A Cognitive Study of Learning to Program in Introductory Programming Courses

A thesis submitted to Middlesex University
in partial fulfilment of the requirements for the degree of Doctor of Philosophy

Saeed Dehnadi
School of Engineering and Information Sciences
Middlesex University

May 2009

Contents

Abstract

Programming is notoriously hard for novices to learn and a substantial number of learners fail in introduction to programming courses. It is not just a UK problem: a number of multi-institutional and multi-national studies reveal that the problem is well-known and is widespread.

There is no general agreement about the causes or the remedies. The major factors which can be hypothesised as a cause of this phenomenon are: learners' psychology; teaching methods; complexity of programming.

In this study, learners' common mistakes, bugs, misconceptions, frequencies and type of errors (syntactic and semantic) in the early stages of learning programming were studied. Noticing the patterns of rationales behind novices' mistakes swayed the study toward investigating novices' mental ability which was found to have a great effect on their learning performance. It was observed that novices reported a recognisable set of models of program execution each of which was logically acceptable as a possible answer and it appeared that some students even used these models systematically. It was suspected that the intellectual strategies behind their reasoning could have been built up from their programming background knowledge and it was surprising when it was found that some of those novices had not even seen a program before.

A diagnostic questionnaire was designed that apparently examined a student's understanding of assignments and sequence but in fact was capturing the reasoning strategy behind their interpretation of each question, regardless of a correct or wrong answer. The questionnaire was administered in the first week of an introductory programming course, without giving any explanation of what the questions were about. A full response from most participants was received, despite the fact that the questions were unexplained.

Confronted with a simple program, about half of novices seem to spontaneously invent and consistently apply a mental model of program execution. They

were called the consistent subgroup. The other half are either unable to build a model or to apply one consistently. They were called the inconsistent subgroup. The first group perform very much better in their end-of-course examination than the rest.

Meta-analysis of the results of six experiments in UK and Australia confirmed a strong effect of consistency on success which is highly significant ($p < 0.001$). A strong effect persisted in *every* group of candidates, sliced by background factors of programming experience (with/without), relevant programming experience (with/without), and prior programming course (with/without) which might be thought to have had an effect on success. This result confirms that consistency is not simply provided by prior programming background.

Despite the tendency in institutions to rely on students' prior programming background as a positive predictor for success, this study revealed that prior programming education did not have a noticeable effect on novices' success. A weak positive effect of prior programming experience was observed overall which appeared to be driven by one experiment with a programming-skilful population.

This study shows that students in the consistent subgroup have the ability to build a mental model, something that follows rules like a mechanical construct. It also seems that when programming skill is measured by a weak assessment mechanism, the effect of consistency on success is reduced.

Acknowledgments

This thesis could not have been written without Prof. Richard Bornat who not only served as my director of study but also encouraged and challenged me, continuously all throughout, from the beginning and up to the end of this study. He never accepted less than my best efforts. I also thank Dr. Ray Adams my second supervisor who patiently guided me through the data analysis in this thesis. I would like to thank Mr. Ed Currie, my teaching line manager for his permission to conduct my experiments in his lectures.

I sincerely thank Dr. Simon, Dr. Peter Rockett, Mr. Christopher Thorpe and Dr. Dimitar Kazakov, the collaborators who provided data for this study and all my peers in PPIG (Psychology of Programming Interest Group) for their valuable advice, especially Thomas Green who has been always supportive and helpful. I also thank Dr. Tracy Cockerton in the Psychology department for her kindness and support.

I thank my wife Homa for her endless support and dedication. She worked so hard to provide all the needs of the family, as well as persuading me to believe on myself when I was down and tired. I like to thank my son Sina who was so patient when I just had a little time to spend with him.

I owe a lot to my student colleagues and friends who showed interest in my work and kept asking about the latest progress.

Chapter 1

Introduction

Students in introductory programming courses find programming notoriously hard to learn and a substantial number of them fail in those courses. Students joining programming courses are keen to find out how a set of instructions can apparently pass on intelligence to an electronic machine, and teachers are keen to help them. But many find it difficult to learn, however hard they try. On the other hand some seem to have a natural aptitude for programming.

Despite considerable research effort, the causes of this phenomenon remain unknown and no testing mechanism is able to separate these two populations at the early stage of their learning.

1.1 Phenomenon

In UK schools, students are prepared for higher education with a number of "Advanced Level" (A-level) studies which provide a background and evidence of aptitude for universities to keep their drop-outs low in the early stage. Computing Science departments were never able to do this, even when a Computer Science A-level existed. They try nowadays to select on mathematical or general abilities, but it has not worked: there has always been a large failure rate. No effective predictor of programming aptitude has ever been found.

A study undertaken in nine institutions in six countries (McCracken et al., 2001) looked at the programming skills of university computer science students at the end of their first year of study. All the participants reported that at the conclusion of their introductory courses very many students still did not know how to program.

Lister et al. (2004) looked at evidence from seven countries and found the same thing. Members of the same team (Raadt et al., 2005) looked at various possible predictors of programming success and found a few weak correlations.

After attending a Psychology of Programming Interest Group (PPIG) meeting and discussing the issue with other researchers, it became apparent to me that the problem is well-known and is widespread.

In introduction to programming courses the majority of novices complain of being confused from the beginning, despite the hard work and attention given to them by their instructors. They describe programming as a peculiar subject which is hardly related to any of the subjects they have encountered in the past. Just a handful of novices in this group could manage to overcome the problem but most of them carried the problem up to the end of the course or looked for alternative course of study. I found them hard to teach.

A minority seem quite confident, ready to take in the relevant materials swiftly, providing opportunity for their teachers to work on their misconceptions and direct them toward the correct models. A vast majority of novices in this group will manage to progress to the next level of programming courses. I found them easy to teach.

Observing two groups of novices with diverse ability in introduction to programming courses was a common phenomenon in Computer Science in many institutions and I was not the only programming teacher who was mystified by it.

1.2 Causes of the phenomenon

The causes of the phenomenon could be:

Students' psychology Some students can comprehend programming much more easily than others. Perhaps it is a matter of mental ability.

Problems in teaching The phenomenon could be caused by a variety of factors related to teaching the subject, such as: inadequate teaching methods; inappropriate materials; incompatibility of teaching and learning styles; and so on.

Complexity of programming The subject could be more difficult than it

needs to be. The underlying programming constructs may have not been designed around learners' needs (Blackwell et al., 2002).

There is no general agreement about the cause of the phenomenon and no agreement on remedies.

1.3 This study

This research is focused on learners' mental ability which I found to be the most effective explanation of the phenomenon.

I observed that some patterns of rational strategies were employed by novices when responding to a programming question at a very early stage of learning to program. Each rationale was recognisable and similar to the mechanisms actually used in program execution although not precisely addressing the correct answer. For example, the mechanism of assigning the value of a variable from left to right is very similar to assigning it from right to left, but only the second is the correct model of Java execution.

This gave me the idea that the novices may have already been equipped with some abilities before the course started. Therefore I decided to give a test in the first week of the course, investigating the mechanisms they brought into the programming learning context before they were affected by teaching methods and materials.

A test was designed that apparently examined a student's knowledge of assignment and sequence but in fact was capturing the reasoning strategy behind their interpretation of each question, regardless of a right or wrong answer. I deliberately chose assignment and sequence because they are simple and fundamental programming constructs. The test was administered in the first week of an introductory programming course, without giving any explanation of what the questions were about.

The result brought up three surprises:

- I received a full response from almost all participants, despite the fact that the questions were unexplained.

- About half of them not only had the ability to create a rational model, they also managed to generalise their models and applied them consistently to

answer most of the questions. The other half did not seem to use the same model consistently or refused to answer most of the questions.

- When I correlated the test result with the subjects' examination mark the consistent subgroup clearly performed better than the others. The consistent subgroup overall had 85% chance of success in the course while the chance of success in the inconsistent subgroup was only 35%.

I showed the test result to the research community at a PPIG (Psychology of Programming Interest Group) conference (Dehnadi and Bornat, 2006). A number of objections were raised. The objections were mainly about the following issues:

- The questionnaire – participants' prior programming background, age and sex were not recorded.
- Interpretation of mental models – judgment of consistency was not clearly defined and was subjective.
- Data analysis methods – were not strong enough.

I responded to these objections by amending the questionnaire, objectifying the process of interpretation by introducing an answer-sheet, a mark-sheet and a marking protocol as well as employing better statistical methods when analysing the test results.

Six experiments, five in UK and one in Australia, were analysed individually and also combined in a meta-analysis. The results and analysis are reported in chapter 6 and 7.

The overall result strongly confirmed the original test result. Consistency does have a strong effect on success in learning programming. Background programming experience, on the other hand, has little or no effect. The mechanisms of rationalisation which students bring to the study of programming have a great effect on their learning performance.

1.4 What is programming?

Programming is generally agreed to consist of a number of activities, including designing, coding, testing, debugging, and maintaining the source code of computer programs. Students in universities develop these skills gradually through a series of courses.

In introductory courses novices learn part of each of the activities involved in programming. Some courses focus on coding and debugging activities from the beginning and some emphasise design activity first. However, there is no evidence that any particular approach has an effect on novices' fail rate. The coding/debugging approach was followed in all the courses which were involved in this study.

Success in an introductory course is proof of ability in some programming activities. But why should my study have any impact in the wider area of programming and what does it contribute to the knowledge of this field? This study looks at recognisable models of assignment and sequence used by candidates which are similar to the mechanical processes behind the execution of programs, which Mayer called an effective mental model of the virtual machine (Mayer, 1981). Ability to use such a model seems to be an essential requirement for novices to pass the course and it is possible that this ability lies behind all activities of programming:

- Without envisaging the underlying activities such as execution of an algorithm, writing or reading a chunk of code is impossible;
- Debugging activity cannot succeed without an understanding of what each line of code does and what they do together;
- Designing also heavily relies on understanding the capability and limitation of programs;
- Large-scale maintenance requires knowledge about small fixes which relies on skill in coding.

Success in an introductory programming course demonstrates candidates' ability to envisage the mechanical processes behind the execution of programs. This ability seems to be an essential foundation which facilitates all programming activities and learning programming may be impossible without it.

1.5 Chapter summaries

Chapter 2 is a review of the literature. Chapter 3 explains the early stages of this study, when I was trying to find a suitable way of understanding the sources

of learners' difficulties. Chapter 4 describes the first methodical experiment that revealed the subpopulations and the effect of consistency. Chapter 5 describes the methodology of later experiments, in which the process of the experiment and methods of analyses are more objective. Chapter 6 shows the results and analyses of those experiments. In order to get an overall view of the effect of consistency among diverse institutions the results are also meta-analysed in Chapter 7. Chapter 8 summarises the outcome of this study and discusses opportunities for future studies.

Chapter 2

Literature Review

Researchers in this area agree that novices find it difficult to learn to program, but have different opinions about causes and cures. Some of the researchers tried to find characteristics which could have an effect on learners' success, while some believed that learning programming needs skills that should be developed in advance. Some other researchers claimed that teaching methods are causing the problem. I found three major directions in the literature which were distinguished by the way they tackled the problem.

- Empirical correlations – look at novice learners' attributes for factors which are correlated to their success in a first programming course.
- Explanations – look for the causes of learners' difficulties in their psychology.
- Interventions – introduce new forms of teaching programming such as concept-first, visual learning tools and Integrated Development Environments (IDE's).

I describe each of these directions, highlighting their strengths, weaknesses and scope of validity.

2.1 Empirical correlations

Since the 1960s, following industrial innovation in computer technology, demand for programming courses has grown rapidly. However the considerable failure rate in these courses prompted a huge interest to find factors which correlate

with learners' success in introductory programming courses. Researchers tried a variety of methods such as:

- Aptitude/Psychometrics test;
- Personality/Self-ranked test;
- Background elements (e.g. sex, age, race, skill at bridge and chess, parental education level and occupation);
- Ability in mathematics.

2.1.1 Aptitude/Psychometrics

Reinstedt et al. (1964), the founders of the Special Interest Group for Computer Personnel Research (SIG/CPR), were concerned to investigate any programming aptitude test that could assist management to select experienced or trainee programmers. They examined the relationship between job performance and certain measures of interest like cognitive abilities and biographical data. They applied a battery of aptitude tests: PAT (Programmer Aptitude Test used by IBM), TSI (Test of Sequential Instructions) and SVI (Strong Vocational Interest). The tests were administered to 534 experienced programmers employed by 24 separate organisations.

Although a significant correlation between PAT scores and supervisory ranking of performance was found in some organisations, there were not significant correlations between these factors for the total sample. Similarly, the correlation between TSI and ranking of performance was not significant across different organisations.

Mayer (1964) from an industrial management position, criticised SIG/CPR for their stress on legitimising current aptitude tests and turned to a traditional method to select programmers. He emphasised direct personal interview for experienced programmers rather than a written test to determine level of capability, as well as an impression of personal traits. Mayer believed that the SIG/CPR aptitude results were ambiguous and claimed (without supporting evidence) that his interviewing technique gave him a 60-70% effectiveness in selection.

Mayer and Stalnaker (1968) reported on more programming aptitude tests such as WPT (Wonderlic Personal Test) and PMA (Test of Primary Mental

Ability) that were widely used in 1960-1970. They concluded "It is true that the PAT measures certain attributes required in programming – numerical capability and spatial relationships – and the TSI probably measures the ability to perform parallel operations, but apparently neither measures the whole cloth of what managers believe to indicate programming ability".

Huoman (1986) invented a programming aptitude test and Tukiainen and Mönkkönen (2002) evaluated it at the beginning of an introduction to programming course. At first they found a very significant correlation between the Huoman test and the final examination, but when candidates' programming background was examined, the test was found to give no evidence of success in the programming exam.

Bennedsen and Caspersen (2006) found no correlation between cognitive development and results in a model-based introductory programming course; and they found, in a three-year longitudinal study (Bennedsen and Caspersen, 2008), that general abstraction ability was not a predictor for success in learning computer science.

2.1.2 Background

Reinstedt et al. (1964) reported that after recognising some serious problems with different kinds of testing, SIG/CPR decided to look for non-cognitive factors to predict programming ability. They reported that age bore no weight in programming performance; that with the exception of a relevant mathematics major in scientific programming, college majors had no effect on programming performance; the college average score of programmers had no predictive value for programming performance; no relationship was found between interest in or frequency of participating in puzzle solving, bridge and chess; parental education level and occupation likewise failed to be significantly correlated.

Wilson and Shrock (2001) used a questionnaire plus the Computer Programming Self-Efficacy Scale test developed by Ramalingam and Wiedenbeck (1998). They administered the test to 105 students after the midterm exam. They examined possible predictive factors including mathematics background, attribution of success and failure (luck, effort, difficulty of task, and ability), domain-specific self-efficacy, encouragement, comfort level in the course, work-style preference, previous programming experience, previous non-programming computer experi-

ence, and gender. They revealed a number of predictive factors in the following order of importance: comfort level, math background, and attribution to luck for success/failure (based on students' beliefs about their reasons for success or failure). Comfort level and math had positive correlations with the midterm score, but attribution of success/failure to luck had negative correlation.

The study also revealed that by considering different types of previous computer experience (including formal programming class, self-initiated programming, internet use, game playing, and productivity software use) that both a formal class in programming and game playing had influence. Formal training had a positive influence and games a negative influence on class grade.

However some of the correlated factors used by this study, like comfort level and attribution to luck for success/failure, could not be provided before midterm at least, so cannot be used as predictors and the authors did not report any succeeding investigation to support their claim.

Besie et al. (2003) examined age, race and sex to see if they were correlated with success in a first programming course, particularly for computer science and information systems. Statistical analysis of their data indicated that neither sex nor age is correlated with success in the first programming class. This study also indicates that the proportion of women is higher in IS majors than CS, and that CS majors have a higher probability of passing the first programming course than IS majors.

2.1.3 Personality

Biamonte (1965) investigated the relationship of the attitude patterns of dogmatism, authoritarianism, and political conservatism versus success in programming training. A sample of 201 trainees was used. It was found that even though there were significant correlations between scores on the tests used to measure each of the three attitudes, scores were related to neither training grades nor scores on an independent measure of intelligence.

Cross (1970) and Mayer and Stalnaker (1968) introduced occupational aptitude tests at the initial stage of joining software industry employment, hoping to predict successful candidates. Cross relied heavily on a psychometric aptitude test instrument, called JAIM (Job Analysis and Interest Measurement), which was a self-report of a candidate's personality. It was designed to measure per-

sonal qualities, aptitudes, training and knowledge which has an influence on job performance.

Applying JAIM to a group of programmers, Cross found them to be rather peculiar individuals, willing to work in isolation, avoiding interaction with different features of the organisation, preferring not to be involved in any possible confrontation with others including supervision and being supervised, avoiding structure and routine, and being motivated primarily by achievement rather than external reward, status, or approval of others. Since this particular personality pattern was found in most programmers, Cross considered it as a good measure of programming aptitude. Unfortunately there is a lack of statistical analysis in his study.

Thirty-three years after Cross's comment about programmers' peculiar personality pattern, Baron-Cohen et al. (2003) exposed this pattern as a classic trait associated with autism. He labeled people such as scientists, mathematicians and engineers as *systematizers*, who are skilled at analysing systems; they also tend to be less interested in the social side of life, and can exhibit behaviour such as an obsession with detail. Wray (2007) inspired by (Baron-Cohen et al., 2001, 2003) administered a number of aptitude tests, using Baron-Cohen's instruments, a self-ranking questionnaire and my test (chapters 4 and 5). He found some interesting associations. Since my research was started four years before Wray's experiment I will discuss his experiment in section 6.7.

Rountree et al. (2004) revealed that the students most likely to succeed are those who are expecting to get an 'A' grade and are willing to say so, in the middle of the course. They rely on students' self consciousness, and believe that a student's expectations may influence their result. They asked questions such as 'what grade do you expect to get in this course' and 'how hard do you expect this paper to be?' showing that the best indicators of success appear to be self-predicted success, attitude, keenness and general academic motivation. They supposed that students have a better idea of their own background strengths than we can determine from their mix of school examination grades, so they asked them what they considered their strongest background. Using decision-tree induction, they found a group of students who seem to be about twice as likely to fail as the others in the class. These students seem likely to be those who are surprised at how different a programming course is from anything else they have encountered. However, these indicators do not distinguish programming

from other disciplines, and have less effect as discipline-specific indicators. They did not reveal the figures that indicate how many students who expected to get 'A' failed, and how many who did not expect 'A' passed the course.

The phenomenon of observing two noticeable subgroups of learners was frequently reported by researchers: the first subgroup struggle from the beginning and give up when they strike the first difficulty; in contrast, the other subgroup seem comfortable when confronting a problem and learn actively. Perkins et al. (1986) described the first group as "stoppers" and the second as "movers" who seemed to use natural language knowledge to get beyond an impasse. Perkins et al. added that some of the "movers" were not genuinely engaging with the problem, but rather carrying on in an extreme trial-and-error fashion that escaped any intellectual challenge with the programming difficulties. Perkins et al. suggested that it would be encouraging to give some positive feedback to prevent "movers" from quitting the study of programming.

2.1.4 Ability in mathematics

Reinstedt et al. (1964) found that relevant mathematics background correlates to success in learning scientific programming and Wilson and Shrock (2001) found math background had positive correlations with the programming score.

McCoy and Burton (1988) studied good mathematical ability as a success factor in beginners' programming. They claimed that understanding mathematical variables was correlated with success in programming courses, but no evidence was cited. In a subsequent paper McCoy (1990) states that understanding programming variables involves knowing that the label can store a value that changes and concludes that programming helps students understand mathematical variables.

2.1.5 Large-scale tests

McCracken et al. (2001) in a multi-national research project tried to find an assessment mechanism to measure first-year students' programming ability. Their collaborators all reported that many students do not know how to program at the conclusion of their programming course. Their explanation for that inability was that students lack knowledge and skills that are precursors to problem-solving. They introduced an assessment mechanism to monitor students' progress in five

steps: ability to take a problem description, decompose it into sub-problems, implement each fragment, assemble the pieces together as a complete solution, evaluate the problem and produce a solution iteratively. The paper does not isolate the causes of the problems.

Lister et al. (2004) followed the McCracken group's study and administered another multi-national research project. First they tested students on their ability to predict the outcome of executing a short piece of code. Second, students were tested on their ability, when given the desired function of a short piece of near-complete code, to select the correct completion of the code from a small set of possibilities. Many students were weak at these tasks, especially the latter, which revealed their lack of skills that are a prerequisite for problem-solving. They added to McCracken et al.'s explanation that the important elements were students' fragile grasp of both basic programming principles and their inability to carry out routine programming tasks systematically, such as tracing through code – students had more difficulties to read code than to write it.

A full report on this study is (Fincher et al., 2005) followed by Simon et al. (2006a) and Tolhurst et al. (2006) with more detail on each component of the study, full analysis of the data and justification of the conclusions.

Simon et al. (2006b) described another multi-national, multi-institutional study that investigated the potential of four diagnostic tasks as success factors in introductory programming courses, in eleven institutions. The four diagnostic tasks were:

1. A spatial visualisation task (a standard paper folding test);
2. A behavioural task used to assess the ability to design and sketch a simple map;
3. A second behavioural task used to assess the ability to articulate a search strategy;
4. An attitudinal task focusing on approaches to learning and studying (a standard study process questionnaire).

It seems possible that a multi-factor model employing tasks such as those used in this study could be used as a reasonable factor of success in introductory programming. However, the study failed to demonstrate any significant effect

and suggests further exploration of possible diagnostic tasks, and a need for clear understanding of their inherent biases.

The study suggested to explore how strongly these tasks are associated with general IQ or standard components of IQ such as verbal and spatial factors, as a potential extension to this study. They added that there is no accepted measure of programming aptitude, therefore they cannot find correlations between performance on simple tasks and programming aptitude. They stated that they had to replace it with the readily quantified measure of exam mark in a first programming course. They believe that there is not a simple linear relationship between programming aptitude and mark in a first programming course, but there is no other easily-measured quantity available.

2.1.6 Summarising empirical correlations

Research looking for a robust factor correlated with programming ability has been actively carried out for five decades, starting with a series of aptitude tests designed to find suitable candidates for employment in the software industry. After recognising serious problems with some of these instruments, researchers returned to traditional alternatives: non-cognitive factors in subjects' background such as age, race and sex; puzzle-solving; skill at bridge and chess; mathematics; parental education level; and occupation. Most studies failed to find any strong correlation.

Two studies, on the other hand, found that subjects' comfort level and high expectation correlated with programming success; but this seems to be a general attitude to success, which works among all disciplines (Wilson and Shrock, 2001; Biamonte, 1965). Another found relevant mathematics background helps scientific programming (Reinstedt et al., 1964).

A peculiar personality pattern was also observed commonly in programmers which was considered as a good measure of programming aptitude; this personality pattern was similar to that described as a classic trait associated with autism thirty years later.

A number of multi-national studies, which were looking for an effective factor correlated to programming success, found that programming difficulty is a global problem. No strong correlations were reported.

Despite all this effort, a reliable method of predicting the success of students

who enter an introductory programming course has not been found. Although some weak evidence has been reported, there is no available effective and solid test which could categorise novices or measure their ability to learn programming and predict their success or failure at the beginning of an introductory course.

2.2 Explanations

Researchers in this group tried to examine a variety of elements in order to understand the causes of learners' difficulties and explain the phenomenon. They tackled the problem from two different directions:

1. Common mistakes – researchers in this area were mostly teachers who were directly involved in teaching programming. They studied novices' errors, recording, cataloguing and analysing them in order to understand what is common amongst them and how misconceptions occur. They looked deeply at novices' misconceptions in order to hypothesise what they are thinking.

2. Psychology – researchers in this area were mostly psychologists who looked at learners' problems like an equation: learners with their mental abilities on one side; and the programming construct with its complexities on the other side. Then they tried to evaluate learners' mental abilities when dealing with the complexities which were found in programming constructs so as to analyse the causes of learners' difficulties.

Researchers in all of these directions reached similar conclusions. Hence I categorise them by the way they explained the phenomenon. Researchers attributed novices' problems to the following factors:

1. Lack of an effective mental model of the machine. An effective mental model can be developed in two stages: first a correct mental model of basic concepts of computer ability; then an effective mental model of the virtual machine.

2. Teaching – issues such as negative impact of misapplication of analogy; or introducing topic "B" which relies on understanding of topic "A" which has not yet been fully understood.

3. Lack of domain knowledge.

4. Complexity in programming constructs (formalism) – they identified the complexities underlying structural characteristics in programming construct which are not designed around the needs of learners.

2.2.1 Mental models

The idea of mental model was frequently used by researchers in this area, although they did not all use exactly the same notion.

The idea of mental model can be traced back to Craik (1943) who suggested that the mind constructs "small-scale models" of reality that it uses to anticipate events. He explained that mental models are representations in the mind of real or imaginary situations. He added that mental models can be built from perception, imagination, or the comprehension of discourse. He believed that mental models underlie visual images, but they can also be abstract, representing situations that cannot be visualised. He stated that each mental model represents a possibility, similar to architects' models or to physicists' diagrams in that their structure is analogous to the structure of the situation that they represent.

Johnson-Laird studied people's competence in deductive reasoning. He attempted to identify alternative mental models used by subjects trying to solve syllogistic puzzles (Johnson-Laird, 1975). Later, Johnson-Laird and Bell (1997) put forward the theory that individuals reason by carrying out three fundamental steps:

1. They imagine a state of affairs in which the premises are true - i.e. they construct a technical/mathematical mental model of them.

2. They formulate, if possible, an informative conclusion true in the model.

3. They check for an alternative model of the premises in which the putative conclusion is false. If there is no such model, then the conclusion is a valid inference from the premises.

Johnson-Laird concludes that comprehension is a process of constructing a mental model (Johnson-Laird, 1981), and set out his theory in an influential book (Johnson-Laird, 1983). Since then he has applied the idea to reasoning about Boolean circuitry (Bauer and Johnson-Laird, 1993) and to reasoning in modal logic (Johnson-Laird and Bell, 1997). The model theory is an alternative to the

view that deduction depends on formal rules of inference similar to those of a logical calculus (Rips, 1994; Braine and O Brien, 1991). The distinction between the two sorts of theories parallels the distinction in logic between proof-theoretic methods based on formal rules and model-theoretic methods based on semantics such as truth tables.

Gentner and Stevens (1983) introduced a different notion of mental model. They explained that people's views of the world, of themselves, of their own capabilities, and of the tasks that they are asked to perform, or topics they are asked to learn, depend heavily on the conceptualisations that they bring to the task. In interacting with the environment, with others, and with the artefacts of technology, people form internal mental models of themselves and of the things with which they are interacting; these models provide predictive and explanatory power for understanding the interaction.

Gentner and Stevens' mental models are about the ability to envisage mechanical processes and changing of states in each process. Kieras and Bovair (1984) termed Gentner and Stevens's mental model a *device model* to distinguish it from the other senses of the term mental model such as that used by Craik and Johnson-Laird. They conducted an experiment to examine the role of mental model in learning how to operate an unfamiliar piece of equipment. In this experiment two groups of subjects learned a set of procedures for operating a simple control panel device consisting of switches, push buttons and indicator lights. The goal of the procedure was to get a certain indicator light to flash. One group learned how-it-works knowledge in the form of internal component and processes of the device. The second group received no how-it-works knowledge, but only learned the procedures by rote. Kieras and Bovair found the first group who received how-it-works knowledge performed significantly better than the second group. They reported that device model information provides a definite and strong facilitative effect.

Craik's mental models are models of structures. Johnson-Laird's mental models are models of static situations. Gentner and Stevens' mental models are models of mechanisms with processes and states. They emphasise peoples' ability to build a *mechanical* mental model of a target system. These are the mental models discussed by researchers into the programming problem, and which I use in this research.

Spohrer and Soloway (1986) were amongst the first researchers to stress the

significance of mental models in learning programming. They indicate that for most programming tasks there is some model – an imagined mechanism – that a novice will use in his/her first attempts and we need to understand when it is appropriate to appeal to this model and when it is necessary to move a novice to some more appropriate model.

They focused on bugs and buggy programs, catalogued them, and classified novices' mistakes and misconceptions. They tried to use novices' mistakes and misconceptions to hypothesise what they were thinking as they programmed. They found that just a few types of bug cover almost all those that occur in novices' programs. They believed that bugs and errors illuminate what a novice is actually thinking. They hypothesised a "programming goals/subgoals/plans" theory that is used widely by novices in order to break down the complexities of program applications. They alerted educators about cognitive bugs and advised them to teach students strategies for putting the pieces of programs together. They stated that learners need more guidance at the beginning, but that their supports need to be changed as they build competence, and become more independent in their learning.

Bayman and Mayer (1983) studied misconceptions in two different types of statements. One statement was an initialisation `D=0` and the other was an assignment `A=B+1`. They describe how in both of these statements students often thought the computer was writing the information somewhere or printing it to the screen as opposed to storing it in a named memory location. The authors suggested that beginners need explicit training concerning memory locations and under what conditions value stored in those locations get replaced.

du Boulay (1986) explained misconceptions about variables, based upon the analogies used in class. He illustrated that the analogy of a box or drawer with a label on it may lead learners to build a wrong mental model that a variable can hold more than one value at a time. Comparing a variable to a box triggers learners' minds to their existing mental model of boxes, rather than building a correct model of variable. He also noted students' misconceptions concerning variables when assigning one variable to another: for example `x=y` may be viewed as linking the second variable to the first, and therefore a change in `x` results in a change in `y`. He highlighted novice users' vital misunderstanding of the temporal scope of variables and explained that they may not understand that a value stays in a variable until it is explicitly changed or the contents of memory are erased

or the machine is switched off.

du Boulay said that novices find the syntax and underlying semantics of a programming language hard to comprehend and have no idea about the capabilities of a computer. He concluded that novices with a lack of mechanical understanding build a poor mental model which is not adequate for their learning requirements. du Boulay showed that when a variable is used to hold a running total, the fact that learners forget to initialise the total to zero comes from the wrong analogy that introduces a variable as a box which, even when empty, has some value.

Perkins and Simmons (1988) explained that novices have misconceptions about the names of variables, even though they know in principle that the choice of variable names is theirs. Having a weak mental model of the virtual machine, students have the notion that a computer program knows that the highest input values should go into variables with names like LARGEST. They highlighted that people commonly fail to notice inconsistencies in their intuitive mental models, and often, when inconsistencies are brought to their attention, cannot notice their importance.

Kessler and Anderson (1986) studied students' errors in learning recursion and iteration. They observed that students who had poor mental models of the mechanical processes of recursion and iteration in programming adopted poor learning strategies. They emphasised that novices' appropriate mental models of such techniques should be developed prior to engaging them in any implementation task.

Mayer (1981) believed that experts are able to think semantically and demonstrates four areas of differences between them and novices in computer programming: semantic knowledge, syntactic knowledge, ability in task management and taking advantage of having an effective mental model of the virtual machine. Novices are at the beginning of their mental model development; syntactic knowledge is hard for them because it's difficult to catch grammatical mistakes; they are inexperienced in problem decomposition; and lack of strategic knowledge forces them to lean on low-level plans during problem solving.

Mayer describes existing knowledge as a "cognitive framework", and "meaningful learning" as a process by which new information is connected to existing knowledge. In (Mayer, 1992) he stated that people who know how their programs work do better than those who do not, and mental models are crucial to learning and understanding programming. He also emphasises background ex-

perience and explains that a person who is an accomplished problem solver in a particular domain tends to build up a range of problem solving techniques and strategies. When confronted with a new problem, such an expert compares it with past problems and, where possible, constructs a solution out of familiar, reliable techniques.

van Someren (1990) investigated novices learning Prolog. He came to the conclusion that those who were successful had a mechanical understanding of the way that the language implementation, the Prolog virtual machine, worked. As a result, he claimed that more complex statements and implicit behaviours, where multiple actions take place as a result of a single statement, are likely to be more confusing to students. He added that the difficult bits of Prolog – unification and depth-first search – gave rise to most difficulties.

Cañas et al. (1994) introduced a tracing mechanism to show novices how a computer works as code is executed, helping them to develop a mechanical mental model. The result showed that students who used the tracing mechanism to follow the code had different mental models than those who did not; the students who used the trace mechanism managed to organise the concepts by semantic aspects of the programming language while those who did not use the tracing mechanism organised the concepts by syntax. They concluded that those who have appropriate mental models are likely to do better on programming tests and show greater understanding than students with poor or inappropriate mental models.

Putnam et al. (1986) examined high school students' misconceptions in the BASIC programming language. They used a screening test and structured interviews to determine their understanding of fundamental concepts such as variables, assignments, loops, flow and control, and tracing and debugging. They found that novices' misconceptions about the capabilities of computers could have a massive negative impact on their success at programming. Many of their subjects tried to use meaningful names for their variables, apparently hoping that the machine would be able to read and understand those names and so perceive their intentions.

Murnane (1993) relates programming to psychological theories and how it can be applied to programming. He suggested that, initially, students require solid, tangible objects to work with. They need good feedback, and a clear path from cause to effect – "when I do `x`, it causes `y`". They confirmed van Someren (1990)

who suggested that a good understanding of the underlying virtual machine is vital for students learning programming languages.

Winslow (1996) described a psychological overview of programming pedagogy and explained the characteristic differences between novices and experts. He explained that most studies differentiate between a task – a goal with a known solution – and a problem – a goal with no familiar solution. A problem to a beginner may be a task to someone more advanced, which is an important characteristic of experts regardless of discipline.

Winslow believed that programming pedagogy should build confidence in novices to put the bits and pieces of the programming puzzles together. He indicated that novices know the syntax and semantics of individual statements but they don't have any idea how to chain them into a valid program; even when they can solve a problem by hand, they have trouble transferring their solution into a comparable computer program. He suggested that practice helps, and complained about students who don't study the problem cautiously or read it in trivial manner.

Ahmadzadeh et al. (2005) looked at programming difficulties and described teaching programming as a problematic issue for computer science. They stressed the difficulty of finding an effective method of teaching suitable for all students. They also described that in order to adopt a better teaching strategy, examining students' mistakes is essential. They recorded compiler errors generated by students' programs and observed them during their debugging process. They initially thought that the majority of good debuggers would be good programmers (students with a high mark); they find that less than half of the good programmers are good debuggers and that this is a major obstacle to their productivity when dealing with complex codes. This might reveal that the half of "good programmers" in this study, in spite of having good marks, did not develop a correct mental model of programming yet. They explained it as lack of knowledge of the actual program implementation which prevents many of the good programmers becoming good debuggers. They suggested skill at debugging increases a programmer's confidence and more emphasis should be placed on debugging skills in the teaching of programming.

2.2.2 Teaching

In the literature researchers described the effect of inappropriate teaching which increases programming learners' difficulties and suggested alternative approaches.

Hewett (1987) tried to use the study of psychology to motivate learning to program. He originally used an existing model from (Shneidermann, 1980) when undertaking the development of a course in software psychology. He gradually modified the course in two different directions: the cognitive psychology of computer programming, the application of cognitive psychology to design and the evaluation of human-computer interfaces. It was a ten-week course involving mixed lectures and discussions without using a computer. Hewett reported that the course strongly supported the "Architect-Builder Metaphor" (Brooks, 1975) that tried to separate the jobs of designers and programmers in the software development life-cycle. He claimed that the course has an impressive impact on the students' design projects. After a year, when equipment allowed, he changed the focus from software psychology to application design and implementation. By changing the structure of the course he lost momentum and never followed up the sustainability of the effect.

Thirteen years after Hewett's study, Vat (2000) reported a similar effect in a junior course, titled Software Psychology, offered in the undergraduate Software Engineering program at university. The course in particular introduced the pedagogy of problem-based learning and addressed issues such as the resources and facilities needed for a programming course.

This study pointed to some evidence that the course developed students' quality of performance in the following characteristics: high level communication, technological literacy, effective problem solving ability, flexibility and adaptability to ease with diversity, creativity, resourcefulness and team-work ability.

Linn and Dalbey (1985) studied programming psychology and mental models of novice programmers. The study suggested that a good pedagogy should keep initial facts, models, and rules simple and only expand and refine them as the student gains experience. They complained about the sophisticated materials that were taught to introductory programming students while study had shown that they failed to understand the basic concept of a simple element – like variables and assignments – even weeks after the course began. This study suggested that spending more time on simple algorithms might pay a much larger return in the

long run.

du Boulay (1986) pointed out that some misconceptions are based on analogies used by teachers. He explained that when a variable is compared to a slate where values are written, learners might not think the existing value gets overwritten. They might think that a variable is a list which keeps all the values that have been assigned to it.

Haberman and Kolikant (2001) implemented a new instructional method, aiming to help novice high-school students to develop a mental model of basic concepts of computation. In this method a basic computational model of input and output, variables and value assignment, and the notion of executing a simple sequential algorithm could be constructed through activating "black boxes". Each "black box" demonstrates the properties of a new concept and its role in the computing process which incorporates a set of correct pieces of knowledge (CPK) associated with the concept. Haberman et al. believed that the "black boxes" method enabled novices to create a valid mental model by incorporating a complete set of CPKs concerning the concept. The study described the CPK for a model of a variable: "the variable is a location in the computer's memory that can contain one value of a specific type at a time; the value of a variable may be used (more than once) as long as it is not changed by any input/assignment statement". The study pointed that the lack of any of the required CPKs, or adapting alternative wrong pieces of knowledge (WPK), gives an incorrect perception of the concept. Haberman et al. also conducted research aimed at assessing the effectiveness of the method on novices' perceptions of basic concepts in computation. They indicated that students who learned according to the "black box" approach gained a better understanding of the basic computational model, compared to students who learned according to the traditional approach.

Lahtinen et al. (2005) organised an international survey with participation of more than 500 students and teachers to get opinions about difficulties in the programming teaching/learning area. The survey result showed that students seem to be quite confident to study alone rather than attending lectures; that learning by doing was considered to be more effective, therefore they asked for more exercise, practical sessions and to be left alone to accomplish their programming coursework on their own; example programs were considered as the most useful type of material both by the students and the teachers; the teachers opined that practical sessions in computer rooms, exercise sessions in small groups and

working alone on coursework are the most effective learning situations; the teachers seemed to think that the students needed guidance more than the students themselves; the teachers considered their teaching more effective than it actually was, because they rated all the guided learning situations more highly than the students did. They concluded that the major difficulties are the lack of effective learning and teaching materials in programming courses.

2.2.3 Lack of domain knowledge

Adelson and Soloway (1985) described how many programming problems comes from a wide range of problem domains and how having a correct mental model of the problem domain is critical.

Pennington (1987) looked at the way that expert programmers understand problem domains and programs. He explained that despite the admonition of the computer science establishment to construct programs top down, experts build them bottom-up. He emphasised that knowledge of the problem domain is one way that experts have an advantage over novices: even if they know no more about programming language than novices, they know a lot more about their problem domain, and they utilise that knowledge. He suggests that five types of programming knowledge are necessarily to enable a novice to overcome the syntactic and semantic requirements of a programming language: control flow, data flow, function, state and operations. Pennington believed that lack of knowledge of the problem domain and imperfect mechanical understanding of programming constructs cause novices' difficulties.

Lahtinen et al. (2005) believed that programming is related to several fields of technology, and universities just provide the basic concepts of those technologies. Students with problems are often stuck at the beginning of the introductory courses, as proved by the high drop-out rates.

2.2.4 Complexities in programming constructs

Researchers in this area tried to examine the complexity of programming in order to identify the complexities underlying structural characteristics of programming constructs. They also tried to evaluate the usability of some programming languages in order to understand novice learners' difficulties.

Bonar and Soloway (1983) were the first to raise questions like: "Why is programming, even at a simple level, so difficult to learn? Are novices' difficulties really inherent in programming or are they related to the nature of the programming tools currently given to them?". They stated that novices' programming difficulties stem from an inappropriate use of natural language specification strategies and suggested that skill in natural language seemed to have a great deal of impact on their conceptions of programming.

Dyck and Mayer (1985) conducted an experiment with two groups of students. Students in the first group were given some statements in BASIC to understand and students in the second group, who had no knowledge of BASIC, were given some statements in English to understand. They also looked at factors which influence the difficulty of the comprehension process for English and BASIC procedural statements; they found that the micro-structure of each statement (the number of actions required) and the macro-structure (the number of other statements in the program) were strongly related to response time performance for both BASIC and English. They concluded that understanding of procedural statements is related to underlying structural characteristics common to both languages.

Thomas Green made enormous contributions identifying novices' obstacles in design and construction of programming language. Green (1997) exposed the falsity of claims that a programming language is easy to use because it is more natural, or because it works the way people think. He explained how imperative programming, object-oriented programming, logic programming, functional programming, graphical programming, and others are all "natural" in someone's eyes but none of them uniformly best for all purposes. He added that the obvious point has been missed out; a programming language cannot be natural; it is not really like natural language, and if it were "natural" it would not be so hard to learn or understand.

Green (2000) put forward a "cognitive dimensions framework" for usability evaluation of all types of information artifact, from programming languages through interactive systems to domestic devices. Cognitive dimensions provides an important list of points to be aware of during the design of any notation. Green introduced some of these cognitive dimensions as:

- Viscosity – resistance to change.

- Premature commitment – constraints on the order of doing things.
- Hidden dependencies – important links between entities are not visible.
- Role-expressiveness – the purpose of the entity is readily inferred.
- Error-proneness – the notation invites mistakes and the system gives little protection.
- Abstractions – types and availability of abstraction mechanisms.

Green (2000) measured the difficulty levels of different languages, and found some were much worse than the others. Green discussed the cognitive complexity of the fact that in some programming languages – such as C, C++ and Java – array indices start at 0, rather than 1 as is more traditional in mathematics. This is a very efficient technique, as array variables in C are pointers to the start of the array, and the index is actually an offset into the array. He added that for programmers, however, it can be difficult to remember and to take into account in all appropriate calculations that a 10 element array actually has indices 0-9, rather than 1-10.

Later, with Blackwell (Blackwell et al., 2002), he evaluated the usability of information-based artifacts and notations such as programming languages. They opened issues such as cognitive ergonomics and language usability. They suggested that a usable programming language should be designed around the needs of the programmer, rather than the needs of the machine.

Shneidermann (1980) investigated different uses of variables and addressed issues such as assignment statements and the difference between the variables and the value stored in the variable, printing, using, changing, and comparing the value stored in a variable as well as the different types of variables (integer, float, character). He mentions three properties or types of variables:

- Counting with a variable
- Summing with a variable
- General uses of a variable

Across the first three types he raised the issues of initialisation, incrementation, final values, and forming totals.

Samurcay (1985) does a similar job by explaining the four ways variables are assigned values through assignment statements:

- Assignment of a constant value (`a=3`)
- Assignment of a calculated value (`a=3*b`)
- Duplication (`a=b`)
- Accumulation (`x=x+1`)

He described how each of these techniques can be used within two different contexts:

- External – where variables are inputs to or outputs of the program, under control of the program user.
- Internal – where variables are necessary only for the solution of the problem and are controlled by the programmer.

He explained that internal variables will be harder for novices to process and supported his claim by illustrating three types of variable, involved in a loop process:

- Update (accumulation variable)
- Test (condition for terminating the loop)
- Initialisation (initial values of loop)

He reported that novices found more difficulties with initialisation than updating or testing.

McIver (2001) explained that a programming language is a type of user interface, and hence that usability principles apply to the design of a programming language in the same way as they apply to the design of any other user interface. She brought up evidence that the first programming language has considerable impact on the novice's learning process. McIver described some pedagogical problems which were frequently found in languages used for introductory programming and introduced a framework (GRAIL) to evaluate these languages.

Lischner (2001) introduced a specific kind of homework which he believed can provide structural dialogues with novices and improve their understanding of

programming and eliminate language obstacles. Lischner's proposed homework is called "an exploration", and was described as: the student must read a short program first and then answer questions about that program, make predictions about the program's behavior, and then test that knowledge by running the program; follow-up questions ask the student to make some predictions; if a prediction was wrong, the student is asked to give a reasonable explanation. Lischner claimed that using the exploration method increased student satisfaction, retention, and learning.

2.2.5 Summarising explanations

Teachers and psychologists tackled learners' difficulties from different directions and despite diversities in the research methods used, they reached similar conclusions in most areas. Hypothesising learners' thinking when they stumbled over programming complexities helped researchers to have a clearer picture of the problem by fitting the bits of the puzzle together.

The lack of a correct mental model of the underlying virtual machine was found to be a problem with enormous consequences. Issues included using meaningful names for variables, hoping that the machine would be able to read and understand it; problems with variables, assignment, sequence, recursion and so on.

Spohrer and Soloway brought up the idea that novices use some mental models in their first attempt, and Mayer added that models can be developed by using relevant experience through meaningful learning, hence some novices who were experienced problem-solving strategists would compare programming problems with past problems and construct a rational solution using familiar techniques. From what Mayer, Spohrer and Soloway indicated, we can speculate that a successful novice may have already developed some mental models required in programming skill in advance, by a variety of factors such as experience, knowledge of the problem domain and so on.

On the other hand a number of complexities were found in programming constructs. Issues such as using variables with different types (integer, float, string and so on) in different roles (loop counter, check, sum) different contexts (internal, external) and with different purposes (initialisation, assignment) were recognised as some of the complexities in the design of programming constructs

which puzzle novices.

Inaccurate teaching such as misapplication of analogies or delivering concepts in an incorrect order (according to their dependencies) were found to be confusing. It was recommended that spending more time on simple algorithms might pay a much larger return in the long run.

Lack of domain knowledge was reported to be a crucial factor in this learning process, therefore avoiding applications with complex domain knowledge, to start with, was strongly recommended.

2.3 Interventions

Researchers in this group are mostly teachers who think they know, or hope they know, the solution for this peculiar problem. They can be categorised in two divisions by their intervention strategies: those who are introducing new languages and tools, advising other teachers on *what* to teach; and those who are presenting new approaches to the teaching of programming, advising *how* to teach.

As we shall see, there are some difficulties in drawing conclusions from research in this area.

2.3.1 What to teach?

The group who advised on what to teach are described briefly here in two sections on programming languages and tools.

The history of invention of programming languages shows that simplicity was an urgent issue from the beginning. From assembly language which associated symbolic names with the machine-language codes, an enormous number of languages were introduced one after the other, aimed to simplify programming. Even children were invited to experience the programming world with Logo with its moving "turtle" (Papert and Minsky, 1969). Programming languages can be grouped as:

- Imperative languages – e.g Fortran, Lisp, Algol/60/68, PL/1, Simula, Pascal, C, BCPL, Ada;
- Logic languages – e.g Prolog;

- Functional languages – e.g Miranda, Haskell;
- Object Oriented languages – e.g Smalltalk, C++, Java.

Although simplicity in a programming language was always the main issue, despite all these efforts the number of drop-outs in introductory courses shows that the problem still persists. All languages were claimed to be simpler to learn and easier to teach than their predecessors, but none are simple or easy; even paying attention to the programming development environment has not changed any fundamental issues to ease the complexities of learning programming. In my view programming languages are designed for the successful programmer not the novice who stumbles over basic concepts. Looking at this situation, I decided to avoid the study of programming languages.

2.3.2 Visual tools

There is a vast quantity of literature describing different tools such as Integrated Development Environments (IDEs). Researchers suggested that by making programming point-and-click, novices will find it easier.

Boyle et al. (2003) tried to tackle this problem with a top-down approach, aiming to improve the learning experience for first-year students. Their study focused on three main areas: curriculum development, organisation of the teaching environment, online learning environment. The evaluation of this research was published a year later, stating that the system had been used by over 1,500 students in Manchester Metropolitan University and two other British universities. The result indicates a 10% to 20% increase in pass rate extracted from different courses but the impacts of individual components used in this project were not exposed. The improvement in pass rate was as a result of many changes made in the course and it is difficult to find out the influences of individual factors. As we shall see, these are typical problems with this kind of research.

Giannotti (1987) designed and implemented VISAL, an interactive visual tool to support learning programming. The tool was designed to stimulate laboratory activities and to facilitate the development and debugging process of a program. VISAL was able to animate the execution of a program, and contained a library of fundamental algorithms to support visualisation. The study claimed that visualising the execution of a given program would enable students to have

a better understanding of the dynamic aspects of programming. An experiment was carried out on undergraduates of a programming course in order to verify the effectiveness of the VISAL implementation as an aid in learning activities. In this study the lack of a statistical evaluation to demonstrate that VISAL had a significant effect on candidates success is noticeable.

Ramadhan (1992) introduced "Discover" as a tool that was designed to simplify the programming development environment using a rapid feedback mechanism. This tool targeted novice programmers, helping them to build up their programming knowledge and skills. "Discover" is an interactive interface that helps novices to program, using common-sense logical phrases. In this study, however, the tool's usability, practicality and its impact on novices' skills enhancement were not evaluated.

Boyle and Margetts (1992) introduced the CORE (Context, Objects, Refinement, and Expression) methodology to the design of interactive multimedia learning environments to provide simplicity through the use of software visualisation tools in programming. Their tools were widely used by students in North London University and several systems have been developed using the CORE method. Boyle et al. (1994) built CLEM (Comprehensive Learning Environment for the Modula-2 language), using a set of design principles extracted from the study of language and cognitive development.

Boyle and Green (1994) described VirCom (Virtual Computer), for learning about hardware by constructing an end-user virtual computer. Boyle et al. (1996) described DOVE (Dynamic Observation in a Virtual Environment), a structured tutorial and virtual field trip in animal ecology. Gray et al. (1998) extended a version of CORE to create a Web-based system in order to facilitate the teaching of the Java programming language in an enriched environment. The outcome of this project has been evaluated as an encouraging and supportive tutorial, with a down side of inadequate feedback mechanism in its prototype system.

Quinn (2002) introduced the "Interrogative Programming tool", in order to ease the process of programming for novices. The tool asks a series of closed ended questions, in order to discover what the user wants to do. The answer will be either a selection from a list of choices or the raw input of a string or number and each choice clarifies some aspect of the program. There were some problems reported with this tool such as: the proposed model could not support functional decomposition; the tool forces the programmer to solve all problems

in a depth-first way while the novices could hardly go that far. Although several unsolved problems have been reported, the authors claimed that the tool is a paradigm with substantial potential to teach programming to novices.

Chalk et al. (2003) described a new approach to solving the problem of teaching first year software development, using web-based multimedia learning objects (LOs), which include student interaction, animation and self-assessment. A variety of evaluation techniques have been applied, including student questionnaires and interviews and server-side tracking. They reported some improvement in student performance in the first semester.

In my view the automated standard tasks provided by IDEs encourage students to deploy a number of pre-built components without understanding the programming mechanism which underlie their actions. It makes it easier for them at first but various essential programming concepts will be hidden from them. Consequently their understanding will be restricted to a shallow and inadequate level. Lack of understanding is revealed when something in the program goes wrong or an essential change is needed which cannot be dealt with in the IDE. I think experienced programmers get more advantage from IDEs than novices because they have the knowledge of the programming mechanism behind their actions and IDEs basically help them to speed up their progress.

2.3.3 How to teach?

The group of teachers who advised how to teach programming proposed a variety of methods, teaching programming through formalisim, programming-first, concept-first, object-first and so on. I try to explain a few of them briefly here.

Programming-first

Programming-first was one of the preliminary approaches which was often used in introductory computer science courses. In this approach most emphasis was given to control structures such as sequence, conditions, loops, recursion and so on. When object-oriented design and implementation were first introduced, programming-first was still the most common approach. Learners were taught those constructs required for imperative programming first and then were exposed to notion of classes and objects later in the course. Hence introducing the constructs required for imperative programming forced teachers to hide the

features and concepts of object-orientation from learners at the beginning. Then the programming-first approach became inadequate.

Concept-first

The concept-first approach with its variety of tools and methods became the most common replacement for the programming-first approach. The notion of analogy between programs and the real world is a widely used method to teach programming concepts. Teachers try to make a link between a real world activity and a program behaviour (execution). By using analogy they compare a programming behaviour with similar activities that learners might be supposed to understand. Teachers first introduce a real world activity, helping novices to observe the features (sequences/repetitions/controls) behind the activity and when the activity is clearly understood, help them to use the knowledge to understand the proposed formal program execution.

Some positive and some negative effects of analogy used in this way are reported by researchers. Curzon (1999) and Curzon (2000) encouraged teaching programming through analogies with games and puzzles. Lopez Jr. (2001) proposed the use of analogy in teaching declarative programming. Neeman et al. (2006) suggested the use of analogy to teach programming. On the other hand du Boulay (1986) and Samurcay (1985) showed how misapplication of analogy can increase confusion (see section 2.2.1).

Klinger (2005) used Stanislavski and Reynolds Hapgood (1984)'s *director* and *actors* analogy in computer science, teaching concepts of Object-Orientation. In Stanislavski's method the director asks the actor to "be an old oak tree". The actor is told to understand what it is to be an old oak tree. What does it see? What does it do? What happens if there is a fire? Klinger substituted programmer for actor in Stanislavski's statement and believed that it is exactly the same sort of thing that programmers are asked to do when they invent a new class in Object-Oriented programming; understanding of what an object must know (its members) and to know how an object will act and react. Klinger found the personification and acting out of computer science concepts to be a powerful teaching technique which enables students to quickly grasp new concepts and gain insights that they otherwise might not have got and which also makes a class more interesting and fun. But this study is another example of teachers'

experiments, described without any scientific evaluation or follow-up report to support the claim.

DuHadway et al. (2002) stated that the concept-first approach is based on three principles:

1. "Drawing on the student's everyday experiences when introducing the principles of computer science in order to ensure that meaningful learning takes place, instructors and designers can employ a variety of strategies to help learners relate their prior knowledge to new information they are to acquire."

2. "Allowing the students to work within a single domain for a period of time before adding a second or third one. Typically programming consists of three domains: general programming concepts; a programming language; and a development environment. It takes time for a student to assimilate new material from any one of these domains. Expecting them to learn new material from all three domains simultaneously may be too much for many students."

3. "Separating computer science concepts from language syntax. Separating concepts from language syntax helps build a cognitive framework that gives students a structure on which they can hang new ideas."

Goldman (2004) used JPie, an integrated programming environment – JPie enables live construction of Java applications through direct manipulation of graphical representations of programming abstractions – to present a concepts-first method in an introduction to programming course, exposing students without programming experience to Object-Oriented programming concepts. He argued that if students could directly manipulate programming abstractions, rather than encoding them textually, syntax difficulties could be by-passed and students could move directly into exploring ideas. He concluded that integrated programming environments allow students to modify programs while they are running and students can learn more easily through experimentation. Since the programming environment supported a standard model of computation, students who continue in a standard computer science curriculum could transfer much of their knowledge and experience. Although most of the students in this course did not intend to major in computer science, they learned a broad set of concepts.

Formalism

Bornat (1986) explained his attempt to teach programming via formal reasoning. He argued that expert programmers can justify their programs, so let's teach novices to do the same. The novices protested that they did not know what counted as a justification, and Bornat was pushed further and further into formal reasoning. In Dehnadi and Bornat (2006) he described how after seventeen years or so of futile effort, he was set free by a casual remark of Thomas Green's, who observed that people don't learn like that, introducing him to the notion of inductive, exploratory learning.

Objects-first

Recently, teachers introduced the idea of an objects-first approach, promoting the notion of teaching classes and objects at the beginning of a course, and many new textbooks have followed this approach. The choice of environment, however, remains an issue. Despite Java being consistently described as an excellent language for teaching, its environments were regularly identified as a significant source of problems and valuable teaching time is spent teaching the students how to use the environment.

Kölling and Rosenberg (2000) introduced "BlueJ" as a Java program development environment, which addressed these issue. They believed BlueJ helps novices to avoid Java's platform setup problems, and that by diagramming classes and objects in UML-like format it simplifies the complexities of introducing objects and their relationships to novices.

Barnes and Kölling (2006) explained how features in the BlueJ environment can be used to create an introductory Java course that fully embraces the "objects first" approach. They added that BlueJ provides graphical support for object-oriented design, abstracts over files and the operating system and provides fully integrated support for a design, edit, compile and test cycle. They also explained how BlueJ supports interactive creation of objects, interactive calling of methods of objects, includes an easy-to-use debugger, support for applications and applets and support for incremental development, one of the major advantages of object-orientation. They believed that BlueJ combines powerful tools with an easy-to-use interface, avoiding the complexity that creates so many problems when using existing environments in a classroom and most importantly is focused on a

teaching context.

Cooper et al. (2003) discussed the challenge of the objects-first strategy with a new approach. The new approach was centered on the visualization of objects and their behaviors using a 3D animation environment. They presented a series of examples, exercises, and projects with solutions. Developing a large collection of examples, despite being a time consuming task, should be done if the associated approach is to be successful. They also compared the pedagogical aspects of this new approach with other relevant work and provided statistical data as well as informal observations as evidence of improved student performance as a result of this approach.

Bruce et al. (2001) explained that although in the objects-first approach many concepts must be introduced before students can understand the construction of classes, students were required to think about the programming process with a focus on methods and objects from the start. They described their invented library "OO-from-the-beginning" developed to support learners in the object-first approach. They used graphical objects with event-driven programming, believing that an interactive graphical environment helps learners to use objects as well as writing methods early while designing and implementing interesting programs. Unexpectedly they proposed to introduce concurrency in the early stage which they believed is a natural extension of single-threaded execution and a way to simplify the interaction of objects.

Cooper et al. (2003) discussed the challenge of the objects-first strategy. They explained that students must dive right into classes and objects, their encapsulation (public and private data, etc.) and methods (the constructors, accessors, modifiers, helpers, etc.); concepts of types, variables, values, references as well as frustrating details of syntax will be added to the other complexities. They added that the objects-first strategy taught through an IDE like BlueJ requires learning of event-driven concepts and the details of its graphical user interface; and all those concept, ideas and skills, which presents various mental challenges, must be grasped almost concurrently.

2.3.4 Difficulty of researching interventions

In my view, there are always some factors hiding underneath teachers' experiments which undermine the accuracy of their claims to produce some effect upon

the learning of programming. I list a few of these hidden elements that I have noticed in teachers' interventions:

- The sustainability of the effects in teachers' interventions was rarely followed up. The effect on students performance, if temporary as I suspect, could be as a result of enthusiastic teaching/learning social individual interactions which are known as the "Hawthorne effect" (Landsberger, 1958).

- Teachers normally conduct experiments in their own classrooms, using small numbers of students without control groups, and try to examine the effect of the new changes by comparing the candidates' final results with the results of students in the same course the year before, ignoring the fact that things change year by year in the normal course of effect – e.g the content of final examinations, the numbers of candidates who chose programming as a major subject or those who had prior experience; as well as economic/social/political changes that may have influence on students' performance in each year.

- The effects in teachers' experiments occurred in the context of a number of changes which may have caused these effects: e.g a web-based feedback mechanism is introduced to support a objects-first approach which is presented in a number of discussion groups; and at the end of the course the effect of the objects-first approach method is evaluated as a result of all these changes.

2.3.5 Summarising interventions

Researchers in this area proposed a variety of methods to teach programming to facilitate learning programming. They also put considerable effort into proposing new languages with more attention on pedagogy. Introducing event-driven languages within a graphical Integrated Development Environment was an attempt to make programming point-and-click, with the hope that novices will find it easier.

Inventing new languages and tools directed more attention on to the capabilities of teaching methods. When object-oriented design and implementation were introduced the programming-first approach became inadequate, because it forced teachers to hide the features and concepts of object-orientation from learners at

the beginning when introducing the constructs required for imperative programming.

The concept-first approach was found to be the most common replacement for the programming-first approach, and it could handle imperative languages as well as Object-Oriented programming languages. The most recent method is the objects-first approach to teach Object-Oriented programming. This approach is supported by a variety of tools, web-based multimedia object libraries, and animations to simulate the execution of a program and a number of IDEs.

Some weak effects were reported as a result of teachers' interventions which were evaluated in the context of several other changes that occurred at the same time and the effect of the intervention was hardly ever followed up to check if it was sustainable or was just a Hawthorne effect. Although most teachers' interventions were reported to have some effect on learners' performance, the number of drop-outs in introductory programming courses remains very high.

2.4 Summary and conclusion

Empirical research has sought a reliable predictor which can categorise novices on the basis of a non-programming attribute at the beginning of an introductory course. Research has turned up very little, after five decades.

The result of most studies in this category convinced me that predicting success in an introductory course is a complicated issue. Large projects (Lister et al., 2004; Fincher et al., 2005; Raadt et al., 2005; Simon et al., 2006a; Tolhurst et al., 2006; Simon et al., 2006b) clearly indicate that a large proportion of students fail entry-level programming, but none found a good predictor. Tukiainen and Mönkkönen (2002) pursuing a reliable success predictor, failed in their latest attempt: the Huoman (1986) programming aptitude test gave no correlation. Even a general abstraction ability does not assist learning programming, as reported by Bennedsen and Caspersen (2006) and in (Bennedsen and Caspersen, 2008) they also found no correlation between cognitive development and results in a model-based introductory programming course.

There was enough evidence in the literature to convince me not to seek a categorising non-programming attribute.

In general I found some difficulties in drawing conclusions from research in teaching intervention. Although most researchers claimed that their proposed

language, tools or teaching methods made programming easier to learn, the number of drop-outs in introductory programming courses remains very high. In fact the effect of their intervention was always evaluated in the context of several other changes that occurred at the same time and the research was hardly ever followed up to check if the improvement was sustainable. Studying the research in this area did not satisfy me that I could find a clear result; therefore I decided not to sway my study in this direction.

This study takes most of its inspiration from the group of researchers who tried to explain programming learning difficulties by looking at learners' psychology and hypothesising learners' thinking when they stumbled over programming complexities. They stated that learners' understanding of program execution plays a major role in the learning process.

Gentner and Stevens (1983) introduced a notion of a mental model which has process and states. It is similar to the processes and states underlying a program execution. Mayer's emphasis on the importance of understanding the mental model of virtual machine as a crucial element in learning programming was influential (Mayer, 1981). The studies of Spohrer and Soloway on novices' rational misconceptions and the way they hypothesised what novices thinking as they programmed were inspiring too. Their findings on the relationship between mistakes and interference of background knowledge on learning programming were valuable clues to research in this area (Spohrer and Soloway, 1986; Soloway and Spohrer, 1989, 1988). du Boulay demonstrated how misunderstanding of tiny elements in programming can have major effects. His categorisation of novices' difficulties and the way he classified them according to mental models (du Boulay, 1986) was a major influence in my study.

Chapter 3

Initial Work

In the early stages of this research I made an effort to find a suitable method to facilitate understanding of difficulties in learning to program. In the literature, research concentrates either on teachers or on learners. In order to build a methodology I needed to study a number of practical research methods. Therefore I decided to make an initial study of methods in both the teaching-centered and learner-centered areas to get some general ideas about the strengths and weaknesses of possible research methods as well as estimating my own skills and limitations to carry on the research confidently within the allocated time. This initial work is briefly explained in this chapter.

One of the most common techniques in research on teaching methods is to look for the effect of a proposed method on learners' success, while research on learners looks for particular characteristics/attributes of learners which have an effect on their success at the end of the course. Observation, interviewing, psychometric tests and the study of common mistakes/misconceptions are the most popular techniques when research concentrates on the learning process.

3.1 Teaching-centered research

I initially had the impression that teaching-centered research would be about studying what teachers do when teaching. I found in the literature that that is not what researchers do; instead they propose a variety of teaching methods and tools for teachers to use. Some example proposals are:

- Use analogy as a tool to build a bridge between the real world and a formal

programming construct (Curzon, 1999, 2000; Neeman et al., 2006).

- Use graphical examples and tools in order to visualise formal programming (Boyle et al., 2003).
- Use novel teaching strategies (DuHadway et al., 2002).
- Use technical tools like an IDE (Integrated Development Environment) which offers a graphical user interface to simplify the process of writing a program (Chen and Marx, 2005; Allowatt and Edwards, 2005).

Ideally the effect of a new teaching method should be examined by comparing two groups within a large scale educational experiment: an experimental group which will be taught with the intervention method; and a control group which will be taught conventionally. Unfortunately this is not usually what happens. The samples are mostly small numbers of students that teachers use in a classroom as a experimental group and rarely have a control group.

Educational effects are in any case hard to assess because of many factors such as enthusiastic teaching/learning and social individual interaction additional to the complexity of the intervention. An example is Chalk et al. (2003) who used web-based multimedia learning objects (LOs), Java graphics library and virtual learning environment (WebCT) all at the same time to improve first year students' performance in software development and claimed that the changes had an effect on students' performance in that particular academic year. It's very hard to say which one of those elements (LOs, Java graphics library, virtual learning) caused the most effect and which one the least. The study also failed to report whether the effect was sustained afterward or not.

3.1.1 Teaching by analogy

In chapter 2, section 2.3.3, I reviewed some studies which proposed using analogy when teaching programming. In an experiment I used an analogy to explain the role of a counter in a loop process. The description of the real world activity was "a cleaner cleans 10 rooms in a day". When the activities of the cleaner's job were discussed and well understood, the associated formal programming construct was introduced. There were several obvious ways of writing the program:

- until `E` do `C`

```
int room = 1;
while(room <= 10)
{
   clean();
   room ++ ;
}
```

Figure 3.1: An example of a "while" loop

- while `E` do `C`
- do `C` until `E`
- do `C` while `E`

The second and fourth styles are used in Java. Because the course was designed to teach programming by Java I decided to use the second (figure 3.1) to build up a program which represents the "cleaner" analogy.

Some students strongly responded to this analogy and managed to understand the link between the cleaner's activities and the associated program's behaviour, while some found it very difficult from the beginning and became confused. There were two points made by students when I tried to explain the code:

1. Students were not happy that the "cleaner" should clean the rooms in the order of room 1 to room 10. They believed that the "cleaner" should choose the rooms in the order of their location or in any order she/he likes. I could have used a different loop construct (figure 3.2) to hide the counter but I would have first had to explain the concept of array, as well as missing the target of the role of a counter in programming.

2. They also were not happy that in the formal programming "10 days" comes first and "clean()" comes later while we described the real world activities of the cleaner's job in the reverse order.

The students first objection to the ordering mechanism in this analogy opened a discussion about the fact that a computer does thing differently from how we do it and this is what they should learn and accept. They failed to understand

```
Room rooms = new Room[10];
for(Room r :  rooms)
{ clean(r); }
```

Figure 3.2: An example of a "for" loop

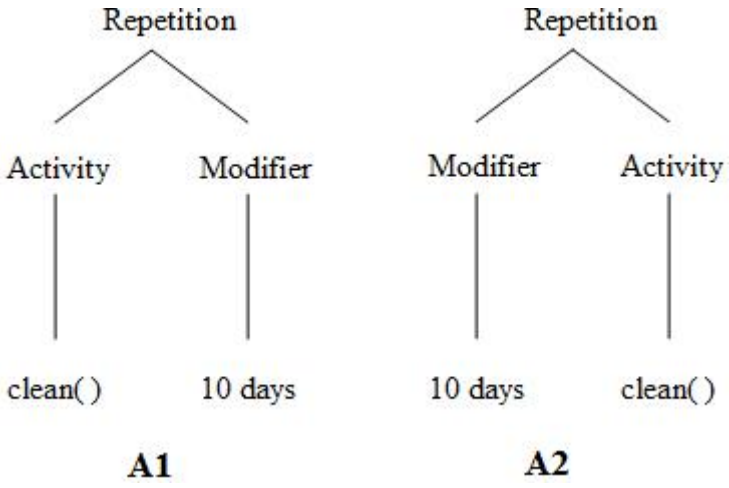


Figure 3.3: Syntactic structures for cleaner's progam

```
L1      int room ;
L2      int day = 1;
L3      while(day <= 7)
L4      {
L5        room = 1;
L6        while(room <= 10)
L7        {
L8          clean();
L9          room ++;
L10       }
L11       day ++;
L12     }
```

Figure 3.4: An example of a "nested while" loop

that although in the real world cleaners clean rooms in any order that suit them, in the programming world, the order is compelled by a computer program.

Regarding their second objection, apparently the differences between the analogy's description and the appearance of the program structure caused the problem. They expected to see all elements of the analogy's description linearly mapped to the structure of the associated program. They did not understand how the structure of program conveys meaning and lack of experience prevented them from realising that in figure 3.3 A1 and A2, despite having different concrete structures, have exactly the same abstract structure. Use of the alternative loop presentation in Java "do `C` while `E`" could be a response to their second objection but the problem with the counter would remain.

Later, I expanded the cleaner analogy to "a cleaner cleans 10 rooms each day of the week" in order to expose them to nested loops. Again when the cleaner's job as a real world activity was discussed and well understood, the associated program was introduced (figure 3.4).

As well as the problems experienced with a single loop, more complications were created when nested loops were introduced.

- When I asked them to write a similar program they often wrote the loops wrongly nested, like repeating "10 rooms" outside and "7 days" inside.

- They kept forgetting to increment the program counters (L9, L11) and re-initialising `room` when the day is changed (L5). They thought the features

```
L1      int room ;
L2      int day = 1;
L3      do
L4      {
L5        room = 1;
L6        while(room <= 10)
L7        {
L8          clean();
L9          room ++;
L10       }
L11       day ++;
L12     }while(day <= 7);
```

Figure 3.5: An example of "nested do-while" loop

like days and months do not need to be incremented, they just happen.

Writing the loops wrongly nested seems to be caused by the same problem of the differences between the real world activity's description and the appearance of the program structure that were observed when the analogy for the single loop was presented. Using the alternative loop presentation in Java "do `C` while `A`" might be a better example here (figure 3.5) but the problem with the counter issue would remain.

It appeared that the "cleaner" analogy, despite having a useful effect at a certain level in some individuals, was not completely useful to present a loop and introduce the role of counter and caused some confusion. I might have imposed the loop presentation without counter but I doubt that I could abolish confusion when nested loops were introduced. Maybe I could have found a better analogy which would cause less confusion. The "cleaner" analogy might be a weak example to judge the capability of analogy but it is a good example to show that even a simple analogy is not an identity and finding an identity for a formal program is hard if not impossible.

I did not find that teaching by analogy was a simple tool to look at programming learners' difficulties. As a research method I needed a simpler method to break down the complexities of the programming learning process. I decided to continue searching in experimental methods, looking for a clearer phenomenon for study.

3.2 Learner-centered methods

Learner-centered research focuses on learners, studying their background education, reasoning strategy, psychological attributes, cognitive behavior and mistakes they make. Three groups of researchers are described in chapter 2:

1. The first group measures learners' attributes and tries to predict their performance (Mayer and Stalnaker, 1968; Rountree et al., 2004).

2. The second group looks at learners' cognitive behavior and mental models. Spohrer and Soloway (1986) explain as follows:

   - Just a few types of bug cover almost all those that occur in novices' programs.
   - For most computerised tasks there is some model that a novice will use in his or her first attempts and we need to understand when is it appropriate to appeal to this model, and, when necessary, how to move a novice to some more appropriate model.
   - Novices plan to deal with the complexity of programming by breaking goals into subgoals.

3. A third group studied common mistakes, bugs and misconceptions in order to describe problematical areas (Bayman and Mayer, 1983; Shneidermann, 1985; Adelson and Soloway, 1985).

Studying common mistakes can reveal valuable indications of novices' misconceptions which can facilitate our understanding of problem areas. I believe that the considerable number of failures in introduction to programming courses reveals that the learning process in this subject is problematic. Thus learner-centered research seems to be appropriate. In learner-centered research, the difficulties of learning programming can be investigated from basic and foundation levels through a series of experiments.

I decided to move my study toward bottom-up learner-centered research and to study learners' difficulties objectively. I investigated the effect of a variety of methods such as asking them to explain their reasoning strategy, investigating their background education, their psychology and also looking at their common mistakes and misconceptions.

3.2.1 Observing problem solving behaviour

A considerable amount of data can be captured by observing novices' analytic behavior in their learning process. As a lab tutor I could observe students' obstacles by watching their debugging process and looking at their draft notes. On the other hand, interaction of researchers and learners is only possible within timetabled teaching slots; there were time restraints that prevented me from relying on this method as the study's main method.

Interviewing is a tool that can help teachers to see where problems start and perhaps to see the roots of misconceptions if the learners are able to express their mindset by describing the strategy behind their decisions. I have interviewed students informally at different stages of their study and logged information which was quite helpful in understanding learners' thinking routes and strategies. On the other hand it is a time consuming process for both teachers and learners and some learners were unable to describe what they think.

After a short while I started to think of adopting a more objective method. The observation method could still be used in parallel as a supplemental process. Observing candidates' misconceptions and problem solving behavior made a substantial contribution to this study and led me toward a valuable source of information.

3.2.2 Background education

One of my initial investigations looked at the effect of learners' background education. A questionnaire was designed in two parts. The first part had 10 questions aimed at assessing students' ability in algebra, numerical reasoning and general IQ and the second part had 10 programming-related questions to test their programming learning progress. The effect of candidates' ability in any of the background elements (first part) on their programming score (second part) was the objective of this study.

A sample question in Algebra:

```
What is the value of A if  A = 5+((2*(9+5))-4)/2
```

A sample IQ question:

```
Your sister is 8 years old. You are three times as old as her.
How old will you be when you are twice as old as her?
```

A sample numerical reasoning question:

```
Which of the following is the odd one out:
3 ... 9 ... 12 ... 24 ... 8 ... 16
```

A programming-related question:

```
int a = 10;  int b = 20;
what is the new value of a and b when:
a = b
```

I administered the test to 40 students in week 5 of a introduction to programming course in Barnet College on an informal voluntary basis after giving them 15 hours of instruction. The scores that candidates achieved in part one (maths, reasoning) did not have any obvious correlation[1] with the score they achieved in part two (programming). Some learners with a high score in maths and reasoning failed to pass the programming section while some with weaker maths and reasoning managed to achieve a good programming score.

I looked again, in more detail, at the effect of candidates' programming education background in experiments which will be analysed in chapter 6.

3.2.3 Psychometric techniques

Since I did not see any obvious effect of candidates' background education on their programming achievement I decided to examine if a psychometric test could predict programming success. During my literature review I came across the work of psychologists who were trying to separate learners according to their positions on a number of scales which indicate the way they receive and process information.

Mahmud and Kurniawan (2005) used psychometric tests for input-device evaluation with older people. Sutton et al. (2005) used conversion of a psychometric test to a web-based study to measure understanding of three-dimensional (3D) concepts as they apply to technical drawing. Borgman (1987), by using a psychometric test, found a wide range of skills in ability to use information-retrieval systems.

I decided to administer a test using one of the psychometric instruments, the "Learning Style and Strategies" model introduced by Felder and Silverman (1988), in order to investigate the correlation between psychological characteristics and scores in a final programming examination. Felder and Silverman's

[1]None of the results that I describe in this chapter were statistically analysed. I was looking for indications of strong association, not pursuing weak effects.

Table 3.1: Attributes of psychometric test (Felder and Silverman (1988))

	Dimension(a) 11 1	**Balance** 0	Dimension(b) 1 11	
ACT				REF
SEN				INT
VIS				VRB
SEQ				GLO

instrument measures learning and teaching dimensions that describe learning styles. They divide learners into "Active/Reflective", "Visual/Verbal", "Sensing/Intuitive" and "Sequential/Global".

In this instrument four different sets of questions were designed targeting the following issues (taken from Felder and Silverman (1988)):

- What type of information do they mostly respond to: sensory (external) sights, sounds, physical sensations, or intuitive (internal) possibilities, insights or hunches?

- Through which sensory channel is external information most effectively perceived: visual – pictures, diagrams, graphs, demonstrations – or auditory – words or sounds? (Other sensory channels – touch, taste, and smell – are relatively unimportant in most educational environments)

- How does the student prefer to process information: actively – through engagement in physical activity or discussion, or reflectively – through introspection?

- How does the student progress toward understanding: sequentially – in continual steps – or globally in large jumps, holistically?

Each student could be fitted into one of several learning styles within the proposed conceptual framework. The test was taken by 30 students of an introductory programming course in Barnet College a few weeks before their final examination. I did not find any obvious correlation between any of the psychometric attributes and programming learning success.

I searched the literature, looking for any reports of the effect of psychometric attributes on programming learning ability. I found Willoughby (1978) that in

a single-page paper, reviewed Penney (1975)'s work who referred to two studies which showed a significant correlation between aspects of systems analysis or programming and scores in standard psychometric tests. Lewis et al. (2005) demonstrated that out of two samples with the same ability to visualise, one could make progress and another could not. They also examined the effects of various measures of prior computer science experience and cognitive abilities on overall performance in a second-level programming course. The first sample was School A, a mid-sized comprehensive university, and the second sample School B, in a large research-intensive university. In school A, the cognitive ability to visualize was significantly related to course performance. However, when examining school B, no significant correlation was found.

Bennedsen and Caspersen (2008) found that general abstraction ability was not a predictor for success in learning programming. Tukiainen and Mönkkönen (2002) found no significant correlation between the Huoman test (Huoman, 1986) and success in the programming exam.

Otherwise I haven't seen any literature concerning whether the predictions made from a psychometric questionnaire significantly distinguish successful programming learners from the rest.

My own experience with Felder and Silverman's instrument, and the lack of studies confirming positive effect of any psychometric attributes on learning programming, made it seem an unproductive area of research. To make progress in this area would seem to require novel psychological measurements and more theoretical psychological insight than I possess. I decided to move on to further investigations in order to find alternative methods to study the programming learning process.

3.2.4 Common mistakes

The literature on common mistakes is quite rich, with considerable outcomes reported by researchers who have studied learners' common mistakes in the early stages of learning programming (Bonar and Soloway, 1983; Adelson and Soloway, 1985).

Observing types and frequencies of novice programmers errors, studying their syntactic and semantic errors, highlights the problematic areas. I decided to look at the most common mistakes, analysing each mistake individually in order to

hypothesise its cause, so as to make a logical explanation of it. Some of the problematic areas in programming constructs are as follows:

- Similarities between a programming construct and students' background knowledge cause interference. For example the "`=`" symbol, used as the assignment sign in Java, is the same as the symbol used to denote equality in school mathematics. In an assignment "`a=3`" might be read, thinking mathematically, as the value of "`a`" is "`3`" and remains "`3`" forever; while in a programming context the value of "`a`" is "`3`" only as long as another number has not been assigned to it.

- Another cause of interference is the same symbol used for different purposes. For example `+` represents concatenation in "`3`"`+i` but addition in `3+i` and incrementation in `i++`.

- In Java indexing, `0` is used as the first ordinal. For example for the array `a` with `n` elements, the elements are arranged `a[0]`, `a[1]`, `a[2]`, `a[3]`, ..., `a[n-1]`, and a loop which initializes the array's elements to `0` is `for (int i = 0; i <10; i++) a[i] = 0`. Students have difficulty in understanding why the loop's counter starts from `0` (not from `1`) and ends with `9` (not with `10`).

- Variables are always problematic. Some novices imagine a variable as a pot which can stack numbers on the top of each other which when it gets a new value keeps its previous value too. Students may have misconceptions about the names of variables (Perkins and Simmons, 1988); assigning a variable to another variable (du Boulay, 1986); different uses of variables as "internal" and "external" variables (Samurcay, 1985).

- Alternative representations of a value always cause confusion, for example when the code:

  ```
  JOptionPane.showInputDialog("Enter a number")
  ```

  in Java returns a numeric result as a string. Understanding that a numeral can be seen as a string as well as representing a numerical value is not easy for novices.

- Soloway and Spohrer (1989) studied novices' errors and explained how their background knowledge interfered with their learning process and caused many of their misconceptions. Bonar and Soloway (1985) compared some programming constructs and put forward evidence to support the startling theory that prior knowledge of one programming language has a negative impact on novices' attempts to program in a second language.

3.2.5 First test

During the investigation of novices' common mistakes I administered a series of short quizzes on an ad hoc basis in the early stage (week 3/4) of an introduction to programming course. I aimed to identify common mistakes, catalogue them and pursue the misconceptions behind each individual mistake. When students' responses, rough notes around the test paper, and verbal explanations were analysed their deductive strategy behind each particular mistake became more visible. It appeared that some of the mistakes were not just slips or guesses or confusion, but there was some rational strategy behind them. It seemed that most of these mistakes had their basis in a series of recognisable models which could be used rationally.

I decided to move from ad hoc quizzes to more methodical test materials with a number of related questions. A new test was designed with a number of related similar questions in order to trace learners' mistakes step by step and illuminate logically related mistakes. I examined each individual candidate's response, looking for answers to the following questions:

1. Can I recognise any rational mistakes?

2. Have these rational mistakes occurred systematically in their answers to similar questions?

I administered the test in the 3rd week of an introductory programming course. The test result suggested that students had the ability to create a rational model, though perhaps not the Java model, logically acceptable as a possible answer to the question.

Some even managed to generalise their models and apply them systematically to answer most of the related questions. It appeared that some simple misconceptions can be extended to a series of related mistakes which are all based on the

same misinterpretation. An example illustrates how a candidate could make a rational mistake based on a simple misconception and apply it systematically. The two questions below were given to novice candidates who had not been exposed to post/pre-increment in programming:

```
Question 1:
int a = 10;
What is the new value for a if:
a++;

Question 2:
int a = 10;
What is the new value for a if:
a--;
```

Most candidates who picked `20` for the new value of `a` in the first question, picked `0` in the second question. They seemed to have the misconception that `a++` means `a=10+10` and `a--` means `a=10-10`. Their explanations confirmed the misconception when I asked them to explain it in an informal interview.

Again, most candidates who picked `30` for the new value of `a` in the first question also picked `-10` in the next question. They had the misconception that `a++` means `a=a+10+10` and `a--` means `a=a-10-10`.

When I interviewed candidates who were able to apply recognisable models behind their deduction process systematically, most declared that they had never been taught programming before. It became clear that there was an intellectual strategy behind their reasoning that had been extracted from what they brought with them, most likely from their prior education.

Capturing candidates' rationalisation patterns with such a simple test gave me an indication that something serious was going on and that I should narrow my study to focus on learners' pre-determined models. It was the first spark in this study that lit up a tiny slice of learners' minds. I decided to follow it in a deeper investigation.

3.2.6 Mental models

Reviewing the literature I became familiar with the notion of "mental model" which was introduced by Gentner and Stevens (1983) (discussed in chapter 2,

section 2.2.1) which directed my investigation toward mental models required for learning programming.

Mayer explained programming as a cognitive activity and said that novices are required to learn new reasoning skills as well as to understand new technical information. He introduced "mental model" as a framework that novices try to build up from their background domain-specific knowledge and their skill in understanding problem description in order to understand new information (described in chapter 2, section 2.2.1). I discussed Perkins in chapter 2, section 2.1.3, who called some novices "stoppers", who appeared to give up at the first difficulty, the others, as "movers" seemed to use a different approach to get beyond an stalemate. Mayer's interpretation of "mental model" and Perkin's explanation were appealing and led me toward a new stage of investigation.

The test that I administered in week 3 with subjects who had never been taught programming before revealed a series of recognisable models which logically were acceptable as a possible answer and it appeared that some students even used these models systematically. At this stage I thought the intellectual strategies behind their reasoning could have been built up from their background domain-specific knowledge to find a rational explanation for unknown phenomena. I decided to follow this in a deeper investigation, applying a more methodical approach in order to get a better understanding of novices' mental models.

Chapter 4

First methodical experiment

The result of the first test – see section 3.2.5 – suggested that students bring different patterns of rationalisation/explanation into the programming context. I narrowed my study by focusing on learners' mental models and moved toward a more methodical approach. I planned to conduct a series of formal experiments in order to examine the following research questions:

- Can we identify the mental models used in novices' responses?
- Can they apply their mental models systematically?
- Are these models pre-determined prior to the programming course?
- Can we categorise learners by the mental models they present?
- Have the mental models affected their success in the final programming exam?

I decided to devise a test and aimed to use it as a detective device to capture candidates' mental models.

4.1 Method used

I designed test materials as a questionnaire that apparently examines students' knowledge of assignments and sequence. I did not seek with this test to judge respondents according to the right/wrong answers they give, but to capture the reasoning strategy behind their interpretation of each question.

1. Read the following statements and tick the box next to the correct answer in the next column. int $a = 10$; int $b = 20$; $a = b$;	The new values of a and b: ☐ $a = 30$ $b = 0$ ☐ $a = 30$ $b = 20$ ☐ $a = 20$ $b = 0$ ☐ $a = 20$ $b = 20$ ☐ $a = 10$ $b = 10$ ☐ $a = 0$ $b = 10$ Any other values for a and b: $a =$ $b =$	Use this column for your rough notes please

Figure 4.1: The first question in the test, a single assignment

4. Read the following statements and tick the box next to the correct answer in the next column. int $a = 10$; int $b = 20$; $a = b$; $b = a$;	The new values of a and b: ☐ $a = 30$ $b = 0$ ☐ $a = 30$ $b = 50$ ☐ $a = 0$ $b = 20$ ☐ $a = 20$ $b = 20$ ☐ $a = 10$ $b = 10$ ☐ $a = 10$ $b = 0$ Any other values for a and b: $a =$ $b =$	Use this column for your rough notes please

Figure 4.2: The fourth question in the test, two assignments

4.2 Test materials

The questionnaire was designed in three columns: questions were in the first column, multiple-choice lists of the alternative answers were in the second column and the blank third column was for rough work, in which I occasionally found very interesting marks that the subjects made.

The questionnaire consisted of 12 questions. Each gave a program fragment in Java, declaring two or three variables and executing one, two or three variable-to-variable assignment instructions. The first three questions had only a variable-to-variable single assignment as illustrated in figure 4.1.

The next three had two variable-to-variable assignment instructions, illustrated in figure 4.2.

The last 6 questions had three variable-to-variable assignment instructions,

7. Read the following statements and tick the box next to the correct answer in the next column. int $a = 5$; int $b = 3$; int $b = 7$; $a = c$; $b = a$; $c = b$;	The new values of a and b: ☐ $a = 0$ $b = 0$ $c = 15$ ☐ $a = 12$ $b = 14$ $c = 22$ ☐ $a = 0$ $b = 0$ $c = 7$ ☐ $a = 7$ $b = 7$ $c = 7$ ☐ $a = 3$ $b = 5$ $c = 5$ ☐ $a = 3$ $b = 12$ $c = 0$ ☐ $a = 8$ $b = 15$ $c = 12$ ☐ $a = 7$ $b = 5$ $c = 3$ ☐ $a = 3$ $b = 7$ $c = 5$ Any other values for a and b: $a =$ $b =$	Use this column for your rough notes please

Figure 4.3: The seventh question in the test, three assignments

illustrated in figure 4.3.

4.3 Mental model exposure

The questionnaire asked the student to predict the effect of the program on its variables and to choose their answer/s from a multiple-choice list of alternatives. The questionnaire did not give any explanation of the meaning of the questions or the equality "`=`" sign that Java uses to indicate assignment. Except for the word "int" and the semicolons in the first column, the formulae employed would have been reasonably familiar to anybody who has experienced algebra. I expected that students would have some notion of what `x=y` might mean, and would use that knowledge in guessing what box to tick in the second column. The test looked like algebra but when the question asked about the "new values" of variables it hinted that the program produces a change.

I had a prior notion of the ways that a novice might understand the programs, and I prepared a list of mental models accordingly. The mental models of assignment that I expected my subjects to use are shown in table 4.1.

In the first three single-assignment questions all models correspond to a single answer. Observing different patterns of rationalisation/explanation brought in by novices I captured five popular strategies that were used most often by candidates in order to handle a single two-variable assignment like `a=b`. These strategies are as follows:

Table 4.1: Anticipated mental models of assignment `a=b` (question 1 figure 4.1 used as an example to explain the mental models)

Models	**Description**
(M1)	Value moves from right to left `a:=b ; b:=0` Ans (`a=20` , `b=0`) 3rd Answer
(M2)	Value copies from right to left `a:=b` Correct model of Java for assignment Ans (`a=20` , `b=20`) 4th Answer
(M3)	Value moves from left to right `b:=a ; a:=0` Ans (`a=0` , `b=10`) 6rd Answer
(M4)	Value copies from left to right `b:=a` the reversed version of Java model Ans (`a=10` , `b=10`) 5th Answer
(M5)	Right value moved and added to the left value `a:=a+b ; b:=0` Ans (`a=30` , `b=0`) 1st Answer
(M6)	Right value copied and added to the left value `a:=a+b` Ans (`a=30` , `b=20`) 2nd Answer
(M7)	Left value moved and added to the right value `b:=a+b ; a:=0` Ans (`a=0` , `b=30`) missed in the answer list
(M8)	Left value copied and added to the right value `b:=a+b` Ans (`a=10` , `b=30`) missed in the answer list
(M9)	Left and right swap values `b:=a; a:=b` Ans (`a=20` , `b=10`) missed in the answer list

1. They "moved" the value from one variable to the other (`a:=b` and `b:=0`). I called this strategy M1 when the "move" was from right to left and called it M3 when the "move" was from left to right.

2. They "copied" the value of one variable to the other (`a:=b` and `b` keeps its previous value). I called this strategy M2 when the "copy" was from right to left and called it M4 when the "copy" was from left to right.

3. They "moved" and "added" the value of one variable to the other (`a:=a+b` and `b:=0`). I called this strategy M5 when the "move" and "add" was from right to left and called it M7 when the "move" and "add" was from left to right.

4. They "copied" and "added" the value of one variable to the other (`a:=a+b` and `b` keeps its previous value). I called this strategy M6 when the "move" and "add" was from right to left and called it M8 when the "move" and "add" was from left to right.

5. They swapped the value of the variables (`a:=b and b:=a`). I called this strategy M9.

The last 8 questions contained more than one assignment and I expected more answers, because the respondents must use a model of composition of commands as well as assignment. I have come across only three models of composition. The effect of the combination of assignment models with sequence models is to increase the complexity of analysis of the results: in single-assignment questions there is more or less one model per tick; with multiple assignments there is considerable ambiguity.

The mental models of composition that I expected my subjects to use are shown in table 4.2.

Sequence (S1): The first assignment has its effect using the initial values of variables; then the second assignment has its effect using the state produced by the first; then the third has its effect using the state produced by the second; and so on for subsequent assignments (the 'correct' answer in Java).

Simultaneous-multiple (S2): Each assignment takes effect using the initial values of variables, and all effects are reported. This model has rarely been observed.

Table 4.2: Anticipated mental models of composition of `a=b; b=a` (question 4 figure 4.2 used as an example to explain the mental models)

Models	**Description**

Sequence (S1)	S1 is `a=b; b=a`
	Conventional sequential execution
	Suppose M1 Applies sequentially through both statements:
	L1) The value of `b` is given to `a` and `b` changes its value to `0`
	`a=20 ; b=0;`
	L2) The value of `a` is given to `b` and `a` changes its value to `0`
	`a=0;` and `b=20;`
	Result is a single answer `a = 0; b = 20;` 3rd Answer

Independent (S2)	S2 is `a=b || b=a`
	Independent assignments, independently reported
	Suppose M1 Applies independently for each line:
	L1) The value of `b` is given to `a` and `b` changes its value to `0`
	`a=20 ; b=0;`
	L2) The value of `a` is given to `b` and `a` changes its value to `0`
	`a=0 ; b=10;`
	Result is two answers:
	`a = 20 ; b = 0;` and `a = 0; b = 10;`
	(These answers are not in the list)

Simultaneous Single (S3)	S3 is `a,b=b,a`
	Simultaneous multiple assignment, ignoring effect upon source variable Suppose M1 applies to each line but effect on right-hand-side is ignored
	L1) The value of `b` is given to `a` and change to `b` ignored :
	`a=20;`
	L2) The value of `a` is given to `b` and change to `a` ignored :
	`b=10;`
	A single answer `a=20 ; b=10;`
	(this answer is not in the list)

Simultaneous-single (S3): Each assignment takes effect using the initial values of variables, but only the effect on the destination side is reported (in figure 4.2, for example, if the assignment model being used was right-to-left (M1/M2/M5/M6) the box would be ticked which reports the effect of the first assignment on `a`, the second on `b` and the third on `c`). This model has been observed more frequently than any others.

Some of the answers associated with these mental models were missed in the questionnaire by mistake and were added to the questionnaire when the methodology was enhanced (see chapter 5).

4.4 Test administration

I decided to administer the test in the first week before the subjects had received any programming teaching. I hoped to identify candidates' mental models at the beginning of the course, before any lectures had been given. It should be emphasised that I was not looking for right or wrong models; any models which offered a rational solution would be interesting.

The test was administered to 30 students on a further-education programming course at Barnet College. In this experiment no information was recorded about earlier education, programming experience, age or sex. I interviewed half of the students before admission, and taught them all.

The same test was then administered to 31 students in the first-year programming course at Middlesex University, once again before they had received any programming teaching. They were mostly male, aged about 18-20, from the middle range of educational attainment. This time I tutored them but did not teach the course.

In the questionnaire, I did not ask the candidates if they have had any previous contact with programming or not. An assumption had been made that they all had enough school mathematics to make the equality sign familiar.

I expected that after a short period of instruction the novices would display the model that corresponds to the way that a Java program actually works. I therefore planned to administer the same test for a second time to the same subjects after the topic had been introduced, and then a third time just before the examination, intending to track changes in subjects' mental models and their

understanding of assignment. I called these three administrations T1, T2, T3 and expected to correlate the results of them with each other as well as with the marks in the official end-of-course examination. Because of what was found on the first and second administrations, the plan for a third administration (T3) was abandoned.

4.4.1 Mental model test (T1)

Despite the risk that taking a programming-related test before giving the relevant instruction might cause participants' rejection, I received a full response from most participants. I combined the two populations (Barnet College and Middlesex University) when the results were analysed.

A small group gave a blank, or mostly blank response (answered none, one or two questions). Of the rest, about half gave answers which corresponded to a single mental model in most or all questions; the other half gave answers which corresponded to different models in different questions, or responded in unexpected ways like ticking three boxes which did not seem to correspond to a rational model.

Table 4.3 details the subdivision into three groups in the first test administration (T1):

1. 27 subjects (44%) appeared to use the same assignment model for all, or almost all, of the questions. I call this the *consistent* group "C".

2. 24 subjects (39%) appeared to use different assignment models in different questions or to use unrecognisable models. I call this the *inconsistent* group "I".

3. The 10 remaining subjects (16%) answered few or none of the questions. I call this the *blank* group "B".

Subjects were not interviewed after the test to determine anything about their answers, so it was not known whether students chose consciously or unconsciously to follow one strategy or another, nor how choices were motivated, nor what any particular choice meant to a subject who made it.

Table 4.3: populations in T1 result

	population
C	27
I	24
B	10
Total	61

Table 4.4: Shift in group membership between T1 to T2

	C (T2)	**I (T2)**	**Total**
C (T1)	25	2	27
I (T1)	11	13	24
B (T1)	5	5	10
Total	41	20	61

4.4.2 Mental model test (T2)

Teaching in the first three weeks of the course concentrated on assignment model M2 (right to left copy) and the sequence model (S1) of sequential composition. When the same test was administered in week 3, it was found that almost all the consistent subjects in T1 remained consistent in the second test (table 4.4, row 1) and that about half of each of the other groups became consistent. There were no blank returns in the second test.

Table 4.4 demonstrates that almost all the consistent subjects in the T1 remained consistent in the second test. It indicates that the T1 result was not an accident; the consistent subgroup was different. My original hypothesis was that subjects brought patterns of reasoning to the course, and changes between T1 and T2 seem to support that.

When I considered not only consistency but also use of the M2/S1 models (the correct model of Java) in the T2 test, I produced table 4.5. In this table, "CC" is used for candidates who were consistent in both tests, "CI" is used for consistent candidates who became inconsistent in the second test and so on.

I observed that:

- Only 29 (47%) of the subjects managed to grasp the meaning of assignment and sequence in Java within the first three weeks of the course.
- 21 (78%) of consistent subjects in T1 used the correct model of Java for

Table 4.5: Tendency toward correct model T1 to T2 (week 0 and week 3)

	Correct	**Incorrect**	**Total**
CC	21	4	25
CI		2	2
IC	5	6	11
II		13	13
BC	3	2	5
BI		5	5
Total	29	32	61

assignment and sequence in T2, while only 5 (21%) of inconsistent subjects and 3 (30%) of blank subjects in T1 used the correct models in T2.

This was the first indication that the consistent subgroup's performance was better than the other two subgroups. I review this table again in section 4.4.4, correlating it with the course's final marks.

4.4.3 Assessment of programming skill

Formal examinations are by no means a perfect measure of programming aptitude, but there is general agreement (Simon et al., 2006a) that they are the only measure available in large-scale surveys. Like other researchers, therefore, I have decided to correlate my test results with course examination results.

There were two in-course quizzes in weeks 7 and 11, which by arrangement were identical between the Barnet and Middlesex groups, and there were distinct end-of-course examinations in week 12. The marks from the three examinations were combined to form the overall mark. I had no access to final examination results. With subject cooperation, however, I was able to obtain their scores on the two quizzes.

For various reasons the examinations were designed to give those with no or minimal programming skills a chance to pass. The first quiz in particular contained a lot of "book-work" questions whose answers could be memorised from lecture notes, and a minimum of technical questions requiring programming skill and understanding. The second quiz was more engaged with programming, asked them to write code fragments and, using dry-run, to pursue the changing of values of a variable.

Table 4.6: T1 population and average grade

	Good	**Average**	**Pass**	**Fail**	**Total**
C	7	10	6	4	27
I	0	5	1	18	24
B	1	1	2	6	10
Total	8	16	9	28	61
$\chi^2 = 24.649,\ df = 6,\ p < 0.0001$ **very highly significant**					

4.4.4 Relating test and quiz results

I recorded the quiz results as a percentage (0-100), a grade (Good, Average, Pass, Fail) and a binary (Pass, Fail). I also worked out an average percentage, an average grade and an average binary from the two quizzes. Table 4.6 demonstrates the association of consistency and subjects average grade result. I correlated with the average grade result here, because averaging was used as part of the final mark in programming courses; I process quiz results individually later in this chapter.

Table 4.6 shows that almost all of the subjects with a "Good" grade (7 out of 8) and a majority of subjects with an "Average" grade (10 out of 16) were from the "C" population. Most of the subjects with a "Fail" grade (18 out of 28) were from the "I" population. Chi-square shows that the subgroups are significantly different on the basis of the observed result but having some cells with small values in this table caused at least 20% of expected frequencies to be less than 5.

The scores' density in the C/I/B subgroups could be seen more clearly (avoiding small numbers) when I merged "Good", "Average" and "Pass" columns in a single "Pass" column. Table 4.7 shows the results of T1 with the average of Quiz 1 and Quiz 2 expressed as pass/fail. Chi-square shows the difference between populations is significant in this table.

This table shows that 85% (23 out of 27) of the C subgroup passed the course while only 25% (6 in 24) of the I subgroup managed to pass. For the B subgroup with a small population (10) it is hard to make any comment but it is clear that the C subgroup performed much better than the I.

I also augmented table 4.5 with the average binary result in order to examine the tendency toward the correct model from T1 (week 0) to T2 (week 3) and how this tendency influenced performance. Table 4.8 shows two interesting effects:

Table 4.7: T1 population and average binary

T1	**Pass**	**Fail**	**Total**
C	23	4	27
I	6	18	24
B	4	6	10
Total	34	27	61
$\chi^2 = 19.491,\ df = 2,\ p < 0.0001$ **very highly significant**			

Table 4.8: Tendency toward correct model in T1 to T2 (week 0/ week 3) and the average binary result

	Correct		**Incorrect**		
	Pass	**Fail**	**Pass**	**Fail**	**Total**
CC	20	1	2	2	25
CI			1	1	2
IC	3	2	1	5	11
II			2	11	13
BC	3	0	1	1	5
BI			0	5	5
Total	26	3	7	25	61

- 90% (26 out of 29) of subjects who grasped assignment and sequence in the first three weeks, passed the course.
- Only 22% (7 out of 32) of subjects who did not grasp assignment and sequence in the first three weeks managed to pass the course.

When I examined T1 and T2 results as a programming success predictor, I found the following details:

- If we take C as a positive result, I and B as negative then T1 gives 30% false-negative (10 out of 34) and 15% false-positive (4 out of 27) shown in table 4.7.
- 22% false-negative (7 out of 32) and 10% false-positive (3 out of 29) were given by T2, shown in table 4.8.

The above figures revealed that T1 (week 0) cannot be considered as a strong predictor of programming success because of the high number of false-negatives

Table 4.9: T1 and the first quiz binary result

T1	**Pass**	**Fail**	**Total**
C	19	5	24
I	9	15	24
B	6	4	10
Total	34	24	58
$\chi^2 = 8.598,\ df = 2,\ p < 0.014$ **significant**			

but the T2 test (week 3) is a better predictor; 22% false-negative is a quite acceptable figure in an educational context.

Despit the temptation to point the research toward a third-week success predictor, I decided to stick with my original research question, focusing on the test which reveals what subjects bring with them to this learning environment.

4.4.5 Analysing methods

The results so far suggest that the C/I/B subgroups are different and that the C subgroup members had much better performance in exams, but this has to be confirmed by statistical analysis. I used a chi-square test in order to examine statistically whether the subgroups (C, I, B) performed differently in the quizzes.

Mental model tests and quiz results

As I mentioned in section 4.4.3 the first quiz was designed to give a chance to weaker students to pass the test but the second quiz was more engaged with programming.

Since association between T1 and the average binary result suggests that the C, I and the B subgroups are significantly different, I tried to examine whether these subgroups were better distinguishable in one rather than the other of these two quizzes. I decided on a threshold significance of $p = 0.05$.

First I examined the result of T1 with the first quiz binary result, shown in table 4.9. The table shows 79% (19 out of 24) of the C subgroup passed the first quiz and 44% (15 out of 34) of I/B subgroups also managed to pass. Chi-square shows that the differences between C/I/B subgroups is significant.

Then I examined the result of T2 with the first quiz binary result as shown in table 4.10. The table shows that 68% (26 out of 38) of the C subgroup passed the

Table 4.10: T2 and the first quiz binary result

T2	**Pass**	**Fail**	**Total**
C	26	12	38
I	8	12	20
Total	34	24	58
$\chi^2 = 4.363,\ df = 1,\ p < 0.037$ **significant**			

Table 4.11: T1 and second quiz binary result

T1	**Pass**	**Fail**	**Total**
C	22	4	26
I	8	15	23
B	4	6	10
Total	34	25	59
$\chi^2 = 13.944,\ df = 2,\ p < 0.0001$ **very highly significant**			

first quiz and 40% (8 out of 20) of the I subgroup managed to pass. Chi-square shows that the C/I difference is still just significant.

Next I examined the result of T1 with the second quiz binary result shown in table 4.11. The table shows that 85% (22 out of 26) of the C subgroup passed the second quiz and only 36% (12 out of 33) of I/B subgroups managed to pass. This time chi-square shows a strongly significant difference between the C and I subgroups ($p < 0.0001$).

Finally I examined the result of T2 with the second quiz binary result shown in table 4.12. The table demonstrates that 74% (29 out of 39) of the C subgroup passed the second quiz and 25% (5 out of 20) of the I/B subgroups managed to pass. Again chi-square shows a strong significant difference between the C and I subgroups ($p < 0.0001$).

4.5 Summary

Test T1 separated the candidates into 3 subgroups at the beginning of the course. One of the subgroups was apparently able to build a systematic strategy and apply it consistently in most of the questions; I called it the "consistent" subgroup. Another group might be able to build a strategy but failed to apply it consistently;

Table 4.12: T2 and second quiz binary result

T2	**Pass**	**Fail**	**Total**
C	29	10	39
I	5	15	20
Total	34	25	59
$\chi^2 = 13.190,\ df = 1,\ p < 0.0001$ **very highly significant**			

I called it the "inconsistent" subgroup. The third subgroup reserved its ideas and handed in the questionnaire incompleted; I called it the "blank" subgroup.

Test T2 revealed that 93% of the consistent population remained consistent and 78% of them corrected their model within 3 weeks, while 46% of the inconsistent population shifted to the consistent subgroup, but only 21% managed to correct their model by week 3. Half of the blank subgroup shifted toward the consistent subgroup, another half joined the inconsistent subgroup and only 30% managed to get the correct model by week 3.

The participants sat two internal examinations (first and second quiz) and the results were averaged and recorded as a number (percentage), a grade and a binary. The results revealed that 93% of subjects who grasped assignment and sequence in the first three weeks passed the course while only 22% of the others managed to pass the course.

Table 4.13 shows a summary of the association of consistency (T1 and T2) and the results of quiz 1 and quiz 2. The consistency captured by the T1 score had a strong positive association with success in the first and the second quiz binary result (79%/43% and 85%/37%). This association was also strong when the T2 score was examined with the first and the second quiz binary result (68%/40% and 74%/25%). Chi-square shows a highly significant result ($p < 0.0001$) when the association of T1/T2 with the second second quiz was examined and shows less significant results ($p < 0.014$, $p < 0.037$) when the first quiz was examined.

This result suggests that the second quiz, with a higher proportion of technical questions was more reliable in separating the C subgroup from the others in this experiment. The data clearly demonstrates that the consistent subgroup had a better chance of success in the second, more technical, quiz than the inconsistent subgroup.

Table 4.13 shows that the test has produced many fewer false positives (15%,

Table 4.13: Summarising the correlation between T1 and T2 and the first, second and average of quiz's binary results

		Success rate		χ^2	*df*	*p*	
		C	notC				
Quiz 1	T1	79%	43%	8.60	2	$p < 0.014$	significant
	T2	68%	40%	4.36	1	$p < 0.037$	significant
Quiz 2	T1	85%	37%	13.94	2	$p < 0.0001$	highly significant
	T2	74%	25%	13.19	1	$p < 0.0001$	highly significant
Average	T1	82%	40%	20.46	1	$p < 0.0001$	highly significant
	T2	71%	33%	7.36	1	$p < 0.007$	highly significant

T1 in the second quiz) than the university's admission system (40%), but it is producing far too many false negatives to be considered as an admission criterion (37% of the inconsistent and blank groups combined passed the examination). I believe that combining this test with a formal assessment with more technical questions might separate the C and I populations even more emphatically.

Chapter 5

Improving Methodology

The results in chapter 4 were encouraging, with an apparent potential to provide a clearer understanding of the patterns of reasoning that learners bring to programming courses. The first test (in week 0) revealed three distinguishable populations and the second test (in week 3) showed that consistency is persistent. A very small number shifted from the consistent subgroup while nearly half shifted from inconsistent to consistent. The correlation with the second quiz result demonstrated that the ability to adopt and maintain a consistent mechanism has a considerable association with success in the first level of programming courses.

Presenting this result (Dehnadi, 2006) to the research community attracted a number of collaborators as well as provoking a number of objections to my methodology. The objections were supplemented with a number of encouraging comments and constructive feedback which generated a series of improvements of the instrument and data analysis. The objections focused on the following issues:

1. The questionnaire did not ask and the analysis did not consider candidates' programming experience, age or sex.

2. Methods to interpret subjects' mental models were not clearly documented and may have been subjective.

3. The sample size was not big enough to support the claimed result.

4. Data analysis was weakly presented and consistency was measured as a binary (black/white) attribute.

In order to respond to these objections I made a number of improvements to the test materials and their analysis, considering various possible elements which may have played a part in the association observed in the initial test.

5.1 Learners' programming background

In the initial experiment the assumption was made that no subjects had prior knowledge in programming. I was advised that the data might be contaminated by subjects' programming background which might have caused the effect – see, for example, (Wilson and Shrock, 2001). Since programming background was not recorded, I decided to rectify the questionnaire and replicate the test to investigate the association with cleaner data. A number of questions were added asking candidates about their programming experience, any formal/informal prior programming courses, age and gender.

5.2 Mental models enhancement

Two more models (observed in candidates responses to the chapter 4 experiments) were added to the list of mental models and to the list of answers in the questionnaire. The conceptions behind the new mental models are as follows:

- Nothing is changed in `a` and `b`; they both keep their original values.
- An assignment is a simple mathematical equation, so all equal values of `a` and `b` are acceptable.

Figure 5.1 shows the first question when the answers, corresponding to the new mental models, were added. I use the question in this figure as an example to describe the new list of the mental models for single assignment in table 5.1.

Mental models for composition remained unchanged from table 4.2.

5.3 Interpretation enhancement

Another main issue raised by collaborators was the unclarity of the test instructions regarding the interpretation of candidates' mental models. In order to objectify the instrument and facilitate replication of the experiment by others, a number of improvements were applied to the test instruments.

Table 5.1: Anticipated mental models of single assignment `a=b` (question 1 figure 5.1 used as an example)

(M1)	Value moves from right to left `a:=b ; b:=0` Ans (`a = 20` , `b = 0`) 8th Answer
(M2)	Value copies from right to left `a:=b` Correct model of Java for assignment Ans (`a = 20` , `b = 20`) 4th Answer
(M3)	Value moves from left to right `b:=a ; a:=0` Ans (`a = 0` , `b = 10`) 3rd Answer
(M4)	Value copies from left to right `b:=a` the reversed version of Java model Ans (`a = 10` , `b = 10`) 1st Answer
(M5)	Right-hand value added to left `a:=a+b` Ans (`a = 30` , `b = 20`) 2nd Answer
(M6)	Right-hand value extracted and added to left `a:=a+b ; b:=0` Ans (`a = 30` , `b = 0`) 10th Answer
(M7)	Left-hand value added to right `b:=a+b` Ans (`a = 10` , `b = 30`) 9th Answer
(M8)	Left-hand value extracted and added to right `b:=a+b ; a:=0` Ans (`a = 0` , `b = 30`) 5th
(M9)	`a` and `b` keep their original values `a:=10 ; b:=20` Ans (`a = 10` , `b = 20`) 6th Answer
(M10)	Assignment is a simple equation, and then all equal values of `a` and `b` are acceptable. Ans (`a = 10` , `b = 10`) 1th Answer Ans (`a = 20` , `b = 20`) 4th Answer
(M11)	`a` and `b` swap their values simultaneously. `a:=b` || `b:=a` Ans (`a = 20` , `b = 10`) 7th Answer

1. Read the following statements and tick the box next to the correct answer in the next column. int $a = 10$; int $b = 20$; $a = b$;	The new values of a and b: ☐ $a = 10$ $b = 10$ ☐ $a = 30$ $b = 20$ ☐ $a = 0$ $b = 10$ ☐ $a = 20$ $b = 20$ ☐ $a = 0$ $b = 30$ ☐ $a = 10$ $b = 20$ ☐ $a = 20$ $b = 10$ ☐ $a = 20$ $b = 0$ ☐ $a = 10$ $b = 30$ ☐ $a = 30$ $b = 0$ Any other values for a and b: $a =$ $b =$ $a =$ $b =$ $a =$ $b =$	Use this column for your rough notes please

Figure 5.1: Question 1 with a single assignment

5.3.1 Answer sheet

An answer sheet was introduced to objectify and simplify the marking process. By looking at a candidate's answers we can find the relevant mental model/s for that particular question in the answer sheet without any prior knowledge of the mental models introduced in this study. For example when a candidate ticks the second box in the answer list in figure 5.1 the examiner, by looking at the answer sheet of question 1 (see figure 5.2), indicates M5 as the candidate's mental model in this question.

In multiple assignments (Q4 onwards) there is more complexity in assessing consistency because of the interaction between models of assignment and composition. I introduced a mark sheet in order to facilitate this process more objectively. More examples in the next section clarify the usability of these components.

Figure 5.4 shows the answer sheet for question 7 (figure 5.3). When a candidate ticked the eighth box in the question, according to the answer sheet the candidate's model for this question could be any one of (M1+S3), (M2+S3) or (M11+S3). This creates a level of ambiguity that has been resolved by introducing a marking protocol.

Question	**Answers/s**		**Model/s**
	$a = 20$	$b = 0$	M1
1.	$a = 20$	$b = 20$	M2
int $a = 10$;	$a = 0$	$b = 10$	M3
int $b = 20$;	$a = 10$	$b = 10$	M4
	$a = 30$	$b = 20$	M5
$a = b$;	$a = 30$	$b = 0$	M6
	$a = 10$	$b = 30$	M7
	$a = 0$	$b = 30$	M8
	$a = 10$	$b = 20$	M9
	$a = 20$	$b = 10$	M11
	$a = 20$	$b = 20$	M10
	$a = 10$	$b = 10$	

Figure 5.2: Answer sheet for question 1

7. Read the following statements and tick the box next to the correct answer in the next column. int $a = 5$; int $b = 3$; int $b = 7$; $a = c$; $b = a$; $c = b$;	The new values of a and b: ☐ $a = 3$ $b = 5$ $c = 5$ ☐ $a = 3$ $b = 3$ $c = 3$ ☐ $a = 12$ $b = 14$ $c = 22$ ☐ $a = 8$ $b = 15$ $c = 12$ ☐ $a = 7$ $b = 7$ $c = 7$ ☐ $a = 5$ $b = 3$ $c = 7$ ☐ $a = 5$ $b = 5$ $c = 5$ ☐ $a = 7$ $b = 5$ $c = 3$ ☐ $a = 3$ $b = 7$ $c = 5$ ☐ $a = 12$ $b = 8$ $c = 10$ ☐ $a = 10$ $b = 8$ $c = 12$ ☐ $a = 0$ $b = 0$ $c = 7$ ☐ $a = 0$ $b = 0$ $c = 15$ ☐ $a = 3$ $b = 12$ $c = 0$ ☐ $a = 3$ $b = 5$ $c = 7$ Any other values for a and b: $a =$ $b =$ $a =$ $b =$ $a =$ $b =$	Use this column for your rough notes please

Figure 5.3: Question 7 with multiple assignments

Question	**Answers/s**			**Model/s**
	$a = 0$	$b = 0$	$c = 7$	M1
	$a = 7$	$b = 5$	$c = 3$	(M1+S3)/(M2+S3)/(M11+S3)
7.	$a = 7$	$b = 7$	$c = 7$	M2
	$a = 3$	$b = 5$	$c = 0$	M3
int $a = 5$;	$a = 3$	$b = 5$	$c = 5$	M4
int $b = 3$;	$a = 3$	$b = 7$	$c = 5$	(M3+S3)/(M4+S3)
int $c = 7$;	$a = 12$	$b = 15$	$c = 22$	M5
	$a = 12$	$b = 8$	$c = 10$	(M5+S3)/(M6+S3)
$a = c$;	$a = 0$	$b = 0$	$c = 15$	M6
$b = a$;	$a = 8$	$b = 15$	$c = 12$	M7
$c = b$;	$a = 8$	$b = 10$	$c = 12$	(M7+S3)/(M8+S3)
	$a = 3$	$b = 12$	$c = 0$	M8
	$a = 5$	$b = 3$	$c = 7$	M9
	$a = 3$	$b = 5$	$c = 7$	M11
	$a = 5$	$b = 5$	$c = 5$	M10
	$a = 3$	$b = 3$	$c = 3$	
	$a = 7$	$b = 7$	$c = 7$	
	$a = 7$	$b = 3$	$c = 0$	(M1+S2)
	$a = 0$	$b = 5$	$c = 7$	
	$a = 5$	$b = 0$	$c = 3$	
	$a = 7$	$b = 3$	$c = 7$	(M2+S2)
	$a = 5$	$b = 5$	$c = 7$	
	$a = 5$	$b = 3$	$c = 3$	
	$a = 0$	$b = 3$	$c = 5$	(M3+S2)
	$a = 3$	$b = 0$	$c = 7$	
	$a = 5$	$b = 7$	$c = 0$	
	$a = 5$	$b = 3$	$c = 5$	(M4+S2)
	$a = 3$	$b = 3$	$c = 7$	
	$a = 5$	$b = 7$	$c = 7$	
	$a = 12$	$b = 3$	$c = 7$	(M5+S2)
	$a = 5$	$b = 8$	$c = 7$	
	$a = 5$	$b = 3$	$c = 10$	
	$a = 12$	$b = 3$	$c = 0$	(M6+S2)
	$a = 0$	$b = 8$	$c = 7$	
	$a = 5$	$b = 0$	$c = 10$	
	$a = 5$	$b = 3$	$c = 12$	(M7+S2)
	$a = 8$	$b = 3$	$c = 7$	
	$a = 5$	$b = 10$	$c = 7$	
	$a = 0$	$b = 3$	$c = 12$	(M8+S2)
	$a = 8$	$b = 0$	$c = 7$	
	$a = 5$	$b = 10$	$c = 0$	
	$a = 7$	$b = 3$	$c = 5$	(M11+S2)
	$a = 3$	$b = 5$	$c = 7$	
	$a = 5$	$b = 7$	$c = 3$	

Figure 5.4: Answer sheet for question 7

Participant code	Age	Sex	Time to do test	Prior programming	A-level's	Prior programming courses	Course result

Questions	Assignment								No effect	Equal sign	Swap values	Remarks (including participants' working notes)
	Assign-to-left		Assign-to-right		Add-Assign-to-left		Add-Assign-to-right		Values don't change (M9)	Assign means equal (M10)	Swap values (M11) /S1..3	
	Lose-value (M1) /S1..3	Keep-value (M2) /S1..3	Lose-value (M3) /S1..3	Keep-value (M4) /S1..3	Keep-value (M5) /S1..3	Lose-value (M6) /S1..3	Keep-value (M7) /S1..3	Lose-value (M8) /S1..3				
1												
2												
3												
4												
5												
6												
7												
8												
9												
10												
11												
12												
C0												
C1												
C2												
C3												

Figure 5.5: A marksheet

5.3.2 Mark sheet

A mark sheet was produced which allowed examination of the judgment of consistency; the means of dealing with ambiguous responses was codified; levels of consistency were defined; judgment of blankness was also clarified. A sample of the mark sheet is illustrated in figure 5.5.

Each column of the mark sheet represents a single model. The examiner ticks the mark sheet according to the ticked model/s in the answer sheet, notionally in pencil. For questions with a single assignment (Q1-Q3) the relevant model/s will be ticked and for questions with multiple assignments (Q4 onwards) instead of just ticking the corresponding model column on the mark sheet, "S1", "S2" or "S3" can be put next to the tick. The logical explanation of these symbols can be found in chapter 4 table 4.2.

A single tick in the first three questions (except M10 which requires two single ticks) maps to a single mental model. In later questions some of the single tick boxes give alternative models. When I made mental models explicit in the answer sheet, it exposed the problem of ambiguity in the S2 and S3 models.

Ambiguity

Suppose a subject consistently applies M1+S3: responses to Q1-Q3 will all be M1; responses to Q4-Q12 will be ambiguous, all containing M1+S3. On the other hand we could obtain the same response from a less consistent subject, starting with M1, but oscillating between different models (all with S3) in Q4-Q12.

If we assess ambiguous responses as inconsistent, all users of the S3 model would be judged inconsistent. If we assess them as consistent, we run the risk of mistakenly increasing the size of the consistent subgroup. I decided to take the second choice, despite the risk that it might weaken my conclusion. The following procedure illustrates how to resolve these ambiguities, indicating a candidate's mental model by using the answer sheet and the marksheet together:

1. Multiple answers in Q1-3 do not indicate a single model (except M10): put a tick in the leftmost (questions) column so that the candidates is not judged 'blank'.

2. Some answers in Q4-Q12 are ambiguous: e.g. the second answer in figure 5.4. We want to maximize judgment of consistency: put pencil ticks in each of the relevant columns. Then, when all the questions are marked, look for the column with the most ticks; and ink pencilled ticks in that column.

3. Finally, sum the inked ticks in each column in the C0 row.

Blankness and Consistency

A protocol defines the criteria for consistency and blankness:

1. A response with six inked ticks in the same column for Q1-Q6 (single and double assignment) is judged consistent.

2. Otherwise, a response with 8 or more inked ticks in the same column is judged consistent.

3. Otherwise, a response with fewer than 8 inked ticks in total (two-thirds of questions) is judged blank.

4. Otherwise, the response is judged inconsistent.

This protocol gives us the basic notions of consistency and blankness and maintains the objectivity of the process. Some deliberate flexibility was included in rules 1 and 2 to let candidates with some inconsistency be judged consistent. As with ambiguity the size of the consistent subgroup increases and consequently the effect of the protocol would be to dilute this subgroup and reduce the correlation; if we still see the correlation, we can be more confident that it is real.

Levels of consistency

In the first experiment (chapter 4) consistency was a factor which was measured as a binary (black/white) attribute. Each candidate was either consistent or not. I was criticised for this and was told that consistency is not black and white and should be measured within a wider spectrum. I decided to try to measure consistency at four different levels in order to examine:

- Correlation of different consistency levels with overall success.
- Whether there is a linear correlation between consistency levels and overall success.

Candidates who used only one model are clearly consistent; candidates who switch between two related models are also consistent, but less so.

For example M1 (left := right; right := 0) and M2 (left := right) are very similar, also M3 and M4, for similar reasons. That gives the first level of the related models (C1) which is illustrated in figure 5.6. I also grouped together M9, M10 and M11, the three non-assignment models. Then similar considerations group M1+M2 with M3+M4 at the next level (C2), and M5+M6 with M7+M8. At the final level (C3) M1 to M8, the assignment models, are grouped together.

In assessing consistency at each level we use the same protocol as before: a candidate's consistency level is the first row with an entry ≥ 8.

I could have joined models in any of three different dimensions: copy/move, left/right, add/overwrite. Because in the S3 model of multiple assignment the copy/move distinction goes away, I chose that as the weakest dimension. Then I decided to ignore direction, and finally addition/overwrite.

In practice joining models in this way was not very successful. It did not expand the C group very much: C0 is always large and C1-C3 were almost always small.

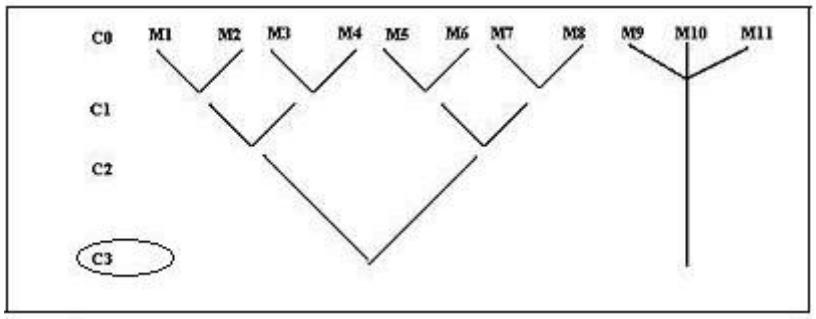


Figure 5.6: Structure diagram to show relationships between models

Questions	Assignment								No effect	Equal sign	Swap values	Remarks (including participants' working notes)
	Assign-to-left		Assign-to-right		Add-Assign-to-left		Add-Assign-to-right					
	Lose-value (M1) /S 1.3	Keep-value (M2) /S 1.3	Lose-value (M3) /S 1.3	Keep-value (M4) /S1.3	Keep-value (M5) /S 1.3	Lose-value (M6) /S1.3	Keep-value (M7) /S 1.3	Lose-value (M8) /S 1.3	Values don't change (M9)	Assign means equal (M10)	Swap values (M11) /S 1.3	
1		1										
2	1											
3		1										
4	1 + S3											
5	1 + S3											
6					1 + S3							
7		1 + S3										
8			1 + S3									
9	1 + S3											
10		1 + S3										
11			1 + S3									
12	1 + S3											
C0	5	4	2	0	1	0	0	0	0	0	0	
C1	9		2		1		0					
C2	11				1					0		
C3	12											

Figure 5.7: Algorithm used to interpret mental models in mark sheet

5.4 Counteracting selection bias

Participants in each experiment of this study were undergraduate students of an introduction to programming course in a University or an institution of higher education. In the initial experiment the assumption was made that no subjects had prior knowledge in programming. In order to allow for the possibility that the data might be influenced by the influence of prior programming experience, I decided to revise the questionnaire and replicate the test to investigate the association with cleaner data.

Seven questions were added to the questionnaire

1. Age
2. Gender
3. A-Level or any equivalent subjects
4. Have you ever written a computer program in any language?
5. If so, in what language(s)?
6. Will this be your first course in programming?
7. If not, what other programming courses have you studied?

The selection process consists of three different types of selection which are considered separately:

1. Experimental selection process
2. Self-selection process
3. Intake selection process

Some of the biases which may be caused by the experimental process, and the corresponding solutions, are:

1. Small groups may give insignificant results and might not represent the whole population of the course: each experiment uses a large sample group (50+).

Table 5.2: Bias caused by programming skill

Population	**Pass**	**Fail**	**Total**
Consistent programmers	20	0	20
Consistent non-programmers	24	16	40
Inconsistent	24	16	40
Total	68	32	100

2. A selected group of students in a class might not represent the whole population of the class: as far as possible, all students in the class participated in each experiment.

3. The population who withdrew from the course might perform differently in the test from those who stay to the end: I shall check the significant differences between these populations in their test performance.

4. The intake policy in different institutions might produce different results: I shall conduct experiments in a wide range of institutions.

The most important source of bias is that some subjects already know how to program. Consider an extreme case: suppose 100 subjects participate in an experiment, of whom 20 are expert programmers and 80 are novices. Suppose that consistency in the test has no effect in the novice subgroup. Suppose also that a novice is equally likely to be judged consistent as inconsistent in the test, and that each novice has a 60% chance of passing the exam. The expert programmers all score consistently in the test, and all pass the examination (row 1 of table 5.2). Of the novices, 40 are judged consistent, and 40 inconsistent; in each group 24 pass and 16 fail (rows 2 and 3 of table 5.2). Overall, it appears that 60 subjects are consistent (rows 1 and 2 of the table 5.2), and 40 inconsistent. But in the "consistent" group, 20 will pass because they are expert (row 1 of table 5.2), and 24 (60% of 40) novices will also pass (row 2 of table 5.2). In the inconsistent group, 24 (60% of 40) novices will pass. Overall, the "consistent" group has a pass rate of 73% (20+24 out of 60, rows 1 and 2 of table 5.2), while the inconsistent group has a pass rate of 60% (24 out of 40, row 3 of table 5.2). The apparent effect of consistency is the result of bias caused by prior programming expertise.

I use two approaches to investigate this bias: the first approach is spotting those who use the correct model of assignment and sequence (Java model) in the

test; the second approach is looking at candidates' responses to prior programming knowledge questions in the questionnaire.

Following the first approach, I can use the marksheet which allows me to pick out those who use the Java/C/imperative language models (M2 and S1). They may have learnt how to program prior to the course start; I call them the *CM2* subgroup. I shall slice the population into two groups of CM2 and notCM2 and examine if the effect of consistency persists in the notCM2 subgroup.

Following the second approach, I shall examine if the effect of consistency is related to prior programming experience by looking at the candidates' responses to the questionnaire. As I mentioned earlier in this section, four questions are related to candidates' prior programming knowledge in the questionnaire. By answering these questions, candidates report if they have written a program or have attempted a programming course. They also reveal which language they have had experienced if they have any programming experience. Using these details I shall examine the effect of consistency in three different group arrangements:

1. I shall divide the population into programmers and non-programmers, and examine the effect of consistency separately in each group.

2. Because the test is based on assignment and sequence in Java, I shall divide the population into programmers who have used a Java/C/imperative language and the rest, and examine the effect of consistency separately in each group.

3. I shall divide the population into those who had previously attempted a programming course and those who are in their first course, and examine the effect of consistency in each group.

Some of the subpopulations produced by these divisions will be small, and the results may therefore not be statistically reliable. It will be necessary, therefore, to combine the results by using a meta-analysis technique to get a reliable result overall, in each of the subdivisions.

5.5 Data Analysis

"Making a strong claim on a basis of a small experiment which was weakly analysed" was an objection that was made when the initial result was presented

Table 5.3: MM1 population and average percentage

MM1	**Mean**	**N**	**Std. Deviation**	**Variance**
C	66.74	27	19.320	373.276
I	43.25	24	16.933	286.717
B	49.30	10	15.699	246.456
Total	54.64	61	20.765	431.168

Table 5.4: Test of Homogeneity of Variance, MM1 population and average percentage

Levene Statistic	$df1$	$df2$	p
.578	2	58	.564

(Dehnadi, 2006). It was a fair objection because I presented the result with only a few tables and figures, analysed with the chi-square test.

In order to improve the data analysing process the T1, T2, first quiz and second quiz test results were added to a SPSS file and data were examined with *parametric* and *nonparametric* statistical tools.

First I compared Means and Variances in the C/I/B categories. Table 5.3 shows the subgroups' Mean, Variance and Standard Deviations with the average percentage.

Differences between subgroups' variances were small. 23% between C and I; 34% between C and B; 14% between I and B. I decided to use a test of Homogeneity of Variance to examine if the subgroups' variances are approximately equal across the sample. The result of the homogeneity test in table 5.4 reveals that the subgroups' variances are approximately equal and the null hypothesis cannot be rejected ($p < 0.564$).

Geng et al. (1982) show that parametric statistical techniques such as ANOVA are robust under minor departure from homogeneity and normal distribution assumptions, such as in this case.

ANOVA was therefore used to examine the differences between subgroups, which revealed significant differences between subgroups ($F = 11.514$, $p < 0.0001$) shown in table 5.5.

Welch's Robust Tests of Equality of Means, which is a part of ANOVA and does not require the data to be normally distributed, confirms the significance

Table 5.5: ANOVA, MM1 population and average percentage

	Sum of Squares	*df*	**Mean Square**	*F*	*p*
Between Groups	7352.280	2	3676.140	11.514	.0001
Within Groups	18517.785	58	319.272		
Totals	25870.066	60			

Table 5.6: Welch's Robust Tests of Equality of Means, MM1 population and average percentage

	Statistic	$df1$	$df2$	p
Welch	10.813	2	26.750	0.0001

of the difference between subgroups. Table 5.6 shows that the null hypothesis of Welch's Robust Tests of Equality of Means significantly rejected the assumption that the C/I/B subgroups were from the same population.

I also used the chi-square test, as an alternative nonparametric statistics tool, which is used for analysing categorical data. Chi-square is a test for detecting changes in responses due to experimental intervention and I used it in order to examine statistically whether the subgroups (C, I, B) performed differently in the quizzes. The chi-square test can be used in two similar but distinct circumstances:

- Chi-square goodness-of-fit test is used for estimating how closely an observed distribution matches an expected distribution.
- The other primary use of the chi-square test is to examine whether two variables are *independent* regardless of what the distribution should look like.

As this was the the first experiment with C/I/B categories, the expected frequencies for these categories were unknown and statistical results for chi-square test analysis were based on the observed rather than expected results. The use of the chi-square "independence" test seems more relevant to examine if C/I/B subgroups were "not correlated with" or were "independent of" each other. If any of these subgroups were correlated, their frequencies tend to move together, either in the same direction or in the opposite. The null hypothesis was that the exam performance is the same for all the subgroup members.

Although SPSS documentation confirms that the "independence" test of chi-square is used, I decided to use table 4.7 and calculate the "independence" test manually and compare the result with the figure obtained by SPSS, to build confidence in using it. The significance value obtained from the manual calculation was $p < 0.0001$, the same as SPSS. Comparing this result with the SPSS result revealed that SPSS uses a very similar method to calculate chi-square. The manual calculation double checked the SPSS chi-square test result and gave me more confidence to use SPSS's chi-square test, for the rest of this experiment.

The result of the SPSS chi-square test confirmed the result of Welch's Robust Tests of Equality of Means, which significantly rejected the null hypothesis that C/I/B subgroups are the same.

For my nominal data, chi-squared methods have been used to explore the relationship between variables. No assumptions about causality have been made. Even where the word "effect" is used, it should be taken to indicate a statistically significant result and not causality.
As is often the case in research conducted in real-life settings, the sample sizes and expected values are not always optimal. However, significant use has been made of advanced meta-analysis methods, where the results of analyses from smaller samples are combined across larger samples to reflect different sample sizes and sample diversity. Where some of the expected values are below five, we can simply note that this result is consistent with the broad pattern of results and with the meta-analysis, so undue reliance is not placed on them.

5.6 Summary

Exposing the initial experiment's result to the research community raised a number of objections about programming background, testing materials and the objectifying of mental models. I augmented the questionnaire to record subjects' programming background as well as age and sex. The list of mental models was enhanced and a number of tools and protocols were introduced to facilitate detection and interpretation of mental models. Administering the test in a variety of institutions was intended to see if there was more than just a local effect.

In order to find appropriate tools both *parametric* and *nonparametric* statistical tools were used to examine the data. Parametric tools appeared to be as applicable as nonparametric alternatives; both confirmed that the C/I/B sub-

groups were significantly different.

Chapter 6

Further Experiments

The results of the initial experiment suggested that the mechanisms of rationalisation which students bring to the study of programming have a correlation with their learning performance. In order to demonstrate the reality of the effect, it had to be verified in large scale experiments. Nine experiments were undertaken: one in Australia (Newcastle University), six in UK (Middlesex University, University of Sheffield, The University of York, University of Westminster, Banff and Buchan College, Royal School of Signals), one in Denmark (Aarhus University) and one in Germany (OSZ TIEM the Department of Informatik in Berlin). Final examination results have not yet been received from the experiments in Banff and Buchan college and in OSZ TIEM.

6.1 University of Newcastle - 2006

Data provided by: Simon[1], School of DCIT (Design, Communication, and Information Technology), Newcastle, Australia.

The experiment was carried out at the beginning of the academic year 2006/2007. The test was administered once, before the course began (week 0). From 90 participants, 17 withdrew before the examination, and two subjects did not attend the end of course examination, leaving 71 active subjects.

[1]Simon has only a single-word name. He contributed the seven additional questions set out on page 89.

Table 6.1: Consistency levels among population

Consistency levels	**Active**	**Withdrawn**
C0	44	11
C1	3	1
C2	1	0
C3	4	1
I	18	6
B	1	0
Total	71	19

Table 6.2: Consistency and active/withdrawn divisions)

Consistency	**Active**	**Withdrawn**
C	52	13
I/B	19	6
Total	71	19
$\chi^2 = 0.173,\ df = 1,\ p < 0.677$ **not significant**		

6.1.1 Assessment methods

The assessment method in the course consisted of two practical tests (each with 10% credit), two assignments (each with 15% credit) and a final examination (with 50% credit). The course mark was the total of these five.

6.1.2 Test result

Table 6.1 shows the distribution of consistency levels in the subject population. 44 subjects (62% of the active population) were assessed as C0 (eight questions or more with a single model), three C1 (eight questions or more with two related models), one C2 (four related models), four C3 (more than four models M1-M8), 18 Inconsistent and one Blank. In this table, a chi-square shows that the withdrawn population is not significantly different from the active population. This table has 8 cells with fewer than five expected entries which weakens the chi-square test result. In order to avoid small numbers I combined consistent subgroups (C0, C1, C2, C3) as C and, the other subgroups (I, B) as I/B. Table 6.2 shows that there is no significant difference between the withdrawn and active populations.

Table 6.3: Consistency (C/notC) and grade course result

Result	**C**	**notC**	**Total**
HD	3	0	3
D	9	2	11
C	10	1	11
P	19	3	22
F	11	13	24
total	52	19	71
$\chi^2 = 14.392,\ df = 4,\ p < 0.006$ **highly significant**			

Measuring the level of consistency in the marksheet (see section 5.3.2) was not very successful in this case. It did not expand the C group very much: C0 is large and C1-C3 are small. Since cells with small expected numbers would not be suitable for data analysis purposes, I decided to combine C1-C3 with other subgroups in two different ways and analyse each of these combinations separately:

1. C1-C3 join C0 to build a single consistent group and I joins B to build a single inconsistent group (C/notC).

2. C0 remains as a single consistent group and C1-C3 joins I and B to build a single group (C0/notC0).

Course results were recorded as a percentage mark (0-100), as a grade P(pass) / C(credit) / D(distinction) / HD(high distinction) / F(fail) and as a binary (Pass / Fail).

I first examined the association of the C/notC populations with grade. Table 6.3 shows a highly significant difference between the C and the notC subgroups. But in this table half of the cells have fewer than 5 subjects. I therefore decided to investigate the strength of the correlation with the binary (pass/fail) result.

Combining columns P, C, D and HD into 'Pass' and FF as 'Fail', gives table 6.4. A chi-square test shows a highly significant association between consistency and binary course mark. The pass rate in the C0-C3 subgroups is 79% (41 out of 52) and in I/B is 32% (6 out of 19).

Second, I examined the association of the C0/notC0 populations with the binary result. Table 6.5 shows a highly significant difference between C0 and the

Table 6.4: Consistency (C/notC) and binary course result

Result	**C**	**notC**	**Total**
Pass	41 (79%)	6 (32%)	47
Fail	11	13	24
Total	52	19	71
$\chi^2 = 13.894,\ df = 1,\ p < 0.001$ **highly significant**			

Table 6.5: Consistency (C0/notC0) and binary course result

Result	**C0**	**notC0**	**Total**
Pass	35 (80%)	12 (44%)	47
Fail	9	15	24
Total	44	27	71
$\chi^2 = 9.213,\ df = 1,\ p < 0.002$ **highly significant**			

other subgroups. The pass rate in the C0 subgroup is 80% (35 out of 44) and 44% (12 out of 27) in notC0.

The C1-C3 subgroups are small in this experiment, and tables 6.4 and 6.5 show that adding them to either the C0 or the I/B subgroups does not much affect the result – 79%/32% in C/notC and 80%/44% in C0/notC0 which is only slightly weaker. Chi-square shows a highly significant difference between subgroups in either case – $p < 0.001$, $p < 0.002$.

6.1.3 Prior programming knowledge

There were four questions about programming background in the questionnaire (see section 5.4). In this experiment 21 out of 46 who reported prior programming experience used the correct model of Java assignment (M2) and the correct model of composition (S1) before the course began. I call them the CM2 subgroup. This subgroup seems likely to contain those who have had some effective prior programming experience with a Java-like language, although it is interesting to note that two of them said they had no previous programming experience. Observing this diversity within the population that reported prior programming experience, I decided to investigate the effect of that experience on candidates' success in two different ways:

Table 6.6: Prior programming experience and binary course results

Result	**Yes**	**No**	**Not answered**	**Total**
Pass	29	13	5	47
Fail	17	6	1	24
Total	46	19	6	71
$\chi^2 = 1.304,\ df = 2,\ p < 0.596$ **not significant**				

Table 6.7: Prior programming experience and binary course results, without ambiguous responders

Result	**Yes**	**No**	**Total**
Pass	29	13	42
Fail	17	6	23
Total	46	19	65
$\chi^2 = 0.170,\ df = 1,\ p < 0.680$ **not significant**			

1. Considering prior programming experience as reported in the questionnaire.

2. Considering the CM2 population as a separate subgroup.

Prior programming experience reported in the questionnaire

Table 6.6 shows that 46 candidates (70%) reported programming experience, 19 reported none and 6 did not give a reply. Almost the same success rate in the population with prior programming experience and the population without (63%/68%), shows that there is no association between prior programming experience and candidates' success. The chi-square test also shows that there is no significant difference between subgroups with or without programming experience. Table 6.7 shows almost the same result when the "not answered" population is removed.

Relevant programming experience

Programming experience was categorised as relevant or irrelevant upon the similarity of assignment and sequence in subjects' prior programming experience to assignment and sequence in Java. For example, experience with languages such

Table 6.8: Prior relevant programming experience and binary course results

Result	**Yes**	**No**	**Not answered**	**Total**
Pass	20	22	5	47
Fail	13	10	1	24
Total	33	32	6	71
$\chi^2 = 1.342,\ df = 2,\ p < 0.511$ **not significant**				

Table 6.9: Prior relevant programming experience and binary course results, without ambiguous responders

Result	**Yes**	**No**	**Total**
Pass	20	22	42
Fail	13	10	23
Total	33	32	65
$\chi^2 = 0.471,\ df = 1,\ p < 0.492$ **not significant**			

as Pascal, C, C++ and Java was considered relevant but experience of HTML, Visual Basic or PHP was considered irrelevant.

Table 6.8 shows whether having relevant programming experience helped subjects to pass the course more often than those who had no such experience. Almost the same success rate in the population with prior relevant programming experience and population without (61%/69%), shows there is no association between relevant programming experience and candidates' success. The chi-square test also shows that there is no significant difference between subgroups with or without prior Java-like programming experience. Table 6.9 shows almost the same result, when the "not answered" population is removed.

Prior programming course

Table 6.10 shows that 30 candidates (49%) reported they had taken a prior programming course, 32 reported they had not and 9 did not reply. Almost the same success rate in population with/without prior programming course (63%/69%) shows that there is no association between prior programming course and candidates' success. The chi-square test also shows no significant difference between subgroups with or without a prior programming course. Table 6.11 shows that the

Table 6.10: Prior programming course and binary course results

Result	**Yes**	**No**	**Not answered**	**Total**
Pass	19	22	6	47
Fail	11	10	3	24
Total	30	32	9	71
$\chi^2 = 0.204,\ df = 2,\ p < 0.903$ **not significant**				

Table 6.11: Prior programming course and binary course results, without ambiguous responders

Result	**Yes**	**No**	**Total**
Pass	19	22	41
Fail	11	10	21
Total	30	32	62
$\chi^2 = 0.203,\ df = 1,\ p < 0.652$ **not significant**			

correlation is still not significant when the "not answered" population is removed.

Summarising the effect of reported prior programming experience

Candidates' prior programming attributes such as programming experience or attendance in a prior programming course appear to have no effect on the binary mark. Chi-square also shows no significant difference between subgroups with/without prior programming background. Table 6.12 shows the significance of prior programming attributes and numeric course mark examined by ANOVA. The results given by ANOVA are similar to the result given by the chi-square test.

CM2 population as a subgroup

In the C0 subgroup 52% of subjects (23 out of 44) used the correct models of assignment (M2) and sequential composition (S1) before the course began (CM2 subgroup). The success rate of this subgroup was 87%. Table 6.13 separates the CM2, C0, C1-3 and (I/B) subgroups. The figure shows that when CM2 is a separate subgroup, the differences between subgroups is strongly significant. But note that the remaining C0 subgroup still has 71% (15 out of 21) success

Table 6.12: (ANOVA) Prior programming attributes and numerical course results

		Sum of Squares	*df*	**Mean Squares**	*F*	*p*
Programming experience	Between groups	6.946	39	.178	.685	.859
	Within groups	6.500	25	.260		
	Total	13.446	64			
Relevant experience	Between groups	27.599	40	.690	.702	.854
	Within groups	29.500	30	.983		
	Total	57.099	70			
Prior course	Between groups	8.817	37	.238	.858	.670
	Within groups	6.667	24	.278		
	Total	15.484	61			

Table 6.13: Consistency and binary course results – CM2 separated

Result	**CM2**	**C0**	**C1-3**	**I/B**	**Total**
Pass	20 (87%)	15 (71%)	6 (75%)	6 (32%)	47
Fail	3	6	2	13	24
Total	23	21	8	19	71
$\chi^2 = 15.139,\ df = 3,\ p < 0.002$ **highly significant**					

compared to I/B's 32% (6 out of 19).

Effect of consistency on success in non-CM2 population

In table 6.14 the CM2 subgroup is excluded. A chi-square test shows that the result is significant, but there is at least one cell with less than 5 expected subjects.

Table 6.15 shows the C/notC populations when the CM2 subgroup is excluded. A chi-square test shows that the difference between the subgroups is highly significant. The pass rate of 72% in C and 32% in notC shows a strong

Table 6.14: Consistency and binary course results – CM2 excluded

Result	**C0**	**C1-3**	**I/B**	**Total**
Pass	15	6	6	27
Fail	6	2	13	21
Total	21	8	19	48
$\chi^2 = 7.808,\ df = 2,\ p < 0.02$ **significant**				

Table 6.15: Consistency (C/notC) and binary course results – CM2 excluded

Result	**C**	**notC**	**Total**
Pass	21 (72%)	6 (32)	27
Fail	8	13	21
Total	29	19	48
$\chi^2 = 7.778,\ df = 1,\ p < 0.005$ **highly significant**			

Table 6.16: Consistency (C0/notC0) and binary course results – CM2 excluded

Result	**C0**	**notC0**	**Total**
Pass	15 (71%)	12 (44%)	27
Fail	6	15	21
Total	21	27	48
$\chi^2 = 3.495,\ df = 1,\ p < 0.062$ **not significant**			

correlation beween consistency and candidates's success.

Table 6.16 shows the C0/notC0 population when the CM2 subgroup is excluded. Although chi-square shows a weaker difference between subgroups, the numbers are still in the right direction: a success rate of 71% (15 out of 21) in C0 and 44% (12 out of 27) in notC0 shows the correlation is still strong.

Separating candidates by programming background factors recorded in the questionnaire failed to indicate any significant difference. Most of the CM2 subgroup (21 out of 23) had prior programming experience, and from that experience they had learned about assignment and sequence. Their 87% success rate in the course can hardly be a surprise. When the result of the initial experiment (reported in section 4.4) was presented to the research community, one of the main objections was that subjects in the consistent subgroup might be simply those who have learned to program before. In this experiment, when this subgroup is removed, despite the chi-square result which shows less significant difference between subgroups, the association of consistency was still strong (71%/44%). Therefore consistency is not simply the effect of learning to program.

Table 6.17: Consistency and binary course results – prior programming experience = yes

Result	**C0**	**notC0**	**Total**
Pass	26	3	29
Fail	7	10	17
Total	33	13	46
$\chi^2 = 12.424,\ df = 1,\ p < 0.001$ **highly significant**			

Effect of consistency on success in sliced populations (prior programming background)

From the evidence shown in tables 6.15 and 6.16 it seems that consistency is not the same as programming skill. In order to look further into the question of whether consistency is an effect of programming background I decided to separate candidates into different slices based on prior programming experience or prior programming course and examine the correlation of consistency with success in each slice separately. By this examination I hoped to see whether the effect of consistency persists, unrelated to programming background.

As tables 6.4 and 6.5 show, adding the C1-C3 subgroups to either the C0 or the I/B subgroups does not much affect the association. I decided to combine them with the I/B subgroup in the rest of my analysis of this experiment.

As a result of slicing the participants into small sub-populations according to their programming background, most tables have cells with very small expected numbers, which affects the reliability of a chi-square test result. In order to overcome this problem I use a meta-analysis procedure in chapter 7 to produce a more reliable result by combining several experiments. We shall see that meta-analysis shows that the effect of consistency on success is strongly significant in every sub-population, even though individual experiments give less definite results.

Table 6.17 shows the subjects who claimed programming experience: 46 subjects with an overall pass rate of 63%. The pass rate in the C0 subgroup is 79% (26 out of 33) which drops to 23% (3 out of 13) in the notC0 subgroup. It shows a strong effect of consistency on candidates' success in this slice and the chi-square test result also reports a highly significant difference between the subgroups.

Table 6.18: Consistency and binary course results – prior programming experience = no

Result	**C0**	**notC0**	**Total**
Pass	11	2	13
Fail	2	4	6
Total	13	6	19
$\chi^2 = 4.997,\ df = 1,\ p < 0.025$ **significant**			

Table 6.19: Consistency and binary course results – prior relevant programming experience = yes

Result	**C0**	**notC0**	**Total**
Pass	19	1	20
Fail	5	8	13
Total	24	9	33
$\chi^2 = 12.698,\ df = 1,\ p < 0.001$ **highly significant**			

Table 6.18 shows the candidates who claimed no programming experience: 19 subjects with an overall pass rate of 68%. The pass rate in the C0 subgroup is 85% (11 out of 13) which drops to 33% (2 out of 6) in notC0 subgroup. It shows a strong effect of consistency on success in this slice and a chi-square test shows a weaker but still significant difference between subgroups.

Table 6.19 shows the subjects who claimed relevant programming experience: 33 subjects with a pass rate of 60%. The pass rate in the C0 subgroup is 79% (19 out of 24) which drops to 11% (1 out of 9) in the notC0 subgroup. It shows the effect of consistency on candidates' success is also strong in this slice and a chi-square test also shows a strongly significant difference between subgroups.

Table 6.20 shows the candidates who claimed no relevant programming experience: 32 subjects with a pass rate of 69%. The pass rate in the C0 subgroup is 80% (12 out of 15) which drops to 59% (10 out of 17) in the others. It shows the effect of consistency on candidates' success is weak in this slice and a chi-square test shows no significant difference between subgroups.

Table 6.21 shows candidates who claimed a prior programming course: 30 subjects with an overall pass rate of 63%. The pass rate in the C0 subgroup is 73% (16 out of 22) which drops to 33% (3 out of 9) in notC0. It shows the effect

Table 6.20: Consistency and binary course results – prior relevant programming experience = no

Result	**C0**	**notC0**	**Total**
Pass	12	10	22
Fail	3	7	10
Total	15	17	32
$\chi^2 = 2.586,\ df = 1,\ p < 0.108$ **not significant**			

Table 6.21: Consistency and binary course results – prior programming course = yes

Result	**C0**	**Others**	**Total**
Pass	16	3	19
Fail	5	6	11
Total	21	9	30
$\chi^2 = 4.178,\ df = 1,\ p < 0.041$ **significant**			

of consistency on candidates' success is strong in this slice and a chi-square test shows a weaker but still significant difference between subgroups.

Table 6.22 shows the candidates who claimed not to have attended a prior programming course: 32 candidates with an overall pass rate of 69%. The pass rate in the C0 subgroup is 93% (14 out of 15) which drops to 47% (8 out of 17) in notC0. It shows the effect of consistency on candidates' success is strong in this slice and the chi-square test result shows a highly significant difference between subgroups.

Table 6.22: Consistency and binary course results – prior programming course = no

Result	**C0**	**notC0**	**Total**
Pass	14	8	22
Fail	1	9	10
Total	15	17	32
$\chi^2 = 7.942,\ df = 1,\ p < 0.005$ **highly significant**			

Table 6.23: Summarising consistency and binary course mark – groups

Groups	**Pass rate**		χ^2	*df*	*p*	**Size**
	Consistent	**Inconsistent**				
CM2/notCM2	87% (21 of 23)	56% (27 of 48)	6.552	1	0.010	71
C0/notC0	80% (35 of 44)	44% (12 of 27)	9.213	1	0.002	71
C/notC	79% (41 of 52)	32% (6 of 19)	13.894	1	0.001	71

6.1.4 Summarising the Newcastle experiment

The result of this experiment supported the result of the initial experiment (section 4.4). The effect of consistency on success was examined in three different group arrangements and with seven different filtering arrangements which might have had an effect on the result. Table 6.23 shows a summary of the group arrangements. Observing a relatively strong effect in the CM2/notCM2 subgroups (87%/56%) is not surprising (discussed in 6.1.3) but the size of the effect in C0/notC0 (80%/44%) and C/notC (79%/32%) group arrangements is even larger. The chi-square test shows the test result significantly separated the subgroups, in favour of the consistent subgroup in each case.

Table 6.24 shows a summary of the effect of consistency in seven subpopulations: candidates outside the CM2 subgroup; candidates with prior programming experience; candidates without prior programming experience; candidates with relevant experience; candidates without relevant experience; candidates with a prior programming course; candidates without a prior programming course. A strong effect persisted in each filtered sub-population and was significant in every case, except for the subgroup without relevant programming experience. The weaker effect of consistency in candidates with no relevant programming experience in this experiment suggests that consistency might be an effect of relevant programming experience. I pursue this issue below in other experiments and examine the overall effect by meta-analysis.

Applicants with prior programming experience may often be preferred to those without, both in university admissions and in employment. But table 6.25 suggests that there is no significant association between programming background and candidates' success. The pass rate of the candidates with/without prior programming experience are almost the same, and attendance at a prior programming course has no effect on the pass rate either. Note also that in *every*

Table 6.24: Summarising consistency and binary course mark – filters

Slices	Pass rate		χ^2	*df*	*p*	Size
	C0	notC0				
CM2 excluded	65% (15 of 23)	44% (12 of 27)	4.428	1	0.062	48
Prior experience	79% (26 of 33)	23% (3 of 13)	12.424	1	0.001	46
No prior experience	85% (11 of 3)	33% (2 of 6)	4.997	1	0.025	19
Relevant experience	79% (19 of 24)	11% (1 of 9)	12.698	1	0.001	33
No relevant experience	80% (12 of 15)	59% (10 of 17)	1.663	1	0.197	32
Prior course	76% (16 of 21)	33% (3 of 9)	4.178	1	0.041	30
No prior course	93% (14 of 15)	47% (8 of 17)	7.942	1	0.005	32

Table 6.25: Summarising prior programming factors and course binary mark

Slices	Pass rate		χ^2	*df*	*p*	Size
	yes	no				
Prior experience	63%(29 of 46)	68%(13 of 19)	0.170	1	0.680	65
Relevant experience	61%(20 of 33)	69%(22 of 32)	0.471	1	0.492	65
Prior course	63%(19 of 30)	69%(22 of 32)	0.203	1	0.652	62

case those without prior programming did better than those with – though the effect is not significant.

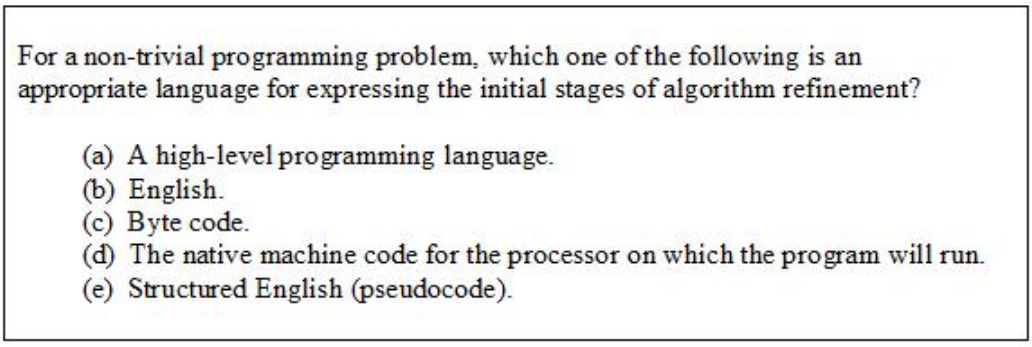
6.2 Middlesex University - 2006

Experiment conducted by the author in the School of Computing Science, Middlesex University, UK.

The experiment was carried out in the academic year 2006/2007. The test was administered once, before the course began (week 0). 118 subjects participated, 26 subjects withdrew before week 7 and 20 more withdrew by the end of the course.

6.2.1 Assessment method

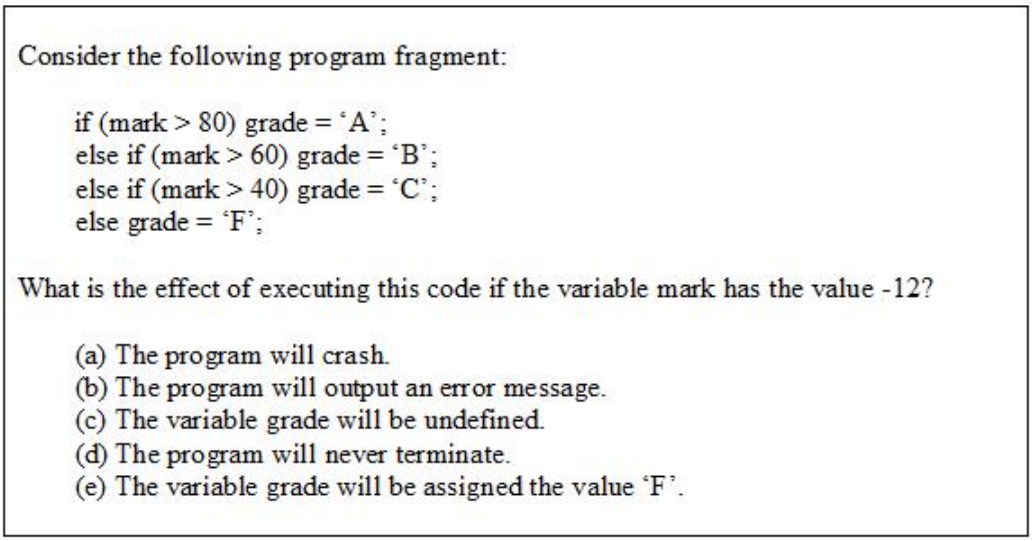
Students had to undertake two quizzes in weeks 7 and 11. The week 7 quiz contained sixteen multiple-choice questions ranging from trivial bookwork (e.g. figure 6.1) to technical analysis (e.g. figure 6.2), and two creative questions (e.g. figure 6.3). The week 11 quiz had write-in bookwork questions (e.g. figure 6.4), technical write-in questions (e.g. figure 6.5) and longer creative questions (e.g. figure 6.6).



For a non-trivial programming problem, which one of the following is an appropriate language for expressing the initial stages of algorithm refinement?

(a) A high-level programming language.
(b) English.
(c) Byte code.
(d) The native machine code for the processor on which the program will run.
(e) Structured English (pseudocode).

Figure 6.1: A bookwork multiple-choice question from the week 7 in-course exam (1 mark)



Consider the following program fragment:

```
if (mark > 80) grade = 'A';
else if (mark > 60) grade = 'B';
else if (mark > 40) grade = 'C';
else grade = 'F';
```

What is the effect of executing this code if the variable mark has the value -12?

(a) The program will crash.
(b) The program will output an error message.
(c) The variable grade will be undefined.
(d) The program will never terminate.
(e) The variable grade will be assigned the value 'F'.

Figure 6.2: A bookwork multiple-choice question from the week 7 in-course exam (1 mark)

The Hendon Used Car dealership price list shows the price of used cars with a mileage of exactly 10000 miles, and adjusts the prices of its cars depending on their individual mileage, as shown in this table:

Price adjuster for mileage:	
Under 10,000 miles	add 200
Over 10,000 miles	subtract 300

Assuming that the price and mileage of a given car are stored in integer variables price and miles, write if..else statement(s) that adjust the value of price according to the table above.

Figure 6.3: A technical creative question from the week 7 in-course exam (5 marks)

The ..... -controlled loop is used to do a task a known number of times.

Figure 6.4: A bookwork write-in question from the week 11 in-course exam (1 mark)

What is wrong with the following main method which is intended to add the numbers from 1 to 4 together and print out the result which should be 10? Give a correct version:

```
public static void main (String[] args) {
    int total = 0;
    for (int i = 1; i < 4; i++)
        total = total+I;
    System.out.println("Total: " + total);
}
```

Figure 6.5: A technical write-in question from the week 11 in-course exam (1 mark)

The following code is an incomplete version of a 'guess the number' program,
in which the user is repeatedly invited to guess a hidden number.
Add the necessary code to the body of the while loop to check the user's guess and
output appropriate messages depending on whether or not the guess is correct.

```
final int HIDDENNUM = 20;
String numStr =
        JOptionPane.showInputDialog("Enter your guess (zero to finish)")
int guess = Integer.parseInt(numStr)}
while (guess != 0) {

//ADD YOUR CODE HERE



String numStr =
        JOptionPane.showInputDialog("Enter your guess (zero to finish)");
int guess = Integer.parseInt(numStr);
}
```

Figure 6.6: A technical creative question from the week 11 in-course exam (5 marks)

Table 6.26: Consistency levels in the first quiz

	Active	**Withdrew**
C0	35	6
C1	3	0
C2	8	2
C3	10	8
I	24	4
B	12	6
Total	92	26

Table 6.27: Consistency and active/withdrew in the first quiz

	Active	**Withdrew**
C	56	16
I/B	36	10
Total	92	26
$\chi^2 = 0.004,\ df = 1,\ p < 0.95$ **not significant**		

Since the difficulty of the questions and the number of subjects who withdrew were considerably different in the first and the second quiz, I decided to analyse each quiz result separately.

6.2.2 Result of the first quiz

From 118 participants, 26 were absent in the first quiz, leaving 92 active subjects. 35 subjects were assessed as C0, three as C1, eight as C2, ten as C3, 24 as I and 12 as B. Table 6.26 shows the population of candidates in consistency subgroups.

In order to examine if the withdrawn population is significantly different from the active population, I combined the consistent subgroups (C0, C1, C2, C3) as C and, the other subgroups (I, B) as I/B to avoid small expected numbers in the chi-square test. Table 6.27 shows that the withdrawn subgroup was not significantly different from the active population.

6.2.3 Analysing the first quiz

C0 was large but unlike the Newcastle experiment C1-C3 were together not very small. Table 6.28 shows a weak effect of consistency on candidates' success in the

Table 6.28: Consistency and the first quiz result

Result	**C0**	**C1-3**	**IB**	**Total**
Pass	27 (77%)	11 (55%)	20 (54%)	58
Fail	8	9	17	34
Total	35	20	37	92
$\chi^2 = 4.825,\ df = 2,\ p < 0.090$ **not significant**				

Table 6.29: Summarising consistency and the first quiz result – grouped

Groups	**Pass rate**		χ^2	df	p	**Size**
	Consistent	**Inconsistent**				
CM2/notCM2	100% (11 of 11)	58% (47 of 81)	7.324	1	0.007	92
C0/notC0	77% (27 of 35)	54% (31 of 57)	4.820	1	0.028	92
C/notC	70% (39 of 56)	53% (19 of 36)	2.675	1	0.102	92

first quiz result. 77% (27 out of 35) in the C0 subgroup passed the course while this figure is 54% (31 out of 57) in notC0. A chi-square test shows there is not a significant difference between subgroups.

Table 6.29 summarises the effect of consistency on CM2/notCM2, C0/notC0 and C/notC slices. As expected the effect is large in CM2/notCM2 subgroup (100%/58%) and highly significant ($p < 0.007$). The effect became weak in the C0/notC0 (77%/54%) and even weaker in C/notC (70%/53%) group arrangemnets. Although the consistent subgroup achieved a better result than the inconsistent subgroup in these two cases, consistency did not significantly separate the subgroups.

Consistency and success rate in sub-populations: first quiz

The effect of consistency on results is shown in table 6.30 within seven different slices. Candidates in the C0 subgroup achieved a better result than the rest in each slice, but the effect of consistency is weak and chi-square shows no significant difference between subgroups in five of the slices. The effect was strong in the subgroup with relevant programming experience (94%/54%) and significant ($p < 0.013$). The effect was almost strong in the subgroup with prior programming course (79%/55%) but chi-square did not separate the subgroups significantly ($p < 0.051$). Notice that the assessment mechanism in the first quiz

Table 6.30: Summarising the consistency (C0/notC0) and the first quiz result – sliced by programming background

Slices	Pass rate		χ^2	*df*	*p*	Size
	C0	notC0				
CM2 excluded	67% (16 of 24)	54% (31 of 57)	1.046	1	0.306	81
Prior experience	75% (9 of 12)	48% (16 of 33)	2.506	1	0.113	45
No prior experience	78% (18 of 23)	62% (13 of 21)	1.411	1	0.235	44
Relevant experience	94% (15 of 16)	54% (7 of 13)	6.237	1	0.013	29
No relevant experience	63% (12 of 19)	54% (22 of 41)	0.477	1	0.490	60
Prior course	79% (19 of 24)	55% (22 of 40)	3.805	1	0.051	64
No prior course	73% (8 of 11)	58% (7 of 12)	0.524	1	0.469	23

Table 6.31: Summarising prior programming factors and the first quiz result

Slices	Pass rate		χ^2	*df*	*p*	Size
	yes	no				
Prior experience	55% (25 of 45)	70% (31 of 44)	2.117	1	0.146	89
Relevant experience	76% (22 of 29)	57% (34 of 60)	3.088	1	0.079	89
Prior course	64% (41 of 64)	65% (15 of 23)	0.01	1	0.920	87

was non-technical and it did not separate consistent and inconsistent subgroups significantly.

Programming background and success rate: first quiz

Table 6.31 shows the effect of programming background on pass rates when candidates who did not answer questions about their prior programming experience are eliminated. Programming experience had a negative effect (55%/70%), relevant programming experience had a low effect (76%/57%) and attending programming course had no visible effect (64%/65%) on candidates' success.

6.2.4 Result of the second quiz

From 118 potential participants in the second quiz, 46 were absent leaving 72 active subjects. 28 subjects were assessed as C0, two as C1, five as C2, seven as C3, 19 as I and 11 as B. Table 6.32 shows the population of candidates and consistency subgroups. Table 6.33 shows that the withdrawn subgroup was not significantly different from the active population.

Table 6.32: Consistency levels among population in the second quiz

	Active	**Withdrew**
C0	28	13
C1	2	1
C2	5	5
C3	7	11
I	19	9
B	11	7
Total	72	46

Table 6.33: Consistency and active/withdrew divisions in the second quiz

Consistency	**Active**	**Withdrew**
C	42	30
I/B	30	16
Total	72	46
$\chi^2 = 0.559,\ df = 1,\ p < 0.455$		
not significant		

6.2.5 Analysing the second quiz

Table 6.34 shows the effect of candidates' consistency in the second quiz result. The effect of consistency on candidates' success is large. 79% (22 out of 28) candidates in C0 passed the couse and the figure drops to 27% (12 out of 44) in the notC0 subgroup. A chi-square test also shows that consistency has a significant effect to separate the subgroups.

Table 6.35 shows a summary of the results in three slices (CM2/notCM2, C0/notC0, C/notC). The table shows that the consistent subgroup performed much better than the inconsistent subgroup. As expected the effect is large

Table 6.34: Consistency and the second quiz result

Result	**C0**	**C1-3**	**IB**	**Total**
Pass	22 (79%)	5 (38%)	7 (29%)	34
Fail	6	8	24	38
Total	28	13	31	72
$\chi^2 = 18.994,\ d = 2,\ p < 0.001$ **highly significant**				

Table 6.35: Summarising consistency and the second quiz result – grouped

Groups	**Pass rate**		χ^2	*df*	*p*	**Size**
	Consistent	**Inconsistent**				
CM2/notCM2	89% (8 of 9)	41% (26 of 63)	7.652	1	0.022	72
C0/notC0	79% (22 of 28)	27% (12 of 44)	18.067	1	0.001	72
C/notC	64% (27 of 42)	23% (7 of 30)	11.776	1	0.001	72

in CM2/notCM2 (89%/41%), but it is also large in C0/notC0 (79%/27%) and in C/notC (64%/23%). The chi-square test shows that the result is strongly significant to separate the candidates in each group arrangement. The second quiz required candidates to write program code and they could not have done this without basic programming knowledge. This quiz seems to have been a stronger assessment mechanism than the first quiz and thus could separate the consistent and inconsistent subgroups.

Consistency and success rate in sub-populations: second quiz

Table 6.36 shows the effect of consistency when candidates are filtered by their programming background. The consistent subgroup within each slice achieved much better results than the inconsistent subgroup and the effect of consistency is large in each slice. The chi-square test shows strongly significant differences between the subgroups in six slices. Consistent candidates performed almost twice as well as inconsistent candidates in the sub-population with no prior experience (67%/33%), although the chi-square test shows only a weak significance. The pattern of results is very similar to that seen in the Newcastle experiment, except for the weak effect in the sub-population without relevant programming experience in the Newcastle experiment (80%/59%).

Programming background and success rate: second quiz

Table 6.37 shows the effect of prior programming experience on pass rates when candidates who did not answer questions about their prior programming experience are eliminated. There is no large or significant effect of prior programming background on candidates' success. Relevant programming experience was observed to have some effect (62%/42%) but prior programming experience (51%/44%) and prior programming course (51%/44%) none. The chi-square re-

Table 6.36: Summarising consistency (C0/notC0) and the second quiz result – sliced by programming background

Slices	Pass rate		χ^2	*df*	*p*	Size
	C0	notC0				
CM2 excluded	74% (14 of 19)	27% (12 of 44)	17.104	1	0.001	63
prior experience	88% (14 of 16)	70% (5 of 17)	11.386	1	0.001	33
No prior experience	67% (8 of 12)	33% (8 of 24)	3.600	1	0.058	36
Relevant experience	100%(10 of 10)	27% (3 of 11)	11.748	1	0.001	21
No relevant experience	67% (12 of 18)	27% (8 of 30)	7.406	1	0.007	48
Prior course	77% (17 of 22)	31% (9 of 29)	10.702	1	0.001	51
No prior course	86% (6 of 7)	20% (2 of 10)	7.137	1	0.008	17

Table 6.37: Summarising prior programming factors and the second quiz result

Slices	Pass rate		χ^2	*df*	*p*	Size
	yes	no				
Prior experience	51% (17 of 33)	44% (16 of 36)	0.345	1	0.557	69
Relevant experience	62% (13 of 21)	42% (20 of 48)	2.398	1	0.121	69
Prior course	51% (26 of 51)	44% (8 of 17)	0.078	1	0.780	68

sult also shows that the prior programming factor did significantly separate the candidates.

6.2.6 Summarising the Middlesex experiments

Although the result of the first quiz indicated a weak effect of consistency on subjects' success rate, the numbers were in the right direction – the consistent subjects did better than the others. The result of the second quiz showed the same effect, but much more strongly. It seems possible that the different level of difficulty of the two quizzes produced this distinction.

In the second quiz, the effect of consistency on success rate was shown to be strong in the CM2/notCM2, C0/notC0 and C/notC divisions and in six out of seven sub-populations filtered by programming background. The effect was weak only when candidates with no prior programming experience were examined.

None of the prior programming elements had a strong effect on candidates' success in the Middlesex experiments. The pass rate of candidates with prior programming experience was slightly higher than those who had never programmed before in the second experiment and very similar in the first one. Attending a prior course had no effect on candidates' success in either experiment.

Table 6.38: Consistency levels

Consistency level	Active
C0	43
C1	0
C2	0
C3	2
I	12
B	1
Total	58

6.3 University of Sheffield - 2007

Experiment conducted by Peter Rockett, Department of Electronic Engineering, University of Sheffield, UK.

In this experiment the test was administered at the beginning of the academic year 2007/2008. The test was administered once, before the course began (week 0). 58 subjects participated in this study and all were present in the final examination.

6.3.1 Assessment method

The assessment mechanism consisted of a compulsory group-work assignment and a formal examination which had to be taken at the end of the academic year. In both the group-work assignment and the formal examination, subjects were required to write program code for particular problem-solving purposes.

6.3.2 Test result

Table 6.38 shows the candidate population. 43 subjects (74%) were assessed as C0, two as C1-3, twelve (21%) as I and one as B.

6.3.3 Analysing the result

Joining models in the marksheet hardly expanded the C group; C0 is large and C1-C3 are very small. Table 6.39 shows the effect of consistency on the binary exam result. Chi-square suggests that the effect is strongly significant, but two cells contain small expected numbers which undermines this result.

Table 6.39: Consistency and the binary course mark

Result	**C0**	**C1-3**	**IB**	**Total**
Pass	39 (91%)	2	5 (38%)	46
Fail	4	0	8	12
Total	43	2	13	58
$\chi^2 = 17.139,\ d = 2,\ p < 0.0001$ **very highly significant**				

Table 6.40: Summarising consistency and the binary course mark – grouped

Groups	**Pass rate**		χ^2	*df*	*p*	**Size**
	Consistent	**Inconsistent**				
CM2/notCM2	72% (8 of 11)	81% (38 of 47)	0.359	1	0.549	58
C0/notC0	91% (39 of 43)	47% (7 of 15)	13.139	1	0.0001	58
C/notC	91% (41 of 45)	38% (5 of 13)	17.039	1	0.0001	58

Table 6.40 shows a summary of the results in three slices (CM2/notCM2, C0/notC0 and C/notC). The table shows, as in the Newcastle experiments, that the consistent subgroup produced much better results than the inconsistent subgroup in both the C0/notC0 and C/notC group arrangement with strongly significant difference between subgroups. Surprisingly there is no effect in CM2/notCM2 and chi-square shows no significance. It can be explained perhaps by the fact that about half of the CM2 population (5 out of 11) in the Sheffield experiment reported no prior programming experience.

Consistency and success rate in sub-populations

The effect of consistency on candidates result is shown in table 6.41 within seven different slices. Subjects in two slices (with prior experience, with relevant experience) were all in C0 and in one slice (with prior course) there was only one in notC0, effectively leaving four slices for analysis. The effect of consistency in these slices are large and chi-square shows a strongly significant difference between subgroups. Note that the effect of consistency in the sub-population without relevant programming experience, which was weak in the Newcastle experiment, is strong in this experiment.

Table 6.41: Summarising consistency and the binary course mark – filtered

Slices	Pass rate		χ^2	*df*	*p*	Size
	C0	notC0				
CM2 excluded	97% (31 of 32)	47% (7 of 15)	16.629	1	0.001	47
Prior experience	93% (13 of 14)	none	none	none	none	14
No prior experience	90% (26 of 29)	47% (7 of 15)	9.744	1	0.002	44
Relevant experience	86% (6 of 7)	none	none	none	none	7
No relevant experience	92% (33 of 36)	47% (7 of 15)	12.675	1	0.001	51
prior course	80% (8 of 10)	0.0% (0 of 1)	2.933	1	0.087	11
No prior course	94% (31 of 33)	50% (7 of 14)	12.258	1	0.001	47

Table 6.42: Summarising prior programming factors and binary course mark – sliced

Slices	Pass rate		χ^2	*df*	*p*	Size
	yes	no				
Prior experience	93% (13 of 14)	75% (33 of 44)	2.064	1	0.151	58
Relevant experience	86% (6 of 7)	78% (40 of 51)	0.199	1	0.655	58
Prior course	93% (13 of 14)	81% (38 of 47)	0.359	1	0.549	58

Background and success rate

Table 6.42 shows the effect of prior programming experience on pass rates when candidates who did not answer questions about their prior programming experience are eliminated. Although none of the programming background elements had a significant effect on candidates' success, prior programming experience is shown to have a slightly bigger effect than the other two factors. The relatively strong effect of prior relevant experience on success, which was observed in both Middlesex experiments, has entirely vanished in this experiment.

6.3.4 Summarising the Sheffield experiment

As in the Newcastle and the second quiz at Middlesex experiments, the effect of consistency on success rate was highly significant. The effect was strong and also significant in the four subgroups sliced by prior programming background which could be analysed.

None of the programming background aspects had a significant positive effect on candidates' success in this experiment, although the pass rate of the candidates with prior programming experience was slightly higher than those who never did programming before.

Table 6.43: Consistency levels

Consistency level	Active
C0	62
C1	3
C2	10
C3	5
I	25
B	5
Total	110

6.4 University of Westminster - 2008

Experiment conducted by Christopher Thorpe, Department of Computing Science, University of Westminster, UK.

The test was administered once, at the beginning of the academic year 2007/2008. 139 subjects participated in this study; 29 were not present in the final examination, leaving 110 active subjects. I did not have access to the test score of withdrawn subjects.

6.4.1 Assessment method

A compulsory formal examination was taken at the end of the academic year. I had no access to the examination paper.

6.4.2 Test result

Table 6.43 shows the candidate population. 62 subjects were assessed as C0, 25 as I and five as B. There were 18 in C1-C3: compared to other experiments, this is quite high.

6.4.3 Analysing the result

Joining models in the mark-sheet expanded the C groups quite a lot in this experiment, but the C0 subgroup, with 62 subjects, is still the largest and I, with 25 subjects, is the second largest. Table 6.44 shows the effect of consistency on success. The effect is weak (77%/60%) and chi-square also shows that the effect

Table 6.44: Consistency and the binary course mark

Result	**C0**	**C1-3**	**I/B**	
Pass	48 (77%)	12 (66%)	17 (57%)	77
Fail	14	6	13	33
Total	62	18	30	110
$\chi^2 = 4.260,\ d = 2,\ p < 0.119$ **not significant**				

Table 6.45: Summarising consistency and binary course mark – grouped

Groups	**Pass rate**		χ^2	*df*	*p*	**Size**
	Consistent	**Inconsistent**				
CM2/notCM2	75% (6 of 8)	70% (71 of 102)	0.103	1	0.749	110
C0/notC0	77% (48 of 62)	60% (29 of 48)	3.724	1	0.054	110
C/notC	75% (60 of 80)	57% (17 of 30)	3.492	1	0.062	110

is not significant. Although the effect is weak and not significant the pattern of success is slightly in favour of consistent subjects.

Table 6.45 shows a summary of the results in three group arrangements (CM2/notCM2, C0/notC0 and C/notC). As in the Sheffield experiment the effect was surprisingly weak in the CM2/notCM2 group arrangement, except with a the difference that this time only one candidate in the CM2 population reported no programming background. The consistent subgroup achieved slightly better than the inconsistent subgroup in the two other cases but not significantly so. The effect on consistency in every group arrangement is weak and the chi-square test did not show significance in any of these cases.

Consistency and success rate

Table 6.46 shows that the consistent subgroup achieved a better result in every slice than the inconsistent subgroup even though the chi-square test result shows no significance in all but one case. The effect more or less disappears in subjects without prior programming experience and those who attended a programming course before.

Despite the fact that the result of this experiment and the first at Middlesex do not significantly support my hypothesis, they are included as they should be in the meta-analysis (chapter 7).

Table 6.46: Summarising consistency and binary course mark – filtered

Slices	Pass rate		χ^2	*df*	*p*	Size
	C0	notC0				
CM2 excluded	78% (42 of 54)	60% (29 of 48)	3.621	1	0.057	102
Prior experience	85% (34 of 40)	60% (15 of 25)	5.182	1	0.023	65
No prior experience	62% (10 of 16)	60% (12 of 20)	0.023	1	0.878	36
Relevant experience	80% (12 of 15)	50% (3 of 6)	1.890	1	0.169	21
No relevant experience	76% (31 of 41)	61% (24 of 39)	1.842	1	0.175	80
Prior course	77% (27 of 35)	70% (18 of 26)	0.483	1	0.487	61
No prior course	93% (13 of 14)	50% (8 of 16)	6.531	1	0.011	30

Table 6.47: Summarising programming background factors and binary course mark – sliced

Slices	Pass rate		χ^2	*df*	*p*	Size
	yes	no				
Prior experience	75% (49 of 65)	61% (22 of 36)	2.261	1	0.133	101
Prior relevant experience	71% (15 of 21)	69% (55 of 80)	0.056	1	0.813	101
Prior course	74% (45 of 61)	70% (21 of 30)	0.143	1	0.705	91

Programming background and success rate

Table 6.47 shows the effect of programming background on pass rates when candidates who did not answer questions about prior programming experience are eliminated. Prior programming experience appeared to have a slight effect, although it is not significant. None of the other programming background factors had even a weak effect on candidates' success in this experiment.

6.4.4 Summarising the Westminster experiment

Although there was no significant effect of consistency in this experiment, the numbers were in the right direction – the consistent subjects did better than the others. It is possible that the assessment method was not strong enough to separate consistent and inconsistent subgroups clearly, or that the candidates were not drawn from the same population as the participants of the other experiments in this study. As in the other experiments, the programming background factors had no significant effect on the candidates' success.

Table 6.48: Consistency levels

Consistency level	Active
C0	99
C1	1
C2	0
C3	3
I	0
B	2
Total	105

6.5 University of York - 2006

Experiment conducted by Dimitar Kazakov, Department of Computer Science, University of York, UK. The test was administered at the beginning of the academic year 2006/2007. 109 subjects participated, of whom 4 withdrew before the final exam.

6.5.1 Assessment method

The assessment mechanism consisted of three compulsory group-work assignments, and a formal examination taken at the end of the academic year.

6.5.2 Test result

Table 6.48 shows the candidate population. 94% (99 out of 105) of subjects were assessed as C0, 4 as C1-3, none as I and two as B.

6.5.3 Analysing the result

Joining models in the marksheet did not expand the C group very much: C0 is huge and C1-C3 is tiny. Table 6.49 shows the effect of candidates' consistency on the binary exam results. The effect is large (92%/50%) and chi-square shows that the effect is highly significant but expected values in most cells are too small for us to rely on the result.

Table 6.50 summarises the effect of consistency in three group arrangements of CM2/notCM2, C0/notC0 and C/notC. Like the experiment in Westminster the effect was surprisingly weak in CM2/notCM2 group arrangement (92%/78%) and

Table 6.49: Consistency levels and the binary course mark

Result	**C0**	**C1-3**	**I/B**	**Total**
Pass	91 (92%)	2	1	94
Fail	8	2	1	11
Total	99	4	2	105
$\chi^2 = 10.599,\ d = 2,\ p < 0.005$ **highly significant**				

Table 6.50: Summarising consistency and binary course mark – grouped

Groups	**Pass rate**		χ^2	*df*	*p*	**Size**
	Consistent	**Inconsistent**				
CM2/notCM2	92% (80 of 87)	78% (14 of 18)	3.196	1	0.074	105
C0/notC0	92% (91 of 99)	50% (3 of 6)	10.599	1	0.001	105
C/notC	90% (93 of 103)	50% (1 of 2)	3.396	1	0.065	105

chi-square shows no significant difference between CM2 and notCM2 populations. In this experiment 7 candidates in the CM2 population reported no programming background but all passed the course, so they are not the cause of the weak effect. The consistent subgroups in the other two group arragements achieved much better results than the inconsistent subgroup and the effect was large (92%/50% and 90%/50%) and strongly significant in C0/notC0 – but again, too few in the inconsistent subgroup to believe it.

Consistency and success rate

Table 6.51 shows that only 18 subjects remained when CM2 subjects are excluded. This means that 87 subjects (83%) were in the CM2 subgroup out of 91 subjects (87%) who reported prior programming experience. Therefore when candidates were filtered in seven slices by background programming factors, most were in the C0 subgroup, and only a small number were left in notC0 in each slice. We should not pay too much attention to the chi-square test results of these slices when the figures in the notC0 group are so small. But the results of this experiment are included as they should be in the meta-analysis (chapter 7) as a top-extreme case, an experiment with a programming-skilful population.

Table 6.51: Summarising consistency and binary course mark – filtered

Slices	Pass rate		χ^2	*df*	*p*	Size
	C0	notC0				
CM2 excluded	92% (11 of 12)	50% (3 of 6)	4.018	1	0.045	18
Prior experience	92% (81 of 88)	100% (3 of 3)	0.259	1	0.611	91
No prior experience	91% (10 of 11)	0.0% (0 of 3)	9.545	1	0.002	14
Relevant experience	95% (54 of 57)	100% (1 of 1)	0.056	1	0.814	58
No relevant experience	88% (37 of 42)	40% (2 of 5)	7.318	1	0.007	47
prior course	94% (61 of 65)	50% (3 of 6)	11.883	1	0.001	71
No prior course	88% (30 of 34)	none	none	none	none	34

Prior programming experience and success rate

Table 6.52 shows the effect of programming background on pass rates when candidates who did not answer questions about prior programming experience are eliminated. Prior programming experience had an effect and relevant experience had a weak effect on candidates success (92%/71% and 95%/85%) and the chi-square test shows the effect significantly separated the subgroups ($p < 0.02$ and $p < 0.049$). There is no effect of prior programming course. There are a number of elements which separate the York experiment from the other experiments which were conducted in this study:

- The distribution of subjects in the consistent subgroups was entirely different. 94% (99 out of 105) of subjects were in C0 and there were none in I.
- 87 subjects (88%) in C0 were in the CM2 subgroup (using the correct model of Java).
- 87% (91 out of 105) had prior programming experience.
- 68% (71 out of 105) had attended a programming course before.

6.5.4 Summarising the York experiment

The population in the York experiment was clearly different from the population in other experiments of this study. The huge number of subjects in the CM2/C0/C and tiny numbers in notC0/notC caused many cells to be too small to give a reliable result.

Table 6.52: Summarising programming background elements and binary course mark – sliced

Slices	Pass rate		χ^2	df	p	Size
	yes	no				
Prior experience	92% (84 of 91)	71% (10 of 14)	5.64	1	0.020	105
Relevant experience	95% (55 of 58)	88% (39 of 47)	3.886	1	0.049	105
Prior course	90% (30 of 34)	88% (64 of 71)	0.089	1	0.765	105

When examining the overall effect of consistency on success, I consider this experiment as a top-extreme case, because the population had a strong programming background, and I consider the first Middlesex experiment as a bottom-extreme case because of its non-technical assessment mechanism.

6.6 University of Aarhus - 2006

Experiment conducted by Michael E. Caspersen, Jens Bennedsen, Kasper Dalgaard Larsen, Department of Computer Science, University of Aarhus, Denmark.

The experiment was carried out in the first week of the academic year 2006/2007. 142 subjects participated in this study and all were present in the final examination.

6.6.1 Assessment method

The assessment mechanism was a computerised online task where students could accumulate marks step-by-step in accomplishing the task.

6.6.2 Test result

Table 6.53 shows the candidate population reported in Caspersen et al. (2007). 124 subjects (87%) were assessed as C, 18 (13%) as notC. The table shows that there is no significant effect of consistency on success in this experiment. I did not have access to any further data for analysis.

As I discussed earlier, the effect of consistency on success was low in the first Middlesex experiment, compared to other experiments. I speculate that the effect of consistency to separate candidates, into consistent and inconsistent subgroups,

Table 6.53: Consistency among population

Result	C	notC	Total
Pass	120	16	136
Fail	4	2	6
Total	124	18	142
$(\chi^2 = 1.888,\ d = 1,\ p < 0.169)$ **not significant**			

could be weakened by a non-technical assessment mechanism. The examination mechanism in Aarhus, accumulating marks through an online process, was totally different from the assessment mechanism of the other experiments in this study.

The experimenters appeared to have believed, based on reading of Dehnadi and Bornat (2006), that my test is psychometric and can be taken at anytime. Their experiment refutes that notion, but it is not one that I put forward in this thesis.

6.7 Royal School of Signals, Blandford - 2007

Experiment conducted by Stuart Wray, Royal School of Signals, Blandford, UK.

Baron-Cohen et al. (2003) exposed an association between classical autism and people such as scientists, mathematicians and engineers who are *systematizers*, skilled at inventing, using and analysing systems.

Wray (2007), inspired by Baron-Cohen et al. (2001) and Baron-Cohen et al. (2003), administered four tests: the *SQ* (Systemizing Quotient) test and the *EQ* (Empathy Quotient) test; a self-ranking test; and the original version of my instrument (chapters 4 and 5) to 19 students of a postgraduate course, five months after the end of the course. Wray correlated the results with students' end of course programming test result. He found some interesting associations.

He found a correlation between the *SQ* and *EQ* results and programming ability which became stronger when he examined *SQ–EQ* (score in Systemizing Quotient minus score in Empathy Quotient). *SQ–EQ* has been shown to be significantly different for males, females and Asperger syndrome individuals (Baron-Cohen et al., 2003).

In contrast, for this group of students, there was no association between pro-

Table 6.54: Success rates

	NewC	Mdx1	Mdx2	Shef	West	York	Overall
Population	71	92	72	58	110	105	508
Success	66%	63%	47%	79%	70%	90%	70%

gramming ability measured by exam mark and either the self-ranking test result or my test.

Wray appears to have believed, based on reading of Dehnadi and Bornat (2006) that my test is psychometric. But that is not a claim that I put forward in this thesis. Wray has not noticed that my test will be ruined by candidates' prior knowledge of assignment and sequence. Conducting the test five months after the end of the course demonstrates only that most subjects have moved to the CM2 subgroups who know the correct model of assignment and sequence.

6.8 Summary of further experiments

The pass rates in the experiments and overall are shown in table 6.54. The pass rate in the second Middlesex experiment is exceptionally low, in Sheffield high and in York exceptionally high. There is a gradient in UK universities in the prior achievement levels of admitted students: Middlesex and Westminster are towards the lower end, Sheffield and York towards the higher. Differences in pass rates may reflect this. The first and second Middlesex experiments were successive in-course examinations of the same cohort.

Table 6.55 shows the effects of background factors on success (the small number who did not reply in each case are ignored). There were few strong differences in the figures. Effects of age and sex are not analysed here: there were very small numbers of women in the experiments, too small to analyse with the tools I had available; and the very small age spread in the populations, typically two or three years, showed almost no differences in the experiments in which I analysed it.

Table 6.56 shows the success rates of consistent subjects against the rest, in C0/notC0 and C/notC divisions.

Table 6.57 shows the effect of consistency on success in subgroups in separate experiments. I added the CM2/notCM2 group arrangement as the top row of this table. To see whether the effect of consistency was simply the effect of prior

Table 6.55: Effect of programming background on success in separate experiments

		NewC		**Mdx1**		**Mdx2**		**Shef**		**West**		**York**		**Overall**	
		pop	succ	pop	succ	pop	succ	pop	succ	pop	succ	pop	succ	pop	succ
Prior	yes	46	63%	45	56%	33	52%	14	93%	65	75%	91	92%	294	74%
experience	no	19	68%	44	70%	36	44%	44	75%	36	61%	14	71%	193	65%
Relevant	yes	33	61%	29	76%	21	62%	7	86%	21	71%	58	95%	169	78%
experience	no	32	69%	50	68%	48	42%	51	78%	80	69%	47	83%	308	68%
Prior	yes	30	63%	64	64%	51	51%	14	93%	61	74%	34	88%	254	69%
course	no	32	69%	23	65%	17	47%	47	81%	30	70%	71	90%	220	76%

Table 6.56: Effect of consistency on success in separate experiments

	NewC		**Mdx1**		**Mdx2**		**Shef**		**West**		**York**		**Overall**	
	pop	succ	pop	succ	pop	succ	pop	succ	pop	succ	pop	succ	pop	succ
C0	44	80%	35	77%	28	79%	43	91%	62	77%	99	92%	311	84%
notC0	27	44%	57	54%	44	27%	15	47%	48	60%	6	50%	197	48%
C0-C3	52	79%	56	70%	42	64%	45	91%	80	75%	103	90%	378	80%
notC	19	32%	36	53%	30	23%	13	38%	30	57%	2	50%	130	42%

learning of programming, I looked at this row, and I looked again at subjects outside the CM2 group (incorrect model) divided by C0/notC0. Then I looked at each of the subgroups defined by the background questions analysed in table 6.55. The CM2 group does generally better than the rest in every experiment but one – the exception is Sheffield. In the rest of the table in *every single case* the C0 group does better than notC0, usually by a considerable margin.

The effect was weakest in the first Middlesex, Westminster and York experiments. In Newcastle, second Middlesex and Sheffield consistent subjects did about twice as well as the rest. At York, as at Aarhus, most subjects scored consistently in the test (99 out of 105 in York, 124 out of 142 in Aarhus). To evaluate more reliably the claim that consistency has a noticeable effect on success in learning to program, the results of the experiments were combined in a meta-analysis. There were so few non-consistent subjects at York that I could put little weight on that particular result, but I can, as I should, include it in the meta-analysis.

Table 6.57: Effect of consistency on success in subgroups, separate experiments

		NewC		Mdx1		Mdx2		Shef		West		York		Overall	
		pop	succ	pop	succ	pop	succ	pop	succ	pop	succ	pop	succ	pop	succ
Correct	CM2	23	87%	11	100%	9	89%	11	73%	8	75%	87	92%	149	89%
model	notCM2	48	56%	81	58%	63	41%	47	81%	102	70%	18	78%	359	62%
Incorrect	C0	21	71%	24	67%	19	74%	32	97%	54	78%	12	92%	162	80%
model	notC0	27	44%	57	54%	44	27%	15	47%	48	60%	6	50%	197	48%
With prior	C0	33	79%	12	75%	16	88%	14	93%	40	85%	88	92%	203	87%
experience	notC0	13	23%	33	48%	17	18%	0	-	25	60%	3	100%	91	44%
No prior	C0	13	85%	23	78%	12	66%	29	90%	16	62%	11	91%	104	80%
experience	notC0	6	33%	21	62%	24	33%	15	47%	20	60%	3	0%	89	47%
With rele-	C0	24	79%	16	94%	10	100%	7	86%	15	80%	57	95%	129	90%
vant exp	notC0	9	11%	13	54%	11	27%	0	-	7	50%	1	100%	40	38%
No rele-	C0	15	80%	19	63%	18	67%	36	92%	41	76%	42	88%	171	80%
vant exp	notC0	17	59%	40	55%	30	27%	15	47%	39	63%	5	40%	146	50%
With prior	C0	21	76%	24	79%	22	77%	10	80%	35	77%	65	94%	177	84%
course	notC0	9	33%	40	55%	29	31%	1	0%	26	69%	6	50%	111	50%
No prior	C0	15	93%	11	73%	7	86%	33	94%	14	93%	34	88%	114	89%
course	notC0	17	47%	12	58%	10	20%	14	50%	16	50%	0	-	69	46%

Chapter 7

Meta-analysis

The Winer procedure of meta-analysis (Winer et al., 1971) was used to examine the overall effect of consistency and/or programming background on success. The procedure combines p values from χ^2 analysis of separate experiments – the probability of obtaining the effect by accident – to give an overall p value. Because this is a meta-analysis of several experiments, our threshold significance value is set at a conservative 0.01 (1%).

7.1 Overall effect of programming background

The overall effect of programming experience, relevant programming experience and prior course on candidates' success was examined. The population, success rate, chi-square (χ^2) and probability (p) values were extracted from tables 6.25, 6.31, 6.37, 6.42, 6.47 and 6.52. Tables 7.1, 7.2 and 7.3 give the results.

Meta-analysis in table 7.1 shows prior programming experience had a weak effect on success overall. The weak effect was driven by 60% of candidates with prior programming experience overall. The success rate in table 7.1 shows that prior programming experience had a negative effect on success in the Newcastle and Middlesex first experiments. The effect was weak in the second Middlesex and Westminster experiments. The York experiment with 86% experienced candidates and 83% CM2 population (see table 6.57) had a strong weight in the overall weak effect. The significance drops to $0.2 < p < 0.3$ if the York experiment is eliminated. The overall effect has not been driven by any other experiments.

Meta-analysis in table 7.2 shows prior relevant programming experience had a weak effect on success overall, also not significant ($0.1 < p < 0.2$), with a negative

Table 7.1: Overall effect of programming experience on success

		pop	succ	χ^2	p	$-ln(p)$
NewC	yes	46	63%	0.170	0.680	0.38566
	no	19	68%			
Mdx1	yes	45	55%	2.117	0.146	1.89712
	no	44	70%			
Mdx2	yes	33	52%	0.345	0.557	0.57982
	no	36	44%			
Shef	yes	14	93%	2.064	0.151	1.69712
	no	44	75%			
West	yes	65	75%	2.261	0.133	2.04022
	no	36	61%			
York	yes	91	92%	5.64	0.020	3.91202
	no	14	71%			
Total	yes	249	74%	$\sum -ln(p)$		10.51196
	no	193	65%			
$df = 2 * (6) = 12;\quad \chi^2 = 2 * (10.51196) = 21.02392$						
$0.05 < p < 0.10$ **not significant**						

Table 7.2: Overall effect of prior relevant programming experience on success

		pop	succ	χ^2	p	$-ln(p)$
NewC	yes	33	61%	0.471	0.492	0.71335
	no	32	69%			
Mdx1	yes	29	76%	3.088	0.079	2.65926
	no	60	57%			
Mdx2	yes	21	62%	2.398	0.121	2.12026
	no	48	62%			
Shef	yes	7	86%	0.199	0.655	0.43078
	no	51	78%			
West	yes	21	71%	0.056	0.813	0.21072
	no	80	69%			
York	yes	58	95%	3.886	0.049	2.99573
	no	47	83%			
Total	yes	169	77%	$\sum -ln(p)$		9.1301
	no	318	66%			
$df = 2 * (6) = 12;\quad \chi^2 = 2 * (9.1301) = 18.2602$						
$0.10 < p < 0.20$ **not significant**						

Table 7.3: Overall effect of prior programming course on success

		pop	succ	χ^2	p	$-ln(p)$
NewC	yes	30	63%	0.203	0.652	0.43078
	no	32	69%			
Mdx1	yes	64	64%	0.010	0.920	0.8338
	no	23	65%			
Mdx2	yes	51	51%	0.078	0.780	0.49430
	no	17	47%			
Shef	yes	14	93%	0.359	0.549	0.59783
	no	47	81%			
West	yes	61	74%	0.143	0.705	0.45667
	no	30	70%			
York	yes	34	88%	0.089	0.765	0.27443
	no	71	90%			
Total	yes	254	68%	$\sum -ln(p)$		2.33739
	no	220	76%			
$df = 2 * (6) = 12; \quad \chi^2 = 2 * (2.33739) = 4.67478$						
$0.95 < p < 0.98$ **not significant**						

impact on success in the Newcastle experiment. The Westminster and Sheffield experiments ($p < 0.813$, $p < 0.655$) respectively had the least weight and the York experiment ($p < 0.049$) had the most, in the overall effect.

Meta-analysis in table 7.3 shows prior programming course apparently had no visible or significant effect on overall success and similar results in each individual experiment. The success rate shows that attending a programming course had a negative effect in the Newcastle, first Middlesex and the York experiments, and overall.

7.2 CM2 population

Meta-analysis in table 7.4 shows an overall significant difference in the CM2/notCM2 division ($p < 0.001$). The candidates in the CM2 subgroup did know about assignment and sequence before the course started and the notCM2 subgroup did not. As we might expect the CM2 candidates with prior programming knowledge should perform better than the notCM2 population in every experiment, and they almost always did. Despite the overall significant difference in the CM2/notCM2 division, the success rates show that the CM2 population performed worse than

Table 7.4: Overall effect of prior programming learning on success, using CM2 subgroup

		pop	succ	χ^2	p	$-ln(p)$
NewC	CM2	23	87%	6.552	0.010	4.60517
	notCM2	48	56%			
Mdx1	CM2	11	100%	7.324	0.007	4.60517
	notCM2	81	58%			
Mdx2	CM2	9	89%	7.165	0.007	4.60517
	notCM2	63	41%			
Shef	CM2	11	73%	0.359	0.549	0.59783
	notCM2	47	81%			
West	CM2	8	75%	0.103	0.749	0.30110
	notCM2	102	70%			
York	CM2	87	92%	3.196	0.074	2.65926
	notCM2	18	78%			
Total	CM2	149	89%	$\sum -ln(p)$		17.3737
	notCM2	359	62%			
$df = 2 * (6) = 12$; $\chi^2 = 2 * (17.3737) = 34.7474$						
$p < 0.001$ **highly significant**						

the others in Sheffield, were almost the same in Westminster and not much different in York. However, it is clear that this division separates the population better than the separations made by prior programming factors (89%/62%, $p < 0.001$ compared to 74%/65%, $0.05 < p < 0.10$, 77%/66%, $0.10 < p < 0.20$, 68%/76%, $0.95 < p < 0.98$) but not as well as the separations made by both the C0/notC0 divisions (84%/48%, $p < 0.001$) or C/notC (82%/40%, $p < 0.001$) and even the C0/notC0 division when the CM2 group is excluded (table 7.7, 80%/48%, $p < 0.001$).

7.3 Overall effect of consistency on success in the whole population

The overall effect of consistency on success in two group arrangements (C0/notC0 and C/notC) was examined. The population, success rate, chi-square (χ^2) and probability (p) values were extracted from tables 6.23, 6.29, 6.35, 6.40, 6.45 and 6.50.

Meta-analysis in table 7.5 and 7.6 shows a highly significant effect of con-

Table 7.5: Overall effect of consistency (C0/notC0) on success

		pop	succ	χ^2	p	$-ln(p)$
NewC	C0	44	79%	9.213	0.002	4.60517
	notC0	27	44%			
Mdx1	C0	35	77%	4.820	0.028	3.50655
	notC0	57	54%			
Mdx2	C0	28	79%	18.067	0.001	4.60517
	notC0	44	27%			
Shef	C0	43	91%	13.139	0.001	4.60517
	notC0	15	47%			
West	C0	62	77%	3.724	0.054	2.99573
	notC0	48	60%			
York	C0	99	92%	10.599	0.001	4.60517
	notC0	6	50%			
Total	C0	306	84%	$\sum -ln(p)$		24.92296
	notC0	202	48%			
$df = 2 * (6) = 12;\quad \chi^2 = 2 * (24.92296) = 49.84592$ $p < 0.001,\quad$ **highly significant**						

Table 7.6: Overall effect of consistency (C/notC) on success

		pop	succ	χ^2	p	$-ln(p)$
NewC	C	52	79%	13.894	0.001	4.60517
	notC	19	32%			
Mdx1	C	56	70%	2.675	0.102	2.30258
	notC	36	53%			
Mdx2	C	42	64%	11.776	0.001	4.60517
	notC	30	23%			
Shef	C	45	91%	17.039	0.001	4.60517
	notC	13	38%			
West	C	80	75%	3.492	0.062	2.81341
	notC	30	57%			
York	C	103	90%	3.396	0.065	2.81341
	notC	2	50%			
Total	C	378	80%	$\sum -ln(p)$		22.25574
	notC	130	42%			
$df = 2 * (6) = 12;\quad \chi^2 = 2 * (22.25574) = 44.51148$ $p < 0.001,\quad$ **highly significant**						

sistency on success in both the C0/notC0 (84%/48%, $p < 0.001$) and C/notC (80%/42%, $p < 0.001$) group arrangements. The first Middlesex and the Westminster experiments had the least weight in the overall significance in both cases (C0/notC0 and C/notC) especially in C/notC. The overall significance has not been driven by any individual experiment: eliminating any of them would not sway the overall result.

7.4 Overall effect of consistency on success in sub-populations

The overall effect of consistency on success in seven sub-populations has been examined. The population, success rate, chi-square (χ^2) and probability (p) values were extracted from tables 6.24, 6.30, 6.36, 6.41, 6.46 and 6.51.

Meta-analysis in table 7.7 shows an overall significant effect (80%/48%, $p < 0.001$) of consistency when the CM2 population is excluded. The Westminster experiment had the least weights in the overall effect (76%/60%, $p < 0.057$) and first Middlesex had none (67%/54%, $p < 0.306$). The success rate in the consistent subgroup is strongly higher than in the inconsistent subgroup in every other experiment. The overall effect has not been driven by any individual experiment: eliminating any of them would not sway the overall result.

Meta-analysis in table 7.8 shows an overall significant effect of consistency on success in the prior programming experience slice (87%/41%, $p < 0.001$). The York experiment had three and Sheffield had no candidates in the notC0 subgroup. The success rate in the consistent subgroup is strongly higher than the inconsistent subgroup in all other experiments. Surprisingly the effect in the Westminster experiment was almost as high as in Newcastle and second Middlesex (85%/60%, $p < 0.023$). The overall significance has not been driven by any individual experiment: eliminating any of them would not sway the overall result.

Meta-analysis in table 7.9 shows an overall significant effect of consistency on success in the no prior programming experience slice (80%/47%, $p < 0.001$). The Westminster (62%/60%) and first Middlesex (78%/62%) experiments had again the least weights in the overall effect, although the consistent subgroup performed slightly better than the inconsistent subgroup even there. The success rates in the consistent subgroup were strongly higher than the inconsistent subgroup in

Table 7.7: Overall effect of consistency (C0/notC0) on success (without CM2 subgroup)

		pop	succ	χ^2	p	$-ln(p)$
NewC	C0	21	71%	4.428	0.062	2.81341
	notC0	27	44%			
Mdx1	C0	24	67%	1.046	0.306	1.20397
	notC0	57	54%			
Mdx2	C0	19	74%	11.793	0.001	4.60517
	notC0	34	35%			
Shef	C0	32	97%	16.629	0.001	4.60517
	notC0	15	47%			
West	C0	54	76%	3.621	0.057	2.81341
	notC0	48	60%			
York	C0	12	92%	4.018	0.045	3.21887
	notC0	6	50%			
Total	C0	162	80%	$\sum -ln(p)$		19.26
	notC0	197	48%			
$df = 2 * (6) = 12$; $\chi^2 = 2 * (19.26) = 38.52$ $p < 0.001$, **highly significant**						

Table 7.8: Overall effect of consistency (C0/notC0) on success (candidates with prior programming experience)

		pop	succ	χ^2	p	$-ln(p)$
NewC	C0	33	79%	12.424	0.001	4.60517
	notc0	13	23%			
Mdx1	C0	12	75%	2.506	0.113	2.20727
	notCo	33	48%			
Mdx2	C0	16	85%	16.102	0.001	4.60517
	notC0	44	27%			
Shef	C0	14	93%	none	none	none
	notC0	0	-			
West	C0	40	85%	5.182	0.023	3.91202
	notC0	25	60%			
York	C0	88	92%	0.259	0.611	0.49430
	notC0	3	100%			
Total	C0	203	87%	$\sum -ln(p)$		15.82393
	notC0	118	41%			
$df = 2 * (5) = 10$; $\chi^2 = 2 * (15.82393) = 31.64786$ $p < 0.001$ **highly significant**						

Table 7.9: Overall effect of consistency (C0/notC0) on on success (candidates with no prior programming experience)

		pop	succ	χ^2	p	$-ln(p)$
NewC	C0	13	85%	4.997	0.025	3.91202
	notC0	6	33%			
Mdx1	C0	23	78%	1.411	0.235	1.46968
	notC0	21	62%			
Mdx2	C0	12	67%	3.600	0.058	2.99573
	notC0	24	33%			
Shef	C0	29	90%	9.744	0.002	4.60517
	notC0	15	47%			
West	C0	16	62%	0.023	0.878	0.13926
	notC0	20	60%			
York	C0	11	91%	9.545	0.002	4.60517
	notC0	3	0%			
Total	C0	104	80%	$\sum -ln(p)$		17.72703
	notC0	89	47%			
$df = 2 * (6) = 12;\quad \chi^2 = 2 * (17.72703) = 35.45406$ $p < 0.001$**, highly significant**						

every other experiment. This overall effect has not been driven by any individual experiment: eliminating any of them would not sway the overall result.

Meta-analysis in table 7.10 shows an overall significant effect of consistency on success in the prior relevant programming experience slice (90%/37%, $p < 0.001$). The Sheffield experiment had none and York had one candidate in the notC0 subgroup. The success rate of the consistent subgroup in every other experiment was strongly higher than the inconsistent subgroup. The overall effect was not driven by any of the individual experiments: eliminating any of them would not sway the overall result.

Meta-analysis in table 7.11 shows an overall significance of consistency on success in the no prior relevant experience slice (80%/50%, $p < 0.001$). The success rate in the consistent subgroup in the Newcastle (80%/59%, $p < 0.197$), Westminster (76%/61%, $p < 0.175$) and the first Middlesex (63%/54%, $p < 0.490$) experiments were slightly higher than the inconsistent subgroup but had no weight in the overall effect. The success rate of the consistent subgroup was significantly higher than the inconsistent subgroup in every other experiment. The overall effect was strong and not driven by any of the individual experiments: eliminating

Table 7.10: Overall effect of consistency (C0/notC0) on success (candidates with prior relevant programming experience)

		pop	succ	χ^2	p	$-ln(p)$
NewC	C0	24	79%	12.698	0.001	4.60517
	notC0	9	11%			
Mdx1	C0	16	94%	6.237	0.013	4.60517
	notC0	13	54%			
Mdx2	C0	10	100%	11.748	0.001	4.60517
	notC0	11	27%			
Shef	C0	7	86%	none	none	none
	notC0	0	-			
West	C0	15	80%	1.890	0.169	1.77196
	notC0	6	50%			
York	C0	57	95%	0.056	0.814	0.21072
	notC0	1	100%			
Total	C0	129	90%	$\sum -ln(p)$		15.79819
	notC0	40	37%			
$df = 2 * (5) = 10;\quad \chi^2 = 2 * (15.79819) = 31.59638$ $p < 0.001$**,** **highly significant**						

any of them would not sway the overall result.

Meta-analysis in table 7.12 shows an overall significant effect of consistency on success in the prior course slice (84%/50%, $p < 0.001$). The Sheffield experiment had no candidates in the notC0 subgroup. The consistent subgroup performed slightly better than the inconsistent group in the Westminster experiment (77%/70%, $p < 0.487$) but significantly better in all other experiments. The overall effect was strong and not driven by any of the individual experiments: eliminating any of them would not sway the overall result.

Meta-analysis in table 7.13 shows an overall significant effect of consistency on success in the no prior programming course slice (90%/46%, $p < 0.001$). York has none in the notC0 subgroup. The consistent subgroup performed slightly better than the inconsistent group in the first Middlesex experiment (73%/58%, $p < 0.469$) but much better in all other experiments. The overall effect was strong and not driven by any of the individual experiments: eliminating any of them would not sway the overall result.

Table 7.11: Overall effect of consistency C0/notC0) on success (candidates with no relevant programming experience)

		pop	succ	χ^2	p	$-ln(p)$
NewC	C0	15	80%	1.663	0.197	1.60944
	notC0	17	59%			
Mdx1	C0	19	63%	0.477	0.490	0.71335
	notC0	41	54%			
Mdx2	C0	18	67%	7.406	0.007	4.60517
	notC0	30	27%			
Shef	C0	36	92%	12.675	0.001	4.60517
	notC0	15	47%			
West	C0	41	76%	1.842	0.175	1.77196
	notC0	39	61%			
York	C0	42	88%	7.318	0.007	4.60517
	notC0	5	40%			
Total	C0	171	80%	$\sum -ln(p)$		17.91026
	notC0	147	50%			
$df = 2 * (6) = 12;\quad \chi^2 = 2 * (17.91026) = 35.82052$ $p < 0.001$, **highly significant**						

Table 7.12: Overall effect of consistency (C0/notC0) on success (candidates with prior programming course)

		pop	succ	χ^2	p	$-ln(p)$
NewC	C0	21	76%	4.178	0.041	3.21887
	notC0	9	33%			
Mdx1	C0	24	79%	3.805	0.051	2.299573
	notC0	40	55%			
Mdx2	C0	22	77%	10.702	0.001	4.60517
	notC0	29	31%			
Shef	C0	10	80%	2.933	0.087	2.52573
	notC0	1	0%			
West	C0	35	77%	0.483	0.487	0.73397
	notC0	26	70%			
York	C0	65	94%	11.883	0.001	4.60517
	notC0	6	50%			
Total	C0	177	84%	$\sum -ln(p)$		17.9885
	notC0	111	50%			
$df = 2 * (6) = 12;\quad \chi^2 = 2 * (17.9885) = 35.977$ $p < 0.001$, **highly significant**						

Table 7.13: Overall effect of consistency (C0/notC0) on success (candidates with no prior programming course)

		pop	succ	χ^2	p	$-ln(p)$
NewC	C0	15	93%	7.942	0.005	4.60517
	notC0	17	47%			
Mdx1	C0	11	73%	0.524	0.469	0.75502
	notC0	12	58%			
Mdx2	C0	7	86%	7.137	0.008	4.60517
	notC0	10	20%			
Shef	C0	33	94%	12.258	0.001	4.60517
	notC0	14	50%			
West	C0	14	93%	6.531	0.011	4.60517
	notC0	16	50%			
York	C0	34	88%	none	none	none
	notC0	0	-			
Total	C0	114	90%	$\sum -ln(p)$		19.1757
	notC0	69	46%			
$df = 2 * (5) = 10;$ $\chi^2 = 2 * (19.1757) = 38.3514$ $p < 0.001$, **highly significant**						

7.5 Summarising the overall effects

Table 7.14 summarises the overall effects of programming background on success, showing the size of the effect, the χ^2 value and significance. None of the programming background factors had a large or a significant effect. Attending a programming course was shown to have no significant effect on success, both overall and in each individual experiment.

On the other hand, meta-analysis shows in table 7.15 a large and highly significant effect of consistency on success in both the C0/notC0 and C/notC

Table 7.14: Overall effect of programming background on success

		pop	succ	χ^2	df	p
Prior experience	yes	294	74%	21.02	12	$0.05 < p < 0.10$
	no	193	65%			
Relevant experience	yes	169	78%	18.26	12	$0.10 < p < 0.20$
	no	308	68%			
Prior course	yes	254	69%	4.67	12	$0.95 < p < 0.98$
	no	220	76%			

Table 7.15: Overall effect of consistency on success

	pop	succ	χ^2	*df*	*p*
C0	311	84%	49.84	12	$p < 0.001$
notC0	197	48%			
C0-3	378	80%	44.51	10	$p < 0.001$
notC	130	42%			

Table 7.16: Overall effect of consistency on success in filtered subgroups

		pop	succ	χ^2	*df*	*p*	**Significance**
Correct model	CM2	149	89%	34.75	12	$p < 0.001$	very high
	notCM2	359	62%				
Incorrect model	C0	162	80%	40.27	12	$p < 0.001$	very high
	notC0	197	48%				
With prior experience	C0	203	87%	31.65	10	$p < 0.001$	very high
	notC0	91	44%				
No prior experience	C0	104	80%	35.45	12	$p < 0.001$	very high
	notC0	89	47%				
With relevant experience	C0	129	90%	31.60	10	$p < 0.001$	very high
	notC0	40	38%				
No relevant experience	C0	171	80%	35.82	12	$p < 0.001$	very high
	notC0	146	50%				
With prior course	C0	177	84%	35.98	12	$p < 0.001$	very high
	notC0	111	50%				
No prior course	C0	114	89%	38.35	10	$p < 0.001$	very high
	notC0	69	46%				

group arrangements. Despite the weak effect in the first quiz at Middlesex and the Westminster experiment, especially in C/notC, none of the experiments is driving the result: if we eliminate any one of them there is still strong significance.

Table 7.16 summarises the overall effect of consistency on success in the eight population divisions characterised by programming background factors. The overall result confirms the result of the initial experiment by demonstrating a strong and significant effect of consistency on success in *every* slice. None of the experiments is driving the overall result: if we eliminate any one we still find a large significant effect.

This analysis shows that consistency is not simply the effect of learning to program. The CM2 group does do better than any other, as might be expected. But there are slightly more individuals who are C0-consistent but not CM2, their success rate is almost as good, and they are almost twice as likely to pass as those who are not consistent. I note that the CM2/notCM2 division (89%/62%) is a more effective predictor of success than prior programming experience (74%/64%), even if we take the weakly significant effect of prior experience at face value, but both give many more false negatives than either measure of consistency (C0/notC0 84%/48%, C/notC 80%/42%). We can see these effects more starkly if we look at failure rates: the CM2 group, which has learnt one of the basics of imperative programming, has an 11% failure rate against 38% for the others; programming background gives 26%/36%; consistency gives either 16%/52% or 20%/58%.

The size of the effect varies according to the population division but it is significant everywhere and it is never small. Programming background, or its absence, does not eliminate the effect of consistency.

During these experiments, some factors have been identified that could damage the result of this test:

Non-standardised assessment mechanism It is an accepted principle that programming ability be measured by final examination mark or by averaging marks of practical tests, assignments and final examination in a majority of institutions. But there are no procedures to keep these assessment mechanisms at the same standard among different institutions.

Population In the York and the Westminster experiments some of the attributes such as the size of the CM2 population, the population with prior experience, the overall success rates, and so on, indicated that candidates may

not be drawn from the same population as the participants of the other experiments in this study.

Psychometric My test is not a psychometric test and will be ruined by candidates' prior knowledge of assignment and sequence. Conducting the test at an inappropriate time of a programming course demonstrates only that most subjects have moved to the CM2 subgroup who know the correct model of assignment and sequence.

Chapter 8

Conclusion and future work

This study started by examining novice programmers' common mistakes and misconceptions, targeting the causes of students' difficulties in learning programming. Noticing the patterns of rationales behind novices mistakes swayed the study toward investigating novices' mental ability which was found to have a great effect on their learning performance.

The test characterises two populations in introductory programming courses which perform significantly differently. More than half of novices spontaneously build and consistently apply a mental model of program execution; the rest are either unable to build a model or to apply one consistently. The results of the experiments described in chapters 5 and 6 confirm that the first group performed very much better in their end-of-course examination than the second: an overall 84% success rate in the first group, 48% in the second (in the C0/notC0 group arrangement, table 7.15).

Administering a test related to assignment and sequence in the first week of an introductory programming course, without giving any explanation of what the test is about despite the risk of participants' rejection, revealed an extraordinary result. It revealed that the ability to build mental models of programming execution were behind some novices' reasoning strategies. This ability was found to be pre-determined and brought into the first programming course.

The test introduced by this study divides students into consistent and inconsistent subgroups on the basis of their mental modelling of program execution. The significant differences between the consistent and inconsistent subgroups persisted in the sub-populations produced when candidates were separated by background factors of programming experience, relevant programming experience and prior

programming course which might be thought to have had an effect on success.

Despite the tendency of institutions to rely on students' prior programming experience as a predictor of success, programming background has only a weak effect on novices' success, and though effective prior learning of assignment and sequence has a stronger effect, it is not as strong as consistency. A weak positive effect of prior programming experience (relevant/irrelevant) was observed which appeared to be driven by one of the experiments with a programming-skilful population.

8.1 Deficiencies of the test

The test results revealed that some candidates apply a mental model systematically. But the present data does not easily distinguish between ability to form models and a mere willingness to do so.

The test was not an ordinary questionnaire. Candidates were asked to answer programming related questions with no explanation, no instruction and at the beginning of a course before any lessons were given. This peculiarity of the test may have affected the result. There are various obvious reasons that some candidates judged inconsistent might still have the ability to use a mental model:

1. They might be able to use a mental model if we explained what sort of answer we are looking for;
2. They might be able to use a mental model of assignment and sequence if we taught it to them;
3. They might be using a mental model not recognised in the test;
4. The might not like to answer a multiple choice questionnaire;
5. They might not like to guess.

All these contribute to the false negative proportion in the experimental result. In section 4.4 I found that when the test was taken in week 3, some of the candidates in groups 1 and 2 had learnt assignment and sequence and shifted from inconsistent to consistent. The false-negative proportion in that test result decreased to 25%. I speculate that if candidates were given a pre-session study, helping them to build a mental model, similar to a model of program execution,

it might help them to become consistent in the test. I cannot think of anything to persuade candidates who do not like to guess.

There are also various reasons why some candidates judged consistent might lack the ability to use a mental model:

- They might guess correctly by chance;
- They might be following some sort of pattern, such as "always choose numbers which add up 11";
- They might have learned the answer in advance.

There were many alternatives in each question, and they were permuted so that the order of models was different in each question. It seems unlikely, therefore, that a significant proportion could guess by chance. There do not seem to be visible patterns in the data. The test is novel and not widely published, so it seems unlikely that any student had prior knowledge. I conclude, therefore, that those judged inconsistent genuinely lacked the ability to use a model.

On the other hand, it seems likely that a significant number of candidates judged inconsistent do have the ability to use a model. If we could reduce this number then the test would be more reliable.

8.2 More about counteracting selection bias

Section 5.4 deals with bias caused by prior knowledge of programming. Intake policy might also cause bias. In general, institutions look at candidates' mathematical, scientific and general education attainment. Some institutions are more restrictive in the level of these requirements and some less restrictive. Candidates' response to questions about background education revealed the different intake policies in institutions. For example, the University of Sheffield only recruited students with a strong background in mathematics, and in the University of York almost all students had a prior programming background (indeed, almost all were CM2), while intake requirements in the University of Westminster and Middlesex University were weaker. I did not examine the effect of candidates' mathematics or general education background on success because, looking at the literature in the field, relevant mathematical background was only found to be weakly correlated to success in learning *scientific programming* (Reinstedt et al., 1964). But is

there a correlation of, say, maths ability and the effect of consistency on success? It would seem not: the signal of the effect of consistency on success is clearly similar in each experiment regardless of the institution's intake policy.

Sex and age may also effect the test result. The questionnaire asks about candidates' age and gender, and I had the intention of analysing their effect on success. I found the age groups in introductory to programming courses very restricted and therefore did not attempt an analysis. The very low proportion of female candidates in Computer Science did not let me examine the effect of gender either.

Another important bias which might cause an effect in the test results is a surprisingly large number of candidates with programming background who failed to learn programming in the course. Most likely they were not able to learn programming in their first attempt either. If they knew assignment and sequence in advance they would be judged consistent and if they then failed the course, would increase the false-positive proportion in the test result. If they did not learn assignment and sequence in advance they would be judged inconsistent and those who cross the pass/fail borderline, would increase the false-negative proportion in the test result.

I have not had the opportunity to directly investigate the level of candidates' success/failure in their past programming experience and its impact on their current performance. However, the present results gave no substantial indication of a major influence of prior programming experience.

8.3 Speculations

Although the study exposed the patterns of rationalisation underlying novices' strategies which make them different from each other, the causes of difficulties to learn programming and why candidates in the consistent subgroup find it easier to learn than the others are still obscure. It might be explained by some speculations.

Cross (1970) found programmers to be rather peculiar individuals, willing to work in isolation, avoiding interaction with different features of the organisation. At that time computer programming was not a widespread skill and Cross proposed such a peculiarity as a good measure of programming aptitude on the basis of a small sample population. The proposal was not taken seriously until thirty-

three years later, when Baron-Cohen et al. (2003) exposed this peculiarity as a classic trait associated with autism. He labeled people such as scientists, mathematicians and engineers as *systematizers*, who are skilled at analysing systems; they can exhibit behaviour such as obsession with detail and are less interested in the social side of life. Baron-Cohen has not addressed any association between ability to systematize and programming skills but Wray (2007) has found a correlation between Baron-Cohen's measures and success in programming, five months after the end of a programming course.

My study shows that students in the consistent subgroup have the ability to build a mental model, a drive to construct a system, something that follows rules like a mechanical construct, and this is what more or less what a systematizer does. In fact my study shows that candidates in the consistent subgroup have the ability to systematize. I can speculate that Wray and I are both measuring a similar trait by different instruments. The results in both studies separate learners with a high ability to systematize, using two different approaches. He used Baron-Cohen's EQ and SQ measures and I chased candidates' mental models.

8.4 Further Work

During the process of this research I have noticed some opportunities to enhance this study.

8.4.1 Repeat the test at week 3

In the first methodical experiment of this study, described in section 4.4, the test was administered twice. The first test was conducted in the week before the subjects had received any programming teaching and the second one in the third week when the assignment model M2 (right to left copy) and the sequence model (S1) of composition had been taught. Table 8.1 shows that 49% of subjects (29 out of 59) had managed to understand assignment and sequence in three weeks, and from them 90% managed to pass the course at the end. The 51% who did not understand assignment and sequence after three weeks had only a 20% chance of passing the course. The effect of consistency captured by week 3 test in the initial experiment was 74%/25% which was large and significant ($p < 0.0001$).

Table 8.2 demonstrates the effect of understanding assignment and sequence

Table 8.1: Early understanding of assignment and sequence and average result

	Yes(+)	**No(−)**	**Total**
Pass	26 (90%)	6 (20%)	32
Fail	3	24	27
	29	30	59

Table 8.2: Consistency and early understanding of assignment and sequence

	C+	**I+**	**C–**	**I–**	**Total**
Pass	20 (95%)	3 (60%)	3 (50%)	3 (17%)	29
Fail	1	2	3	16	22
	21	5	6	19	51

in the success of the C and I subgroups (51 out of 59). The notation "+" and "–" in the labels is used to indicate understanding assignment and sequence after three weeks. The table shows that 72% of those who learned assignment and sequence in the first three weeks were in the C subgroup in week 0 and had a 95% chance to pass the course.

Table 8.3 shows the same effect in the blank subgroup (small population). Only subjects who understood assignment and sequence within three weeks managed to pass the course and the rest failed.

As a result of this evidence I suggest that the additional test in week 3 has a valuable potential to be used as a success predictor. It would also show a group of students who are likely to struggle in the course and might be supported by remedial teaching.

Table 8.3: Blankness and early understanding of assignment and sequence

	B+	**B–**	**Total**
Pass	3 (100%)	0 (0%)	3
Fail	0	5	5
	3	5	8

Table 8.4: Distribution of subgroups among experiments

Experiment	C0	C1-3	IB	Total
Sheffield	43	2 (3%)	13	58
York	99	4 (4%)	2	105
Newcastle	44	8 (11%)	19	71
Westminster	62	18 (16%)	30	110
Middlesex	35	21 (23%)	37	93
	283	53 (12%)	101	437

8.4.2 Reviewing the judgment of mental models by combining columns differently in the mark sheet

In section 5.3.2, I explained that candidates who used only one model are clearly consistent and candidates who switched between two related models are also consistent, but less so. In section 5.3.2 I described my judgment of related models.

I could have combined models in any of three different dimensions:

- copy/move
- left/right
- add/overwrite

Because in the S3 model of multiple assignment the copy/move distinction goes away, I chose that as the weakest dimension. Then I decided to ignore direction, and finally addition/overwrite.

In practice joining models in this way was not very successful. It did not expand the C group very much: C0 is always large and C1-C3 are usually small. Table 8.4 illustrates the pattern of C1-3 population in five experiments with average of 12%.

Decomposing the C/I subgroups with a more accurate test might provide some answers to the excess number of false negative results – those inconsistent subjects who pass the exam – observed in these experiments.

8.4.3 Mental models of another hurdle

Assignment and sequence are not the only hurdles which trip up novices. Two other major hurdles which trip novice imperative programmers are recur-

sion/iteration and concurrency. Concurrency trips not only novices, it is a hurdle jumped by just a few. Iteration trips novices at an early stage, particularly when all the alternative kinds of iteration (while, do-while, for) are introduced at the same time. In fact iteration is conceptually difficult, and the proper treatment of iteration is mathematically complicated.

One of the potential extensions to this study is looking at the mental mechanism that students use when thinking about a simple iteration. Since studying students' mental models of assignments and sequences revealed distinguishable populations, understanding the mechanism involved when students think about an iteration might reveal some other unsuspected phenomena.

8.4.4 Revising the mental models list

I prepared the list of mental models by analysing novices' rational mistakes and enhanced it during the progress of this study. Failing to notice a mental model would cause some novices to be misplaced in their subgroup which would effect the accuracy of the test result. The potential of revising the list of mental models of assignment and sequence should not be overlooked.

8.4.5 Testing high school students

By the method introduced in this study, students with the ability to construct and apply a model systematically can be distinguished at the beginning of an introductory programming course. Having this result before the course starts shows that this capability has not been developed by the University's education. Therefore there is a potential that the test could separate learners at an earlier stage of their academic study. I would not be surprised if half of the students in high school have the ability to use a consistent mental model systematically and I would be interested to see how far back it goes. Understanding when novices start to develop this ability is one of the major potentials to extend this study.

8.4.6 Interviewing

In the initial stage of this study I interviewed some of the students and captured valuable information which helped me to analyse the mistakes behind their misconceptions. Unfortunately none of the later experiments included a formal

interview. Novices' strategies were judged on the basis of their responses to the questions and the draft notes around their test papers. Interviewing candidates might light up some ideas which could lead us to a better understanding of their thinking strategies.

8.4.7 Follow-up performance

The result of this study confirmed that the ability to use a recognisable model consistently has a considerable impact on novices' success in the first level of programming courses. But how this ability could assist their success in the next level of programming courses remains unknown. There were no follow-up experiments in this study to trace the effect.

Pursuing candidates' progress in a higher level of programming course (second semester, second year, final year) could expose the scope of the effect of consistency on candidates' performance.

8.5 Open problems

During the process of this research I have faced some problems which remained unsolved.

8.5.1 Alternative methods

In this study I followed the notion of mental models as a vehicle to derive a rationale behind novices' reasoning when solving a series of unexplained problems. Although the significance of mental models in novice learners was repeatedly confirmed in the literature, the method is surely not the only way of looking at their difficulties.

The result of this study showed that 49% of candidates from the inconsistent population managed to pass the first programming course. Thus is quite a high negative occurence, against the initial hypothesis in this study. The study cannot suggest a remedial enhancement to reduce the false-negative occurence in this result.

8.5.2 Measuring programming skills

This study introduced a test that measures ability to learn programming which is confirmed to be associated with course marks at the end of the introductory programming course. It should be taken into consideration that the course mark is not a perfect mechanism to reflect programming skills, but there is no well-defined alternative.

Measuring programming skills by course mark in this study showed noticeable variation in the strengths of consistency among institutions. Each institution has its own assessment mechanism which is designed under restraint of many individual factors.

At Middlesex, where I have access to the examination materials, I know that the first Middlesex exam was a non-technical first in-course quiz, largely book-work, whereas the second Middlesex exam was a more technical second quiz. The second exam separated students more radically and showed a stronger effect of consistency, with far fewer passes in the non-consistent groups. It may be that the effect of consistency is generally weaker in non-technical examinations. I believe that this matter is worth further investigation.

Simon et al. (2006b) explained the lack of an accepted measure of programming aptitude which does not let us to find correlations between performance on simple tasks and programming aptitude. He added "We do not pretend that there is a linear relationship between programming aptitude and mark in a first programming course, or that different first programming courses are assessed comparably; but we have succumbed to the need for an easily measured quantity."

Although the test introduced by this study measures the ability to learn programming with some accuracy, without a solid method of measuring programming skill, an optimum result cannot be achieved. Now we have a mechanism which measures programming aptitude, what we need is a well-defined and effective assessment mechanism to measure programming skills at the end of the introductory programming course.

Bibliography

Beth Adelson and Elliot Soloway. The role of domain experience in software design. *IEEE Trans. Softw. Eng.*, 11(11):1351–1360, 1985. 4578.

Marzieh Ahmadzadeh, Dave Elliman, and Colin Higgins. An analysis of patterns of debugging among novice computer science students. In *ITiCSE '05: Proceedings of the 10th annual SIGCSE conference on Innovation and technology in computer science education*, pages 84–88, New York, NY, USA, 2005. ACM. ISBN 1-59593-024-8.

Anthony Allowatt and Stephen Edwards. IDE support for test-driven development and automated grading in both Java and C++. In *Eclipse '05: Proceedings of the 2005 OOPSLA workshop on Eclipse technology eXchange*, pages 100–104, New York,USA, 2005. ACM. ISBN 1-59593-342-5.

David J. Barnes and Michael Kölling. *Objects First With JAVA: A Practical Introduction Using BlueJ (3rd Edition)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2006. ISBN 013197629X.

S. Baron-Cohen, S. Wheelwright, R. Skinner, J. Martin, and Clubley. The autism spectrum quotient (AQ): Evidence from Asperger's syndrome/high functioning autism, males and females, scientists and mathematicians. *Journal of Autism and Developmental Disorders*, 31:5–17, 2001.

S. Baron-Cohen, J. Richler, D. Bisarya, N. Gurunathan, and S. Wheelwright. The systemising quotient (SQ): An investigation of adults with Asperger's syndrome or high functioning autism and normal sex differences. *Philosophical Transactions of the Royal Society, Series B, Special issue on "Autism: Mind and Brain"*, 358:361–374, 2003.

M.I. Bauer and P.N. Johnson-Laird. How diagrams can improve reasoning: Mental models and the difficult cases of disjunction and negation. In *Cognitive Science*, Boulder, Colorado, 1993. Lawrence Erlbaum Associates.

Piraye Bayman and Richard E. Mayer. A diagnosis of beginning programmers' misconceptions of BASIC programming statements. *Commun. ACM*, 26(9): 677–679, 1983. ISSN 0001-0782.

Jens Bennedsen and Michael E. Caspersen. Abstraction ability as an indicator of success for learning computing science? In *ICER '08: Proceedings of the fourth international workshop on Computing education research*, pages 15–26, New York, NY, USA, 2008. ACM. ISBN 978-1-60558-216-0. doi: http://doi.acm.org/10.1145/1404520.1404523.

Jens Bennedsen and Michael. E. Caspersen. Abstraction ability as an indicator of success for learning object-oriented programming? *SIGCSE Bull.*, 38(2): 39–43, 2006. 1138430.

C. Besie, M. Myer, L. VanBrackle, and N. Chevli-Saroq. An examination of Age, Race, and Sex as Predictors of Success in the First Programming Course. *Journal of Informatics Education Research*, 5:51–64, 2003.

A. J. Biamonte. A study of the effect of attitudes on the learning of computer programming. In *CPRC 65: Proceedings of the third annual computer personnel research conference*, pages 68–74, New York, NY, USA, 1965. ACM.

Alan Blackwell, Peter Robinson, Chris Roast, and Thomas Green. Cognitive models of programming-like activity. In *CHI '02: CHI '02 extended abstracts on Human factors in computing systems*, pages 910–911, New York, NY, USA, 2002. ACM. ISBN 1-58113-454-1. doi: http://doi.acm.org/10.1145/506443.506654.

Jeffrey Bonar and Elliot Soloway. Preprogramming knowledge: a major source of misconceptions in novice programmers. *Hum.-Comput. Interact.*, 1(2):133–161, 1985. ISSN 0737-0024.

Jeffrey Bonar and Elliot Soloway. Uncovering principles of novice programming. In *POPL '83: Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on principles of programming languages*, pages 10–13, New York, NY, USA, 1983. ACM. ISBN 0-89791-090-7.

C. Borgman. Why do some people have more difficulty learning to use an information retrieval system than others? In *SIGIR '87: Proceedings of the 10th annual international ACM SIGIR conference on Research and development in information retrieval*, pages 61–71, New York, NY, USA, 1987. ACM. ISBN 0-89791-232-2.

Richard Bornat. *Programming from first principles*. Prentice Hall International (UK) Ltd., Hertfordshire, UK, 1986. ISBN 0-137-29104-3.

T Boyle, B Stevens-Wood, F Zhu, and A Tika. Structured learning in virtual environments. *Computer and Education*, 261(3):41–49, 1996.

Tom Boyle and Thomas Green. Build your own virtual computer: a multimedia simulation using an active learning approach. In *Proceedings of New Media for Global Communications, International Conference on Multimedia, Hypermedia and Virtual reality*, Moscow, 1994.

Tom Boyle and Sue Margetts. The core guided discovery approach to acquiring programming skills. *Comput. Educ.*, 18(1-3):127–133, 1992. 134897.

Tom Boyle, John Gray, Brian Wendi, and Martyn Davies. Talking the plunge with CLEM: the design and evaluation of a large scale CAL system. In *CAL '93: Selected contributions from the 93 symposium on CAL into the mainstream*, pages 19–26, Tarrytown, NY, USA, 1994. Pergamon Press, Inc.

Tom Boyle, Claire Bradley, Peter Chalk, Ken Fisher, Ray Jones, and Poppy Pickard. Improving pass rates in introductory programming. *4th Annual LTSN-ICS Conference, NUI Galway 2003 LTSN Centre for Information and Computer Science*, 2003.

M. D. S. Braine and D. P. O Brien. A theory of if: a lexical entry reasoning program, and pragmatic principles. *Psychological Review*, pages 182–203, 1991.

Fred P. Brooks, Jr. The mythical man-month. In *Proceedings of the international conference on Reliable software*, pages 193–193, New York, NY, USA, 1975. ACM. doi: http://doi.acm.org/10.1145/800027.808439.

Kim B. Bruce, Andrea Danyluk, and Thomas Murtagh. Events and objects first: an innovative approach to teaching JAVA in CS 1. *J. Comput. Small Coll.*, 16 (4):1–1, 2001. ISSN 1937-4771.

José Juan Cañas, Maria Teresa Bajo, and Pilar Gonzalvo. Mental models and computer programming. *Int. J. Hum.-Comput. Stud.*, 40(5):795–811, 1994. ISSN 1071-5819.

Michael E. Caspersen, Kasper Dalgaard Larsen, and Jens Bennedsen. Mental models and programming aptitude. In *ITiCSE '07: Proceedings of the 12th annual SIGCSE conference on Innovation and technology in computer science education*, pages 206–210, New York, NY, USA, 2007. ACM. ISBN 978-1-59593-610-3.

Peter Chalk, Claire Bradley, and Poppy Pickard. Designing and evaluating learning objects for introductory programming education. *SIGCSE Bull.*, 35(3): 240–240, 2003. ISSN 0097-8418.

Zhixiong Chen and Delia Marx. Experiences with Eclipse IDE in programming courses. *J. Comput. Small Coll.*, 21(2):104–112, 2005. ISSN 1937-4771.

Stephen Cooper, Wanda Dann, and Randy Pausch. Teaching objects-first in introductory computer science. In *SIGCSE '03: Proceedings of the 34th SIGCSE technical symposium on Computer science education*, pages 191–195, New York, NY, USA, 2003. ACM. ISBN 1-58113-648-X.

Kenneth Craik. *The Nature of Explanation.* Cambridge University Press, 1943.

E. Cross. The behavioral styles of computer programmers. In *Proc 8th Annual SIGCPR Conference*, pages 69–91, Maryland, WA, USA, 1970.

Paul Curzon. Learning computer science through games and puzzles. In *Computers and Fun 2*, York, UK, 1999.

Paul Curzon. Learning computer science through games and puzzles. In *Proceedings of C@MDX'00, Middlesex University School of Computing Science, Research Student's Conference*, pages 14–15, London, UK, 2000.

Saeed Dehnadi. Testing programming aptitude. In *PPIG '06: Proceedings of the 18th Annual Workshop of the Psychology of Programming Interest Group*, England, 2006.

Saeed Dehnadi and Richard Bornat. The camel has two humps. In *Little PPIG*, Coventry, UK, 2006.

Benedict du Boulay. Some difficulties of learning to program. *Journal of Educational Computing Research*, 2(1):57–73, 1986.

Linda P. DuHadway, Stephen W. Clyde, Mimi M. Recker, and Donald H. Cooley. A concept-first approach for an introductory computer science course. *J. Comput. Small Coll.*, 18(2):6–16, 2002. ISSN 1937-4771.

Jennifer L. Dyck and Richard E. Mayer. BASIC versus natural language: is there one underlying comprehension process? In *CHI '85: Proceedings of the SIGCHI conference on Human factors in computing systems*, pages 221–223, New York, NY, USA, 1985. ACM. ISBN 0-89791-149-0.

R.M. Felder and L.K. Silverman. Learning and teaching styles in engineering education. *Engr. Education*, 78(7):674–681, 1988.

Sally Fincher, Raymond Lister, Tony Clear, Anthony Robins, Josh Tenenberg, and Marian Petre. Multi-institutional, multi-national studies in CSEd research: some design considerations and trade-offs. In *ICER '05: Proceedings of the first international workshop on Computing education research*, pages 111–121, New York, NY, USA, 2005. ACM. ISBN 1-59593-043-4.

Shu Geng, Paul Schneeman, and Wen-Jan Wang. An empirical study of the robustness of analysis of variance procedures in the presence of commonly encountered data problems. *American Journal Enology and Viticulture*, 33(3): 131–134, 1982.

Dedre Gentner and Albert L. Stevens. *Mental Models*. Lawrence Erlbaum Associates, New Jersey, 1983.

Elena Giannotti. Algorithm animator: a tool for programming learning. *SIGCSE Bull.*, 19(1):308–314, 1987. ISSN 0097-8418.

Kenneth J. Goldman. A concepts-first introduction to computer science. In *SIGCSE '04: Proceedings of the 35th SIGCSE technical symposium on Computer science education*, pages 432–436, New York, NY, USA, 2004. ACM. ISBN 1-58113-798-2.

John Gray, Tom Boyle, and Colin Smith. A constructivist learning environment implemented in Java. In *ITiCSE '98: Proceedings of the 6th annual conference*

on the teaching of computing and the 3rd annual conference on Integrating technology into computer science education, pages 94–97, New York, NY, USA, 1998. ACM. ISBN 1-58113-000-7.

Thomas Green. Cognitive approach to software comprehension: results, gaps and limitations. In *Extended abstract of talk at workshop on Experimental Psychology in Software Comprehension Studies 97*. University of Limerick, University of Limerick, Ireland, 1997.

Thomas R. G. Green. Instructions and descriptions: some cognitive aspects of programming and similar activities. In *AVI '00: Proceedings of the working conference on Advanced visual interfaces*, pages 21–28, New York, NY, USA, 2000. ACM. ISBN 1-58113-252-2.

Bruria Haberman and Ben-David.Yifat. Kolikant. Activating "black boxes" instead of opening "zipper" - a method of teaching novices basic CS concepts. In *6th annual conference on innovation and technology in computer science education*, pages 41–44, Canterbury, 2001. ACM Press.

Thomas T Hewett. An undergraduate course in software psychology. *SIGCHI Bull.*, 18(3):43–49, 1987. ISSN 0736-6906.

J. Huoman. Predicting programming aptitude, 1986. Master's Thesis, Department of Computer Science, University of Joensuu, Finland.

P.N. Johnson-Laird. Comprehension as the construction of mental models. *Philosophical Transactions of the Royal Society*, 295:353–374, 1981.

P.N. Johnson-Laird. *Mental Models*. Cambridge University Press, Cambridge, 1983.

P.N. Johnson-Laird. Models of deduction. In R Falmagne, editor, *Reasoning: Representation and Process in Children and Adults*. Lawrence Erlbaum Associates, Springdale, NJ, 1975.

P.N. Johnson-Laird and V.A. Bell. A model theory of modal reasoning. In *Nineteenth Annual Conference of the Cognitive Science Society*, pages 349–353, 1997.

C. M Kessler and J. R Anderson. Learning flow of control: Recursive and iterative procedures. *Human-Computer Interaction*, 2:135–166, 1986.

David Kieras and Susan Bovair. The role of a mental model in learning to operate a device. *Cognitive Science*, 3(8):255 – 273, 1984.

William J. Klinger. Stanislavski and computer science. *SIGCSE Bull.*, 37(4): 111–114, 2005. ISSN 0097-8418.

Michael Kölling and John Rosenberg. Objects first with JAVA and BlueJ (seminar session). In *SIGCSE '00: Proceedings of the thirty-first SIGCSE technical symposium on Computer science education*, pages 429–429, New York, NY, USA, 2000. ACM. ISBN 1-58113-213-1.

Essi Lahtinen, Kirsti Ala-Mutka, and Hannu-Matti Järvinen. A study of the difficulties of novice programmers. *SIGCSE Bull.*, 37(3):14–18, 2005. ISSN 0097-8418.

Henry A. Landsberger. *Hawthorne Revisited*. Ithaca: Cornell University Press, 1958.

Tracy L. Lewis, J. D. Chase, Manuel A. Pérez-Quiñones, and Mary Beth Rosson. The effects of individual differences on CS2 course performance across universities. In *SIGCSE '05: Proceedings of the 36th SIGCSE technical symposium on Computer science education*, pages 426–430, New York, USA, 2005. ACM. ISBN 1-58113-997-7.

M Linn and J Dalbey. Cognitive consequences of programming instruction. *Educational Researcher*, 14(5):14–29, 1985.

Ray Lischner. Explorations: structured labs for first-time programmers. *SIGCSE Bull.*, 33(1):154–158, 2001. ISSN 0097-8418.

Raymond Lister, Elizabeth S. Adams, Sue Fitzgerald, William Fone, John Hamer, Morten Lindholm, Robert McCartney, Jan Erik Moström, Kate Sanders, Otto Seppälä, Beth Simon, and Lynda Thomas. A multi-national study of reading and tracing skills in novice programmers. In *ITiCSE-WGR '04: Working group reports from ITiCSE on Innovation and technology in computer science education*, pages 119–150, New York, NY, USA, 2004. ACM.

Antonio M. Lopez Jr. Supporting declarative programming through analogy. In *CCSC '01: Proceedings of the sixth annual CCSC northeastern conference. The journal of computing in small colleges*, pages 53–65, USA, 2001. Consortium for Computing Sciences in Colleges.

Murni Mahmud and Hastuti Kurniawan. Involving psychometric tests for input device evaluation with older people. In *OZCHI '05: Proceedings of the 19th conference of the computer-human interaction special interest group (CHISIG) of Australia on Computer-human interaction*, pages 1–10, Narrabundah, Australia, 2005. Computer-Human Interaction Special Interest Group (CHISIG) of Australia. ISBN 1-59593-222-4.

David B. Mayer. A computer manager speaks: "my problems are...". In *SIGCPR '64: Proceedings of the second SIGCPR conference on Computer personnel research*, pages 117–125, New York, NY, USA, 1964. ACM.

D.B. Mayer and A.W. Stalnaker. Selection and evaluation of computer personnel: the research history of SIG/CPR. In *Proc 1968 23rd ACM National Conference*, pages 657–670, Las Vegas, NV, USA, 1968.

Richard Mayer. *Thinking, Problem Solving, Cognition.* W. H. Freeman and Company, New York, 2nd edition, 1992. ISBN 0716722151.

Richard E. Mayer. The psychology of how novices learn computer programming. *ACM Comput. Surv.*, 13(1):121–141, 1981. ISSN 0360-0300.

L. P McCoy. Literature relating critical skills for problem solving in mathematics and computer programming. *School Science and Mathematics*, 90(1):48–60, 1990.

L. P. McCoy and J. K Burton. The relationship of computer programming and mathematics in secondary students. *Computers in the Schools*, 4(3/4):159–166, 1988.

Michael McCracken, Vicki Almstrum, Danny Diaz, Mark Guzdial, Dianne Hagan, Yifat Ben-David Kolikant, Cary Laxer, Lynda Thomas, Ian Utting, and Tadeusz Wilusz. A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. In *ITiCSE-WGR '01: Working*

group reports from ITiCSE on Innovation and technology in computer science education, pages 125–180, New York, NY, USA, 2001. ACM.

L McIver. *Syntactic and Semantic Issues in Introductory Programming Education.* PhD thesis, Monash University, 2001.

J Murnane. The psychology of computer languages for introductory programming courses. *New Ideas in Psychology*, 11(2):213–228, 1993.

Henry Neeman, Lloyd Lee, Julia Mullen, and Gerard Newman. Analogies for teaching parallel computing to inexperienced programmers. *SIGCSE Bull.*, 38 (4):64–67, 2006. ISSN 0097-8418.

Seymour Papert and Marvin Minsky. *Perceptrons.* MIT Press, Cambridge, Massachusetts, 1969. ISBN 0-262-63111-3.

G. Penney. Aptitude testing for employment in computer jobs. *Computers in education, IFIP*, pages 425–428, 1975.

N. Pennington. Stimulus structures and mental representations in expert comprehension of computer programs. *Cognitive psychology*, 19:295–341, 1987.

D. N. Perkins and R Simmons. Patterns for misunderstanding: An integrative model for science, math, and programming. *Review of Educational Research*, 58(3):303–326, 1988.

D.N Perkins, C Hancock, R Hobbs, F Martin, and R Simmons. Conditions of learning in novice programmers. *Journal of Educational Computing Research*, 2(1):37–55, 1986.

Ralph T Putnam, D Sleeman, Juliet A Baxter, and Laiani K Kuspa. A summary of misconceptions of high school BASIC programmers. *Journal of Educational Computing Research*, 2(4):459–472, 1986.

Alexander Quinn. An interrogative approach to novice programming. In *HCC '02: Proceedings of the IEEE 2002 Symposia on Human Centric Computing Languages and Environments (HCC'02)*, pages 83–85, Washington, DC, USA, 2002. IEEE Computer Society. ISBN 0-7695-1644-0.

M. Raadt, M. Hamilton, R. Lister, J. Tutty, B. Baker, I. Box, Q. Cutts, S. Fincher, J. Hamer, P. Haden, M. Petre, A. Robins, Simon, K. Sutton, and D. Tolhurst. Approaches to learning in computer programming students and their effect on success. *Higher Education in a changing world: Research and Development in Higher Education*, 28:407–414, 2005.

Haider Ramadhan. An intelligent discovery programming system. In *SAC '92: Proceedings of the 1992 ACM/SIGAPP Symposium on Applied computing*, pages 149–159, New York, NY, USA, 1992. ACM. ISBN 0-89791-502-X.

V. Ramalingam and S. Wiedenbeck. Development and validation of scores on a computer programming self efficacy scale and group analyses of novice programmer self-efficacy. *Journal of Educational Computing Research*, 19(4):367–381, 1998.

R. N. Reinstedt, Beulah C. Hammidi, Sherwood H. Peres, and Evelyn L. Ricard. *Programmer performance prediction study.* Computer personnel research group, Rand Corporation, Santa Monica, Calif., 1964.

L.J. Rips. *The Psychology of Proof.* MIT Press, Cambridge, 1994.

Nathan Rountree, Janet Rountree, Anthony Robins, and Robert Hannah. Interacting factors that predict success and failure in a CS1 course. *SIGCSE Bull.*, 36(4):101–104, 2004. ISSN 0097-8418.

R Samurcay. The concept of variable in programming: Its meaning and use in problem-solving by novice programmers. *Education Studies in Mathematics*, 16(2):143–161, 1985.

B. Shneidermann. *Software Psychology: Human Factors in computer and Information systems.* Winthrop, Cambridge, 1980.

B. Shneidermann. When children learn programming: Antecedents, concepts and outcomes. *The Computing Teacher*, 12(5):14–17, 1985.

Simon, Quintin Cutts, Sally Fincher, Patricia Haden, Anthony Robins, Ken Sutton, Bob Baker, Ilona Box, Michael de Raadt, John Hamer, Margaret Hamilton, Raymond Lister, Marian Petre, Denise Tolhurst, and Jodi Tutty. The ability to articulate strategy as a predictor of programming skill. In *ACE '06:*

Proceedings of the 8th Australasian Computing Education Conference, pages 181–188, Darlinghurst, Australia, 2006a. Australian Computer Society, Inc. ISBN 1-920682-34-1.

Simon, Sally Fincher, Anthony Robins, Bob Baker, Ilona Box, Quintin Cutts, Michael de Raadt, Patricia Haden, John Hamer, Margaret Hamilton, Raymond Lister, Marian Petre, Ken Sutton, Denise Tolhurst, and Jodi Tutty. Predictors of success in a first programming course. In *ACE '06: Proceedings of the 8th Australasian Computing Education Conference*, pages 189–196, Darlinghurst, Australia, 2006b. Australian Computer Society, Inc. ISBN 1-920682-34-1.

Elliot Soloway and James C. Spohrer. Some difficulties of learning to program. In *Studying the Novice Programmer*, pages 283–299. Lawrence Erlbaum Associates, 1989.

Elliot Soloway and James C. Spohrer. *Studying the Novice Programmer*. L. Erlbaum Associates Inc., Hillsdale, NJ, USA, 1988. ISBN 0805800026.

James C. Spohrer and Elliot Soloway. Novice mistakes: are the folk wisdoms correct? *Commun. ACM*, 29(7):624–632, 1986.

Constantin Stanislavski and Elizabeth Reynolds Hapgood. *An Actor Prepares*. Theatre Arts Books, New York, 38th edition, 1984. ISBN 0878309837.

Ken Sutton, Andrew Heathcote, and Miles Bore. Implementing a web-based measurement of 3d understanding. In *OZCHI '05: Proceedings of the 19th conference of the computer-human interaction special interest group (CHISIG) of Australia on Computer-human interaction*, pages 1–4, Narrabundah, Australia, 2005. Computer-Human Interaction Special Interest Group (CHISIG) of Australia. ISBN 1-59593-222-4.

Denise Tolhurst, Bob Baker, John Hamer, Ilona Box, Raymond Lister, Quintin Cutts, Marian Petre, Michael de Raadt, Anthony Robins, Sally Fincher, Simon, Patricia Haden, Ken Sutton, Margaret Hamilton, and Jodi Tutty. Do map drawing styles of novice programmers predict success in programming?: a multi-national, multi-institutional study. In *ACE '06: Proceedings of the 8th Australasian Computing Education Conference*, pages 213–222, Darlinghurst, Australia, 2006. Australian Computer Society, Inc. ISBN 1-920682-34-1.

Markku Tukiainen and Eero. Mönkkönen. Programming aptitude testing as a prediction of learning to program. In *PPIG '02: Proceedings of the 18th Annual Workshop of the Psychology of Programming Interest Group*, pages 45–57, London, England, 2002.

Maarten W. van Someren. What's wrong? understanding beginners' problems with Prolog. *Instructional Science*, 19(4/5):257–282, 1990.

Kam Hou Vat. Teaching software psychology: expanding the perspective. *SIGCSE Bull.*, 32(1):392–396, 2000. ISSN 0097-8418.

T. C. Willoughby. Review of Aptitude testing for employment in computer jobs by Penney, G. in Computers in education, O. Lecarme and R. Lewis (Eds.), IFIP, North-Holland/American Elsevier Publ. Co., New York, 1975, 425-428. *SIGCPR Comput. Pers.*, 7(4):15–15, 1978. ISSN 0160-2497. doi: http://doi.acm.org/10.1145/382072.1039657.

Brenda Cantwell Wilson and Sharon Shrock. Contributing to success in an introductory computer science course: a study of twelve factors. *SIGCSE Bull.*, 33 (1):184–188, 2001. ISSN 0097-8418.

B.J. Winer, Donald R. Brown, and Kenneth M. Michels. *Statistical principles in experimental design.* Mc.Graw-Hill, Series in Psychology, 1971. ISBN 0070709815.

Leon E. Winslow. Programming pedagogy: a psychological overview. *SIGCSE Bull.*, 28(3):17–22, 1996.

Stuart Wray. SQ minus EQ can predict programming aptitude. In *Proceedings of the PPIG 19th Annual Workshop*, Finland, 2007.

Appendix A

Questionnaire

Multiple-choice test for the Research Project:

Test created by Saeed Dehnadi & Prof Richard Bornat, School of Computing, Middlesex, University, UK

Researcher: .....

Date: .....

The following questionnaire will be recorded in a database. It will never be revealed to any person who could in any way identify you from the data given above. The questionnaire will never be used for assessment purposes.

I consent to the use of the following questionnaire for the research project conducted by .....

Please sign here: .....

Name	
Student No:	
Age	
Gender	**M** ☐ **F** ☐
A-Level or any equivalent subjects:	
Have you ever written a computer program in any language? **If so, in what language(s)?**	
Will this be your first course in programming? **If not, what other programming courses have you studied?**	

1. Read the following statements and tick the box next to the correct answer in the next column. int $a = 10$; int $b = 20$; $a = b$;	The new values of a and b: ☐ $a = 10$ $b = 10$ ☐ $a = 30$ $b = 20$ ☐ $a = 0$ $b = 10$ ☐ $a = 20$ $b = 20$ ☐ $a = 0$ $b = 30$ ☐ $a = 10$ $b = 20$ ☐ $a = 20$ $b = 10$ ☐ $a = 20$ $b = 0$ ☐ $a = 10$ $b = 30$ ☐ $a = 30$ $b = 0$ Any other values for a and b: $a =$ $b =$ $a =$ $b =$ $a =$ $b =$	Use this column for your rough notes please
2. Read the following statements and tick the box next to the correct answer in the next column. int $a = 10$; int $b = 20$; $b = a$;	The new values of a and b: ☐ $a = 0$ $b = 30$ ☐ $a = 30$ $b = 10$ ☐ $a = 0$ $b = 10$ ☐ $a = 20$ $b = 0$ ☐ $a = 20$ $b = 20$ ☐ $a = 20$ $b = 10$ ☐ $a = 30$ $b = 0$ ☐ $a = 10$ $b = 20$ ☐ $a = 10$ $b = 10$ ☐ $a = 10$ $b = 30$ Any other values for a and b: $a =$ $b =$ $a =$ $b =$ $a =$ $b =$	

3. Read the following statements and tick the box next to the correct answer in the next column.
 	
int big = 10;
int small = 20;
 	
big = small;
	The new values of big and small:
	☐ big=30 small=0
	☐ big=20 small=0
	☐ big=0 small=30
	☐ big=20 small=10
	☐ big=10 small=10
	☐ big=30 small=20
	☐ big=20 small=20
	☐ big=0 small=10
	☐ big=10 small=20
	☐ big=10 small=30
	
	Any other values for big and small:
	big= small=
	big= small=
	big= small=
		Use this column for your rough notes please

4. Read the following statements and tick the box next to the correct answer in the next column. int $a = 10$; int $b = 20$; $a = b$; $b = a$;	The new values of a and b: ☐ $a = 10$ $b = 0$ ☐ $a = 10$ $b = 10$ ☐ $a = 30$ $b = 50$ ☐ $a = 0$ $b = 20$ ☐ $a = 40$ $b = 30$ ☐ $a = 30$ $b = 0$ ☐ $a = 20$ $b = 20$ ☐ $a = 0$ $b = 30$ ☐ $a = 30$ $b = 30$ ☐ $a = 10$ $b = 20$ ☐ $a = 20$ $b = 10$ Any other values for a and b: $a =$ $b =$ $a =$ $b =$ $a =$ $b =$	

5. Read the following statements and tick the box next to the correct answer in the next column. int $a = 10$; int $b = 20$; $b = a$; $a = b$;	The new values of a and b: ☐ $a = 30$ $b = 50$ ☐ $a = 10$ $b = 10$ ☐ $a = 20$ $b = 20$ ☐ $a = 10$ $b = 0$ ☐ $a = 0$ $b = 20$ ☐ $a = 30$ $b = 0$ ☐ $a = 40$ $b = 30$ ☐ $a = 0$ $b = 30$ ☐ $a = 20$ $b = 10$ ☐ $a = 30$ $b = 30$ ☐ $a = 10$ $b = 20$ Any other values for a and b: $a =$ $b =$ $a =$ $b =$ $a =$ $b =$	Use this column for your rough notes please
6. Read the following statements and tick the box next to the correct answer in the next column. int $a = 10$; int $b = 20$; int $c = 30$; $a = b$; $b = c$;	The new values of a and b: ☐ $a = 30$ $b = 50$ $c = 30$ ☐ $a = 60$ $b = 0$ $c = 0$ ☐ $a = 10$ $b = 30$ $c = 40$ ☐ $a = 0$ $b = 10$ $c = 0$ ☐ $a = 10$ $b = 10$ $c = 10$ ☐ $a = 60$ $b = 20$ $c = 30$ ☐ $a = 30$ $b = 50$ $c = 0$ ☐ $a = 20$ $b = 30$ $c = 0$ ☐ $a = 10$ $b = 20$ $c = 30$ ☐ $a = 20$ $b = 20$ $c = 20$ ☐ $a = 0$ $b = 10$ $c = 20$ ☐ $a = 20$ $b = 30$ $c = 30$ ☐ $a = 10$ $b = 10$ $c = 20$ ☐ $a = 30$ $b = 30$ $c = 50$ ☐ $a = 0$ $b = 30$ $c = 50$ ☐ $a = 30$ $b = 30$ $c = 30$ ☐ $a = 0$ $b = 0$ $c = 60$ ☐ $a = 20$ $b = 30$ $c = 20$ Any other values for a and b: $a =$ $b =$ $a =$ $b =$ $a =$ $b =$	

7. Read the following statements and tick the box next to the correct answer in the next column. int $a = 5$; int $b = 3$; int $b = 7$; $a = c$; $b = a$; $c = b$;	The new values of a and b: ☐ $a = 3$ $b = 5$ $c = 5$ ☐ $a = 3$ $b = 3$ $c = 3$ ☐ $a = 12$ $b = 14$ $c = 22$ ☐ $a = 8$ $b = 15$ $c = 12$ ☐ $a = 7$ $b = 7$ $c = 7$ ☐ $a = 5$ $b = 3$ $c = 7$ ☐ $a = 5$ $b = 5$ $c = 5$ ☐ $a = 7$ $b = 5$ $c = 3$ ☐ $a = 3$ $b = 7$ $c = 5$ ☐ $a = 12$ $b = 8$ $c = 10$ ☐ $a = 10$ $b = 8$ $c = 12$ ☐ $a = 0$ $b = 0$ $c = 7$ ☐ $a = 0$ $b = 0$ $c = 15$ ☐ $a = 3$ $b = 12$ $c = 0$ ☐ $a = 3$ $b = 5$ $c = 7$ Any other values for a and b: $a =$ $b =$ $a =$ $b =$ $a =$ $b =$	Use this column for your rough notes please
8. Read the following statements and tick the box next to the correct answer in the next column. int $a = 5$; int $b = 3$; int $b = 7$; $a = c$; $b = a$; $c = b$;	The new values of a and b: ☐ $a = 3$ $b = 5$ $c = 7$ ☐ $a = 15$ $b = 10$ $c = 22$ ☐ $a = 12$ $b = 8$ $c = 22$ ☐ $a = 7$ $b = 7$ $c = 7$ ☐ $a = 3$ $b = 5$ $c = 3$ ☐ $a = 0$ $b = 0$ $c = 7$ ☐ $a = 5$ $b = 3$ $c = 7$ ☐ $a = 3$ $b = 3$ $c = 3$ ☐ $a = 7$ $b = 5$ $c = 3$ ☐ $a = 3$ $b = 5$ $c = 0$ ☐ $a = 3$ $b = 7$ $c = 5$ ☐ $a = 8$ $b = 10$ $c = 12$ ☐ $a = 5$ $b = 5$ $c = 5$ ☐ $a = 15$ $b = 8$ $c = 10$ ☐ $a = 10$ $b = 5$ $c = 0$ ☐ $a = 0$ $b = 0$ $c = 15$ Any other values for a and b: $a =$ $b =$ $a =$ $b =$ $a =$ $b =$	Use this column for your rough notes please

9. Read the following statements and tick the box next to the correct answer in the next column. int $a = 5$; int $b = 3$; int $b = 7$; $c = b$; $a = c$; $b = a$;	The new values of a and b: ☐ $a = 15$ $b = 18$ $c = 10$ ☐ $a = 7$ $b = 5$ $c = 3$ ☐ $a = 7$ $b = 0$ $c = 5$ ☐ $a = 0$ $b = 3$ $c = 0$ ☐ $a = 10$ $b = 0$ $c = 5$ ☐ $a = 5$ $b = 3$ $c = 7$ ☐ $a = 3$ $b = 3$ $c = 3$ ☐ $a = 12$ $b = 8$ $c = 10$ ☐ $a = 7$ $b = 7$ $c = 7$ ☐ $a = 15$ $b = 10$ $c = 12$ ☐ $a = 7$ $b = 7$ $c = 5$ ☐ $a = 8$ $b = 10$ $c = 12$ ☐ $a = 0$ $b = 15$ $c = 0$ ☐ $a = 7$ $b = 3$ $c = 5$ ☐ $a = 5$ $b = 5$ $c = 5$ ☐ $a = 3$ $b = 7$ $c = 5$ Any other values for a and b: $a =$ $b =$ $a =$ $b =$ $a =$ $b =$	Use this column for your rough notes please
10. Read the following statements and tick the box next to the correct answer in the next column. int $a = 5$; int $b = 3$; int $b = 7$; $b = a$; $c = b$; $a = c$;	The new values of a and b: ☐ $a = 0$ $b = 7$ $c = 3$ ☐ $a = 12$ $b = 8$ $c = 10$ ☐ $a = 15$ $b = 0$ $c = 0$ ☐ $a = 0$ $b = 7$ $c = 8$ ☐ $a = 3$ $b = 7$ $c = 3$ ☐ $a = 5$ $b = 3$ $c = 7$ ☐ $a = 3$ $b = 3$ $c = 3$ ☐ $a = 7$ $b = 5$ $c = 3$ ☐ $a = 20$ $b = 8$ $c = 15$ ☐ $a = 3$ $b = 7$ $c = 5$ ☐ $a = 5$ $b = 0$ $c = 0$ ☐ $a = 8$ $b = 10$ $c = 15$ ☐ $a = 5$ $b = 5$ $c = 5$ ☐ $a = 8$ $b = 10$ $c = 12$ ☐ $a = 5$ $b = 7$ $c = 3$ ☐ $a = 7$ $b = 7$ $c = 7$ Any other values for a and b: $a =$ $b =$ $a =$ $b =$ $a =$ $b =$	

11. Read the following statements and tick the box next to the correct answer in the next column. int $a = 5$; int $b = 3$; int $b = 7$; $b = a$; $a = c$; $c = b$;	The new values of a and b: ☐ $a = 8$ $b = 18$ $c = 15$ ☐ $a = 7$ $b = 0$ $c = 8$ ☐ $a = 5$ $b = 5$ $c = 5$ ☐ $a = 12$ $b = 8$ $c = 15$ ☐ $a = 7$ $b = 0$ $c = 5$ ☐ $a = 3$ $b = 7$ $c = 5$ ☐ $a = 7$ $b = 5$ $c = 3$ ☐ $a = 0$ $b = 15$ $c = 0$ ☐ $a = 0$ $b = 3$ $c = 0$ ☐ $a = 3$ $b = 3$ $c = 3$ ☐ $a = 7$ $b = 7$ $c = 7$ ☐ $a = 12$ $b = 8$ $c = 10$ ☐ $a = 8$ $b = 10$ $c = 12$ ☐ $a = 7$ $b = 5$ $c = 5$ ☐ $a = 5$ $b = 3$ $c = 7$ ☐ $a = 7$ $b = 3$ $c = 5$ Any other values for a and b: $a =$ $b =$ $a =$ $b =$ $a =$ $b =$	Use this column for your rough notes please
12. Read the following statements and tick the box next to the correct answer in the next column. int $a = 5$; int $b = 3$; int $b = 7$; $a = c$; $c = b$; $b = a$;	The new values of a and b: ☐ $a = 0$ $b = 12$ $c = 3$ ☐ $a = 5$ $b = 5$ $c = 5$ ☐ $a = 0$ $b = 7$ $c = 3$ ☐ $a = 8$ $b = 10$ $c = 12$ ☐ $a = 15$ $b = 0$ $c = 0$ ☐ $a = 3$ $b = 7$ $c = 5$ ☐ $a = 12$ $b = 15$ $c = 10$ ☐ $a = 5$ $b = 7$ $c = 3$ ☐ $a = 3$ $b = 3$ $c = 3$ ☐ $a = 7$ $b = 7$ $c = 7$ ☐ $a = 12$ $b = 8$ $c = 10$ ☐ $a = 5$ $b = 0$ $c = 0$ ☐ $a = 5$ $b = 3$ $c = 7$ ☐ $a = 7$ $b = 7$ $c = 3$ ☐ $a = 20$ $b = 15$ $c = 12$ ☐ $a = 7$ $b = 5$ $c = 3$ Any other values for a and b: $a =$ $b =$ $a =$ $b =$ $a =$ $b =$	

Appendix B

Answer sheet

Question	**Answers/s**		**Model/s**
	$a = 20$	$b = 0$	M1
1.	$a = 20$	$b = 20$	M2
int $a = 10$;	$a = 0$	$b = 10$	M3
int $b = 20$;	$a = 10$	$b = 10$	M4
	$a = 30$	$b = 20$	M5
$a = b$;	$a = 30$	$b = 0$	M6
	$a = 10$	$b = 30$	M7
	$a = 0$	$b = 30$	M8
	$a = 10$	$b = 20$	M9
	$a = 20$	$b = 10$	M11
	$a = 20$	$b = 20$	M10
	$a = 10$	$b = 10$	
	$a = 0$	$b = 10$	M1
2.	$a = 10$	$b = 10$	M2
int $a = 10$;	$a = 20$	$b = 0$	M3
int $b = 20$;	$a = 20$	$b = 20$	M4
	$a = 10$	$b = 30$	M5
$b = a$;	$a = 0$	$b = 30$	M6
	$a = 30$	$b = 20$	M7
	$a = 30$	$b = 0$	M8
	$a = 10$	$b = 20$	M9
	$a = 20$	$b = 10$	M11
	$a = 20$	$b = 20$	M10
	$a = 10$	$b = 10$	

Question	Answers/s		Model/s

3.
int big = 10;
int small = 20;
 	
big = small;
	big=20	small=0	M1

	big=20	small=20	M2
	big=0	small=10	M3
	big=10	small=10	M4
	big=30	small=20	M5
	big=30	small=0	M6
	big=10	small=30	M7
	big=0	small=30	M8
	big=10	small=20	M9
	big=20	small=10	M11
	big=20 big=10	small=20 small=10	M10
4. int $a = 10$; int $b = 20$; $a = b$; $b = a$;	$a = 0$	$b = 20$	M1+S1
	$a = 20$	$b = 10$	(M1+S3)/(M2+S3)/(M3+S3)/ (M4+S3)/ (M11+S3)
	$a = 30$ $a = 20$	$b = 30$ $b = 20$	(M5+S3)/(M6+S3)/(M7+S3)/ M2+S1
	$a = 10$	$b = 0$	M3+S1
	$a = 10$	$b = 10$	M4+S1
	$a = 30$	$b = 50$	M5+S1
	$a = 0$	$b = 30$	M6+S1
	$a = 40$	$b = 30$	M7+S1
	$a = 30$	$b = 0$	M8+S1
	$a = 10$	$b = 20$	(M9+S1)/(M11+S1)
			(M8+S3)
	$a = 20$ $a = 10$	$b = 20$ $b = 10$	(M10+S1)/(M2+S2)/(M4+S2)
	$a = 20$ $a = 0$	$b = 0$ $b = 10$	(M1+S2)/(M3+S2)
	$a = 30$ $a = 0$	$b = 0$ $b = 30$	(M5+S2)/(M7+S2)
	$a = 30$ $a = 10$	$b = 20$ $b = 30$	(M6+S2)/(M8+S2)
	$a = 20$ $a = 20$	$b = 10$ $b = 10$	(M11+S2)

Question	**Answers/s**		**Model/s**
	$a = 10$	$b = 0$	M1+S1
	$a = 20$	$b = 10$	(M1+S3)/(M2+S3)/(M3+S3)/
5.			(M4+S3)
	$a = 10$	$b = 10$	M2+S1
int $a = 10$;	$a = 0$	$b = 20$	M3+S1
int $b = 20$;	$a = 20$	$b = 20$	M4+S1
	$a = 40$	$b = 30$	M5+S1
$a = b$;	$a = 30$	$b = 30$	(M5+S3)/(M6+S3)/(M7+S3)/
$b = a$;			(M8+S3)
	$a = 30$	$b = 0$	M6+S1
	$a = 30$	$b = 50$	M7+S1
	$a = 0$	$b = 30$	M8+S1
	$a = 10$	$b = 20$	(M9+S1)/(M11+S1)/
			(M11+S3)
	$a = 20$	$b = 20$	(M10+S1)/(M2+S2)/(M4+S2)
	$a = 10$	$b = 10$	
	$a = 0$	$b = 10$	(M1+S2)/M3+S2)
	$a = 20$	$b = 0$	
	$a = 30$	$b = 20$	(M5+S2)/(M7+S2)
	$a = 10$	$b = 30$	
	$a = 0$	$b = 30$	(M6+S2)/(M8+S2)
	$a = 30$	$b = 0$	
	$a = 10$	$b = 20$	(M11+S2)
	$a = 10$	$b = 20$	

Question	**Answers/s**			**Model/s**
	$a = 20$	$b = 30$	$c = 0$	M1+S1
6.	$a = 20$	$b = 30$	$c = 30$	(M2+S1)/(M2+S3)/(M1+S3)
	$a = 0$	$b = 0$	$c = 10$	M3+S1
int $a = 10$;	$a = 10$	$b = 10$	$c = 10$	M4+S1
int $b = 20$;	$a = 10$	$b = 10$	$c = 20$	(M3+S3)/(M4+S3)/(M6+S3)
int $c = 30$;	$a = 30$	$b = 50$	$c = 30$	(M5+S1)/(M5+S3)
	$a = 30$	$b = 30$	$c = 0$	M6+S1
$a = b$;	$a = 10$	$b = 30$	$c = 60$	M7+S1
$b = c$;	$a = 0$	$b = 0$	$c = 60$	M8+S1
	$a = 10$	$b = 30$	$c = 50$	(M7+S3)/(M8+S3)
	$a = 10$	$b = 20$	$c = 30$	M9+S1
	$a = 20$	$b = 30$	$c = 10$	(M11+S1)/(M11+S3)
	$a = 10$	$b = 10$	$c = 10$	M10+S1
	$a = 20$	$b = 20$	$c = 20$	
	$a = 30$	$b = 30$	$c = 30$	
	$a = 20$	$b = 0$	$c = 30$	(M1+S2)
	$a = 10$	$b = 30$	$c = 0$	
	$a = 20$	$b = 20$	$c = 30$	(M2+S2)
	$a = 10$	$b = 30$	$c = 30$	
	$a = 0$	$b = 10$	$c = 30$	(M3+S2)
	$a = 10$	$b = 0$	$c = 20$	
	$a = 10$	$b = 10$	$c = 30$	(M4+S2)
	$a = 10$	$b = 20$	$c = 20$	
	$a = 30$	$b = 20$	$c = 30$	(M5+S2)
	$a = 10$	$b = 50$	$c = 30$	
	$a = 30$	$b = 0$	$c = 30$	(M6+S2)
	$a = 10$	$b = 50$	$c = 0$	
	$a = 10$	$b = 30$	$c = 30$	(M7+S2)
	$a = 10$	$b = 20$	$c = 50$	
	$a = 0$	$b = 30$	$c = 30$	(M8+S2)
	$a = 10$	$b = 0$	$c = 50$	
	$a = 20$	$b = 10$	$c = 30$	(M11+S2)
	$a = 10$	$b = 30$	$c = 20$	

Question	**Answers/s**			**Model/s**
	$a = 0$	$b = 0$	$c = 7$	M1
	$a = 7$	$b = 5$	$c = 3$	(M1+S3)/(M2+S3)/(M11+S3)
7.	$a = 7$	$b = 7$	$c = 7$	M2
	$a = 3$	$b = 5$	$c = 0$	M3
int $a = 5$;	$a = 3$	$b = 5$	$c = 5$	M4
int $b = 3$;	$a = 3$	$b = 7$	$c = 5$	(M3+S3)/(M4+S3)
int $c = 7$;	$a = 12$	$b = 15$	$c = 22$	M5
	$a = 12$	$b = 8$	$c = 10$	(M5+S3)/(M6+S3)
$a = c$;	$a = 0$	$b = 0$	$c = 15$	M6
$b = a$;	$a = 8$	$b = 15$	$c = 12$	M7
$c = b$;	$a = 8$	$b = 10$	$c = 12$	(M7+S3)/(M8+S3)
	$a = 3$	$b = 12$	$c = 0$	M8
	$a = 5$	$b = 3$	$c = 7$	M9
	$a = 3$	$b = 5$	$c = 7$	M11
	$a = 5$	$b = 5$	$c = 5$	M10
	$a = 3$	$b = 3$	$c = 3$	
	$a = 7$	$b = 7$	$c = 7$	
	$a = 7$	$b = 3$	$c = 0$	(M1+S2)
	$a = 0$	$b = 5$	$c = 7$	
	$a = 5$	$b = 0$	$c = 3$	
	$a = 7$	$b = 3$	$c = 7$	(M2+S2)
	$a = 5$	$b = 5$	$c = 7$	
	$a = 5$	$b = 3$	$c = 3$	
	$a = 0$	$b = 3$	$c = 5$	(M3+S2)
	$a = 3$	$b = 0$	$c = 7$	
	$a = 5$	$b = 7$	$c = 0$	
	$a = 5$	$b = 3$	$c = 5$	(M4+S2)
	$a = 3$	$b = 3$	$c = 7$	
	$a = 5$	$b = 7$	$c = 7$	
	$a = 12$	$b = 3$	$c = 7$	(M5+S2)
	$a = 5$	$b = 8$	$c = 7$	
	$a = 5$	$b = 3$	$c = 10$	
	$a = 12$	$b = 3$	$c = 0$	(M6+S2)
	$a = 0$	$b = 8$	$c = 7$	
	$a = 5$	$b = 0$	$c = 10$	
	$a = 5$	$b = 3$	$c = 12$	(M7+S2)
	$a = 8$	$b = 3$	$c = 7$	
	$a = 5$	$b = 10$	$c = 7$	
	$a = 0$	$b = 3$	$c = 12$	(M8+S2)
	$a = 8$	$b = 0$	$c = 7$	
	$a = 5$	$b = 10$	$c = 0$	
	$a = 7$	$b = 3$	$c = 5$	(M11+S2)
	$a = 3$	$b = 5$	$c = 7$	
	$a = 5$	$b = 7$	$c = 3$	

Question	Answers/s			Model/s
	$a = 3$	$b = 5$	$c = 0$	M1
	$a = 7$	$b = 5$	$c = 3$	(M1+S3)/(M2+S3)/(M11+S3)
8.	$a = 3$	$b = 5$	$c = 3$	M2
	$a = 0$	$b = 0$	$c = 7$	M3
int $a = 5$;	$a = 7$	$b = 7$	$c = 7$	M4
int $b = 3$;	$a = 3$	$b = 7$	$c = 5$	(M3+S3)/(M4+S3)
int $c = 7$;	$a = 15$	$b = 8$	$c = 10$	M5
	$a = 12$	$b = 8$	$c = 10$	(M5+S3)/(M6+S3)
$c = b$;	$a = 10$	$b = 5$	$c = 0$	M6
$b = a$;	$a = 15$	$b = 10$	$c = 22$	M7
$a = c$;	$a = 8$	$b = 10$	$c = 12$	(M7+S3)/(M8+S3)
	$a = 0$	$b = 0$	$c = 15$	M8
	$a = 5$	$b = 3$	$c = 7$	M9
	$a = 3$	$b = 5$	$c = 7$	M11
	$a = 5$	$b = 5$	$c = 5$	M10
	$a = 3$	$b = 3$	$c = 3$	
	$a = 7$	$b = 7$	$c = 7$	
	$a = 5$	$b = 0$	$c = 3$	(M1+S2)
	$a = 0$	$b = 5$	$c = 7$	
	$a = 7$	$b = 3$	$c = 0$	
	$a = 5$	$b = 3$	$c = 3$	(M2+S2)
	$a = 5$	$b = 5$	$c = 7$	
	$a = 7$	$b = 3$	$c = 7$	
	$a = 5$	$b = 7$	$c = 0$	(M3+S2)
	$a = 3$	$b = 0$	$c = 7$	
	$a = 0$	$b = 3$	$c = 5$	
	$a = 5$	$b = 7$	$c = 7$	(M4+S2)
	$a = 3$	$b = 3$	$c = 7$	
	$a = 5$	$b = 3$	$c = 5$	
	$a = 5$	$b = 3$	$c = 10$	(M5+S2)
	$a = 5$	$b = 8$	$c = 7$	
	$a = 12$	$b = 3$	$c = 7$	
	$a = 5$	$b = 0$	$c = 10$	(M6+S2)
	$a = 0$	$b = 8$	$c = 7$	
	$a = 12$	$b = 3$	$c = 0$	
	$a = 5$	$b = 10$	$c = 7$	(M7+S2)
	$a = 8$	$b = 3$	$c = 7$	
	$a = 5$	$b = 3$	$c = 12$	
	$a = 5$	$b = 10$	$c = 0$	(M8+S2)
	$a = 8$	$b = 0$	$c = 7$	
	$a = 0$	$b = 3$	$c = 12$	
	$a = 5$	$b = 7$	$c = 3$	(M11+S2)
	$a = 3$	$b = 5$	$c = 7$	
	$a = 7$	$b = 3$	$c = 5$	

Question	Answers/s			Model/s
	$a = 0$	$b = 3$	$c = 5$	M1
	$a = 7$	$b = 5$	$c = 3$	(M1+S3)/(M2+S3)/(M11+S3)
9.	$a = 3$	$b = 3$	$c = 3$	M2
	$a = 7$	$b = 0$	$c = 5$	M3
int $a = 5$;	$a = 7$	$b = 7$	$c = 5$	M4
int $b = 3$;	$a = 3$	$b = 7$	$c = 5$	(M3+S3)/(M4+S3)
int $c = 7$;	$a = 15$	$b = 18$	$c = 10$	M5
	$a = 12$	$b = 8$	$c = 10$	(M5+S3)/(M6+S3)
$c = b$;	$a = 0$	$b = 15$	$c = 0$	M6
$a = c$;	$a = 15$	$b = 10$	$c = 12$	M7
$b = a$;	$a = 8$	$b = 10$	$c = 12$	(M7+S3)/(M8+S3)
	$a = 10$	$b = 0$	$c = 5$	M8
	$a = 5$	$b = 3$	$c = 7$	M9
	$a = 7$	$b = 3$	$c = 5$	M11
	$a = 5$	$b = 5$	$c = 5$	M10
	$a = 3$	$b = 3$	$c = 3$	
	$a = 7$	$b = 7$	$c = 7$	
	$a = 5$	$b = 0$	$c = 3$	(M1+S2)
	$a = 7$	$b = 3$	$c = 0$	
	$a = 0$	$b = 5$	$c = 7$	
	$a = 5$	$b = 3$	$c = 3$	(M2+S2)
	$a = 7$	$b = 3$	$c = 7$	
	$a = 5$	$b = 5$	$c = 7$	
	$a = 5$	$b = 7$	$c = 0$	(M3+S2)
	$a = 0$	$b = 3$	$c = 5$	
	$a = 3$	$b = 0$	$c = 7$	
	$a = 5$	$b = 7$	$c = 7$	(M4+S2)
	$a = 5$	$b = 3$	$c = 5$	
	$a = 3$	$b = 3$	$c = 7$	
	$a = 5$	$b = 3$	$c = 10$	(M5+S2)
	$a = 12$	$b = 3$	$c = 7$	
	$a = 5$	$b = 8$	$c = 7$	
	$a = 5$	$b = 0$	$c = 10$	(M6+S2)
	$a = 12$	$b = 3$	$c = 0$	
	$a = 0$	$b = 8$	$c = 7$	
	$a = 5$	$b = 10$	$c = 7$	(M7+S2)
	$a = 5$	$b = 3$	$c = 12$	
	$a = 8$	$b = 3$	$c = 7$	
	$a = 5$	$b = 10$	$c = 0$	(M8+S2)
	$a = 0$	$b = 3$	$c = 12$	
	$a = 8$	$b = 0$	$c = 7$	
	$a = 5$	$b = 7$	$c = 3$	(M11+S2)
	$a = 7$	$b = 3$	$c = 5$	
	$a = 3$	$b = 5$	$c = 7$	

Question	Answers/s			Model/s
	$a = 5$	$b = 0$	$c = 0$	M1
	$a = 7$	$b = 5$	$c = 3$	(M1+S3)/(M2+S3)/(M11+S3)
10.	$a = 5$	$b = 5$	$c = 5$	M2
	$a = 0$	$b = 7$	$c = 3$	M3
int $a = 5$;	$a = 3$	$b = 7$	$c = 3$	M4
int $b = 3$;	$a = 3$	$b = 7$	$c = 5$	(M3+S3)/(M4+S3)
int $c = 7$;	$a = 20$	$b = 8$	$c = 15$	M5
	$a = 12$	$b = 8$	$c = 10$	(M5+S3)/(M6+S3)
$b = a$;	$a = 15$	$b = 0$	$c = 0$	M6
$c = b$;	$a = 8$	$b = 10$	$c = 15$	M7
$a = c$;	$a = 8$	$b = 10$	$c = 12$	(M7+S3)/(M8+S3)
	$a = 0$	$b = 7$	$c = 8$	M8
	$a = 5$	$b = 3$	$c = 7$	M9
	$a = 5$	$b = 7$	$c = 3$	M11
	$a = 5$	$b = 5$	$c = 5$	M10
	$a = 3$	$b = 3$	$c = 3$	
	$a = 7$	$b = 7$	$c = 7$	
	$a = 0$	$b = 5$	$c = 7$	(M1+S2)
	$a = 5$	$b = 0$	$c = 3$	
	$a = 7$	$b = 3$	$c = 0$	
	$a = 5$	$b = 3$	$c = 3$	(M2+S2)
	$a = 7$	$b = 3$	$c = 7$	
	$a = 5$	$b = 5$	$c = 7$	
	$a = 5$	$b = 7$	$c = 0$	(M3+S2)
	$a = 0$	$b = 3$	$c = 5$	
	$a = 3$	$b = 0$	$c = 7$	
	$a = 5$	$b = 7$	$c = 7$	(M4+S2)
	$a = 5$	$b = 3$	$c = 5$	
	$a = 3$	$b = 3$	$c = 7$	
	$a = 5$	$b = 3$	$c = 10$	(M5+S2)
	$a = 12$	$b = 3$	$c = 7$	
	$a = 5$	$b = 8$	$c = 7$	
	$a = 5$	$b = 0$	$c = 10$	(M6+S2)
	$a = 12$	$b = 3$	$c = 0$	
	$a = 0$	$b = 8$	$c = 7$	
	$a = 5$	$b = 10$	$c = 7$	(M7+S2)
	$a = 5$	$b = 3$	$c = 12$	
	$a = 8$	$b = 3$	$c = 7$	
	$a = 5$	$b = 10$	$c = 0$	(M8+S2)
	$a = 0$	$b = 3$	$c = 12$	
	$a = 8$	$b = 0$	$c = 7$	
	$a = 5$	$b = 7$	$c = 3$	(M11+S2)
	$a = 7$	$b = 3$	$c = 5$	
	$a = 3$	$b = 5$	$c = 7$	

Question	**Answers/s**			**Model/s**
	$a = 7$	$b = 0$	$c = 5$	M1
	$a = 7$	$b = 5$	$c = 3$	(M1+S3)/(M2+S3)/(M11+S3)
11.	$a = 7$	$b = 5$	$c = 5$	M2
	$a = 0$	$b = 3$	$c = 0$	M3
int $a = 5$;	$a = 3$	$b = 3$	$c = 3$	M4
int $b = 3$;	$a = 3$	$b = 7$	$c = 5$	(M3+S3)/(M4+S3)
int $c = 7$;	$a = 12$	$b = 8$	$c = 15$	M5
	$a = 12$	$b = 8$	$c = 10$	(M5+S3)/(M6+S3)
$b = a$;	$a = 7$	$b = 0$	$c = 8$	M6
$a = c$;	$a = 8$	$b = 18$	$c = 15$	M7
$c = b$;	$a = 8$	$b = 10$	$c = 12$	(M7+S3)/(M8+S3)
	$a = 0$	$b = 15$	$c = 0$	M8
	$a = 5$	$b = 3$	$c = 7$	M9
	$a = 7$	$b = 3$	$c = 5$	M11
	$a = 5$	$b = 5$	$c = 5$	M10
	$a = 3$	$b = 3$	$c = 3$	
	$a = 7$	$b = 7$	$c = 7$	
	$a = 0$	$b = 5$	$c = 7$	(M1+S2)
	$a = 5$	$b = 0$	$c = 3$	
	$a = 7$	$b = 3$	$c = 0$	
	$a = 5$	$b = 3$	$c = 3$	(M2+S2)
	$a = 7$	$b = 3$	$c = 7$	
	$a = 5$	$b = 5$	$c = 7$	
	$a = 5$	$b = 7$	$c = 0$	(M3+S2)
	$a = 0$	$b = 3$	$c = 5$	
	$a = 3$	$b = 0$	$c = 7$	
	$a = 5$	$b = 7$	$c = 7$	(M4+S2)
	$a = 5$	$b = 3$	$c = 5$	
	$a = 3$	$b = 3$	$c = 7$	
	$a = 5$	$b = 3$	$c = 10$	(M5+S2)
	$a = 12$	$b = 3$	$c = 7$	
	$a = 5$	$b = 8$	$c = 7$	
	$a = 5$	$b = 0$	$c = 10$	(M6+S2)
	$a = 12$	$b = 3$	$c = 0$	
	$a = 0$	$b = 8$	$c = 7$	
	$a = 5$	$b = 10$	$c = 7$	(M7+S2)
	$a = 5$	$b = 3$	$c = 12$	
	$a = 8$	$b = 3$	$c = 7$	
	$a = 5$	$b = 10$	$c = 0$	(M8+S2)
	$a = 0$	$b = 3$	$c = 12$	
	$a = 8$	$b = 0$	$c = 7$	
	$a = 5$	$b = 7$	$c = 3$	(M11+S2)
	$a = 7$	$b = 3$	$c = 5$	
	$a = 3$	$b = 5$	$c = 7$	

Question	**Answers/s**			**Model/s**
	$a = 0$	$b = 7$	$c = 3$	M1
	$a = 7$	$b = 5$	$c = 3$	(M1+S3)/(M2+S3)/(M11+S3)
12.	$a = 7$	$b = 7$	$c = 3$	M2
	$a = 5$	$b = 0$	$c = 0$	M3
int $a = 5$;	$a = 5$	$b = 5$	$c = 5$	M4
int $b = 3$;	$a = 3$	$b = 7$	$c = 5$	(M3+S3)/(M4+S3)
int $c = 7$;	$a = 12$	$b = 15$	$c = 10$	M5
	$a = 12$	$b = 8$	$c = 10$	(M5+S3)/(M6+S3)
$a = c$;	$a = 0$	$b = 12$	$c = 3$	M6
$c = b$;	$a = 20$	$b = 15$	$c = 12$	M7
$b = a$;	$a = 8$	$b = 10$	$c = 12$	(M7+S3)/(M8+S3)
	$a = 15$	$b = 0$	$c = 0$	M8
	$a = 5$	$b = 3$	$c = 7$	M9
	$a = 5$	$b = 7$	$c = 3$	M11
	$a = 5$	$b = 5$	$c = 5$	M10
	$a = 3$	$b = 3$	$c = 3$	
	$a = 7$	$b = 7$	$c = 7$	
	$a = 0$	$b = 5$	$c = 7$	(M1+S2)
	$a = 5$	$b = 0$	$c = 3$	
	$a = 7$	$b = 3$	$c = 0$	
	$a = 5$	$b = 3$	$c = 3$	(M2+S2)
	$a = 7$	$b = 3$	$c = 7$	
	$a = 5$	$b = 5$	$c = 7$	
	$a = 5$	$b = 7$	$c = 0$	(M3+S2)
	$a = 0$	$b = 3$	$c = 5$	
	$a = 3$	$b = 0$	$c = 7$	
	$a = 5$	$b = 7$	$c = 7$	(M4+S2)
	$a = 5$	$b = 3$	$c = 5$	
	$a = 3$	$b = 3$	$c = 7$	
	$a = 5$	$b = 3$	$c = 10$	(M5+S2)
	$a = 12$	$b = 3$	$c = 7$	
	$a = 5$	$b = 8$	$c = 7$	
	$a = 5$	$b = 0$	$c = 10$	(M6+S2)
	$a = 12$	$b = 3$	$c = 0$	
	$a = 0$	$b = 8$	$c = 7$	
	$a = 5$	$b = 10$	$c = 7$	(M7+S2)
	$a = 5$	$b = 3$	$c = 12$	
	$a = 8$	$b = 3$	$c = 7$	
	$a = 5$	$b = 10$	$c = 0$	(M8+S2)
	$a = 0$	$b = 3$	$c = 12$	
	$a = 8$	$b = 0$	$c = 7$	
	$a = 5$	$b = 7$	$c = 3$	(M11+S2)
	$a = 7$	$b = 3$	$c = 5$	
	$a = 3$	$b = 5$	$c = 7$	

Appendix C

Marksheet

Participant code	Age	Sex	Time to do test	Prior programming	A-level's	Prior programming courses	Course result

Questions	Assignment								No effect	Equal sign	Swap values	Remarks (including participants' working notes)
	Assign-to-left		Assign-to-right		Add-Assign-to-left		Add-Assign-to-right					
	Lose-value (M1) /S1..3	Keep-value (M2) /S1..3	Lose-value (M3) /S1..3	Keep-value (M4) /S1..3	Keep-value (M5) /S1..3	Lose-value (M6) /S1..3	Keep-value (M7) /S1..3	Lose-value (M8) /S1..3	Values don't change (M9)	Assign means equal (M10)	Swap values (M11) /S1..3	
1												
2												
3												
4												
5												
6												
7												
8												
9												
10												
11												
12												
C0												
C1												
C2												
C3												

Figure C.1: A marksheet

Appendix D

Marking protocol

The instruction below was used in the six experiments of this study:

In the answer sheet for Q1-Q3 (single assignment questions) there are ten single-tick boxes (M1 to M11) and one double-tick box (M10). If the subject gives one tick, we use a single-tick box. If they give two ticks in the positions specified, we use the double-tick box. We can't interpret anything else.

In multiple assignments (Q4 onwards) there is more complexity. First, some of the models are decorated with S1, S2 or S3. Instead of just ticking the corresponding model column on the mark sheet, put the S1, S2 or S3 next to the tick.

Second, some of the single-tick boxes give alternative models. In this case tick all of the alternative models on the mark sheet, in pencil. Then, when you have marked all the questions, try to maximise the coherence of the subject's answers by inking in on of the pencil ticks on each row, so as to maximise the numbers in the summary row (labelled C0 on the mark sheet).

Subjective marking is needed to decide what to do with not-entirely- blank scripts. At present we use the following rule:

Rule 1: A consistent response to Q1- Q3 (all the ticks in a single column or in two adjacent columns) can be considered non-blank, but if all we get is three ticks all over the place and nothing else, it's blank. If we could get consistent responses to all the double-assignments or the triple-assignments, then that was non-blank too.

Using joined columns; we can investigate four different levels of consistency in the rows that represent by labels C0, C1, C2 and C3. Level C0 contents of

the 11 single models and demonstrates the highest rate of consistency while sliding toward level C3 leads to lower rate and poorer sign of consistency.

Level C1 contents of 4 columns that each is created by joining two adjacent models, logically carried common concepts. M1 and M2, M3 and M4, M5 and M6, M7 and M8. Each of these new columns logically approved Assignment, assigning value to the left or to the right. Level C2 contents 2 columns that each is created by joining 4 adjacent models, logically carried common concepts. M1 and M2 and M3 and M4, M5 and M6 and M7 and M8. Each of these new columns logically approved Assignment, assigning value to the left and to the right. Level C3 contents of a single column that created by joining 8 other models, logically carried common concepts. M1 and M2 and M3 and M4 and M5 and M6 and M7 and M8. The new column logically approved assignment.

Rule 2: Any C level can be considered as subject's level of consistency if:

1. (mode value in C level) >= *abs* (no. of answered questions * 80%) and (no. of answered questions) >= *abs* (no. of questions * 80%).
2. According to the above rule the subject in figure E.2 is consistent in C3 level. This method creates around 20% flexibility in C level of subject's that answered 80% of the questionnaire.

Appendix E

Samples

3. Read the following statements and tick the box next to the correct answer in the next column.
 	
`int big = 10;`
`int small = 20;`
 	
`big = small;`
	The new values of big and small are:
	
	☐ big = 30 small = 0
	☐ big = 20 small = 0
	☐ big = 0 small = 30
	☐ big = 20 small = 10
	☐ big = 10 small = 10
	☐ big = 30 small = 20
	☐ big = 20 small = 20
	☐ big = 0 small = 10
	☑ big = 10 small = 20
	☐ big = 10 small = 30
	
	Any other values for big and small :
	
	big = small =
	big = small =
	big = small =
		Use this column for your rough notes please
		
		M9

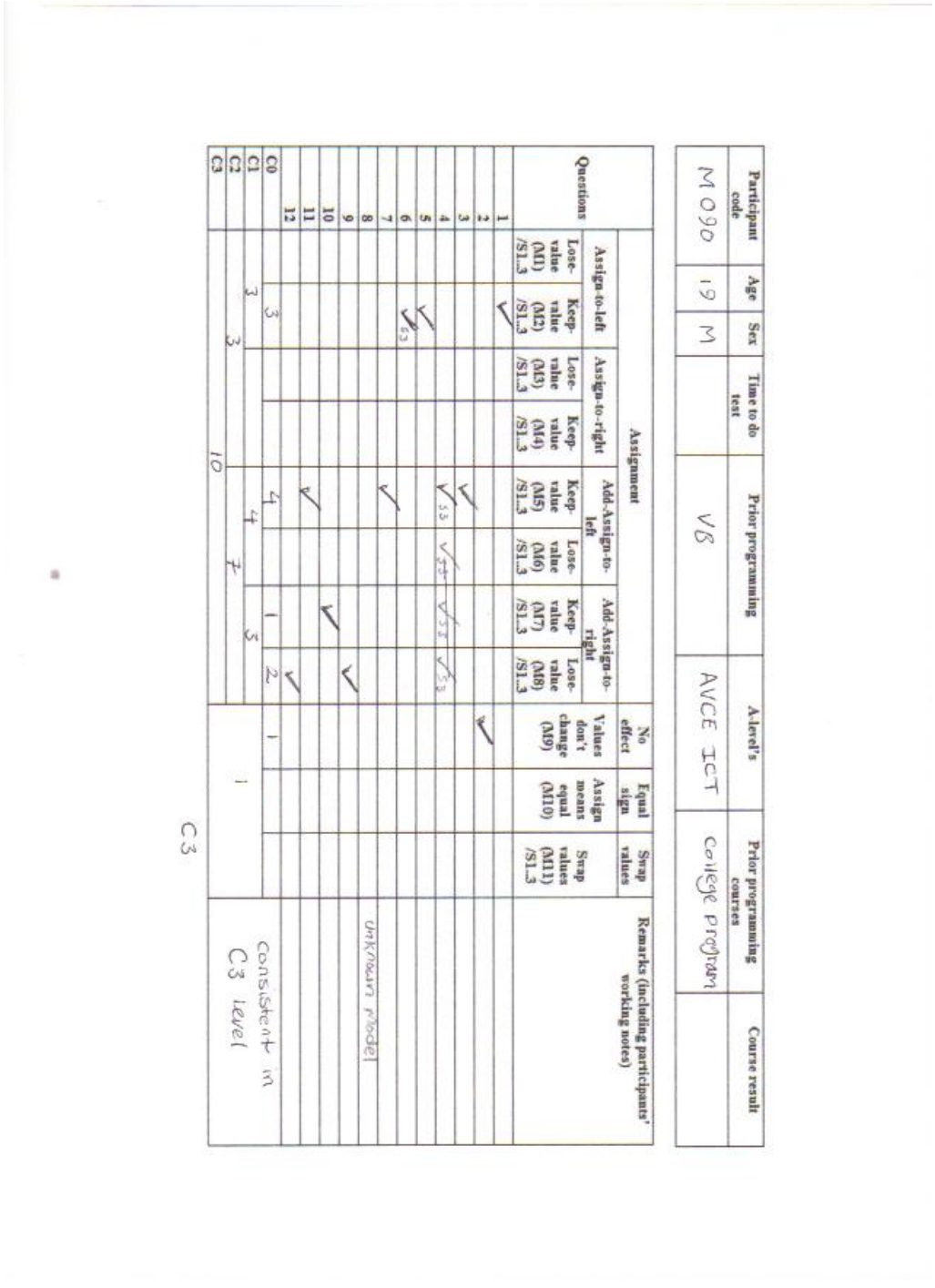
4. Read the following statements and tick the box next to the correct answer in the next column.
 	
`int a = 10;`
`int b = 20;`
 	
`a = b;`
`b = a;`
	The new values of a and b are:
	
	☐ a = 10 b = 0
	☑ a = 10 b = 10
	☐ a = 30 b = 50
	☐ a = 0 b = 20
	☐ a = 40 b = 30
	☐ a = 30 b = 0
	☐ a = 20 b = 20
	☐ a = 0 b = 30
	☑ a = 30 b = 30
	☑ a = 10 b = 20
	☐ a = 20 b = 10
	
	Any other values for a and b:
	a = b =
	a = b =
	a = b =
		M4 +
		M9 / M11

5. Read the following statements and tick the box next to the correct answer in the next column.
 	
`int a = 10;`
`int b = 20;`
 	
`b = a;`
`a = b;`
	The new values of a and b are:
	
	☐ a = 30 b = 50
	☑ a = 10 b = 10
	☑ a = 20 b = 20
	☐ a = 10 b = 0
	☑ a = 0 b = 20
	☑ a = 30 b = 0
	☐ a = 40 b = 30
	☐ a = 0 b = 30
	☐ a = 20 b = 10
	☐ a = 30 b = 30
	☐ a = 10 b = 20
	
	Any other values for a and b:
	a = b =
	a = b =
	a = b =
		M4 +
		M6

Participant code	Age	Sex	Time to do test	Prior programming	A-level's	Prior programming courses	Course result
M116	19	M		HTML	CBSE	NO	

Questions	Assignment								No effect	Equal sign	Swap values	Remarks (including participants' working notes)
	Assign-to-left		Assign-to-right		Add-Assign-to-left		Add-Assign-to-right					
	Lose-value (M1) /S1..3	Keep-value (M2) /S1..3	Lose-value (M3) /S1..3	Keep-value (M4) /S1..3	Keep-value (M5) /S1..3	Lose-value (M6) /S1..3	Keep-value (M7) /S1..3	Lose-value (M8) /S1..3	Values don't change (M9)	Assign means equal (M10)	Swap values (M11) /S1..3	
1	✓								✓			
2												
3								✓				
4	✓											
5	✓								✓			
6												
7									✓			
8	✓S1	✓S3	✓	✓							✓S3	
9												
10							✓S2	✓S2				
11									✓			
12	✓S1	✓S3									✓S3	
C0	2						2	2	3	1		Inconsistent
C1		2	1	1		2						
C2					5							
C3		3										

Figure E.1: A marksheet sample



Participant code	Age	Sex	Time to do test	Prior programming	A-level's	Prior programming courses	Course result
M090	19	M		VB	AVCE ICT	College program	

Questions	Assignment								Swap values	Remarks (including participants' working notes)
	Assign-to-left		Assign-to-right		Add-Assign-to-left		Add-Assign-to-right		No effect	Equal sign		
	Lose-value (M1) /S1.3	Keep-value (M2) /S1.3	Lose-value (M3) /S1.3	Keep-value (M4) /S1.3	Keep-value (M5) /S1.3	Lose-value (M6) /S1.3	Keep-value (M7) /S1.3	Lose-value (M8) /S1.3	Values don't change (M9)	Assign means equal (M10)	Swap values (M11) /S1.3	
1		✓							✓			
2												
3					✓							
4					✓S3	✓S3	✓S3	✓S3				
5		✓S3										
6												
7					✓							
8												unknown model
9							✓					
10						✓						
11					✓							
12							✓	✓	✓			
C0	3			4		1	3	2	1			consistent in C3 level
C1					4							
C2	3					7				1		
C3			3									
				10								

C3

Figure E.2: A marksheet sample

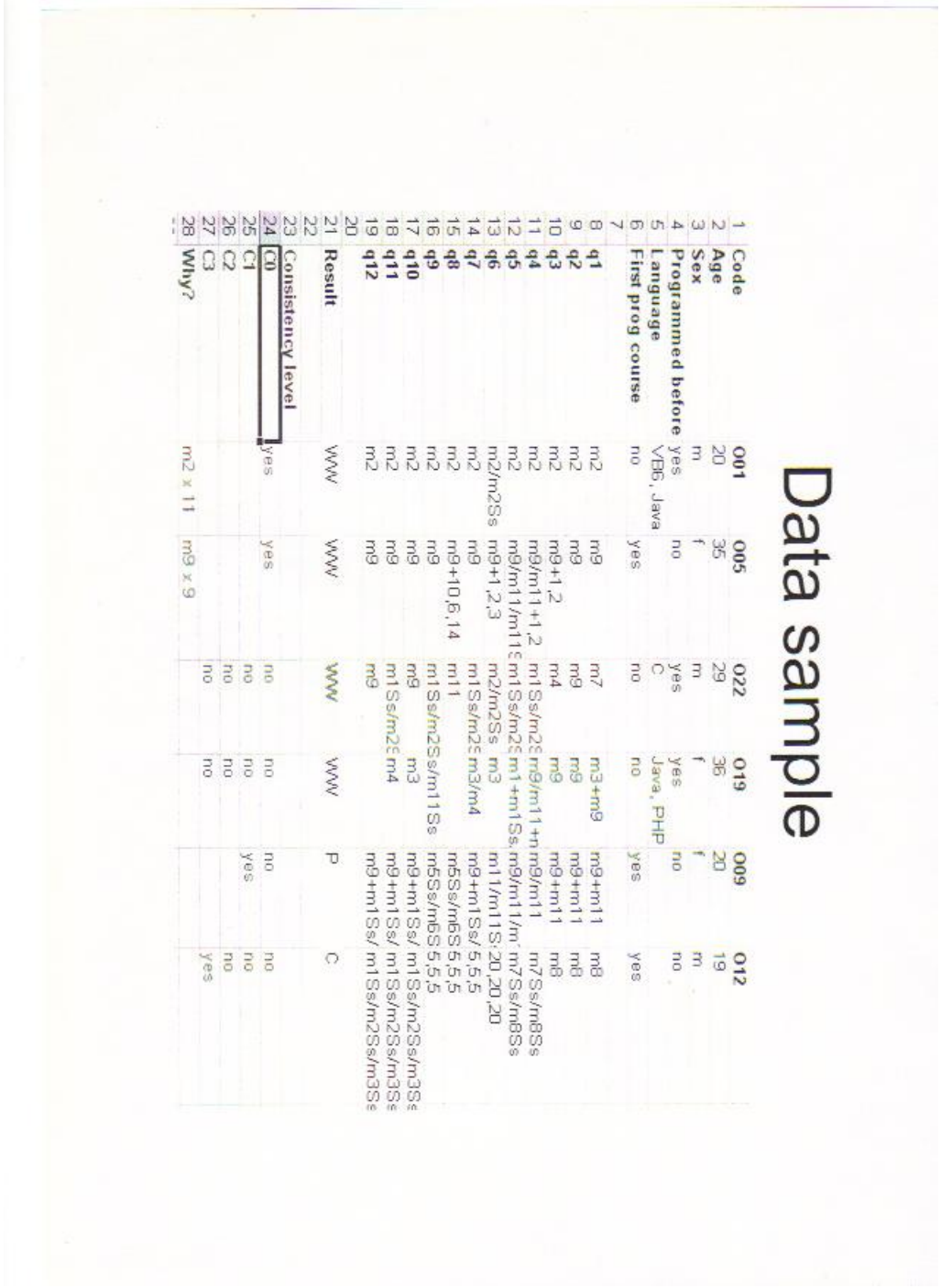


Figure E.3: A marksheet sample