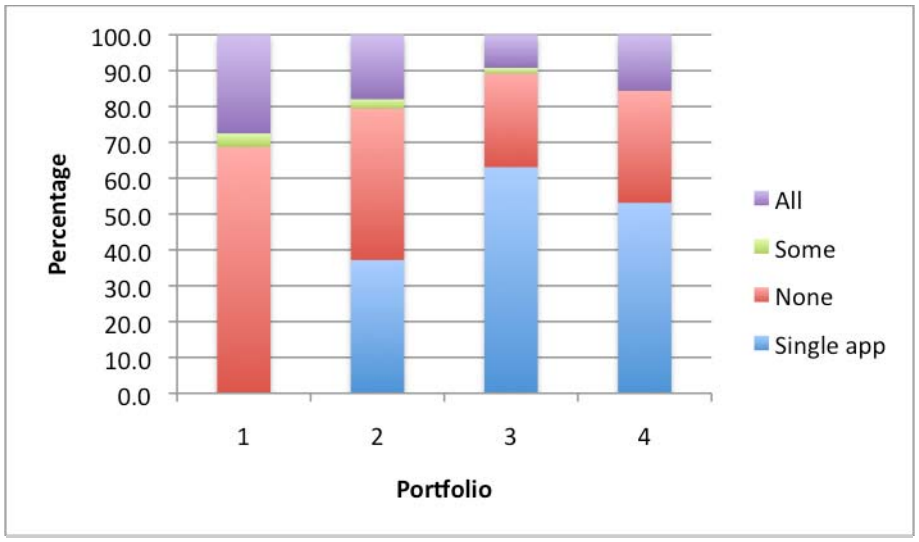


Figure 24 – Cohort C: Distributions of Build Batch Files Dates and Times

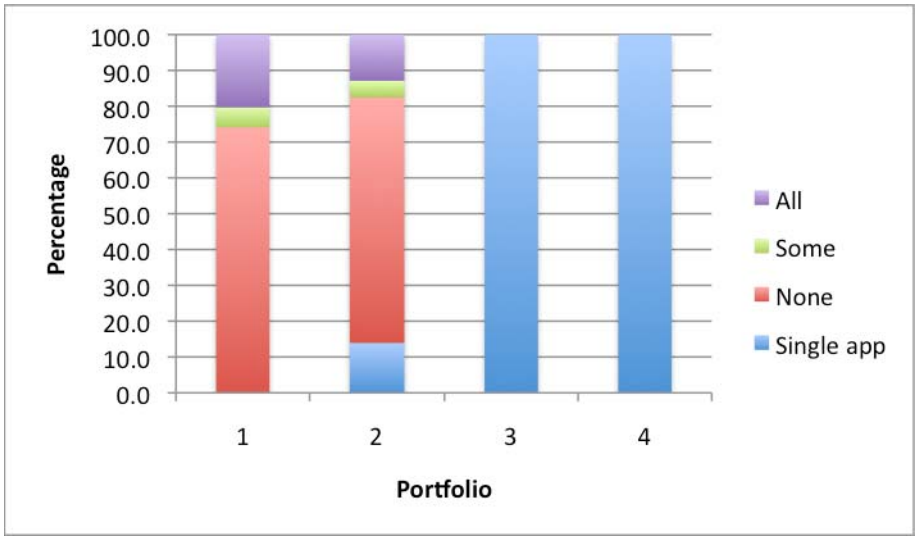
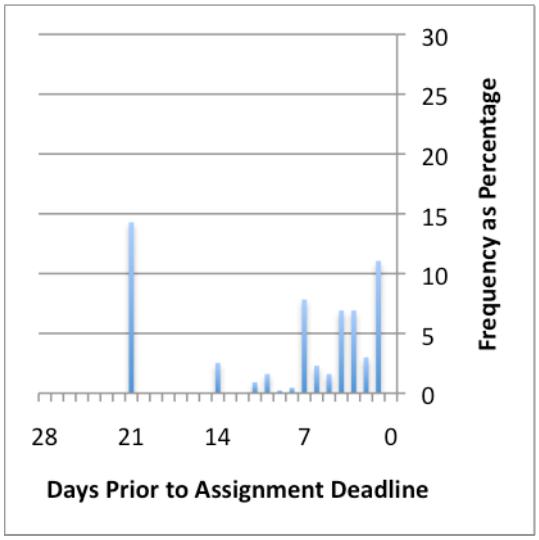


Figure 25 – Cohort D: Distributions of Build Batch Files Dates and Times

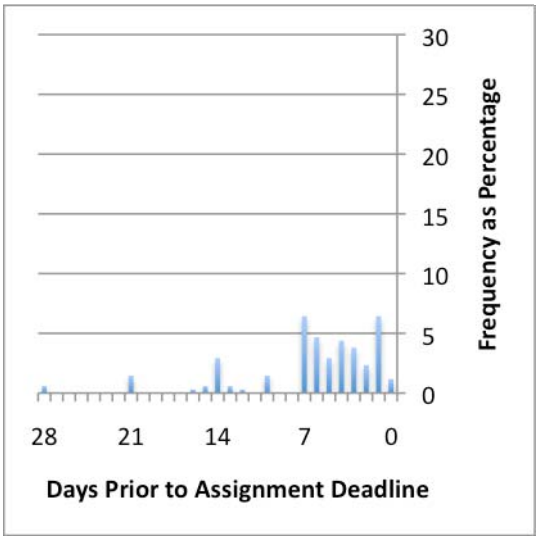
The influence of the introduction of the project builder application can be seen in the figures for cohorts C and D. A number of students (almost 30% in the case of Cohort C Portfolio 1) chose to use the 'copy directory' method. The incidence of this strategy declined as the academic year progressed, with the exception of the last portfolio in Cohort C. This last portfolio had a deadline that was only one week after the submission deadline for a double assignment associated with two other units, so there is a sense that some of the students who submitted may have been pressed for time.

A close look at the creation times for these batch files supports the notion that many students started their applications close to the assignment deadline, thus continuing to adopt the 'next assignment' strategy, even where weekly exercises could be used in the construction of a portfolio. (In a 'next assignment' strategy, a student only works on one assignment at a time, often termed 'assignment driven' learning.)

Figures 26 to 29 show the distribution of batch files creation times as the assignment deadlines approach. Percentages have again been used, rather than absolute numbers, to illustrate the trends more clearly.

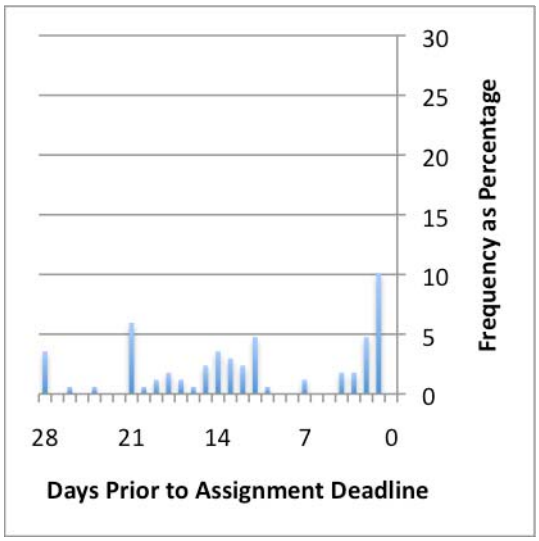


Portfolio 1

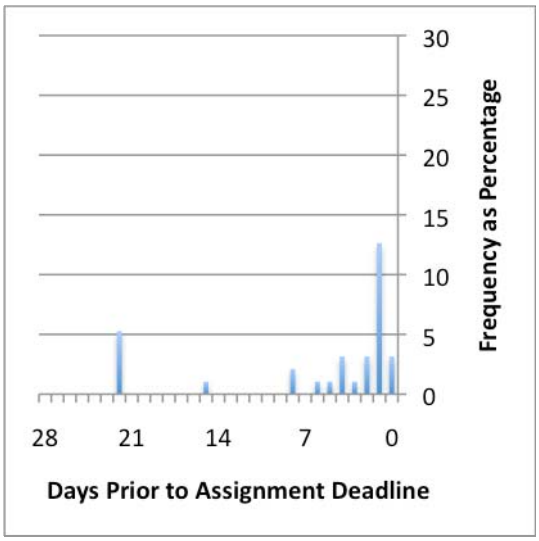


Portfolio 2

Figure 26 – Cohort C: Batch File Creation Times for Portfolios 1 and 2

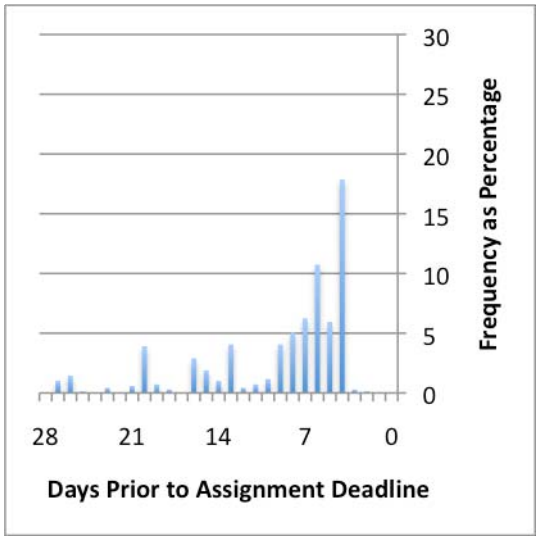


Portfolio 3

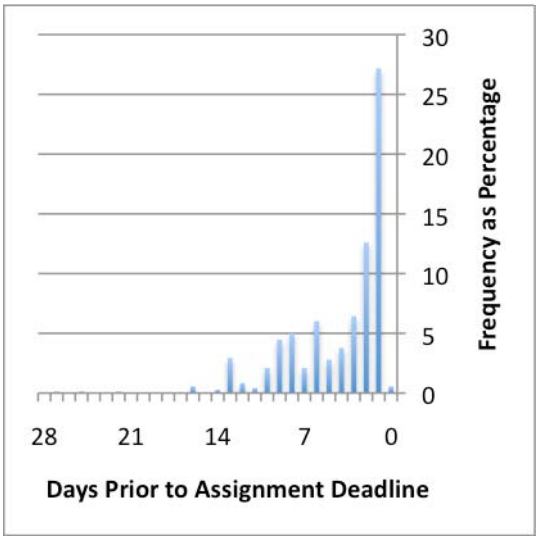


Portfolio 4

Figure 27 – Cohort C: Batch File Creation Times for Portfolios 3 and 4

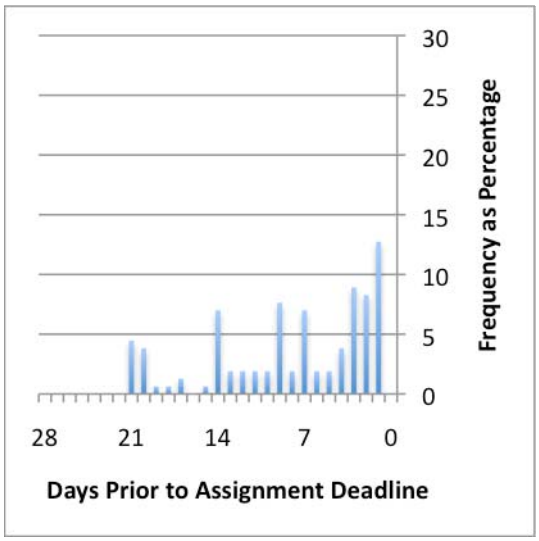


Portfolio 1

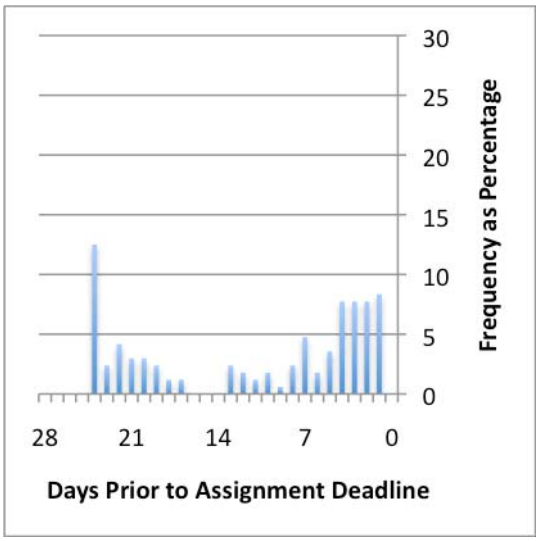


Portfolio 2

Figure 28 – Cohort D: Batch File Creation Times for Portfolios 1 and 2



Portfolio 3



Portfolio 4

Figure 29 – Cohort D: Batch File Creation Times for Portfolios 3 and 4

The cohorts behaved in generally similar ways, in that there is a spread of batch file creation times, although a number of applications were begun quite close to the assignment deadline. Cohort C contained more students who began the applications that formed part of their portfolio earlier than was the case for Cohort D, whilst more students (as a proportion) in Cohort D began their applications closer to the eleventh hour, so to speak.

One can see spikes in the batch file creation times, coinciding with the weekly workshops.

When one examines the numbers of batch files created more than 28 days before the assignment deadline, one can see that this may not be an indicator of committed and enthusiastic students. The numbers and proportions of applications apparently started more than 28 days prior to the assignment deadline are given in Table 18.

Cohort	Portfolio	More than 28 days	More than 365 days
C	1	3 (1.1%)	0 (0.0)
C	2	94 (40.5%)	12 (5.2%)
C	3	47 (43.8%)	4 (3.0%)
C	4	39 (54.9%)	4 (3.0%)
D	1	199 (28.9%)	199 (28.9%)
D	2	153 (21.4%)	76 (10.6%)
D	3	31 (19.7%)	10 (6.4%)
D	4	31 (18.5%)	10 (6.0%)

Table 18 – Cohorts C, D: Build Batch Files created more than 28 Days Prior to the Assignment Deadline

The high proportion of ‘elderly’ batch files in Cohort D can be partially explained as the first application presented to the students was in the form of a zip file. Provided the student extended this application, he or she could submit it as part of the first portfolio. The situation is more complex, however, as there were a total of 15 different times amongst the 199 files created more than a year before the deadline for the first portfolio. In the case of Cohort C, students were encouraged to revisit and extend applications and resubmit them as part of subsequent portfolios. One can see that this was also the case in Cohort D, between the first and second portfolios (the

latter two ‘portfolios’ in Cohort D were specified applications).

For information, the numbers of applications with coincident build batch file dates and times were: Cohort B – 94.83% (N=422), Cohort C – 44.86% (N=314), Cohort D – 32.31% (N=556).

Given that it is far simpler for a student to use the project builder application than manually create the directory structure and files for an application, it is not surprising to note that each cohort made more use of the project builder application than the previous cohort. The pattern of use remained consistent, with around a third of application directory structures being created within a day of the portfolio being submitted.

No reliable attendance figures were maintained for workshops, but the majority of seats were occupied, certainly in the first term in each workshop. As students were presented with a number of exercises each week, most of which involved writing applications, it is reasonable to suggest one of two reasons for this late creation of the batch files. The first is that most students did not manage to complete sufficient applications during the workshops (or at home between workshops), so had to write applications was the assignment deadline approached. The second alternative was that students did not use the project builder application as part of the workshop exercises, and then used it to create the directory structure specifically for assignment submission.

4.8.2. Detecting ‘Unsporting’ Behaviour

In a unit with such visibility as programming, where sharing between students can easily extend beyond information, one would expect (and possibly encourage) a degree of collaboration. Some may be tempted to go beyond that. A dozen students were found guilty of plagiarism in the final portfolio of Cohort D, but these may well have formed the most obvious tip of a much larger iceberg.

4.8.2.1 Copying within Portfolios

Comparisons of the submitted source files were used to detect whether two or more

students submitted the same file. With portfolio assignments, one cannot use the conventional plagiarism detection software (e.g., TurnItIn) as students were permitted to submit the same application, and there are very few ways in which a very small application can be written. Instead, all comments were removed, and then the resulting files compared for size. This is a crude mechanism, as modification of (say) the variable names would be sufficient to pass this test. Nonetheless, it will detect the situation where a student has been sufficiently lazy (or lacking in confidence) to modify code written by another student. No directly comparable applications were found.

It is also possible that code could be copied from the Web, or be commissioned from another student, or from one of the many contract programming websites. Commissioned code cannot be detected even with the use of plagiarism detection software.

A demonstration (by the students) was included in all the assignments. This does not preclude or prevent plagiarism, but it does at least require that the student be able to explain his or her code.

4.8.2.2 Concurrent Consultation of Lesson (Practice) Tests

Requests for access to the lesson tests were monitored for Cohort D. When these requests were compared with the start and end times of the student's assessed tests, it became possible to identify those students who felt the need to 'consult' this resource. This activity was not penalised, as the overall effect would be negligible, unless the student was making a number of requests to view the lesson tests within the time of the assessed test. In the event, no student exhibited that behaviour. For selective consultation, the student would first need to feel confident that a question in the assessed test had formed part of a particular lesson test, but not confident enough to answer the question. This would not be possible for those students who had not done any of the practice tests. It was therefore anticipated that few students would consult the lesson tests, and that those would have worked through the lesson tests. This proved to be the case, as 7.0% (N=14) and 11.2% (N=22) students accessed the lesson tests during the assessed tests in Cohort D.

4.9. Conclusions

The results provide indicative answers for the research questions. Within each cohort there was a steady progression when measured in terms of the sizes of the portfolios submitted and the range of concepts used in those portfolios. The pedagogical re-ordering of the introduction of programming concepts (typified by introducing collections early) were well received by students, and many students were confident enough to be more creative and explore their own interests in the later assignments (based around a specific set of concepts rather then functionality).

5. Pedagogical Innovations

The term ‘innovation’ is here used in connection with two facets: entirely novel as far as the institution is concerned, and relatively novel compared with their use elsewhere. The five main innovations introduced into the introductory teaching of programming were:

1. A paradigm shift towards programming fundamentals and away from programming language fundamentals.

2. The requirement for students to explicitly identify those elements of their work that satisfy particular requirements.

3. The use of a code generator to assist in gaining understanding of comprehension and modification.

4. The use of a library to simplify the production of applications.

5. The use of a system to semi-automate the process of the submission, assessment, and the provision of feedback on students’ work.

The first is the main focus for the study, and has been considered in detail in the previous chapter. The second appears to be unique in the context of the introductory teaching of programming. One other example of the third has been found (Ganapathi and Fischer, 1985). Many examples of the fourth innovation exist, although few contain as many elements as the one created and employed in this context.

5.1. Assessing Concept Realisation Directly in Introductory Programming

One of the difficulties in developing a learning and teaching scheme for introductory programming is deciding on the visibility of the understanding pertaining to the realisation of the underlying programming concepts within the students’ work. At one end of the spectrum is ‘implicit realisation’, where the production of ephemera

(such as working applications) implies understanding of the relevant programming concepts. This was the approach taken with Cohort A, a continuation of the practice of many years within the institution in question. Explicit realisation requires the learner to demonstrate understanding of the rationale behind the use of a given concept at a point in a particular application. This can be managed through the use of face-to-face demonstrations, stylised comments, unseen examination questions, or by requiring the student to provide written evidence as part of assessed coursework. All four techniques were used in delivering the revised learning and assessment regime to Cohorts A and B. The stylised comments were included as part of the demonstrations and are considered elsewhere as part of the assessment analysis. The performance of students in the unseen examination questions form part of the continuation analysis. The final mechanism (that of explicit inclusion of documentary evidence by students as part of coursework assignments) is considered separately because of its innovative nature, and potential in measuring students' understanding of specific programming concepts.

5.1.1. Documentary Evidence

Starting with Cohort C students were required to submit a document with each portfolio assignment specifying where nominated programming concepts were realised. These concepts were identified as part of the assignment specification. This document consisted of a number of lines, each of which identified a concept and where that concept was realised (i.e., the lines which contained statements which together formed the implementation of one of the required concepts). A concept could appear on more than one line, which is necessary where a concept was realised in more than one location (as commonly would be the case). The submission system checked for the existence of this document, and verified that the applications and lines specified in the document existed. If any errors were found, the submission was rejected.

The demonstration software was modified to highlight (select) those lines that (the student claimed via the documentary evidence) related to a given concept. The marker could then view these snippets in the demonstration, and either asks the student relevant questions, or uses them as the basis for explanations. Utilising one of the other views of the code available in the demonstration (i.e., the raw code), the

marker could then assess whether there were other occasions in which concepts were realised, but for which claims were not made. Due to time pressures, much of this was done away from the demonstrations. Students received more marks where claims matched the realisation.

5.1.2. The Method

Each portfolio submitted by students in Cohorts C and D were further analysed. Five measures were calculated for each concept for each portfolio in every assignment: the number of claims made (via the supplied document); the number of those claims that could be considered valid; the number of under-claims; the number of over-claims; and the number of occasions within the code that the concept was realised, but for which no claim had been made. ‘Under claims’ and ‘over claims’ are forms of partial validity. Partial validity exists where a claim either fails to fully satisfy the concept (an ‘under claim’), or the concept is realised within the claim, but additional lines have been included (an ‘over claim’). An example of an under claim would be where a claim is associated with the specification of a ‘for’ statement, and the concept required is a ‘for’ loop. If the reverse is true, and the claim is associated with the whole of the ‘for’ loop, but the concept required to be realised was a ‘for’ loop specification, then an over claim has been made.

5.1.2.1 Results - Claims

The first portfolio (Figures 30 and 31) was submitted after only a few weeks, and it is clear that a minority of students are not confident of (or focused on) identifying where they have realised concepts within their applications. This minority all but disappears in the second and third portfolios (Figures 32 and 33). The rubric of the coursework mark, which was formed from the best three portfolio marks plus the marks from the four tests, meant that few students need complete the final portfolio to attain a pass. It is reasonable to suggest that the students who completed the final assignment were the ones who were more motivated. It is interesting to note that the numbers of claims reduced. It is reasonable to suggest that the students were more focused on the creative aspects of the program(s) than on improving their grades.

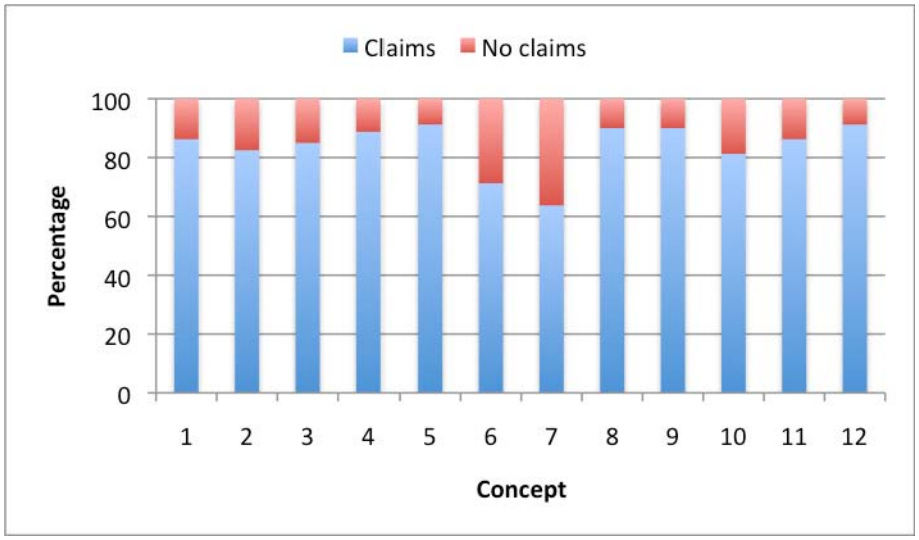


Figure 30 – Cohort C: Proportion of Students Making Claims Against the Concepts for Portfolio 1

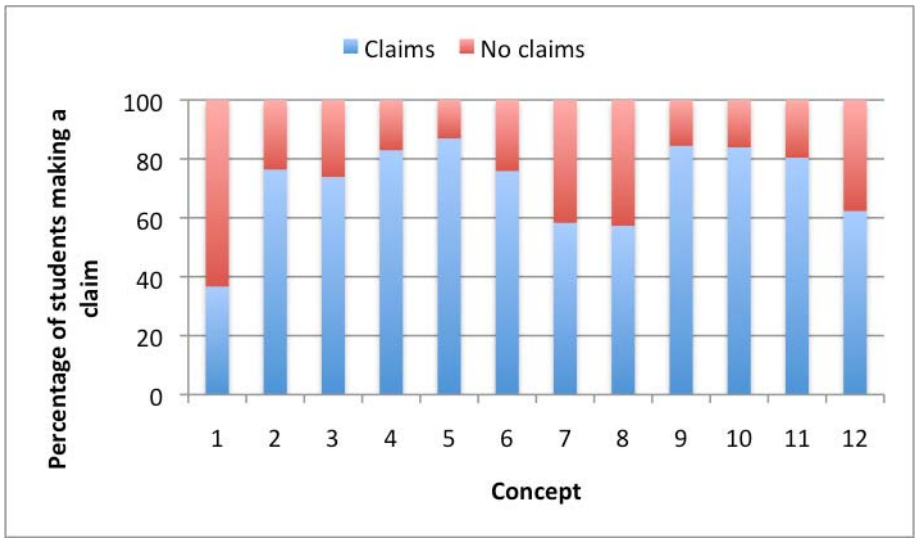
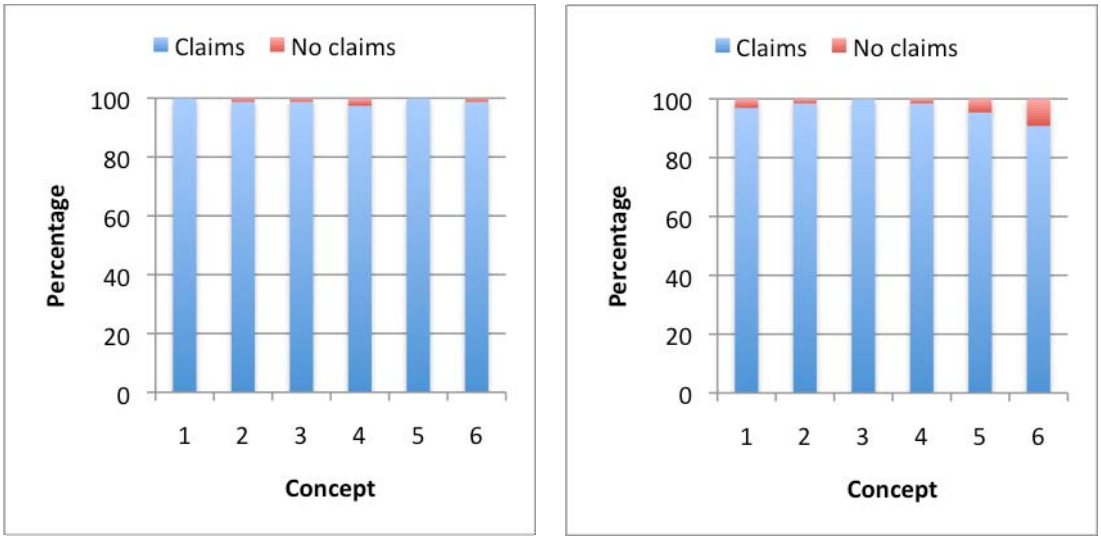


Figure 31 – Cohort D: Proportion of Students Making at Least One Claim in Portfolio 1



Portfolio 2 *Portfolio 3*

Figure 32 – Cohort C: Proportion of Students Making at Least One Claim in Portfolios 2 and 3

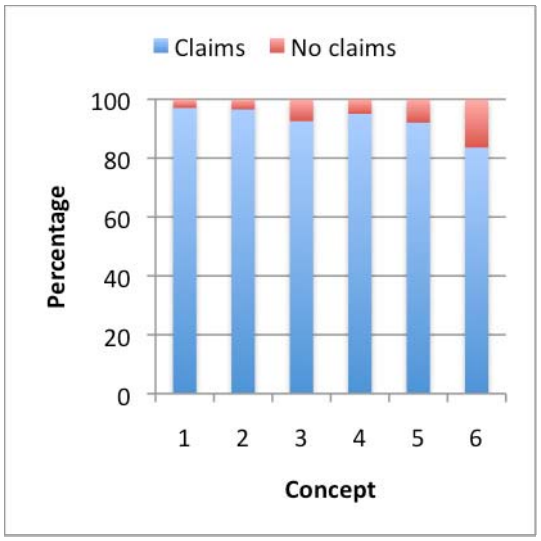


Figure 33 – Cohort D: Proportion of Students Making at Least One Claim for Portfolio 2

The veracity of the claims was not systematically analysed, due to time constraints. Assessing a claim as partially valid is not a simple task. Typographical errors could be as significant a factor as failure to link the concept to the code. Anecdotally, many of the claims in the first portfolio were partially or totally incorrect, but this was very largely rectified in the later portfolios. This adds more weight to the notion that

student confidence increased. Further study would be needed to ascertain a more systematic view of the student ability to link concept and code.

5.2. JCodeGen: Using Code Generation in Introductory Programming

Two of the fundamental questions software developers need to be able to answer are: 'how can I set a piece of software to do that?' and 'what does this program do?' Each question is inextricably linked to the other, because every act in a software development involves creative and comprehensive activities. To be able to answer each question implies understanding of a range of inter-connected concepts and techniques. In the case of the latter question answers exist at several levels. There is the highly abstracted summary layer (e.g., 'this program calculates the trajectory of a firework') through to the physical electronics layer, with a number of layers in between. Commonly, novice programmers are first introduced to a high level programming language: the term 'high-level language' was coined to convey that the relationship between the code and its implementation is linguistically formal, but the representations at either end are quite different.

The relationship between program statements and electrons is one of the dimensions of program comprehension. Each statement is also linked to those before and afterwards, to other functions or methods that it invokes, and to other statements that use or manipulate the same variables. Given the plethora of relationships, none of which is trivial to understand, the designer of the delivery of a teaching scheme for introductory programming needs to select and prioritise.

The 'implementation' dimension (from statement to electrons) cannot be explored to any extent without both sacrificing understanding of the other relationships and introducing a number of other concepts, not least the physics involved. In terms of programming, the 'machine code' layer is the most logical point along the implementation dimension to aim for. This is complicated by the use of virtual machines in languages such as Java or C#. Comprehension of the implementation dimension to this machine layer involves coverage of machine architectures, whether real or virtual. It is therefore necessary to consider how important comprehension is,

and what form it should take within the students' learning experience.

5.2.1. The Rise of Comprehension

In the software industry, the concept of 'green fields' software development has been under considerable pressure for many years from four main sources: the high cost of production, the availability of software components of reasonable quality, systems integration, and the relatively limited progress in developing higher levels of abstraction than the high-level languages which trace their history to the 1960's. One consequence is that comprehending, assessing and modifying software is now a vital skill required by a modern software developer.

This changes the nature of the linkage between creation and comprehension. In a 'green fields'-oriented approach, the linkage between the development of creative and comprehensive skills can be largely sequential: one learns to create, and then reflect and comprehend whether the effects of the created artefact match the requirements. With the majority of modern software development including a substantial 'brown fields' component, comprehension must be developed in parallel: in particular, the novice programmer needs to be able to comprehend software he or she has not written.

When writing software, the author has a notion of the intention that motivates the development. This guides the development and provides the framework for reflection. In addition the author is exposed to the development process over a period of time: his or her 'interface' (time spent in manipulation) with the artefact has time to mature. With unseen software, one needs to infer the intention following a number of deductive steps. Moreover, the interface with the artefact is necessarily much more shallow, hence comprehension represents a significant level of cognitive challenge. Whether this is greater or less than the cognitive challenge of creation is debatable. What is certain is that neither is less than highly complex.

5.2.2. Comprehension of the Implementation Dimension

Debuggers (software used to step through the execution of a program) are a feature of most interactive development environments. Whilst these can show the state and

highlight which statement is being executed, there is no mediating element to illuminate the process.

It seems plausible that an animated, visual representation could assist the learner in connecting the program statements to their consequences. Many research projects have explored this area, from the early beginnings (e.g., Brown and Sedgewick, 1984; Brown, 1988) to ongoing projects including Jeliot (Moreno *et al.*, 2004) and ANIMAL (Rößling *et al.*, 2000). The viewer is presented with at least two simultaneously animated sections: the program code (with highlighting used to identify the 'active' statement), and the 'effects' area, showing statements being executed and the values in storage locations being changed. Further areas are devoted to the program output, for instance. A detailed study of a number of animation projects by Hundhausen *et al.* (2002) found that the results were almost universally disappointing.

The visual language used for the animation has a symbolism that might not be trivially accessible. A more substantial hurdle may be managing one's perception and dividing one's attention between the simultaneous modification of at least two areas of the screen: the highlighting of the statement being executed and the effect (in memory) of the statements. A reasonable level of expertise will be needed to use any of these animation tools; with programming, the programming language, and the tool.

5.2.3. Sample Code in the Curriculum

Most tutors provide code examples that the students use, copy, modify or (possibly) enhance or correct. It is reasonable to suggest that this will be pedagogically desirable, as it provides students with another dimension of experience that will complement those gained in creating the solution from scratch. From a motivational perspective, copying (or typing) an example will both shorten the creative time, and significantly increase the chance that the program will work. If many students have the same example, they can confer and advise (and receive advice from) their peers. As students begin to gain some understanding, the provision of common examples might prove counter-productive, with students either ignoring the supplied code in favour of developing their own applications, or simply copying from each other. A

sense of individualism is important for novice programmers.

5.2.4. An Alternative Dimension: five-dimensional symbology

Symbology is a term devised by Turner (1974) to describe the use of symbols in rituals in different cultures, where those symbols achieve a level of significance beyond their traditional context. The term is used here in essentially the same manner. Most programming languages do not require a particular layout of the code (Python and Occam are two exceptions). Yet most professionals and all textbooks emphasise the importance of using a consistent layout. This layout has two additional facets to the sequential, left-to-right direction of the code itself. Line feeds are used, much as they are in conventional poetry, to convey some degree of discreteness of (or within) elements. In addition, indentation is used as a physical manifestation of the nesting of elements within the code. It is reasonable to suggest that good layout thus aids the speed of comprehension that can then be used to navigate the code both more quickly and with fewer mistakes. The experimental evidence on the reading of poetry indicates that removing the graphic layout of the poem can reduce the recall of the textual elements. Hanauer (1998) manipulated poems by creating two versions: the original ('high-graphic') layout, and the modified ('low-graphic') layout in which the poem was rendered as a piece of prose. Participants were asked to recall textual elements, and those presented with the high-graphic versions tended to recall more of those elements.

The purpose behind the experiment was to compare three theories of comprehension, formalist, stylistic, and conventionalist. The conventionalist viewpoint is that reading conventions provide the main influence on reading; the other theories suggest that the textual content itself is more important. In terms of computer programs, the compiler can be seen as formalist. Hanauer suggests that conventions play a significant part in understanding, hence the greater recall of poems presented graphically.

One can imagine that one explanation is that the two-dimensional structure adds cues that aid recall. It is not unreasonable to extrapolate that the three-dimensional physical symbology in correct program layout will have an even greater effect.

All programming languages have a limited range of 'keywords', none of which are synonyms. In English, one might say 'if...' one minute then 'provided...' another. Such choice is not possible when writing software. Potential confusion can arise if variable naming (the one aspect within the scope of the developer) is inconsistent or contradictory. The 'goodness' of variables names can thus be expressed in terms of congruence with that context. If the names chosen clash with the context, one would expect comprehension to be compromised. The context generally transcends the current application: experienced developers create their own rationale for variable naming, hence creating a fourth dimension of symbology which may impede or facilitate the comprehension of the code.

One complexity for the learner is that many texts use the concepts of implicit property and rationale in variable naming. This produces shorter names, which might have benefits typographically, but which may impede comprehension. For instance, if an application is concerned with finding the average of students' marks in an assignment, suggestions for the name of the collection containing the values might include 'marks' or 'students'. A more accurate name would be 'studentsAssignmentMarks'. This incorporates the properties (mark, assignment) and the object (student). The use of 'marks' implies a symbology where the object and associated properties are gleaned from the context of use, which may be difficult for the learner (or another developer) to comprehend.

The fifth dimension of the symbology is commenting. In Java, one comments statements, methods and classes. Good comments do not echo the action of the program elements: they provide insight into the intention that lies behind the selection and use of those elements.

5.2.5. A Comparison of Comprehension Dimensions

It is tempting to consider the implementation dimension to be of greater importance than the symbology dimension. The high-level language statements do not (of themselves) 'do' anything – they need to be translated into an equivalent sequence of machine instructions first. It is therefore desirable that this dimension of comprehension exists at some level, but this does not, of itself, require that this type of understanding is developed *ab initio,* unless this comprehension plays a more

significant role in software development.

Traditionally, it was necessary to understand the implementation dimension because computer resources (memory and processor) were both expensive and slow. For instance, there was a time when developers might limit the use of collections of data, simply because the computational overhead of calculating the offset of an individual item from the start of the collection.

Nowadays, this justification for the pre-eminence of the implementation dimension has disappeared – one may even suggest that knowledge of this dimension might be undesirable. The complexity of modern systems is such that run-time optimisation might negate the negative impact of using what might seem to be resource-intensive program constructs. In addition, in modern software development, the parsimonious use of computer resources is almost never relevant, compared with satisfying the users requirements.

A final difficulty with the implementation dimension is the relative lack of success associated with the various animation research projects.

The symbology dimension represents a conundrum: syntactically, the punctuation dictates the program structure, not the layout (with the exceptions noted above). If one presents the two as congruent (by always showing the learners well laid out code with meaningful variable names), there is a danger that this distinction will not be understood.

5.2.6. One Solution: JCodeGen

The principle behind JCodeGen is to provide novice learners with unique code elements that they can then manipulate and comprehend. It is a mechanism to help the students to acquire the necessary symbology relevant to good coding practices. In this web-based system, students select from a limited range of options, including whether a code snippet or an entire class is generated, and what form of 'obfuscation' should be applied. Currently the only language supported is Java, although the system could be easily adapted to handle other programming languages. A decision was taken to ensure that the generated code elements contain no syntax errors. The

range of programming concepts used in the code generation increases throughout the delivery of the unit. For the initial weeks, the code generated will execute without error; that may not be the case later on. The student copies and pastes the generated code into a framework document (source file), and manipulates the text using a conventional text editor.

5.2.7. Obfuscation – Layout and Naming

JCodeGen provides three levels of layout obfuscation: none, minor, and major. The 'none' alternative not only provides the student with an ideal layout, but this option is also annotated using HTML highlighting aspects of the generated code. Minor and major layout obfuscation offer different degree of manipulation of whitespace: inserting tabs, spaces and line feeds, and removing syntactically insignificant tabs and line feeds.

Four naming strategies are available in JCodeGen. These are: normal, arbitrary, cryptic, and obfuscated. In normal mode the full names of the variables are provided. With arbitrary naming, the names are randomly derived from the dictionary of words available with Linux systems. Cryptic names consist of only one or two characters, whilst longer sequences of randomly generated names are a feature of obfuscated naming. The use of the three levels of 'meaninglessness' is not just to emphasise the importance of choosing meaningful names, as explained in the rationale later.

In an earlier version, the arbitrary and cryptic modes were not available. Students reported finding the random characters of the obfuscated names made it impossible to understand what the code was doing.

5.2.8. Templates

When presented with the menu, students select the naming and layout options, and identify whether a snippet or a class is to be generated. The generator then randomly chooses a template from a list of those available. The template includes three or four sections: keywords, variables, code, and (optionally) data. The keywords section lists the programming concepts being realised. The next section (variables), defines the configuration of the variables. The data items associated with these variables are

generated, and then used in the processing of the code section. The code section includes optional elements, where one of each of the alternatives is randomly selected. If the code implies the use of additional data items (such as input or random data), these are included in the optional data section.

An external file links the keywords to points within the delivery programme, thus ensuring that code generation does not include features not yet introduced.

The permutations of random selection of template, layout, naming, values of variables and coding elements mean that there is little chance that any code generated will match any other.

Note that each example is separately generated: no access is provided to the original (unmodified) code. This was a deliberate decision, removing the temptation to retrieve the unmodified code and thus short-circuit the de-obfuscation.

5.2.9. The Rationale in More Detail

With good variable names and well laid out code, and the natural tendency of the novice to focus on the more accessible semantic layer, the tendency will be for students to answer the “what does it do?” question with an approximation to the semantic interpretation. This may impede the understanding of the more syntactic level, something which is widely recognised, and which has contributed to the development of the visualisation tools. The intention behind JCodeGen is to provide opportunities for both levels of comprehension to be developed simultaneously, and be able to demonstrate understanding of the two by “de-obfuscating” the generated code back to a well laid out piece of code using sensible variable names.

The first process in de-obfuscation of the code would be to correct the layout. This is useful in terms of learning, not only to reinforce the understanding of the (quite simple) rules of layout, but also to increase the students’ interaction with the three-dimensional symbology (textual layout) of program code.

The second stage of de-obfuscation is to select variable names that make sense in the given context. Many alternatives will be equally plausible, but not all. If the code

involves integers, then 'names' would be a poor choice for the name of a collection, for instance. The final process requires the students to add appropriate comments for each line, and for the methods and classes (typically one method and one class).

Another subtext behind JCodeGen is to increase the number of modifications and hence the time students spend interacting with a particular piece of code. Typically, around fifty spaces and linefeeds would need to be removed, and roughly an equal number would need to be added. In addition, the variable names need to be replaced with more appropriate ones. Finally, as the obfuscated code compiles (and will run) without error, the student knows that any subsequent errors must be due to his or her mistake. It is likely that the whole de-obfuscation process to produce a well laid-out, appropriately named and commented piece of code might take around 30 minutes, even for a moderately expert developer. Within that time, the student will be physically and cognitively continuously interacting with the code.

5.2.10. Additional Motivation

There were additional reasons why this system was developed: design of curriculum content, design of assessment, and plagiarism mitigation.

The curriculum was designed around certain principles: the fundamental programming pattern, and maintenance of student motivation. The fundamental pattern is considered to be: populate 'collection' then iteratively manipulate the collection to produce secondary data. There is therefore a direct relationship between a collection and a loop. The first collection data structure introduced was an array. The commitment to the concept of the fundamental pattern was demonstrated from the outset in that the first program to which the students were introduced was not 'Hello World'; it displayed the contents of an array using a loop. This pattern formed the basis for all the exercises within the unit. One of the objectives was to minimise the redefinitions of problem spaces inherent in more traditional delivery of introductory programming. See Jones (2008) for more information.

Feedback plays a vital role in student motivation, one component of which is assessment of coursework. For those new to programming, assessment can be a daunting prospect, so there is a temptation to delay the first assessment until a

reasonable number of concepts have been introduced. This delay reduces the extent of formal feedback, and can increase, rather than decrease, anxiety. By building all the exercises around the fundamental pattern, an early assignment (after 4 sessions) became feasible, and more assessments could take place within the unit. The students could (and some did) submit generated code for these assessments.

Plagiarism is of considerable concern throughout higher education; none more so than in programming. It is difficult to imagine any introductory programming exercise not having a plethora of alternative solutions readily available and easily accessible, even without the possibility of commissioned code. Various approaches have been taken to mitigate the level of plagiarism, including the use of software (Culwin and Lancaster, 2000), and detecting the incidence of contract cheating (Clarke and Lancaster, 2006). JCodeGen creates individualised solutions that require attention before they can be submitted. This does not, of itself, eliminate plagiarism, but the intention is to provide an alternative avenue for students to participate in programming assessment without necessitating the writing of original code.

5.2.11. Use of Generated Code in Assessed Work

The assessment used in the unit incorporates portfolios, in which the student (electronically) presented applications that together demonstrated certain required programming concepts (such as arrays, loops, etc.). A portfolio could consist of between 1 and 8 applications. JCodeGen generated code could be included, (hence the use of unique identifiers which are recorded in a database). For submission, the code had to be de-obfuscated and fully commented, with 'line' (//) comments for each statement, and JavaDoc comments for the class and each method (the same requirements as for the non-JCodeGen applications). The students were informed that more marks would be awarded for a greater degree of manipulation selected. In fact, all generated (obfuscated) code was treated equally – the statement of differentiation was intended as a bias within the data. No marking differentiation was made between 'original' and generated code. This equality of marking between original and generated code was intended to suggest to the students that, if one is able to transform the obfuscated code, the comprehension involved is commensurate with the creation of an 'original' piece of code.

5.2.12. Experience with the System

Activities associated with code de-obfuscation were used for each of the cohorts. The software itself was rewritten twice, as a result of student feedback. The first version was based on previously submitted student work. The second version used a pseudo-random generation of program statements. The intention was that the generated code should have no underlying semantics. However, students reported that the complete lack of semantics produced a feeling of dissociation. The intention with the revised system was to provide a sliding scale of obfuscation.

Prior to Cohort D, no formal record was maintained over student use of the system, other than in submissions as part of portfolio assignments. For Cohort D, containing a cohort of some 220 students, overall 3,033 code generation requests were made by a total of 216 students. 123 students made 12 or fewer requests. 20 students made 30 or more requests, with the highest number of requests by a single student being 107.

Of the 947 applications submitted in total by a cohort of 85 students for 4 portfolio assignments in Cohort C, 43 (4.54%) were de-obfuscated versions of code that had been generated. In 2008-09, 11.78% (N=176) of the 1,494 applications submitted by 216 students had been generated by JCodeGen. Note that in Cohort D, three programmes joined to share the same introductory programming unit. The extent to which comparisons between the cohorts' use of the generator for assessment is limited not only by the different nature of the cohorts, but also different versions of the generator were used in each of the academic years. Anecdotally, it was the weaker students who chose to use the code generator. Further investigation is underway to gain more insight into the attitudes of students to the system.

A presentation on JCodeGen is available (Jones, 2009).

A simple analysis of all 3,033 requests yielded:

Naming/Layout	None	Minor	Major
Normal	2425 (79.95%)	38	16
Arbitrary	4	9	3
Cryptic	2	5	2
Obfuscated	98 (3.23%)	124 (4.09%)	307 (10.12%)

Table 19 – Profile of Naming and Layout Obfuscation for All JCodeGen Requests

The 9 students with at least 40 requests (19.39% of the total (N=588)):

Naming/Layout	None	Minor	Major
Normal	310 (52.72%)	6	2
Arbitrary	0	0	0
Cryptic	0	1	0
Obfuscated	26 (4.42%)	44 (7.48%)	199 (33.84%)

Table 20 – Profile of Naming and Layout Obfuscation by the Students with the Most Requests

This group accounted for 20.45% (N=36) of the code-generated applications submitted. The implication is that frequency of use is not particularly related to likelihood to submit generated code.

The 207 students with less than 40 requests (80.61% of the total (N=2,445)):

Naming/Layout	None	Minor	Major
Normal	2115 (86.50%)	32	14
Arbitrary	4	9	3
Cryptic	2	4	2
Obfuscated	72 (2.94%)	80 (3.27%)	108 (4.42%)

Table 21 – Profile of Naming and Selection Obfuscation by the Students with the Fewest Requests

Details of 19 of the 176 code generation submissions were unavailable. Of the remaining 157, the distribution was:

Naming/Layout	None	Minor	Major
Normal	16 (10.19%)	14 (8.92%)	3
Arbitrary	1	4	0
Cryptic	0	3	0
Obfuscated	9 (5.73%)	80 (50.96%)	27 (17.20%)

Table 22 – Profile of Naming and Selection Obfuscation in Submitted Applications

These distributions are to be expected. The HTML highlighting where no layout obfuscation was applied would have helped in creating the comments (in other applications) that were a required element in the marking criteria. The lure of more marks for the greater degree of manipulation would tend to favour the obfuscated naming alternative. That those using the generator more would favour this

combination is also consistent – they should have more confidence with the system.

5.2.13. Further Work

A final year student project is currently examining the responses of the students to the system in more detail, and more closely examining whether the system assists the students in becoming more aware of the symbology of the Java programs.

This project will also evaluate the technical aspects of the system and potentially implement some of the following features, as well as examine the potential benefits of using a system such as Jeliot (Moreno *et al*, 2004). There is the possibility that the greater time spent interacting with the code during the de-obfuscation process might aid the development of the appropriate mental models.

The structure of the templates is both awkward and restricting. A revised formal grammar will be developed which allows for more interpretation with regard to the way in which elements within the template are modified. In particular, greater choice between statement types will be included, as well as choice within the statements themselves.

It is the intention that the students engage in an evaluation between generated code segments with regard to a given problem. Students will be set a problem, and supplied with a range of generated code segments. The exercise will involve selecting one of the segments and (potentially) modifying it to solve the required problem.

5.2.14. Summary

The style of program code, including layout, variable naming and commenting has long been considered important. Most Interactive Development Environments (IDEs) and a number of editors and other tools provide 'pretty printing' capabilities to enforce particular layout principles. This availability, coupled with empirical evidence from comprehension in reading, suggests that students learning of programming will be enhanced if they acquire the appropriate skills in styling. It is also suggested that this 'dimension' of program comprehension is becoming more important, with the transition from 'green' to 'brown' fields in many areas within the

software industry.

The principle that code generation can be used within an introductory programming unit to provide styling activities has been demonstrated. The use of the system both for exploration and assessment shows that many students perceive some benefits from using the system. It is also the case that the present system, based on templates rather than random generation of program elements, is more accessible. Further analysis is underway to ascertain the nature of these perceived benefits in more depth.

5.3. The Simple Development Environment

The intention behind the production of this library was to simplify the production of the framework of software applications in a manner that was comprehensible to novices, and in line with best practice in the software industry. Integrated Development Environments (IDEs) offer many productivity benefits, but the mechanisms are not available for examination, even where they are straightforward. Expecting novices to structure their applications as a professional would is too optimistic, given the many other challenges in learning to program. The library provides a middle way.

The main elements of the simple development environment are:

1. *A batch file to set up the environment appropriately*. This batch file also opens the relevant windows, and a tutorial is available to assist the students in replicating the environment on their own computers.

2. *An application to create and project structure*. This application creates the relevant directories, batch files, and a skeleton application class.

3. *An application to create (business) classes*. The user (student) enters the names and types of attributes, and the application creates the corresponding business class. Apart from saving on typing and allowing a much earlier introduction of classes than would otherwise be the case, the classes created can be considered from in a more academic context. In particular, the notion

of ‘over-engineering’ can be considered.

4. *An application to generate sample data*. This application creates a data file based on the attributes in a given business class. Apart from saving on typing, this application enables a greater focus on testing than would be the case otherwise. This facilitates a number of academic discussions, including performance analysis of applications (for large datasets) and the strategies for the generation of the data in order to cover various combinations.

These applications are provided in a library that the students install in a particular location. Installing libraries is also something that professional developers do on a regular basis. The library includes a large number of other features, including the capability to read data from Excel spreadsheets, speech to text, and simple graphics.

A central element in the library is that novices should be able to comprehend how the features work, even if that comprehension is not necessary in order to use the feature. For instance, reading an entire file into a array is a feature that students use in the first 5 sessions. Less than 10 lines of code are used in this feature, so this can be opened up for study later in the delivery of the unit. There is a pedagogical justification behind this: learning in programming must be top-down as well as bottom-up, and students must become comfortable with both. Developing a capability to understand enough to use something without being concerned as to the detail of its function is a vital intellectual skill in programming.

5.4. The Electronic Assessment System (EAS)

The EAS system consists of the following components:

1. *Online (portfolio) assignment submission.* The submissions are fully validated, which requires the students to structure their work precisely in a format consistent with best programming practice. Multiple submissions are allowed, with the last submitted prior to the deadline being assessed.

2. *Automatic compilation of the submitted work*. A manual process then follows where the problems are ‘fixed’. All such modifications are recorded.

3. *Production of a file ready for demonstration*. All the (now syntactically) correct applications are aggregated into demonstration files – one per student. These are downloaded and used in the demonstration phase, allowing rapid focus on key aspects of the assignment. The marks for the demonstration are written, collected, and entered into the system.

4. *Production of feedback*. The demonstration marks, together with the automated assessment of the submitted (corrected) files, are used to calculate the marks, and create a feedback document for the students (and one overall document for academic records). The feedback is then verified and audited by another lecturer, before being uploaded ready for the students to collect. At this time the marks have to be copied manually into the university’s student record system, but this is likely to be automated in the near future.

The feedback document includes the use of stock phrases that are selected from a bank of comments, and the ability to aggregate points into stylised paragraphs. This aspect of marking has received considerable attention, and many systems exist (e.g., Waypoint XXX). These generally require manual association of phrases with the feedback for an individual student. The EAS allows the tutor to include a paragraph specific to a given student, and produces feedback much more quickly, due to the automatic selection of phrases using the mark allocated. It was estimated that 30 hours of staff time would be needed to process and mark the submissions of 200 hundred students, and produce a detailed feedback document with more than 20 individual feedback elements generated from a consideration of every aspect of the work submitted. Of those 30 hours, most was spent in supervising the student demonstrations, thus providing opportunities for more informal feedback.

Despite the introduction of marking assistance software in the VLE, it is likely that the EAS will continue to be used, because of the ease of use and the level of integration between the various components.

The phrases, marks and printing information are all held in a spreadsheet. All the marks for each criterion for all the students can be viewed readily, and secondary analyses are easily created. When creating the framework for another assignment, the spreadsheet can be copied and then modified.

6. Reflections on the Research

Research is an on-going process – no matter what has been demonstrated, the method can be improved, more can always be discovered, or the results can be validated. All of these are true in this study.

6.1. The Pedagogical Objectives and Achievements

The main objectives of the research were to measure, and gain understanding of the process of pedagogical transformation in an introductory programming unit. To that end, the approach selected was the conducting of a holistic, longitudinal study covering all the most obvious ephemera of all the students in introductory programming in four successive cohorts. The first level of this approach was achieved with data gathered and analyses performed on more than 4,500 Java source files and 1,000 tests. The analyses showed that the range of programming concepts used increased and the volume of work submitted grew during the academic year. Both would be expected in any learning environment, but these were achieved in the context of a paradigm shift in the pedagogy. A regime where concepts were introduced according to their importance in programming, replaced one where the concepts were introduced in a sequence dictated by the lecturer's perception of their pedagogical complexity. This radical shift was achieved in an evolutionary manner by progressively introducing elements of the new paradigm over the period of the study, despite the unanticipated disruption of a complete revalidation of all computing-related programmes.

This shift also permitted students to explore their interests in programming at a much earlier stage, an opportunity that many took up in designing their own applications around the realisation of a specified set of required concepts. This specification of required concepts, rather than the traditional functional specification, was seen as a key element in the paradigm shift. It built on the assumption that there would always be a natural interest in the functional elements of a program, i.e., in 'creating a program that works'. The students were required to submit an additional document stating where certain concepts were realised. Initially, not all students engaged in this activity, but, by the second portfolio assignment, almost all students submitted a

reasonably complete set of statements linking the required concepts to particular lines of program code.

The real purpose in introducing this element into the pedagogy was to enable students to engage in a much richer evaluation when planning their own applications. The fascination of creating a working computer program is enhanced considerably where one is the originator, but there is a danger that novices will be unable to grasp the consequences of their ideas, or that they will explore areas that are only tenuously related to the fundamental concepts of programming. An example of the latter phenomenon was experienced by the author some years ago when a student created a program that created a picture of a famous person simply by a long sequence of output statements.

A focus on the underlying (required) concepts has benefits in addition to providing a framework for their creativity whilst ensuring that particular concepts are covered. All creativity is subject to external pressures, often expressed in terms of resource limitations or functional requirements. Unfettered creativity might seem ideal for motivation, but is not feasible in any environment, so it would be unrealistic and unhelpful to the students to allow them that freedom.

The study examined three different facets of learning to program: comprehension, modification, and construction. Modification was introduced in the form of an obfuscator, which both generated code, and then obfuscated one or more elements, such as layout or naming. This was not wholly successful, and two main faults were identified: the code lacked sufficient semantics and it was introduced before students had a firm grasp of the fundamental concepts of programming. The approach was then modified to focus on comprehension and modification, and to maintain key elements of the semantics.

The paradigm shift and the innovations would not have been feasible without the introduction and development of a comprehensive submission, assessment, and feedback system. Students submitted their portfolios online, and received electronic copies of feedback within a short timescale. Likewise, online tests replaced written ones (in the main), with immediate production of feedback. Students were able to

practice for the tests, which most students did. The effectiveness of the assessment system facilitated a doubling of the number of assessments and an increase in the quality of feedback, and more rapid production of feedback, all whilst reducing the overall marking load.

The net effect of these innovations was to empower both the students and the staff. The students received more feedback and were able to plot their progress more effectively. The greater number of assessments permitted the use of a rubric that selected the subset of portfolios with the highest marks. This meant that, by halfway through the delivery, most of the students achieved a pass level in the coursework component of the assessment of the unit. This added to the sense of empowerment, with students being free to decide where they should focus their efforts. In the event, most were not satisfied with a bare pass, and continued to work on their programming.

The assessment system included some automatic analysis of the work submitted, which then allowed a more focussed and in-depth evaluation of the work submitted. This provided a richer picture of student progress, and enabled the unit leader (the author) to feel confident that the vast majority of the students were engaging in the learning programming as individuals. In the context of a large number of people working on similar activities in an electronic environment, one can never know the extent of collusion, but it is reasonable to suggest that empowered students may not feel the need to copy from their peers to the same extent as those with less sense of their own progress. Further study in this area would be required.

Between-cohort studies also showed that, as each element in the paradigm shift was introduced, the level of student engagement increased. This was true of each cohort except the last, where a hybrid regime was used, necessitated by the amalgamation of three programmes, and two teaching teams.

All domains have their associated languages, and it may be a truism to suggest that all such languages contain contradictory elements that can confuse or confound the novice. That, combined with the challenges inherent in investigating such a complex area, might have suggested that this study avoid this area, were it not for its all-

embracing effect on the learning of programming. Rather than attempt an in-depth examination of the semiotics of programming, the approach taken was more of a scoping exercise – attempting to delineate the more tractable areas, as well as gain some insight into the learning issues involved.

Two aspects on the semiotics were identified: presentational and lexicographical. An analysis of a traditional delivery regime highlighted faults in the presentational semantics, notably the use of advanced concepts without explanation, and the contradiction of earlier statements. The learning regime was designed around what was identified as the fundamental programming pattern, with each stage exploring an aspect in more detail. This approach avoided the pitfalls of the traditional regime, providing the students with a consistent, rather than an episodic, view of programming.

The contradictions in the lexicographical semantics and semiotics are much less straightforward to obviate, largely because, in the case of programming languages, most design decisions were made out of necessity, linked to the limitations of the computer, or designed for an environment very different from the one in which it is currently being used.

Programming is like speaking almost entirely in idioms. To a novice, a program might seem like a sequence of cockney rhymes, where the ‘rhymes’ have been omitted to confound any analogical reasoning. Indeed, the design of many programming languages is frequently an object lesson in how to negate opportunities for analogical reasoning. The assignment statement (e.g., A = B) is but one example.

Analysis of the semiotics of programming, including the language itself and the way it is presented in the context of learning, serves to remind all those responsible for the learning process of the intellectual complexity of the task, as well as highlighting those elements that are most likely to impede learning. This can inform and reform the delivery and assessment regimes, as has been demonstrated in the course of these four cohorts.

Traditional learning regimes tacitly recognised this, and were designed to introduce

programming concepts according to their perceived complexity (as has been stated before). This understandable, but misguided, approach had two main drawbacks. Firstly, no analysis underpinned the allocation of a level of complexity to a given concept. Were one done, a tutor would readily see that programming is a self-referential system with no ‘atomic’ concepts. Secondly, there was a significant effect in the use of manufactured, unrealistic examples that had the effect of subverting the comprehensibility of the presentational semantics. For example: writing a program to find the average of a set of numbers leads to a double dissociation: from the semantics of the numbers, and from the appropriateness of the vehicle chosen. Why not use a spreadsheet? This use of artificial or inappropriate examples resulted in students having difficulty in learning to program, but many tutors assumed was linked with the complexity of the concepts. In turn, this style of thinking leads to the design of learning objects (e.g., McGreal, 2004) that aim to assist the assimilation of specific concepts. Tutors have tried different programming languages, or sub-domains of programming (e.g., graphics), (seemingly) rather than more closely examine the semiotics and pedagogy.

Challenging these notions was the original motivation of this pedagogical transformation and for its associated study. By conducting a holistic analysis of the students’ performance over a number of cohorts, the author aimed to validate this paradigm shift, and provide a large volume of evidence that may influence more traditional tutors.

6.2. The Research Approach

In the main, quantitative instruments were used in the study in order to perform some level of analysis on the learning of all of the students as this progressed through the academic year. This analysis extended from the work submitted, to include initial surveys and an intermediate one. Questionnaire-based surveys can be limited. Their purpose here was to provide an overall picture of students as they arrived at the start of the unit, and how they felt their learning was progressing. The general picture of each incoming cohort was one of many, if not most, students with very little prior understanding of programming. This emphasised an additional dimension of the complexity of designing a learning regime, as the students’ prior

experience and aspirations formed a continuum, with some students knowing they wanted to become software developers based on their programming experience, sitting alongside those with no knowledge at all, and who may be considered unlikely to aspire to a similar career path.

One of the main limitations of a quantitative approach is that it generally has to assume levels of motivation, comprehension, causality and accuracy. One cannot easily establish the motivation behind a participant's actions, whether he or she understands what is being asked of him or her, what caused the resulting effects, or whether the data is complete or accurate.

Where more than one instrument is used, one can begin to triangulate data, in order to gain a more detailed view. The comparatively low correlations between professed prior programming experience and eventual achievement increase the sense of care that needs to be taken with questionnaire results, as well as highlighting potential limitations in the quality and quantity of programming in courses that aim to support students entering higher education.

A limited experiment with semi-structured interviews was conducted that provided useful insights into the limitations of the initial survey with regard to the prior computing experiences of the students. Reasons behind reactions were also highlighted, for instance in connection with the obfuscator. The aim was to capture the students' views *in loco*, whilst engaged in the programming experience. Their utterances on their current tasks were less reflective than when they were asked about their prior experiences, as one would hope and expect. As the interviewees were free to discuss their interviews with other participants, another level of potential bias was added to those of the tutor as interviewer, researcher, and assessor.

Insight into and experience of the interviewing processes were gained, which showed that these provided a different view of the four 'horsemen' of participants' attitudes: motivation, comprehension, causality and accuracy. The author was not convinced of the advantages of one-to-one interviews conducted under such conditions with a few students, as compared with questionnaire survey with the whole cohort. Neither is ideal, and both have significant limitations.

6.3. Recommendations for the Design and Delivery of an Introductory Programming Unit

There are a number of factors that influence the success of the delivery of an introductory programming unit. Chief among these will be the embracing discipline (computing or computer science) and the content of other units. These recommendations are intended to apply equally to computing and computer science programmes where ‘programming’ is covered entirely in a single unit, and there is not a separate ‘algorithms’ unit.

1. *Recognise the Main Characteristics of Learning to Program*. These include domain complexity and individualism. Thresholds of capability in multiple dimensions within the domain of programming must be achieved before one can write non-trivial programs. These dimensions are: macro- and micro-problem solving, the development environment, and the syntax and semantics of the programming language. Each dimension is multi-faceted and contains specific pieces of information that limit or confound the extent to which analogical reasoning can be employed.

   No other discipline has been shown to be a reliable indicator of how challenging an individual will find learning to program. The consequence is that the more individual the learning regime, the greater the likelihood of student progress, provided that each student can measure his or her progress.

2. *Select a Genre of Application*. Each application genre is associated with a different interpretation of the term ‘elegance’. An elegant program is one that uses no more statements than is necessary to solve the problem. The elements that contribute to elegance will permeate the design of the pedagogy. The genre selected will determine the pedagogy and will influence the selection of programming language and software tools.

   The genre selected should reflect the overall focus of the programme, and not be chosen for any perceived short-term increase in student motivation. The rationale is that the focus of the programme will permeate the presentation of

other units and provide more long-term support for programming.

3. *Remember that Programming Concepts form a Network*. The programming concepts appropriate to a genre will form a network, not a tree. The tutor must design a logical path to follow within the network, but need not be hidebound by any notions of inherent sequential dependencies.

   This notion of a network of programming concepts will assist the learner, as the network presents multiple opportunities for the learner to grasp a concept. Learners should not be deprived of these opportunities in a misguided attempt to 'simplify' programming. Any simplification will resonate with some students, and alienate others, increasing the likelihood of a bi-modal profile of student performance.

4. *Present a Realistic Universe*. In a data processing application, loops are used to process collections of data items. It is therefore unrealistic to separate the presentation of loops and data collections. There is a danger that serial presentation of coincident concepts will foster the development of inappropriate learning strategies forcing some students to 'unlearn' when the realistic 'universe' of coincident loops and collections is introduced.

5. *Maintain the Pre-eminence of the Pedagogy*. Programming languages (in particular) have idiosyncrasies, and it is tempting to include these in designing the unit delivery and assessment. This should be resisted, as the intention must be to focus on programming, with the language as servant, not the master. Where necessary, libraries and tools should be employed, provided the motivation is to maintain the pedagogy, not to simplify programming.

6. *Identify General Measures of Progress*. These will affect how student progress will be measured. One dimension can be characterised as 'comprehension-modification-construction'. The growing importance of using existing code in new applications means that each of these elements is as important as the others. This should be communicated to the students otherwise they will see construction as pre-eminent.

The range of programming concepts realised is another dimension that can be measured with minimal effort, provided the necessary software is available. Another general measure is the sophistication of the combination of programming concepts. In the early stages of learning to program it will be important to minimise nesting concepts wherever possible in order to limit the intellectual demands being placed on the student.

7. *Identify Specific Measures of Progress*. These will be linked to the genre. All non-trivial programs involve multiple manipulations of data. In the data processing genre each manipulation tends to be unique, with the result that a program with 5 manipulations will generally be larger than one with two manipulations. Therefore program size (in the data processing genre) will tend to be an indicator of sophistication.

8. *Include Multiple Distinct Classes of Progress Measurement.* This may seem counter-intuitive, given the intellectual complexity of learning to program. The key word is 'distinct'. In addition to comprehension, modification and construction and the range of programming concepts, multiple-choice questions, and program style were identified as measures of progress within the delivery and assessment regimes. Each class involves a distinctive, complementary combination of intellectual processes. It is likely that a cohort will contain sub-groups, each of which will find one class more tractable than the others.

9. *Make the Classes of Measure Visible*. The classes must be visible to the students and their nature and significance explained. One cannot expect all students to understand immediately every aspect of each class. By outlining the landscape one is providing a realistic view of the complexity of learning to program.

10. *Emphasise the Importance of Any Progress*. The pedagogical approach must present each class as important and emphasise progress and de-emphasise lack of progress. This can be achieved by structuring workshops around multiple threads, where threads correspond to classes of progress. Provided this is the case, students will always be able to follow a thread they find more comprehensible, should they struggle in one of the other threads. In this way the

use of multiple classes will optimise the proportion of students who feel that they are (in some sense) learning to program.

The other essential element is coherence. The complexities of programming need not be hidden from the students provide the message is clear and consistent.

6.4. The Future

The research will continue with the final phase of the paradigm shift, the early introduction of classes, featuring in the 2009-10 delivery of the unit. Further developments of the analysis software are planned, with a version being included in the library supplied to the students at some point in the same cohort. The code generator will be developed, and integrated with the online tests, facilitating a more formal examination of students' comprehension of program code.

7. Learning to Program: the future

Information technology has grown steadily in importance over the past half century to the point where it is now crucial in many aspects of business around the globe. The proliferation of software genres has both been a result of this development and a driver in its progress. The graphical user interfaces of the client/server systems of the 1980s and 1990s have been superseded by web-based applications that are now being replaced with rich Internet applications (RIAs) and applications for mobile devices. These ‘front office’ technologies connect to server applications that are far more integrated than their predecessors, so the capability demanded of software continues to expand in all directions.

One could suggest that, although the technology has advanced, the underlying conceptual framework of software remains relatively stable. Data is still read, processed, and output. This is an over-simplification, as the architecture of software systems is radically changing due to the mismatch between the growth in the demand and diversification of software and the development of more sophisticated software development techniques and tools. Basically, the latter are failing to progress at sufficient speed, requiring a fundamental change in how software is created.

The production of modern software is only viable if much use is made of existing software. Software libraries have always been used in this regard, but these have now been augmented by frameworks and (particularly) by connecting with other software systems. The role of the software developer is largely being commuted into that of a systems integrator and coordinator. The skill set required relies more on comprehension and communication than on algorithm creation. The communication (with others) is a necessary by-product of the proliferation of software technologies. The complexity of these technologies is such that a developer will tend to specialise in a limited number, requiring collaboration with others where (as is normally the case) multiple technologies are required in a given system.

This pattern shows every sign of continuing for the foreseeable future. One is therefore left to ponder whether changes need to be made in the education of software developers and others within (or aiming to be part of) the software industry.

One additional complication is that it would be impractical to create a new programming language as there is a gigantic effort and/or cost in generating the associated paraphernalia, such as libraries, development aids, not to mention a sufficiently large and well-trained workforce capable of exploiting the new language. No language in common use today is less than 15 years old, and almost all those in use are based on much earlier languages.

7.1. The Effect on Programming Education

There are three possible approaches to meeting the demands of modern economies and businesses, which can be characterised in terms of length, width, and depth.

(Undergraduate) programmes could be lengthened to facilitate the consideration of the modern challenges in software development, as well as the fundamentals currently studied. Alternatively, more specialist programmes could be created that focus on the programming aspects of the software processes. Neither of these is viable from several points of view. Higher education is expensive, and both of these pathways would exacerbate the cost. The first would simply add a multiplier, whilst the second would reverse some of the economies of scale currently employed to manage the growth of higher education.

The third alternative is to be more selective in what is studied. This is only feasible if there are support mechanisms to offset the likely diminution of the conceptual area begin covered. Two such mechanisms can be identified: computing power and the obsolescence of techniques and tools. Modern computers are far more powerful and much cheaper than their predecessors, a trend that is set to continue for a little while yet. Also, with the advent of new technologies, one sees the importance of others reduce.

Ideally, these mechanisms would compensate entirely for the growth in demand and diversity obviating the need for substantial changes in pedagogy. Sadly, this is not the case, mainly due to the continued presence of the same programming languages. The result is that the pedagogy must change to reflect the increased importance given to comprehension and integration, at the expense of algorithm creation.

Integration is the key element within this pairing. One can manage increasing complexity if there exist partial solutions that can be selected and integrated to produce the solution required. This depends on gathering the relevant information, understanding the goal, evaluating potential partial solutions (existing systems, technologies, frameworks and libraries), and then creating the integrated solution, probably using some bespoke software as ‘glue’.

Comprehension is therefore needed in several levels. There is the ‘comprehension for use’ that largely depends on reading the documentation (and consulting other sources). Should some adaption be needed, or additional bespoke elements created, then ‘comprehension for modification’ will become necessary. Deeper levels may be appropriate where, for instance, novel hardware is involved.

This is essentially a ‘top down’ environment, where the main strategy is to minimise the exploration of a potentially useful partial solution wherever possible. This runs contrary to traditional scientific teaching, and can lead to accusations of trivialising the subject.

7.2. The Readiness of the Current Approach

The paradigm shift that forms the central theme being studied in this research has moved the initial teaching of programming in the direction of more emphasis on comprehension. The use of a code generator and the electronic assignment system together emphasise another underlying principle, that of utilising software systems wherever possible and appropriate. The supplied library contains applications that simplify the building of classes and applications, without hiding the mechanisms employed. Tests are used to highlight the importance of accruing specific knowledge, as well as allowing the main bulk of the delivery to focus more on general principles.

The current approach remains linked with procedural desktop applications. Event-driven systems, such as those used in mobile applications and graphical user interfaces, require fundamentally different application architectures. At present, these are covered in a later unit, but that unit is under pressure to include a wider diversity of web technologies, so one cannot rule out the possibility of revisiting the

orientation of the introductory unit.

7.3. The Role of Research

The current study has shown that it is possible to change the pedagogy and domain focus to enhance students' learning without detriment to any sub-group. It has been shown that conducting a holistic study is appropriate and feasible, provided the relevant software systems are available. The research approach and tools form a basis on which further study can be carried out.

7.4. In Conclusion

This study has enabled the author and others to gain a more detailed perspective of students' learning as they grapple with the complexities and vagaries of programming, The rigour of the research approach has brought another level of discipline into all aspects of the design and implementation of delivery and assessment regimes. There is now a platform for further development, both within the current programme and institution, and with the wider academic community. The author foresees this as opening a number of opportunities, based on the conduct and consequences of this study. To paraphrase Winston Churchill: this is not the end, or the beginning of the end, but it is the end of the beginning.

References

ACM, 2008. Undergraduate Computing Curricula, Available at: http://www.acm.org/education/ComputerScience2008.pdf. Last viewed: 20 July 2009.

Apple, 2008. Xcode, Available at: http://developer.apple.com/tools/xcode/) Last viewed: 10 August 2008.

Ausubel, D. P., 1960. The use of advance organizers in the learning and retention of meaningful verbal material. *Journal of Educational Psychology*, 51 (5), 267-272.

Barnes, D. and Kölling, M., 2002. *Java and BlueJ*, Harlow, Essex, UK: Prentice-Hall.

Bergin, S. and Reilly, R., 2005. Programming: factors that influence success. *ACM SIGCSE Bulletin*, 37 (1), 411-415.

Blevins-Knabe, B., 1992. The ethics of dual relationships in higher education. *Ethics and Behavior*, 2 (3), 151-163.

Bloom, B. S., 1965. *The Taxonomy of Educational Objectives (Handbook 1),* Harlow, UK: Longman.

Bloom, B. S. E., Englelhart, M. D., Furst, E. J., Hill, W. H. and Krathwohl, D. R., 1956. *Taxonomy of educational objectives: The classification of educational goals. Handbook I: Cognitive domain,* New York, New York, USA: David McKay.

British Psychological Society, 2005. Code of Conduct for Psychologists, Available at: http://www.bps.org.uk/the-society/ethics-rules-charter-code-of-conduct/code-of-conduct/a-code-of-conduct-for-psychologists.cfm Last viewed: 19 February 2006.

Brown, M. H. and Sedgewick, R., 1984. A system for algorithm animation. *ACM SIGGRAPH Computer Graphics*, 18 (3), 177-186.

Brown, M. H., 1988. Perspectives on algorithm animation. In O'Hare, J. J., (Ed.) *Proceedings of the SIGCHI conference on Human factors in computing systems*, Washington, D.C., United States, May 15-19 1988, 33-38.

Brunkhorst, H., 2009. The transformation of solidarity and the enduring impact of monotheism. *Philosophy & Social Criticism*, 35 (1-2), 93-103.

Byrne, P. and Lyons, G., 2001. The effect of student attributes on success in programming. *ACM SIGCSE Bulletin*, 33 (3), 49-52.

Caspersen, M. E., Kaspar Dalgaard, L. and Bennedsen, J., 2007. Mental models and programming aptitude. *ACM SIGCSE Bulletin*, 39 (3), 206-210.

Cavaye, A. L. M., 1996. Case study research: a multi-faceted research approach for IS. *Information Systems Journal*, 6 (3), 227-242.

Chen, X., Francia, B., Li, M., McKinnon, B. and Seker, A., 2004. Shared Information and Program Plagiarism Detection. *IEEE Transactions on Information Theory*, 50 (7), 1545-1551.

Chisholm, R. M., 1967. Brentano on Descriptive Psychology. In Lee, E. N. & Mandelbaum, M. (Eds.) *Phenomenology & Existentialism*. Baltimore, Maryland, USA: The Johns Hopkins Press.

Christians, C. G., 2005. Ethics and Politics. In Denzin, N. K. & Lincoln, Y. S. (Eds.) *The Sage Handbook of Qualitative Research*. 3rd Revised Edition. Thousand Oaks, California, USA: SAGE Publications.

Clarke, R. and Lancaster, T., 2006. Eliminating the Successor to Plagiarism? Identifying the use of contract cheating sites. In *International Plagiarism Conference*, Gateshead, UK, 19-21 June 2006. JISC.

Clawson, E. U. and Barnes, B. R., 1973. The Effects of Organizers on the Learning of Structured Anthropology Materials in the Elementary Grades. *The Journal of Experimental Education*, 42 (1), 11-15.

comScore, 2009. Nine Out of Ten 25-34 Year Old U.K. Internet Users Visited a Social Networking Site in May 2009, Available at: http://www.comscore.com/Press_Events/Press_Releases/2009/7/Nine_Out_of_Ten_25-34_Year_Old_U.K._Internet_Users_Visited_a_Social_Networking_Site_in_May_2009. Last viewed: 10 January 2010.

Cooper, S., Dann, W. and Pausch, R., 2000. Alice: a 3-D tool for introductory programming concepts. *Journal of Computing in Small Colleges*, 15 (5), 107-116.

Cross, J. H. and Hendrix, T. D., 2007. jGRASP: an integrated development environment with visualizations for teaching Java in CS1, CS2, and beyond. *Journal of Computing in Small Colleges*, 23 (1), 5-7.

Culwin, F. and Lancaster, T., 2000. A Review of Electronic Services for Plagiarism Detection in Students Submissions In The *1st LTSN-ICS Conference on the Teaching of Computing*, Edinburgh, UK, 28-31 August 2000.

Davis, H. C., Carr, L. A., Cooke, E. C. and White, S. A., 2001. Managing Diversity: Experience Teaching Programming Principles. In *The 2nd LTSN-ICS Annual Conference*, London, UK, 28-30 August 2001.

Dehnadi, S., 2006. Testing Programming Aptitude. In *18th Annual Workshop of the Psychology of Programming Interest Group*, Brighton, UK, 7-8 September.

Dehnadi, S., Bornat, R. and Adams, R., 2009. Meta-analysis of the effect of consistency on success in early learning of programming. In *21st Annual Workshop of the Psychology of Programming Interest Group*, University of Limerick, Ireland, 24-26 June.

Dewey, J., 1966, originally 1916. *Democracy and Education*, New York: Free Press, Macmillan.

dictionary.com, 2009a. Definition of geek, Available at: http://dictionary.reference.com/browse/geek. Last viewed: 12 March 2009.

dictionary.com, 2009b. Definition of nerd, Available at: http://dictionary.reference.com/browse/nerd. Last viewed: 12 March 2009.

e-skills, 2005. Quarterly Bulletin - Q1 2005, Available at: http://www.e-skills.com/cgi-bin/go.pl/online-services/index.html?docurl=http%3A%2F%2Fwww.e-skills.com%2Fcgi-bin%2Forad.pl%2F261%2Fbulletin12_Q1_2005.pdf. Last viewed: 17 December 2008.

Fincher, S., 2005. A Disciplinary Commons in ITP, Available at: http://www.cs.kent.ac.uk/people/staff/saf/dc/. Last viewed: 19 March 2009.

Fincher, S., Bibby, P., Bown, J., Bush, V., Campbell, P., Cutts, Q., Jamieson, S., Jenkins, T., Jones, M., Kazakov, D., Lancaster, T., Ratcliffe, M., Seisenberger, M., Shinners-Kennedy, D., Wagstaff, C., White, L. and Whyley, C., 2006. Some Good Ideas from the Disciplinary Commons. In 7th Annual HEA-ICS Conference on the Teaching of Computing, Trinity College, Dublin, Ireland, 29 August - 1 September.

Fleming, H., 2004. Peer assisted learning: experienced students facilitating interactive study sessions. In: *Retention in Higher Education Conference: innovative practice to support a diverse student base*, London, UK, 13 May 2004.

Fowler Jr., F. J., 1997. Design and Evaluation of Survey Questions. In Bickman, L. & Rog, D. J. (Eds.) *Handbook of Applied Social Research Methods*. Thousand Oaks, California, USA: SAGE Publications.

Flyvbjerg, B., 2006. Five Misunderstandings About Case-Study Research. *Qualitative Inquiry*, 12 (2), 219-245.

Ganapathi, M. and Fischer, C. N., 1985. Affix grammar driven code generation. ACM Transactions on Programming Language Systems, 7 (4), 560-599.

Grandin, T., Duffy, K., 2008. *Developing Talents: Careers for Individuals with Asperger Syndrome and High-Functioning Autism (Updated and Expanded Edition)*, Shawnee Mission, Kansas, USA: Autism Asperger Publishing Company.

Green, T. R. G., 1989. Cognitive Dimensions of Notations. In Sutcliffe, A. G. & Macaulay, L. (Eds.) *People and Computers V*. Cambridge, UK: Cambridge University Press.

Guba, E. S. and Lincoln, Y. S., 1994. Competing Paradigms in Qualitative Research. In Denzin, N. K. & Lincoln, Y. S. (Eds.) *Handbook of Qualitative Research*. Thousand Oaks, California, USA: Sage Publications.

Gurwitsch, A., 1967. Husserl's Theory of the Intentionality of Consciousness in Historical Perspective. In Lee, E. N. & Mandelbaum, M. (Eds.) *Phenomenology & Existentialism*. Baltimore, Maryland, USA: The Johns Hopkins Press.

Halasz, F. and Moran, T. P., 1982. Analogy considered harmful. In: *Proceedings of the 1982 conference on Human factors in computing systems*, Gaithersburg, Maryland, United States. March 15 - 17, 1982, 383-386.

Hale, C., 1998. *Research Ethics: Standards of Care*, London, UK: RCN Publishing Company.

Hanauer, D., 1998. Reading Poetry: An Empirical Investigation of Formalist, Stylistic and Conventionalist Claims. *Poetics Today*, 19 (4), 565-580.

Heinrich, D., 1999. What is Metaphysics - What is Modernity? Twelve Themes against Jürgen Habermas. In Dews, P. (Ed.) *Habermas: A Critical Reader*. Malden, Massachusetts, USA: Blackwell.

HEA-ICS, 2008. Reusable Learning Objects, Available at: http://www.ics.heacademy.ac.uk/resources/rlos/index.php. Last viewed: 10 February 2010.

Honneth, A., 1999. The Social Dynamics of Disrespect:Situating Critical Theory today. In Dews, P. (Ed.) *Habermas: A Critical Reader*. Malden, Massachusetts, USA: Blackwell.

Huet, I., Pacheco, O. R., Tavares, J. and Weir, G., 2004. New challenges in teaching introductory programming courses: a case study. In: *34th Annual Conference in the Frontiers in Education (FIE 2004)*, Savannah, Georgia, USA: 20 – 23 October 2004, T2H/5-T2H/9.

Hundhausen, C. D., Douglas, S. A. and Stasko, J. T., 2002. A Meta-Study of Algorithm Visualization Effectiveness. *Journal of Visual Languages & Computing*, 13 (3), 259-290.

Husserl, E., 1973. *Cartesian Meditations: an introduction to phenomenology*, Den Haag, Netherlands: Martinus Nijhoff.

Hutchings, P. and Shulman, L. S., 1999. The Scholarship of Teaching: New Elaborations, New Developments. *Change*, 31 (5), 11-15.

Hutchings, P., 1998. The Course Portfolio: How Faculty Can Examine Their Teaching to Advance Practice and Improve Student Learning, Washington D.C.: American Association for Higher Education.

Jadalla, A. and Elnagar, A., 2008. PDE4Java: Plagiarism Detection Engine for Java source code: a clustering approach. *International Journal of Business Intelligence and Data Mining*, 3 (2), 121-135.

JCQ, 2006. A, AS, VCE, AEA Results Summer 2006.

JCQ, 2009a. GCSE, Applied GCSE and Entry Level Certificate Results Summer 2009.

JCQ, 2009b. A, AS and AEA Results Summer 2009.

Jenkins, T. and Davy, J.R.., 2000. Dealing with Diversity in Introductory Programming. In: *The 1st LTSN-ICS Annual Conference*, University of Edinburgh, August 29 - 31 2000, Edinburgh, UK.

Jenkins, T., 2001. The motivation of students of programming. *ACM SIGCSE Bulletin*, 33 (3), 53-56.

Jenkins, T., 2002. On the difficulty of learning to program. In: *3rd Annual LTSN-ICS Conference*, Loughborough University, UK, 28 - 30 August, 2002.

Jones, M., Main, A. and Polkinghorne, M., 2003. SPOCE KTP: Final Report, Bournemouth, UK: Bournemouth University.

Jones, M., 2007. The redesign of the delivery of an introductory programming unit. *Italics*, 6 (4). Available at: http://www.ics.heacademy.ac.uk/italics/vol6iss4/MJones.pdf. Last viewed: 14 May 2009.

Jones, M., 2008. Avoiding Newspeak. In: *HEA-ICS 8th Annual Workshop on the Teaching of Programming*, 31 March 2008, University of Glasgow. Glasgow, UK. Available at: http://www.ics.heacademy.ac.uk/events/presentations/691_Teaching of Programming Workshop 2008-final.ppt. Last viewed: 11 January 2010.

Jones, M., 2009. JCodeGen: a Java source code generator. In: *HEA-ICS 9th Annual Workshop on the Teaching of Programming*, 6 April 2009, University of Bath. Bath, UK. Available at: http://www.ics.heacademy.ac.uk/events/presentations/740_JCodeGen.ppt. Last viewed: 11 January 2010.

Joy, M. and Luck, M., 1999. Plagiarism in programming assignments. *IEEE Transactions on Education*, 42 (2), 129-133.

Kelleher, C. and Pausch, R., 2007. Using storytelling to motivate programming. *Communications of the ACM*, 50 (7), 58-64.

Kennedy, M. M., 1979. Generalizing from Single Case Studies. *Evaluation Quarterly*, 3 (4), 661-678.

Kernighan, B. W. and Ritchie, D. M., 1988. *The C Programming Language (2nd edition)*, Harlow, UK: Pearson Education Limited.

Lancaster University Management School, 2006. *Can the UK Succeed in a Global Software Economy?* Lancaster, UK: Lancaster University.

Ma, L., Ferguson, J., Roper, M. and Wood, M., 2007. Investigating the viability of mental models held by novice programmers. *ACM SIGCSE Bulletin*, 39 (1), 499-503.

Magee, B., 1973. *Popper*, Aylesbury, UK: Fontana.

Maxwell, J. A., 1998. Designing a Qualitative Study. In Bickman, L. & Rog, D. J. (Eds.) *Handbook of Applied Social Research Methods*. Thousand Oaks, California, USA: SAGE Publications.

Mayer, R. E., 1981. The Psychology of How Novices Learn Computer Programming. *ACM Computing Surveys*, 13 (1), 121-141.

Mayer, R.E., Dyck, J. and Vilberg, W., 1986. Learning to program and learning to think: what's the connection? *Communications of the ACM*, 29 (7), 605-610.

McCarthy, T., 1978. The Critical Theory of Jürgen Habermas, Cambridge, Massachusetts, USA: MIT Press.

McGreal, R., 2004. Online education using learning objects, London, UK: Taylor & Francis Limited.

Meulman, J.J., Van der Kooij, A.J., Heiser, W.J., 2004. Principal components analysis with non-linear optimal scaling transformations for nominal and ordinal data. In: Kaplan, D. (Ed.) 2004. *The Sage Handbook of Quantitative Methodology for the Social Sciences*. Thousand Oaks, California, USA: SAGE Publications.

Misra, S. and Akman, I., 2008. A Model for Measuring Cognitive Complexity of Software. In: *Proceedings of Knowledge-Based Intelligent Information and Engineering Systems (KES 2008)*, Berlin / Heidelberg, Germany, 879-886.

Mooney, T. and Moran, D., 2002. Edith Stein: Phenomenology and the Interpresonal. In Mooney, T. & Moran, D. (Eds.) *The Phenomenology Reader*. London, UK: Taylor & Francis Ltd.

Moore, B., Nuttall, D. and Willmott, A., 1973. *Data Collection*, Bletchley, UK: Open University Press.

Moreno, A., Myller, N., Sutinen, E. and Ben-Ari, M., 2004. Visualizing programs with Jeliot 3. In: *AVI '04: Proceedings of the working conference on Advanced visual interfaces*, Gallipoli, Italy, May 25 - 28, 373-376.

Office of National Statistics, 2010. *Vacancies in Information and Communication*, Available at: http://www.statistics.gov.uk/StatBase/TSDtimezone.asp. Last viewed: 12 March 2010.

Penuel, W. R., 2006. Implementation and Effects Of One-to-One Computing Initiatives: A Research Synthesis. *Journal of Research on Technology in Education*, 38 (3), 329-348.

Peters, R. S., (Ed.) 1967. *The Concept of Education*, London, UK: Routledge & Kegan Paul.

Peters, R. S., (Ed.) 1977. *John Dewey Reconsidered*, London, UK: Routledge & Kegan Paul.

Peters, R. S., 1966. *Ethics and Education*, London, UK: George Allen & Unwin.

Petre, M. and Green, T. R. G., 1993. Learning to Read Graphics: Some Evidence that 'Seeing' an Information Display is an Acquired Skill. *Journal of Visual Languages & Computing*, 4 (1), 55-70.

Reaves, C. C., 1992. *Quantitative Research for the Behavioural Sciences*, New York, New York, USA: John Wiley & Sons.

Reynolds, D., Treharne, D. and Tripp, H., 2003. ICT - the hopes and the reality. *British Journal of Educational Technology*, 34 (2), 151-167.

Ricoeur, P., 1967. *Husserl: an analysis of his phenomenology*, Evanston, Illinois, USA: Northwestern University Press.

Rorty, R., 1978. *Philosophy and the Mirror of Nature*, Princeton, New Jersey, USA: Princeton University Press.

Ross, P., 1983. *LOGO Programming*, Harlow, Essex, UK: Addison-Wesley Higher Education.

Rößling, G., Schüer, M. and Freisleben, B., 2000. The ANIMAL algorithm animation tool. *ACM SIGCSE Bulletin*, 32 (3), 37-40.

Russill, C., 2005. The Road Not Taken: William James's Radical Empiricism and Communication Theory. *The Communication Review*, 8 (3), 277-305.

Scheffler, I., 1960. *The Language of Education*, Springfield, Illinois, USA: Thomas.

Schön, D. A., 1984. The Reflective Practitioner: How Professionals Think in Action, New York, New York, USA: Perseus Book Group.

Solomon, C., J. and Papert, S., 1976. A case study of a young child doing turtle graphics in LOGO. In *Proceedings of the National Computer Conference and Exposition*, 7-10 June 1976, New York, New York, USA.

Soloway, E., 1985. The psychology of programming. In: Healy, B., Schlesinger, Judith D. (Eds.) *Proceedings of the 1985 ACM annual conference on the range of computing : mid-80's perspective*, Denver, Colorado, USA, 252-252.

Soloway, E., 1986. Learning to program = learning to construct mechanisms and explanations. *Communications of the ACM*, 29 (9), 850-858.

Speigelberg, H., 1967. Husserl's Phenomenology & Sartre's Existentialism. In Kockelmans, J. J. (Ed.) *Husserl's Phenomenological Philosophy in the Light of Contemporary Criticism*. Garden City, NY: Doubleday & Company, Inc.

Stake, R. E., 1995. *The art of case study research*, Thousand Oaks, CA: Sage Publications.

Sternberg, R. J., 1977. *Intelligence, Information Processing and Analogical Reasoning: the Componential Analysis of Human Abilities*, Hillsdale, New Jersey, USA: Laurence Erlbaum Associates.

Stickney, J., 2006. Deconstructing Discourses about 'New Paradigms of Teaching': A Foucaultian and Wittgensteinian perspective. *Educational Philosophy and Theory*, 38 (3), 327-371.

Thomas, L., Ratcliffe, M., Woodbury, J. and Jarman, E., 2002. Learning styles and performance in the introductory programming sequence. In: Gersting, J, Walker, H. M. and Grissom, S., (Eds.) *Proceedings of the 33rd SIGCSE technical symposium on Computer Science Education*, Cincinnati, Kentucky, USA, 27 February - 3 March, 33-37.

Tiryakian, E. A., 1978. Durkheim and Husserl: A comparison of the Spirit of Positivism and the Spirit of Phenomenology. In Bien, J. (Ed.) *Phenomenology and the Social Sciences*. Den Haag, Netherlands: Martinus Nijhoff.

Turing, A. M., 1936. On Computable Numbers, with an Application to the Entscheidungsproblem. In: *Proceedings of the London Mathematical Society*, 42 230-265.

Turner, V., 1974. Liminal to liminoid, in play, flow, and ritual: an essay in comparative symbology. *Rice University Studies*, 60 (3), 53-92.

Vonnegut, K., 2003. Knowing what's nice, Available at: http://www.vonnegutweb.com/archives/arc_nice.html. Last viewed: 28 September 2009.

Weinberg, G. M., 1998. *The Psychology of Computer Programming: Silver Anniversary Edition*, New York, New York, USA: Dorset House Publishing Company Inc.

Wittgenstein, L., 1953. *Philosophical Investigations*, Oxford: Blackwell.

Yin, R. K., (Ed.) 2004. *The Case Study Anthology*, Thousand Oaks, California, USA, SAGE Publications.

Yin, R. K., 1998. The Abridged Version of Case Study Research: Design and Method. In Bickman, L. and Rog, D. J. (Eds.) *Handbook of Applied Social Research Methods*. Thousand Oaks, California, USA: SAGE Publications.

Yin, R. K., 2003. *Applications of Case Study Research,* Thousand Oaks, California, USA: SAGE Publications.