A Study of Novice Programmer Performance and Programming Pedagogy

Michael Edwin Dacey BEng MSc

Director of Studies: Dr Carlene Campbell

Supervised by: Dr Kevin Palmer,

Dr Glenn Jenkins

Submitted in partial fulfilment for the award of

the degree of Doctor of Philosophy

University of Wales Trinity St David

2018

Author's Declaration

This work has not previously been accepted in substance for any degree and is not being concurrently submitted in candidature for any degree.

Signed ..... (candidate)

Date 17/09/18

STATEMENT 1

This thesis is the result of my own investigations, except where otherwise stated. Where correction services have been used the extent and nature of the correction is clearly marked in a footnote(s). Other sources are acknowledged by footnotes giving explicit references. A bibliography is appended.

Signed ..... (candidate)

Date 17/09/18

STATEMENT 2

I hereby give consent for my thesis, if accepted, to be available for photocopying and for inter-library loan, and for the title and summary to be made available to outside organisations.

Signed ..... (candidate)

Date 17/09/18

STATEMENT 3

I hereby give consent for my thesis, if accepted, to be available for deposit in the University's digital repository.

Signed ..... (candidate)

Date 17/09/18

Abstract

Identifying and mitigating the difficulties experienced by novice programmers is an active area of research that has embraced a number of research areas. The aim of this research was to perform a holistic study into the causes of poor performance in novice programmers and to develop teaching approaches to mitigate them. A grounded action methodology was adopted to enable the primary concepts of programming cognitive psychology and their relationships to be established, in a systematic and formal manner. To further investigate novice programmer behaviour, two sub-studies were conducted into programming performance and ability.

The first sub-study was a novel application of the FP-Tree algorithm to determine if novice programmers demonstrated predictable patterns of behaviour. This was the first study to data mine programming behavioural characteristics rather than the learner's background information such as age and gender. Using the algorithm, patterns of behaviour were generated and associated with the students' ability. No patterns of behaviour were identified and it was not possible to predict student results using this method. This suggests that novice programmers demonstrate no set patterns of programming behaviour that can be used determine their ability, although problem solving was found to be an important characteristic. Therefore, there was no evidence that performance could be improved by adopting pedagogies to promote simple changes in programming behaviour beyond the provision of specific problem solving instruction.

A second sub-study was conducted using Raven's Matrices which determined that cognitive psychology, specifically working memory, played an important role in novice programmer ability. The implication was that programming pedagogies must take into consideration the cognitive psychology of programming and the cognitive load imposed on learners.

Abstracted Construct Instruction was developed based on these findings and forms a new pedagogy for teaching programming that promotes the recall of abstract patterns while reducing the cognitive demands associated with developing code. Cognitive load is determined by the student's ability to ignore irrelevant surface features of the written problem and to cross-reference between the problem domain and their mental program model. The former is dealt with by producing tersely written exercises to eliminate distractors, while for the latter the teaching of problem solving should be delayed until the student's program model is formed. While this does delay the development of problem solving skills, the problem solving abilities of students taught using this pedagogy were found to be comparable with students taught using a more traditional approach. Furthermore, monitoring students' understanding of these patterns enabled micro-management of the learning process, and hence explanations were provided for novice behaviour such as difficulties using arrays, inert knowledge and "code thrashing".

For teaching more complex problem solving, scaffolding of practice was investigated through a program framework that could be developed in stages by the students. However, personalising the level of scaffolding required was complicated and found to be difficult to achieve in practice.

In both cases, these new teaching approaches evolved as part of a grounded theory study and a clear progression of teaching practice was demonstrated with appropriate evaluation at each stage in accordance with action research.

Table of Contents

List of Tables

List of Figures

1 Introduction

As Sheared, *et al*. [1] have observed:

> *"Programming is clearly a difficult topic for many students, and is understandably a key area of computing education research."*

An important starting point in grounded theory study is maintaining an open mind and avoid imposing preconceptions on the development of the theory while "*…ensuring that the knowledge and experience you possess is used effectively…"*[2]. A researcher must begin by acknowledging their existing assumptions, experience and knowledge of the area as an "*effective mechanism for establishing where you stand in relation to your proposed study*"[3]. In the author's experience over a number of years of teaching programming to first year undergraduate students the bimodal nature of student results has been quite notable. This reflects the similar experiences of many teachers across many courses across institutions, and in research studies[4]. For a significant percentage of students the art of programming proves overly challenging. The problems of high failure rate was the subject of a detailed investigation in 2001 by the McCracken working group in what has become the most cited paper [5] in the SIGCSE section in the ACM library [6]. In this paper, the McCracken working group assessed the programming ability of a large number of first year computer science students across four universities. The students were required to write programs that parsed and evaluated arithmetic operations. Most of the students performed poorly, with the average student score being 21%. They concluded that the majority of programming students failed to gain even the most rudimentary skills required to get a program ready to run. Furthermore, the results were independent of country and education system. Although this study was unable to identify specifically why these students struggled, they did note that the students were weak in problem solving skills i.e. following the five step iterative process:

1. Abstract the problem from its description.

2. Generate sub-problems

3. Transform sub-problems into sub-solutions

4. Re-compose

5. Evaluate and iterate

Programming requires abstraction of a problem into a form suitable for conversion into a program. As Lui, *et al* [7] observes:

> "*Computer programming is all fabricated and finds few parallels in the physical world…*"

The abstract nature of programming has been the subject of a number of research papers [8, 9]. There is broad agreement that abstract thinking is a core component of programming and a difficult skill for novice programmers to develop.

But, how do we define abstract thinking in the context of programming? To investigate this, we have to consider the role of cognitive psychology [10], working memory [11], fluid intelligence (gF) [12] and the mental models constructed by programmers [13].

The questions posed are:

- Is success or failure predictable? If so can those identified as being at risk of failure be given additional assistance?
- Is it possible that these students may have benefited from a different teaching approach?

Research such as that conducted by the McCracken working group[5], has found no single explanation for the difficulties experienced by novice programmers. Thus, the research contained in this thesis started with no preconceived ideas that a "*silver bullet*"[14] would be found to solve all novice programmers' problems. A more holistic viewpoint was taken, and as such a broad overview of the research area was conducted using a grounded theory approach. Originally, grounded theory espoused the idea that the research problem itself must "emerge" from the research [15]. However, a more middle ground approach is to "*state your research question broadly and in terms that reflect a problem-centered perspective of those experiencing or living the phenomenon to be studied*"[3]. The "general" aim of the research presented in this thesis was to conduct a study into the process of learning to program, and to determine whether any factors or patterns of behaviour were associated with or indicative of success.

Grounded research allows us to create theories that may help explain the poor performance of students. These theories can be further explored, confirmed and if validated form the basis of approaches to mitigate the underlying causes of the problems. The action research methodology addresses the process by which an individual's practice

(or work), is modified and the effects of these changes are evaluated. Applying a mixed methodological approach, the changes to be made are driven by the theories generated from the grounded action research. In the context of the research presented here, this involved studying the problem(s) encountered in teaching programming through grounded research and determining the actions to be taken to try to improve the teaching approach (practice). Thus, the rationale for change was developed and documented during the grounded theory phase, while the actions taken as a result of this research and the effects of those changes were documented to form the action research [16]. The aim and objectives of the research presented here, have been refined as a result of the research itself.

The aim of this study is to investigate the causes of poor novice programmer performance and develop approaches to mitigate them. To meet this aim, three objectives had to be met:

1. To develop a systematic understanding of the cognitive psychology associated with learning to program and to review current pedagogy to identify limitations in the current approaches.
2. To analyse the factors associated with poor performance and to develop an understanding of how these relate to cognitive psychology and how they impact on the student learning experience.
3. To change current teaching practice by applying principles and concepts from the cognitive psychology and to critically evaluate the effectiveness of the new approaches adopted.

In Chapter 2, the Literature Review begins by investigating existing background research to set the context for this study. It contains discussion of research into abstraction, cognitive psychology, software comprehension, problem solving and pedagogy. The structure of this review reflects the results of the grounded theory research, since this is the most appropriate way of exploring those results. Hence, a number of concepts and their relationships are illustrated by figures generated from that research.

Chapter 3 discusses the research methodology used for the research undertaken and presented in this thesis. In this case, a mixed methodology was “selected” and this chapter presents arguments for this approach. Perhaps a more accurate description is that the work was inspired by this methodology since some modifications were made.

One implication of the selected approach was that notes or memos were made as secondary data sources were analysed. These have been written into the literature review, but in-line with grounded theory these notes were first coded and classified. The primary research contains a number of experiments that were performed to obtain more data in order to refine the analysis. In grounded theory terms this process is referred to as theoretical sampling, and provides support for the theories generated. These experiments are documented in Chapters 5 and 6.

Chapter 4 describes the grounded research process employed and a new approach to visualizing the incident data gathered using an edge-weighted graph. An overview of the results of the grounded research is provided and the resultant research phases are introduced.

Chapter 5 identifies a number of metrics to assess student performance and analyses their effectiveness in measuring and predicting student difficulties. The research involved using data mining pattern analysis to determine whether specific patterns of behaviour could be associated with good student performance.

Chapter 6 evaluates the role of working memory in programming, by using code and Raven Matrices tests to determine if there is a correlation between the results. A correlation would suggest that some students have an inherent advantage in problem solving within a programming context.

Chapter 7 and Chapter 8 apply the grounded research findings to the development of teaching approaches aimed at overcoming the constraining factors affecting student performance. In the first instance, by abstracting the teaching of software constructs and concentrating on building the student's mental model of them. The problems presented at this level, were very short and basic with an emphasis on repetition to aid recall. While in Chapter 8, the focus switches to the more advanced problem solving required for more real-world problems and the provision of appropriate scaffolding to support this learning.

Chapter 9 outlines the stages of development of teaching practice and the research that influenced and motivated these changes. This chapter also suggests a course structure for teaching programming based on the action research conducted.

Chapter 10 provides an overview of the results, overall conclusions and describes areas where future research should be conducted to expand upon the findings presented in this thesis.

The contributions to knowledge are summarised below:

- Problem solving is the key indicator of good novice programmer performance, and no other patterns of behaviour, as measured by common performance metrics, are associated with or are predictors of coding ability.
- A correlation is demonstrated between Fluid intelligence (gF) (and working memory) to programming ability, providing evidence of both the importance of problem solving skills in programming and also offering an explanation for the bimodal distribution of marks often seen at the completion of programming courses aimed at novice programmers. An interesting conclusion being that some novice programmers have an initial inherent disadvantage that must be overcome.
- A new Abstracted Construction Instruction pedagogy can be used to teach software constructs as patterns, supporting a more gradual learning of programming skills based on an analysis of software comprehension in the development of expertise. This research also investigated the teaching of problem solving skills in programming, and found that a key aspect of novice difficulties is the failure to recognise the difference between coding and problem solving, with many issues arising due to poor mapping between the problem definition (or domain) and the student's mental model of the solution.
- A new Structured Problem Solving pedagogy can be used to promote the development of advanced problem solving, by developing software frameworks to support scaffolding for practice. A clear motivational advantage of this approach is the creation of an environment within which problems emerge and can be identified by the students themselves. However, some limitations of the scaffolding for practice were also identified.

In universities it is common to refer to the course being studied by a student as the "programme" and the individual units of study within it as "modules". Thus a student may be enrolled on a Computer Science degree programme as part of which they are studying a module of introductory programming. For other institutions or educational sectors these terms may be alien. Given this thesis only considers the study of programming, any

potential confusion will be avoided by only using the term “course” and defining it as a unit of study forming part of the students’ overall studies.

2 Literature Review

The literature review presented here is divided broadly into two halves. The first half is structured to reflect the grounded theory analysis and explores the relationships between the identified concepts. As illustrated in Figure 2-1 code abstraction was the most commonly occurring theme and is discussed in Section 2.1. Two concepts that are directly related to abstraction are software comprehension and problem solving skills i.e. mental models [17] , plans [18] and program goals [13, 19]. These are covered in Sections 2.4 and 2.6 respectively. The nature of expertise and a comparison of novice and expert programmer behaviour is the subject of Section 2.4.6. Figure 2-1 also illustrates that there are different characteristics associated with novice and expert programmers. There is a relationship between expertise, the mental picture of the code and the background knowledge acquired through solving similar problems or from working in a similar context. Novices are more associated with weak problem solving skills and fixating on unnecessary details that prevent them from seeing the generic abstract solution. Memory, domain specific knowledge, the impact of surface features in problem solving and the effect of loading working memory when solving problems, are all topics related to cognitive psychology and are described in Section 2.2.

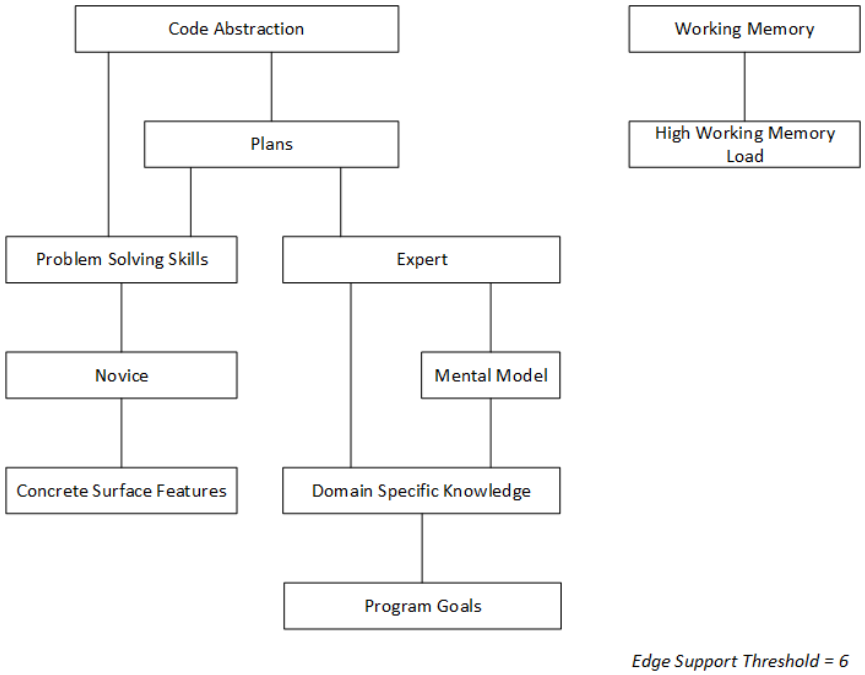


Figure 2-1 Overview of Concepts

The second half of the literature review (starting at Section 2.7) discusses programming methodologies and teaching approaches, and investigates possible approaches to address the issues raised in the grounded theory analysis.

2.1 Abstraction in Programming

The importance of abstraction to programming is summed up by Dijkstra in his Turing Award lecture "The Humble Programmer" [20]:

> "*It has been suggested that there is some law of nature telling us that the amount of intellectual effort needed grows with the square of program length. But, thank goodness, no one has been able to prove this law. And this is because it need not be true. We all know that the only mental tool by means of which a very finite piece of reasoning can cover a myriad of cases is called "abstraction"; as a result the effective exploitation of his powers of abstraction must be regarded as one of the most vital activities of a competent programmer.*"

Abstraction allows programmers to develop solutions to a multitude of problems, and makes programs short and efficient to write. The challenge is to convert real-world problems into abstract solutions that can be executed as a program. By using abstraction a complicated problem can be reduced to a simpler concept which succinctly encapsulates the essential details of the problem. Being able to identify the key requirements of a solution is a difficult skill that requires practice [21].

> "*Understanding that computation is merely symbol manipulation, and that the power of computers is predicated on a tremendous amount of abstraction, is crucial in understanding what computers can, and cannot, do.*" [21]

Unfortunately, as Ben-Ari [22] points out:

> "*Abstraction is essential as a way of 'forgetting' detail, and software development would be impossible without it, but it seems to me that there must be an object oriented paradox: how is it possible to forget detail that you never knew or even imagined?*"

It seems reasonable to argue that Object Oriented Programming should be taught after standard procedural programming. Often introductory programming courses make use of GUI libraries, but this suggests a potential problem [22]. If students are struggling to build viable mental models for simple concepts such as variables, how will they build viable mental models for objects like radio buttons [22]. Furthermore, when abstraction is taught, it must not be assumed that the student will construct the same mental model the instructor has [22].

Adelson [23] found that when recalling code, expert programmers used abstract representations while novices focused on the syntax of the code. The participants were asked to recall 16 random lines of code using a Multitrial Free Recall (MTFR) procedure. The code they were shown could be organised either conceptually into three programs or syntactically into five categories according to the key words that they contained. In the trial, experts clustered the lines into programs while the novices clustered them according to syntactic categories.

> "*With increasing expertise , there is a gradual change in people's focus of attention from aspects that are not relevant to the solution to those that are*" [24]

Corritore *et al* [25] observed that novices develop concrete mental representations of program text, while more advanced novices use more abstract concepts. Hence, an important aspect of learning to program is the ability to apply abstract thinking to real-world problems. Research by Koppleman *et al* [26], found that experienced programmers were able to separate and concentrate on individual levels of abstraction e.g. by creating separate functions to handle different aspects of the problem. However, novice programmers are unlikely to come up with these abstract solutions because they will "see" the concrete solutions first [26]. The difficultly novices have in seeing abstract solutions is related to their inability to see beyond the concrete surface features of the problem (Figure 2-1) [27, 28]. Indeed, many textbooks reinforce this idea by asking students to solve a problem "by hand". Often when teaching programming the flow of control of the program is emphasized, typically accompanied by flow charting exercises. Flow charting is simply an alternative approach to simulating the flow of a program "by hand". Therefore, students need to be taught abstraction because forces exist to prevent them spontaneously developing abstract solutions [26].

> "*More generally, there is a gap between the way introductory programming is taught and mastering the skills of abstraction.*" [26]

For example, students should be taught that a function call is not only a way of sub-dividing code by transferring flow of control to a separate code segment but also a way of suppressing irrelevant detail. They should resist the inclination to determine how a function works and instead concentrate on the effect of the function call [26].

This process of subdivision of code is identical to the process of subdividing problems into smaller sub-problems using a divide and conquer approach [5, 29]. Functions in essence

are just “smaller” problems to solve. Unsurprisingly then, there is significant evidence that problem solving skills are an important factor [30-34] in learning to program (Figure 2-1), including new research presented in this thesis.

Koppleman *et al* [26] made three recommendations:

1. Teach abstraction early using simple problems.
2. Teach abstraction consciously. Instructors must highlight where abstraction is being used and illustrate it with a concrete example.
3. Stress the benefits of abstraction. Many students see abstraction as hard and obscure, so instructors must demonstrate the benefits that it brings and that it makes life easier once mastered.

Mostrom *et al* [35] asked the question “How is abstraction manifested in students’ transformative experiences?”. Students were asked to write a description (a biography) of how a computing concept had transformed the way they saw or experienced computing. The study included 86 students from five institutions across three countries. Of these students, 47 discussed topics related to abstraction. The general areas of these topics were Modularity, Data Abstraction, Object Oriented Concepts, Code Reuse, Design Patterns and Complexity. Although abstraction per se may not be an indicator of likely success, this research suggests that these topic areas exhibit the characteristics of such indicators. In terms of abstraction, many of the students in the study developed an appreciation of abstraction as their programs became larger or more complex.

> “*….they were unable to deal with the complexity of programming without the concept, and applying the concept makes the complexity manageable.*” [35]

In cognitive development terms we would say they moved from “*late concrete operational*” to “*formal operational*” stage [36]. Other students discussed learning an abstract concept but having to implement it concretely in order to gain a full understanding of it. In fact, all of the students discussed “Applying” the concept concretely. This suggests that many of the students learnt gradually about abstraction by applying it in concrete examples [26]. The process of extracting a generic or abstract solution by reviewing or developing solutions to a number of problems requiring a similar solution is related to analogous transfer of knowledge [37]. However, the implication from this particular study is that students will only develop an understanding of

abstraction when the scale and the complexity of programs become too great to solve in simpler ways.

> "*… students were transformed after facing a level of complexity where their normal practices no longer were effective. Finding approaches that did not require this level of failure could be less frustrating and more efficient*" [35]

Unfortunately, the results of attempting to measure the effectiveness of directly teaching abstraction have been mixed. Starting with the hypothesis "*General abstraction ability has a positive impact on learning computer science*", Bennedsen *et al* [38] conducted a number of tests and found hardly any correlation between cognitive development (abstraction ability) and the final grades obtained by the students. They repeated this study over three years and again found hardly any correlation [39]. Their conclusion was that "abstraction" in a computer science context is very hard to define, and that further research is required into how it can be measured.

Clearly abstract thinking is a critical element of programming, but it is primarily developed by practice through solving larger problems that demand more generic solutions. This leads to the chicken and the egg causality dilemma: to learn abstraction the student needs to be able to solve fairly large programming problems, but to program the student needs to be able to learn to create abstract solutions. Overcoming this issue requires the development of a more effective teaching methods that must take into consider the role of cognitive psychology and software comprehension in the creation of a programmer's abstract mental model.

2.2 Cognitive Psychology

During the 1940s, Craik and Bartlett [10] proposed that theoretical models for human memory could be developed and modelled in a computer (which were analogue at the time). This led to a new approach in psychology based on the computer metaphor, and during the 1950s and 60s this information processing approach to psychology became very influential [40] and was summarised by Ulric Neisser[41] in his book "*Cognitive Psychology*" which gave its name to this field of research. The fundamental concept is that any memory system requires the ability to *encode* (enter information into the system), the capacity to *store* it and the ability to *retrieve* it [40]. Although these are distinct stages, they do interact. Typical of these models was the modal model [42]. Broadly this model assumed that we experience the world through our senses involving

sensory memory, through which information is passed into temporary *short term memory (STM)* before being stored in *long term memory (LTM)*.

> "*Short-term memory consists of the information that is maintained at the surface level of coding within the grasp of immediate consciousness or the focus of attention. Thus, short-term memory is a subset of working memory, which in turn is a subset of long-term memory*" [43]

The latest research [44] suggests that memories are actually simultaneously formed in both short and long term memory. Over time, the short term memories decay while the long term memories become stronger.

Based on the assumption that learning and reasoning depend on a mental work space, *working memory (WM)* is related to STM and provides storage for information used for performing complex tasks [40]. It is also thought to be related to attention and is able to draw on resources from both short term and long term memory [40]. *LTM* stores data over long periods of time and consists of both explicit and implicit memory [40]. Implicit memory is associated with skills such as riding a bike. Explicit memory is involved in the remembering of facts or information, and it is sub-divided into both episodic and semantic memory. *Semantic memory* stores general knowledge or real-world facts while *episodic memory* allows us to remember single episodes or events [40]. For example, if you hear that a friend has won the lottery that information becomes part of your semantic memory but where and when you heard the news becomes part of your episodic memory. Hence, the event (the lottery being won) becomes part of both types of memory. One possible explanation for this relationship is that information enters semantic memory as a result of one or more episodic events [40]. Learning the same information through multiple events or sources reinforces the memory of that information.

2.2.1 Short Term Memory (STM)

Short term memory (STM) is a subset of working memory [43] and is an active area of research with a number of competing theories [40]. A simple test for short term memory is the *digit span test* [40]. This test involves remembering short sequences of numbers of increasing length until the test subject fails to accurately recall the numbers. Increasing the number of items to be recalled also increases the total time required to rehearse them, which in turn increases the chance of them fading before recall. The longest sequence of numbers that can be recalled is the memory span for that individual and for

most people this span is about six or seven [40]. This was first described by Miller in his 1956 article "The Magical Number Seven, Plus or Minus Two: Some Limits on Our Capacity for Processing Information" [45] , which hypothesized that there is a fixed capacity for the information received by the human brain i.e. the brain can only receive a certain amount of information per unit time. Miller termed this the "channel capacity" which is a kind of mental bandwidth [17]. He [46] suggested that our capacity to remember is based not on the number of items but the *number of chunks* to be recalled.

The generic definition of a memory chunk is "*a collection of elements having strong associations with one another, but weak association with elements within other chunks*" [47]. For example, splitting numbers into groups of threes [48] makes them easier to remember, probably because we are familiar with this from the natural flow of speech. This is also true for letters [40], for example CONTRAPOSTABLE is much easier to remember than ECTPANRSLBOTPO even though the letters are identical. Given the role that STM plays in natural language processing, we may conclude that it must play a similar role programming. For example, spaces are not allowed in variable names leading to quite cryptic but hopefully descriptive names that are meaningful to the programmer. Using meaningful names is important for program comprehension, especially for novice programmers [49].

2.2.2 Working Memory (WM)

Baddeley *et al* [11] modelled working memory as three components (Figure 2-2) consisting of a phonological loop, a visual-spatial sketchpad and a central executive. However, only the central executive is of interest to us.

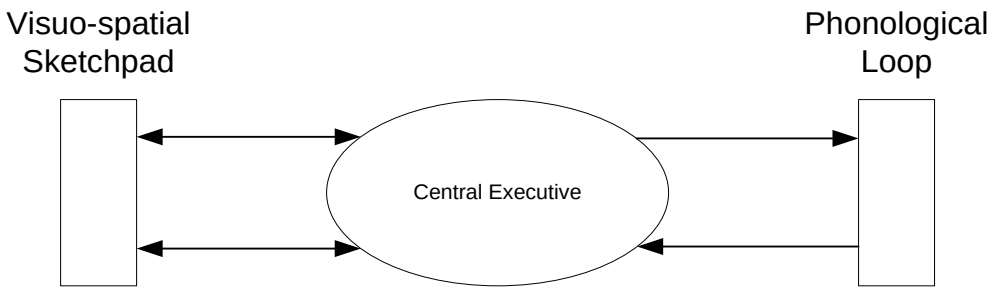


Figure 2-2 The Baddeley and Hitch Working Memory Model

A major function of the Central Executive is focus or concentration. The executive provides two modes of control, an automatic mode and a second mode that depends on a Supervisory Attentional System [50]. People tend to perform many tasks either

automatically or semi-automatically. During a semi-automatic task, actions are performed unconsciously requiring little attention until something out of the ordinary occurs, at which point the executive can resolve the issue through learned procedures. Alternatively, when people are unable to make an automatic unconscious decision or they are faced with a novel situation or problem then they have to pay attention to it and a Supervisory Attentional System [50] has to intervene to make a conscious decision or select a strategy for finding an alternative solution.

As Clarke [51] puts it:

> "*Attention is the gateway to our brain. That's why gaining and sustaining attention is an early and ongoing central consideration in any learning event*"

If we want to understand the limitations on complex mental challenges we need to understand how attention is controlled. There is much discussion in psychology of how attention works [51] and there are a number of mental models [17]. Hidi [52] refers to *early* attention as responsible for automatic detection and prioritization, while *late* attention is responsible for focused effort devoted to learning [51]. A simplified model is described by Klingberg [17] and consists of controlled attention, stimulus-driven attention and arousal. Stimulus-driven attention describes the involuntary attraction to an unexpected event, while controlled attention requires conscious focus. Controlled attention in related research [53] is also referred to as selective or goal-driven attention.

As the load on working memory increases people become more easily distracted [54, 55]. People with higher working memory are more able distinguish between relevant and irrelevant information [56]. In effect, the distractors get stored in working memory instead of the relevant information [17], resulting in people with lower working memory being more easily distracted [57, 58] i.e. higher working memory makes it easier to ignore distractions [56].

The importance of working memory in education must not be underestimated. As Kirschner *et al* [59] note:

> "*Any instructional theory that ignores the limits of working memory when dealing with novel information or ignores the disappearance of those limits when dealing with familiar information is unlikely to be effective."*

Kyllonen *et al* [60] compared working memory tests with a number of tests taken from standard IQ tests and found a high correlation. Engle *et al* found a similar result [12] when

studying fluid intelligence (gF), because the ability to solve problems also depends significantly on the amount of information that can be stored in working memory. He found that the correlation was usually between 0.6 and 0.8 [61] a similar result was obtained through latent variable analysis [62]. Thus, when comparing people who are good at solving problems against those who are not, half the variance can be attributed to working memory capacity [17]. Halford *et al* [63] also hypothesise that working memory and intelligence share a common capacity constraint. This constraint is determined either by the working memory span or by the *"number of interrelationships between elements in a reasoning task*" [64]. The common capacity constraint is thought to arise because of a common demand for attention that is required when forming representations in reasoning tasks [63].

> "*At present, working memory capacity is the best predictor of intelligence that has yet been derived from theories and research on human cognition*" [65]

Although this research will not enter into the debate about measurement of intelligence, it is generally accepted that there are two types crystallized (gC) and fluid (gF) [66] with evidence that they exist being found in studies involving university students [67] and when using MR scanners to monitor brain activity [68]. The term "g" was first described by Spearman [69] who believed that there was one central intellectual ability "g" and numerous specific abilities [69]. "g" is a numerical score-factor (general factor) that was generated after performing factor analysis to examine a number of mental aptitude tests i.e. it refers "*to the determinants of shared variance among tests of intellectual ability*" [70]. However, it should be noted that there is a strong relationship between crystallized and fluid intelligence, and consequently they are not mutually exclusive [71]

It might be expected that programming could enhance students' general cognitive ability, given that it is a skill that requires characteristics such as rigorousness, systematicity, the usage of problem sub-division (i.e. a divide and conquer strategy) and the diagnosis/debugging of problems. [72]. Programming languages may indeed promote procedural thinking and reveal more about how the mind works [72]. However, there are a number of studies that have shown that programming produces little cognitive enhancement [73, 74]. Although, Mayer *et al* [75] noted that learning to program can lead to improvement in a specific aspect of fluid intelligence:

> "*...learning a programming language – even a language with as many critics as BASIC has – can result in changes in thinking skills. The improvement appears to be*

limited to thinking skills that are specifically tied to specific concepts underlying BASIC, however, and there is no evidence of any enhancement of intellectual ability in general" [75]

Given the complexity of the process of programming, is there a relationship between working memory and programming? Shute [76] investigated the relationship between programming skills acquisition and a number of measurements of individual abilities including prior knowledge and cognitive skills, the ability to decompose problems into constituent parts and learning styles (e.g. the priming required in the form of hints). This study looked at teaching Pascal programming using a tutorial application, to determine if learning programming skills could be predicted from measures of specific problem solving abilities which were assumed to be:

i. *understanding*: of the problem and being able to identify the basic elements. Establishing the initial and final state then hypothesizing the operations required to achieve the solution.
ii. *method-finding*: decomposing and sequencing the problem elements into an outline solution to a programming problem that identifies and arranges the relevant operators of commands.
iii. *coding*: translate the natural language solution from the previous stages into code.

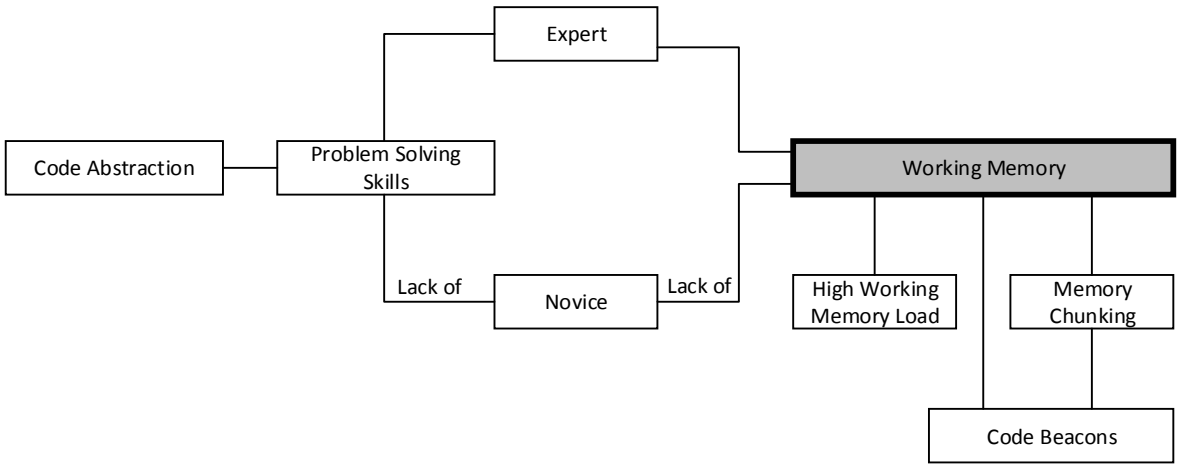
Two sets of data were used to assess prior knowledge. Firstly general vocabulary and mathematical ability was assessed using the Armed Services Vocational Aptitude Battery (ASVAB) tests. Secondly cognitive processes including working memory capacity and information processing speed were measured using computerized tests developed in the Learning Abilities Measurement Program (LAMP) by the US Air Force. An algebra word problem test battery was used to estimate problem solving abilities. A statistical approach was used to determine if there was anything unique to the problem solving as estimated from this test, that would predict who would succeed in learning to program.

The study consisted of 260 test subjects who had no prior Pascal programming experience. A Pascal programming intelligent tutoring system (Pascal ITS) was used to assess programming ability and the test consisted of 25 questions of increasing difficulty. There were 3 learning phases associated with each question, firstly to generate a natural language solution to problem, secondly to convert it into a program implementation plan and flowchart then finally to translate the solution to Pascal code. Subjects could ask for

unlimited hints from the tutor, and there were three levels of hints which became more specific and less abstract. Three progressively more complex programming post-tests consisting of 12 problems per test, were used to assess the subjects programming abilities.

After factor analysis (principal axis with varimax rotation) was conducted working memory was found to be the best predictor of Pascal programming skill acquisition. Similar results were also obtained for a course on logic gates [77]. A limitation of this study is that it did not use the more formally recognised Raven Matrices tests when measuring performance. However, a similar study conducted in this thesis using these matrices also found similar results (Chapter 6).

We conclude that there is a relationship between working memory and programming. Poor working memory is likely to limit the learning of novice programmers when the working memory load increases. Furthermore, the relationship between working memory, gF and problem solving [60] may also explain why low problem solving skills are also associated with poor programming performance. Through grounded theory analysis, an overview of working memory and related concepts is shown in Figure 2-3. This figure also suggests that there is a relationship between working memory to both memory chunking [46] which is the ability to memorise patterns, and the related concept of recognising software keywords or function names known as "*Code Beacons*" [78]. These concepts will be discussed later.



Edge Support Threshold = 3

Figure 2-3 Overview of Working Memory Concepts

2.2.2.1 Testing Working Memory

A classic problem solving test used by psychologists to measure general intellectual ability is Raven Progressive Matrices test. The test subject is presented with a 3x3 matrix of symbols one of which is missing and the subject must deduce the rules required and specify the missing symbol. Thus, each matrix represents a visual analogy problem [70]. An example question is shown in Figure 2-4.

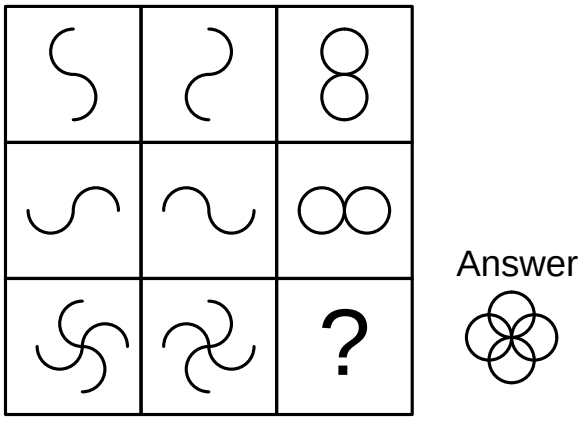


Figure 2-4 Example of a Raven Matrix with Answer

This test relies on working memory [17], since solving a matrix requires retaining and manipulating visual information in working memory while also remembering the instructions. These matrices have been found to be strongly correlated with gF [17] and that they measure processes that are central to analytical thinking [70]. Individual differences in the Raven test have been found to highly correlate with those found in other complex cognitive tests [70]. More difficult Raven test tend to involve more abstract rules, and the level of abstraction also appears to differentiate tests intended for children from those intended for adults [70]. In studying the test subjects Carpenter *et al* [70] found that all the test subjects processed the matrices in the same way, by breaking it into progressively smaller sub-problems and then proceeding to solve each sub-problem. The induction of the rules was incremental in two respects, firstly each rule was induced one at a time and secondly the induction of each rule required a number of small steps generated by a pair-wise comparison of elements of adjoining entities. During the pair-wise comparison, the subjects were encoding some of the figural elements and comparing their attributes in an attempt to identify the "*rule tokens*" i.e. differences that might contribute to a new rule. The error rate was found to increase with the number of "*rule tokens*" in a problem, suggesting that the test subject's ability to keep track of the figural attributes and rules accounted for their individual performance. Here "*keeping*

track" means creating new sub-goals in working memory and remembering when they have been attained. Schraw *et al* [67], also showed that these matrices loaded the verbal crystallized ability of subjects confirming the findings of Prabhakaran *et al* [68]. Actually, Prabhakaran *et al* [68] also studied the brain activity of students while they were performing Raven tests. They concluded that these matrices reflected the status of many working memory systems because they activated many of the domain-dependent and domain-independent working memory systems [68]. This explains the strong correlation between these matrices with working memory and their ability to predict performance in many other tasks [68]. It may be common for the performance of many tasks to rely on multiple working memory systems [68] or a core cognitive ability that spans fluid and crystallized intelligence [67].

A study comparing coding performance to Raven test results was conducted and is included in this thesis (Chapter 6).

2.2.3 Episodic Memory (Long Term Memory)

Tulving [79] defines episodic memory as:

> "*Episodic memory is recently evolved, late-developing, and early-deteriorating past-oriented memory system ..... It makes possible mental time travel through subjective time, from the present to the past, thus allowing one to re-experience, through autoneotic awareness, one's own previous experiences*"

Of most interest is the relationship between episodic memory, semantic memory and learning. The episodic details learned during a task eventually form the semantic structures (or schema) of expertise and an understanding of episodic memory may help improve how experts are trained [80]. There is thought to be an Episodic Buffer [40, 81] in working memory that links together short term, working, episodic and semantic memory. A detailed understanding of this process is not required for this study, but it is an area of active research.

Expertise is developed by repeated exposure programming through a series of programs over long periods of time, months, years or decades [82]. When reading source code a programmer may come across a familiar code fragment which may trigger a memory e.g. of a previous mistake or when modifying code they will remember the file and last modification made yesterday. Furthermore, programmers may "*associate development*

activities with actions and episodic events that may take place outside the world of the text editor or debugger" [82].

2.2.4 Semantic Memory (Long Term Memory)

Semantic memory stores concepts of various kinds. The spreading activation model [83] describes how concepts are organised in semantic memory and assumes that semantic memory is organized based on semantic relatedness or semantic distance i.e. how closely related concepts are.

More complex concepts may also be stored in semantic memory as large structures in what are known as schema. A *schema* is a well-integrated chunk of knowledge about the world, events, people or actions [40] and includes scripts and frames. *Scripts* contain knowledge of events and consequences of events e.g. actions/events in a restaurant such as sitting down, ordering and eating. *Frames* have knowledge of structures e.g. buildings have floors and walls. Events which do not conform to the schema are unexpected so become distinctive and memorable. This of course relates back to episodic memory.

For Anderson [84], cognition depends on the knowledge encoded and the effective deployment of that encoded knowledge. In the Adaptive Character of Thought (ACT-R) theory [84], complex cognition arises from the interaction of two types of long-term knowledge [84, 85], procedural and declarative. Procedural knowledge is represented by production rules and declarative knowledge is represented by chunks. These chunks are initially simple encoding of objects in the environment (or facts) while production rules are encodings of transformation in the environment (how things should be done) that can transform previously stored chunks. In transforming these chunks, new declarative structures (i.e. chunks) may be created [84].

> "*All there is to intelligence is the simple accrual and tuning of many small units of knowledge that in total produce complex cognition. The whole is no more than the sum of its parts, but it has a lot of parts*" [84]

This is the process by which learning from worked examples occurs, thus allowing related problems to be solved [84]. This knowledge acquisition process is very simple as it just requires "*modest inferences about the rules underlying the transformations*" from chunk(s) to chunk [84].

> "*To get behaviour, general interpretative productions must convert this [declarative] knowledge into behaviour....problems arise because of this indirection through these interpretative productions*" [86]

Learning then is just a matter of slowly acquiring more and more production plans and declarative knowledge [84]. Current research [84, 86, 87] does not specifically describe how this long-term knowledge is stored in semantic memory or the role of episodic memory. However, the most likely scenario is that as production rules are reinforced by repeated exposure (e.g. by multiple worked examples [88]) and they are transferred from episodic memory to semantic memory. Likewise, declarative knowledge starts as chunks in working memory but some must be transferred to semantic memory to allow later recall of production rules. Some support for these assumptions is provided by Anderson's own description of how ACT-R would account for the learning of the letter "H" [84]. On seeing "H" it is encoded in a chunk as shown in Figure 2-5.

object		
	isa	H
	left-vertical	bar1
	right-vertical	bar2
	horizontal	bar3

Figure 2-5 ACT-R Encoding of the Letter H [84]

This chunk assumes that another chunk describing what a "bar" is, already exists or as Anderson puts it:

> "*We assume that before the recognition of the object, these features (the bars) are available as parts of the object but that the object itself is not recognized*" [84].

Therefore, it seems reasonable to assume that the declarative structure defining "bar" must be stored in long-term memory i.e. semantic memory.

Interestingly, Anderson goes on to say that:

> "*A basic assumption is that the process of recognizing a visual pattern from a set of features is identical to the process of categorizing an object given a set of features*" [84]

Clearly, this provides support for the role of perceptual learning in which perceptual chunks are used to recognize incoming stimuli that then develop to allow experts to quickly recognize patterns.

Cheng *et al* [89]proposed reasoning involves clusters of generalized abstract rules defined with respect to classes of goals and types of relationships known as *pragmatic reasoning*

schemata [90]. Errors in reasoning can be induced by manipulating the semantic features of a problem and corrected by presenting the problem based on an abstract description of a situation that mapped to a schema [90]. Thus, the number of errors that occur depend on the mapping between the pragmatic schema and the concrete situation, and the degree to which the schema rules allow inferences to be made that conform to the standard logic [90].

In programming the relationship between episodic and semantic is neatly captured by Kolodner [91]:

> *"...even if a novice and an expert had the same semantic memory..., the expert's experience would have allowed him to build up better episodic definitions of how to use it."*

Through experience and practice the expert programmer builds up the episodic knowledge of how to program [92].

2.3 Cognitive Psychology and Programming

It is now possible to begin to see how programming is related to cognitive psychology and Figure 2-6 summarises the steps by which programming is learnt [30]:

(a) The programmer reads the code which represents a new stimulus to which they must pay attention

(b) Appropriate pre-requisite concepts must be found from Long Term Memory (LTM) and the new information is then assimilated.

(c) Actively using this pre-requisite knowledge causes the new information to be associated with it.

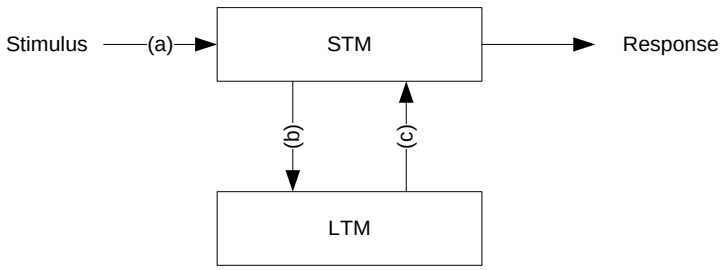


Figure 2-6 Information Processing [30]

Although not specified, it can be inferred that this pre-requisite knowledge [30] is gained firstly through offline activities such as programming manuals [30], training materials and software design and secondly by analysis of software through reading and writing code.

Information is encoded into episodic memory within LTM as an episodic event [80]. These events may be hours, days or week apart, each event adding new nodes or cues to semantic memory storing and reinforcing long term memory of that knowledge. For example, a student may attend a lecture during which the definition of a variable is explained and days later read a book describing the same concept reinforcing and supplementing it. Eventually, the lecture and the book may both be forgotten but the concept remains in semantic memory [93]. Code is inherently abstract [21], so each episodic event enables the programmer to infer more about the abstract coding constructs and principles they are learning. These must be encoded in semantic memory as schema [13] to allow them to be applied in solving future programming problems. Episodic memory also allows programmers to remember previous code and which file contained it [94]. Working memory plays a role in both allowing the student to focus on the required details [56] and in the fluid intelligence required to extract the abstract principle [12].

In reading and writing code, learning progresses in a similar way through worked examples or the student's own code. However, as the tasks become more complex two additional constraints arise [80]. Firstly as the complexity increases so does the quantity and diversity of information that must be stored, demanding a less selective approach to the storage of information. In addition, a continual encoding process is required because any item processed may become important to remember later. By implication:

> "*...memory contains vast amounts of information, much of which is never retrieved, but any of which could be retrieved and could critically affect the course of behaviour*" [80]

Secondly the retrieval of information for a complex task may involve a problem solving search or a dynamic environment in which any item can be required at any time. Encoding specificity suggests that the more time spent elaborating on an item the better the chance of recalling it [95]. Unfortunately, expert problem solving generally affords little time for such elaboration limiting the storage of item cues that are critical to retrieving details about the task.

These constraints were identified by Altman [80], who proposed a variation of the traditional model of memory that used a construct called "*near-term memory*" that was developed using computational behavioural simulation of a programmer at work. Altman was interested in the cognitive processes a programmer uses when browsing, scrolling

and reading through code i.e. how a programmer tries to make sense of code by stepping through it in detail [80]. Near-term memory combined the analytical goals of long-term working memory with real-world studies of memory and cognitive simulation. It describes a component of memory that bridges the gap between large quantities of episodic detail and semantic memory, by providing links that allowed critical items to be retrieved as required when the cues existed. One finding of this research was that the success of building mental models depends on the success of adding the right cues to working memory at the right time.

> "*In the model a stream of internal episodic symbols or event tags is produced automatically by the cognitive system ..... Then, when the attention process adds some new item to WM in service of the current comprehension goal, it automatically associated the current tag with the attended item....to link the tag and the item together in memory, forming an episodic trace whose cue is the item itself.*" [80]

The primary focus of Altman's simulation was the chunking of knowledge associated with expert behaviour and this was simulated using event, perceptual and semantic chunks [80].

Event chunks [80] were recorded each time an object was read from the source code to indicate that it had been "*attended to*" providing a cue that mapped it to the semantics of the item. If this item was encountered again then the event chunk was fired. By implication, the non-existence of such an event indicated that the item was novel and it was selected in preference to any older items. This is similar if not identical to *goal-oriented chunking* [47] which occurs as a result of conscious selective attention where the individual concentrates on remembering the stimulus e.g. the variable name being read from the source code [47].

Perceptual chunks [80] were used to map externally displayed features in the source code and various internal cues in working memory to the associated internal representation of those features. They were created each time cues in the working memory identified a novel external feature (from the source code) that was required to allow a goal to be comprehended. That is, the cues brought the new feature to the attention of the simulation. This process mimics selective attention in humans, and is necessary because when a large number of stimuli are presented simultaneously only a limited subset of the available information can be processed. *Perceptual chunking* is an automatic and

continuous process, as incoming stimuli arrive they are evaluated against existing chunks in short term memory or information from semantic memory (schema) [47].

> "*The concept of selective attention [in humans] is intimately related to that of limited capacity. If our capacity to process, decide about, and remember information were not limited, then selective attention would serve no purpose. It is because processing capacity is overloaded in numerous situations that a subset of the information arriving must be given special attention. Any selective-attention deficit, therefore, implies a corresponding capacity limitation*" [96]

Using perceptual chunks alleviates this cognitive bottleneck by caching the features. If the same feature is identified using the same cues then the cached chunk provides a direct link to the internal representation allowing it to be loaded immediately into working memory [80]. This approach efficiently handles concepts such as nested data structures and nested code, where the comprehension goal might change [80]. Other simulations [47] have also recognized the importance of perceptual chunking.

There is evidence that the human brain does optimize in this way. For example, in the way incremental interpretation occurs when semantic interpretations are developed on an almost word-by-word basis as the text is read [97]. This allows aspects of language comprehension to be performed very rapidly [97]. There is considerable evidence that programmers understanding of code is also "*chunked*", which is briefly discussed here but is covered in more detail in Section 2.4. McKeithen *et al* [98] described an experiment where novice, intermediate and expert programmers were shown one of two versions of an ALOGOL W program: a normal version or a randomly scrambled version that preserved the indentation. They were then asked to recall each line of code and its position in the program. 53 subjects were tested across the all skill levels. This experiment showed that the level of recall for the normal version correlated to the skill levels of the participants. However, no such distinction was evident for the scrambled versions, demonstrating that experienced and novice programmers remember normal programs differently. On closer inspection, it could be seen that expert programmers were recognizing the lines as chunks, such as counting loops. In a similar experiment, Guerin *et al* [99] demonstrated that expert programmers had better semantic knowledge than novices by asking the test subjects to write a summary of each program and how that purpose was achieved. They found that recall of the programs highly correlated with software comprehension and the experts were always better than novices. As before, when the program lines were randomized the experts advantage over novices

disappeared because the experts could no longer apply their semantic knowledge. This effect does not extend to well-ordered programs that are more or less meaningful (typical and atypical), where experts always do better [92]. In fact, for atypical programs the difference between expert and novice programmer performance actually increases [100]). Widowski [100] found evidence that experts do use plans (as described by Pennington [101]) for stereotypical programs , "*with a significant interaction between expertise and semantic complexity*" [92]. But on atypical programs, experts shifted to different strategies while novices did not [100]. Experts adopted two strategies, control-structure oriented and variable-oriented [100]. Experts consistently used the variable-oriented strategy more than novices, and varied the control-structure oriented processing according to the complexity of the program [92, 100].

Expert programmers "chunk" the information they learn differently to novices. Simon [102] estimates that to become an expert in a domain requires about 50000 chunks of domain-specific information. An expert might encode the code segment given in Figure 2-7 as "Calculate the sum of the array" [30].

```
int[] numberArray = new int[10];
int sum = 0;
int i;
for(i = 0; i < 10; i++)
{
   sum = sum + numberArray [i];
}
```

Figure 2-7 Example Code for Sum of Array

> "*Wiedenbeck [103] empirically verified that recognizable patterns with the source code, which serve as indicators of a stereotypical structures or operations, can be considered beacons. … Experienced programmers tend to rely more on code pattern beacons rather than naming style when comprehending a program's source code*" [49]

The central role that the concept of memory chunking plays in the way programmers develop schema to interpret code is reflected in the grounded theory analysis conducted as shown in Figure 2-8. For programmers the mental models and schemata created in their memory are directly derived from and influenced by this process. In particular, by the ability to spot important cues or "*code beacons*" [78] in the code. The way programmers develop a mental mode or schema is a subject much discussed in research into software comprehension, since it represents the programmer's expertise acquired through experience. These schema contain static elements such as text structure

knowledge, plans, and hypotheses with which they create higher level abstractions [104]. To reflect the more specialised nature of many of the "*memory chunks*" involved in programming, we begin referring to coding "*plans*" [101]. Figure 2-8 also illustrates the role that working memory plays in potentially limiting the learning of these plans and how practice can improve memory [40, 105].

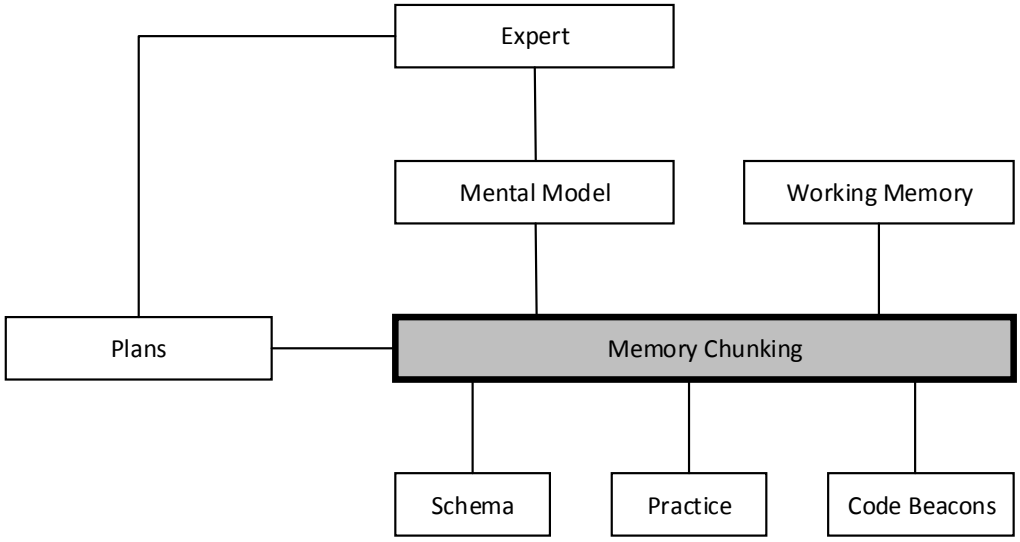


Figure 2-8 Overview of Memory Chunking and Related Concepts

2.3.1 Perceptual Learning and Teaching

Using pre-existing schema, any novel stimuli can be encoded into a new chunk which can then be used to improve the recognition of future stimuli. Generally termed *perceptual learning*, this process continually alters the perception of the problem or task, allowing attention to focus on the new and novel. Schemata in long term memory allow us to "fill in the gaps" and enhance our understanding [40] of what we read or hear i.e. they allow us to make "calculated guesses". Thus, these schemata form our understanding of natural language syntax. Alternatively, if a schema is incorrect then it may distort our memory or understanding [40]. A number of successful studies have found that receiving teaching to underpin this memory chunking process has enabled students to learn a "solution plan" mimicking expert behavior [106-108].

Given the potential importance of perceptual learning in programming, we need to develop a clearer understanding of the nature of the programmer "chunking" mechanisms, and this leads us into a discussion of software comprehension.

2.4 Software Comprehension

The study of how programmers build mental models of the software they are developing or maintaining is known as *program or software comprehension* and is defined as:

> "*a process whereby a software practitioner understands a software artifact using both knowledge of the domain and/or semantic and syntactic knowledge, to build a mental model of its relation to the situation*" [109]

There are numerous cognition models that attempt to explain this process, but the essentials are found in Letovsky's model [13] which consists of an external representation, existing knowledge, an assimilation process and a mental model (Figure 2-9)

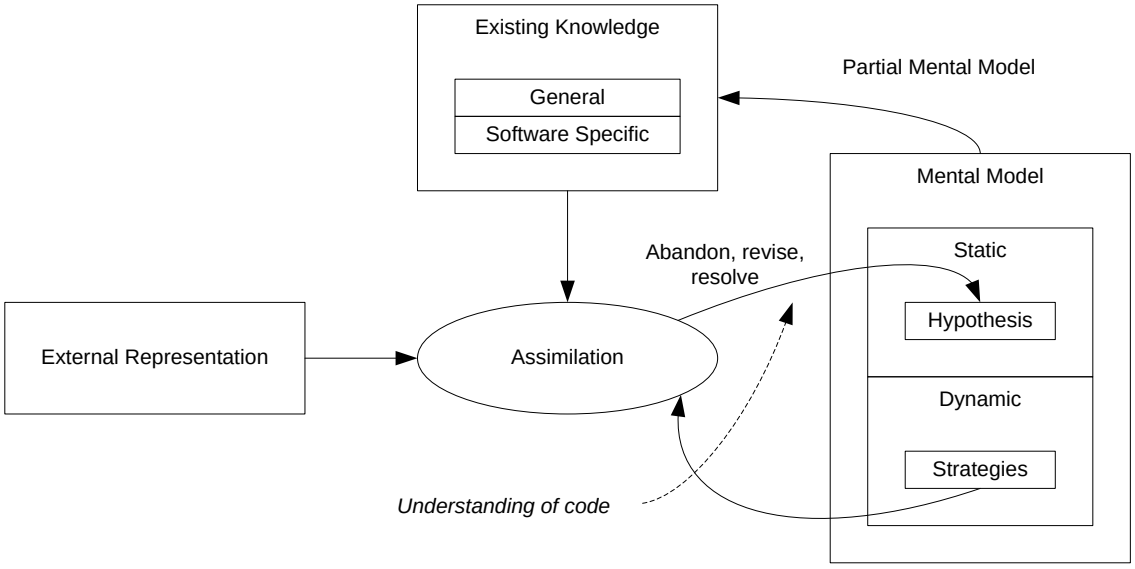


Figure 2-9 Generic Software Comprehension Model

A programmer's accumulated existing knowledge includes programming language syntax, software constructs, programming principles, concepts, techniques, algorithms and domain specific knowledge [13]. If the programmer has worked with the same code previously then they may also have a partial mental model of it [13]. By "External Representation", we mean any *external* information that is available and can be read by the programmer including the code itself, any system documentation, manuals, expert advice and other pertinent sources of information. [13, 109]. Assimilation involves the understanding of the code using a top-down or a bottom-up process, using various cognition strategies to formulate hypotheses that can then be resolved, revised or abandoned [104]. This process is opportunistic and programmers will change strategies

with ease depending on external cues and the approach that will yield the highest gain in knowledge [104, 109]. There are three major types of hypotheses [104]:

- *why* conjectures hypothesize the purpose of a function or design choice
- *how* conjectures hypothesize the method for accomplishing a program goal
- *what* conjectures hypothesize classification e.g. concepts such as a variable or a function.

> "*Hypotheses drive the direction of further investigation. Generating hypotheses about code and investigating whether they hold or must be rejected is an important facet of code understanding*" [104]

For Brook [110], the mental model is developed in a top-down process where hypotheses are iteratively refined by passing through a number of knowledge domains e.g. accounting, mathematics and programming, until they match either the code or related documentation [104]. Hypotheses are checked against the External Representation to seek support [104] for them. The programmer starts with a general hypothesis about what the program does which is formulated from a number of sources (excluding the source code) that document the programs purpose [49]. These sources include source code headers, inline source code comments, user manuals and API reference texts. [49]. This initial hypothesis guides the programmer when they read the code, because Brooks [110] believes they do not read the code line-by-line instead they scan it for 'beacons' which they use to "*elaborate their current hypothesis by forming more specific, sub-hypothesis*" [49]. Slowly a hierarchical structure of hypotheses is developed, starting from the initial hypothesis and leading to the lower-level subsidiary hypothesis which are "*more closely bound to specific parts of the programs source code*" [49]. The larger this hierarchical structure the better the programmer understands the source code [49]. The concept of code beacons is further discussed in Section 2.4.4.

While a detailed description of the of the many cognition models is beyond the scope of this research, a brief summary is justified. In the Letovsky model [13], the mental model has three layers: a specification, implementation and an annotation layer. The program goals are described by the specification layer (highest abstraction level), while the implementation layer (lowest abstraction level) provides abstractions of data structures and functions. Finally, the annotation layer links these goals with their realization in the implementation layer. The implications of program goals are described in Section 2.4.3.

For the Shneiderman and Mayer model [111] the programmer chunks the program while reading it and these chunks are held in a number of levels of abstraction in working memory. A mental model is stored in long term memory as syntactic or semantic knowledge. Like working memory, the semantic knowledge is layered and incorporates high-level concepts and low-level details [104]. While the syntactic knowledge is program language specific, the semantic knowledge is abstract and applicable across many problems.

Soloway, Anderson and Ehrlich [18] looked more closely at the memory chunking process and developed the concept of plans and defined a program as a set of plans [112] which when merged together in the correct way achieve the goal of the program [19]. If a plan is common, the program code for the plan can be abstracted and stored as a schema or a "chunk" of knowledge [19]. A program can therefore be considered to have a *plan structure* consisting of basic plans or "canned solutions" that have been created from previously learnt *plan schemata* [19].

> "*… understanding a program is finding a set of underlying plans such that parts of the program match the roles in the hypothesized plans. Comprehension of a program, under this view, would proceed by partial pattern matches activating candidate plans, causing programmers to search for further evidence to instantiate a plan. According to this concept of comprehension the program is mentally represented as a set of linked descriptions, like blue prints, rather than a set of instructions to be executed.*" [101]

For example, a sort algorithm would be represented by a microstructure containing all the instructions in the specific programming language while the macrostructure is an abstraction of the concept which is just labeled "sort" [104]. In short, programmers often just remember the purpose of a piece of code or a function and identify it by some label or name e.g. "it sorts values, so it is a sort function".

Higher level chunks can contain lower level chunks building up the knowledge of the structure of the program [104]. It is also worth noting that unlike normal text, in program text there is implicit information "in the text" that gives meaning to the program [101], such as the sequence of statements and certain program language keywords provide information about the sequence in which program statements will be executed.

Pennington [101] suggested that programmers maintain two mental models [104]: a program model that represents the source code and a situation model that takes into consideration the problem domain.

2.4.1 Pennington's Program Model: Programming Plan Knowledge

To explain program model development, Pennington [101] used the concepts of text structure knowledge and programming plan knowledge.

2.4.1.1 Text Structure Knowledge

Programs can be described by a limited number of control flow constructs including sequence, loop constructs, branch constructs, variable definitions, function call hierarchies and function parameter definitions. [104]. These units are known as structured programming units [101] or alternatively as prime programs [101] because programs can be decomposed into them in the same way a number can be decomposed into prime factors. Prime programs are the lowest level of decomposition and can be aggregated into a higher level sequence, such as a branch or loop construct, so that "*the entire program text can be represented as a hierarchy of prime units*" [101]. Soloway, Anderson and Ehrlich [18, 104] refer to these as *implementation plans*. A programmer's knowledge of these prime programs or constructs is their text structure knowledge [101]. Proponents of structured programming hypothesize that strict use of these constructs makes code easier to understand because it corresponds to programmer's mental organization [113]. Furthermore, the process of understanding a program is similar to decomposing a program into prime programs [101, 114]. Considerable evidence exists that suggests text structure knowledge does play an important role in software comprehension [98]. Alluding to episodic memory, Pennington indicates that structured programming or prime program units may be considered to be a kind of "episode" for programs [101].

2.4.1.2 Plan Structure Knowledge

Plan structure knowledge [101] emphasizes that "*programmers' understanding that patterns of program instructions 'go to together' to accomplish certain functions*" and corresponds to an intermediate level of programming concepts such as searching, summing, hashing, and counting. Higher level concepts may involve algorithms and data structures. These plan structure representations of a program are primarily based on data flow relations and function (purpose) [101].

2.4.1.3 Slot Types and Fillers

Von Mayrhauser [104, 115] proposed that plans are schemata (or frames) which consist of two parts: *slot types* (templates) and *slot fillers*. A slot filler is developed specifically to solve a particular problem, using a slot type which is an abstraction of multiple slot fillers. Therefore, slot types which are sometimes just referred to as "*slots*", can be considered templates that can be applied to a number of problems. Examples of slot types include data structures such as lists or trees [104]. These structures are linked by either a *Kind-of* or an *Is-a* relationship.

Thus, the concept of "frames" suggests that programmers abstract a generic "textual pattern" with elements i.e. "placeholders" where specific changes must be made to craft a working solution. For example, the pattern for declaring a variable might read:

type variablename;

Variables are declared with an appropriate type, the allowed types have to be memorized and written at the start of the declaration. The variable name follows the type and must follow certain variable naming rules. Finally, the instruction statement must end with semicolon. Thus, perceptual learning in the context of programming may, at least in some part, involve software constructs being taught as "abstract programming plans" rather than a learning through examples approach.

2.4.1.4 Supporting Evidence for Plan Knowledge

In examining the psychology of learning BASIC [116], Mayer refers to "*levels of knowledge*" when considering what is learnt when programming and these seem closely related to the concepts of plans. For example, a "*mandatory chunk*" consists of two or more statements that must occur in some configuration [116] while a program is a set of chunks and statements.

> "*As a learner gains more experience, the size and number of chunks(or 'superstatements') he/she knows will grow*" [116]

As a result, Mayer [116] recommended teaching programming by explicitly "presenting" the chunking process at various levels and emphasizing techniques for generating subroutines and structured programming to help the novice programmer to develop additional chunks[116].

Ebrahimi [32] in a study of 80 novice programmers demonstrated that there is a strong correlation between plans and software constructs stating "*Language constructs are used as building blocks to form a plan*" and that "*…programmers that make more plan composition errors also tend to make more language construct errors and vice-versa*".

A class of bugs that novice programmers often encounter are *plan composition* problems, which occur as a result of the difficulty they experience in putting groups of plans or parts of plans together correctly [117].

A surprising demonstration of the importance of developing and consistently applying a mental program model may have "accidentally" been provided by Dehandi [118]. It was discovered that some students were able to determine or imply a mechanistic sequential code execution process despite receiving no programming instruction. The most successful programming students that later emerged were those that were initially able to create some internal model of the process themselves and then apply it consistently [119]. Dehandi referred to these students as "*programming sheep*" [118]. The implication being that the students were spotting patterns and execution rules and were therefore employing untrained chunking behaviour. A subsequent meta-analysis including an improved version of the test [119], demonstrated that this effect did indeed exist. Thus, we can conclude that two vital components in learning to program are extracting the "rules" for writing code (in software comprehension these are the "plans" [18]) and developing a systematic approach to applying them consistently.

2.4.2 Pennington's Situation Model: Domain Plan Knowledge

Pennington [104] proposed that programmers construct a situation model for addressing the problem domain based on the concept of domain plan knowledge.

> "*By inferences and additional domain knowledge, a situation model is also constructed which includes comprehension of the function, the goals and purposes of the program*" [120]

Domain plans incorporate the non-code related knowledge about the problem and concrete real-world objects [104] (i.e. the problem domain specific knowledge) which is crucial for understanding program functionality. These plans exclude the implementation detail, such as the code or the low-level algorithms required [104]. For example, the code "cost = cost + productPrice" would be described in the situation model as "increase the

cost by the price of the product purchased". The concept of plans relates directly to work of Soloway, Anderson and Ehrlich [18].

2.4.3 Program Goals

Spohrer *et al* [117] make two important points about the relationship between goals and plans:

i) a goal decomposes into subgoals, and plans organize the subgoals of a goal;

ii) there are usually many different plans for achieving the same goal.

Building on the work of Soloway[112], Rist looked at schema creation in programming [121, 122] and software design [19, 123]. For Rist, the creation of a basic plan involves setting a goal and working backwards from it one action at a time until the plan is complete [19]. Likewise, complex plans are created by merging basic plans together working backwards from the goal [19]. A plan schema consists of a *surface structure* which is the actions (lines of code) executed in program order forming a linear structure and a *plan structure* which is the set of data and control flow dependencies (i.e. actions later in the plan are supported by earlier actions) traced backward from the goal producing a non-linear structure [19, 123]. A plan is a branch of a plan structure, so plans appear at many levels of composition [123] i.e. there can be many compound plans. For example, suppose we want to calculate the average rainfall for a month [19]. Working backwards from the goal, to calculate the average we need the total rainfall for the month. Each daily rainfall must be added to the total, for this code to work each days rainfall value must be entered in a loop and running total must be kept. Before a running total can be kept it must be initialized to zero. The plan structure is thus defined by working backwards from the goal which was to calculate the average. One plan structure can generate multiple different surface structures depending on the choices made in constructing the plan from the goal and how the plan structure is coded [123].

If a programmer has no previous experience of the programming problem, the solution is typically constructed by focal expansion [121] using a bottom-up or backward design approach. As a programmer gains knowledge, they develop the program design using a top-down and forward design approach [121]. This effect is also seen in the way experts read normal and abnormal programs [124]. Three levels of expertise can be defined, novice, advanced novice and competent [122, 123]. A novice knows the syntax and general programming principles and usually solves a problem by backward design [123].

An advanced novice possesses a small set of schema and can apply these to new problems as well as developing their own schemata. They also have a set of design rules but apply these with some difficulty [123]. A competent designer has a large set of abstract schema that they can apply in multiple languages and construct complete algorithms. They can easily apply design rules to select the best solution [123].

> "*Only a rank novice working on a new problems shows pure bottom-up design, and only an expert working in a familiar domain shows pure top-down design*" [123]

Rist [123] promotes the teaching of many variations of these plans over presenting a single solution. By planning code, the student is forced to address the initial focus of each plan and build backwards from it (the typical approach when faced with a novel problem). It emphasizes that the plan structure contains the fundamental solution and not the code, since the same plan can be implemented in many ways. Students are forced to "chunk" the solution making the small re-usable plans easier to remember, reducing the cognitive load imposed in attempting to remember a complete solution. Since plans can be implemented in many ways, the student "sees" that the same solution can be coded in a number of ways and the act of choosing is made explicit. Finally, the approach is systematic and forces the student to identify the essential details of the design.

Rist [122] uses the concept of "*slots*" [104, 115] as previously discussed (Section 2.4.1.3) to replace variable names when translating the basic plan structures to a concrete code solution (surface plan).

2.4.4 Cues or Code Beacons

A beacon can be text, a component or other knowledge that invokes a particular mental schema. For example, a function name "sort" is an obvious beacon for a sorting procedure but just the presence of a swap statement inside a loop may actually be enough [78] to trigger the same connection. Evidence exists to support this concept of "beacons". Gellenbeck *et al* [78], studied the importance of procedure and variable names as beacons. Short Pascal procedures for searching and sorting were presented to 96 computer science students for one minute and then they had to produce a written description of the function of each procedure. They found that meaningful procedure and variable names served as beacons of high-level comprehension [78] but in some cases the presences of strong code beacons [78], such as a swap in a sort procedure, were more significant than the procedure name for high-level comprehension [78] i.e. the

programmers were able to infer the behaviour of the procedure even when the procedure name was deliberately misleading [78]. Pennington [101] also found that when the code is completely new to a programmer they build up their understanding of a program from the bottom-up using code beacons [104, 110].

In order to improve software development tools, Ko *et al* [125] investigated how programmers modified existing code. The study used 31 Java developers of different abilities and required them to complete 5 maintenance tasks over a 70 minute period during which they would be interrupted a number of times. For 10 of the more experienced developers their actions were recorded and analysed in further detail. From the results of this study, a new model of software comprehension emerged (Figure 2-10) that describes a process of searching, relating and collecting relevant information. it involves forming perceptions of relevance from cues in the programming environment [125] .

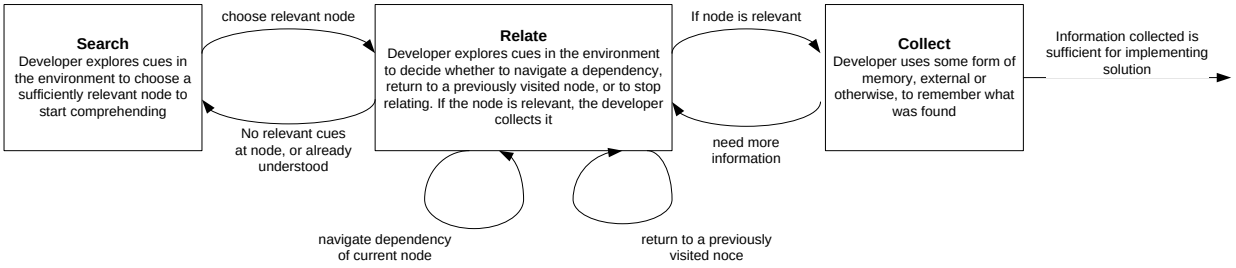


Figure 2-10 Model of Program Understanding in which Developers Search, Relate and Collect Information. [125]

In this model, two factors about the development environment were found to be important [125]. Firstly it must provide clear and representative cues to allow developers to judge the relevance of the information, and secondly it must provide a way to collect information so that they do not have to retrace their steps to locate information that has already been found. An important point this study raised was that the visual representation of code is an important influence in software comprehension [125] i.e. visual stimulus plays an important role.

Central to the development of this model was *information foraging theory* [126], which theorizes that people adapt their strategies and their environment to maximise extraction of relevant information per unit cost. This adaption process uses a concept known as

information scent which is the imperfect "*perception of the value, cost, or access path of information sources obtained from a proximal cue*" [126].

In the software development environment, cues included names of source-code entities, comments and source file names. [125]. The model predicted that the assimilation strategy chosen [125] e.g. top-down, depended on the cues in the environment.

Other research [94] [125] also recognizes that reading source code involves visual stimulus in the form of the spatial cues/beacons that alert the programmer to important aspects of the code. The speed at which these spatial beacons are identified strongly supports the findings of Altman's [80] simulation and the role that perceptual chunks play in reading code. It is therefore possible to conclude that perceptual learning plays an important part in gaining programming expertise as previously discussed in Section 2.3.1 and in Section 2.7.3.

2.4.5 Cognitive Strategies used to Read and Write Code

The mental model can be said to contain static and dynamic entities [104], where the static entities represent the remembered information and the dynamic entities are the strategies and reasoning process by which the programmer assimilates information from the source code [120]. If the goal is to understand a block of code, the strategy may be to systematically read and understand each line of code while building an increasingly abstract mental representation [104]. A top-down strategy is used when the programmer takes knowledge from the problem domain and maps it to the microstructure of the code [120]. Alternatively, this can be viewed as a process of taking programming plans and the rules of discourse to decompose plans into lower level plans [120, 127]. A bottom-up strategy is where the programmer takes the code statements and chunks or groups them into abstractions, then these abstractions are further chunked and grouped at successively higher levels of abstraction until the mental representation of the program is complete [120]. Finally, an opportunistic strategy views the programmer as being an "*opportunistic processor*" able select the appropriate strategy as required [13, 120].

The strongly supported view [101] is that when reading code the abstract knowledge of program text structures plays the initial organisational role but control flow and procedural relations dominate in the macrostructure representation i.e. text structure knowledge theory dominates in practice (implementation plans). Pennington [101] found that when reading code experts do not apply a kind of mental library of plans to

understand programs from the top-down as suggested by Soloway *et al* [18] and that plan structure knowledge (as opposed to text structure knowledge i.e. implementation plans) does not form the organizing principle for "*memory structures*" [101]. When the code is completely new to a programmer they construct a program from the bottom-up [101] using cues from the code (beacons) [104, 110]. Since they were using a bottom-up strategy (Figure 2-11) Pennington also concluded that the programmers analyzed code first by developing a program model and then subsequently the situation model [120] i.e. first they learnt how the code worked then they related it to the problem being solved.

> "*… the understanding of program control flow and procedures precedes understanding of program functions [purpose or goals]*" [101]

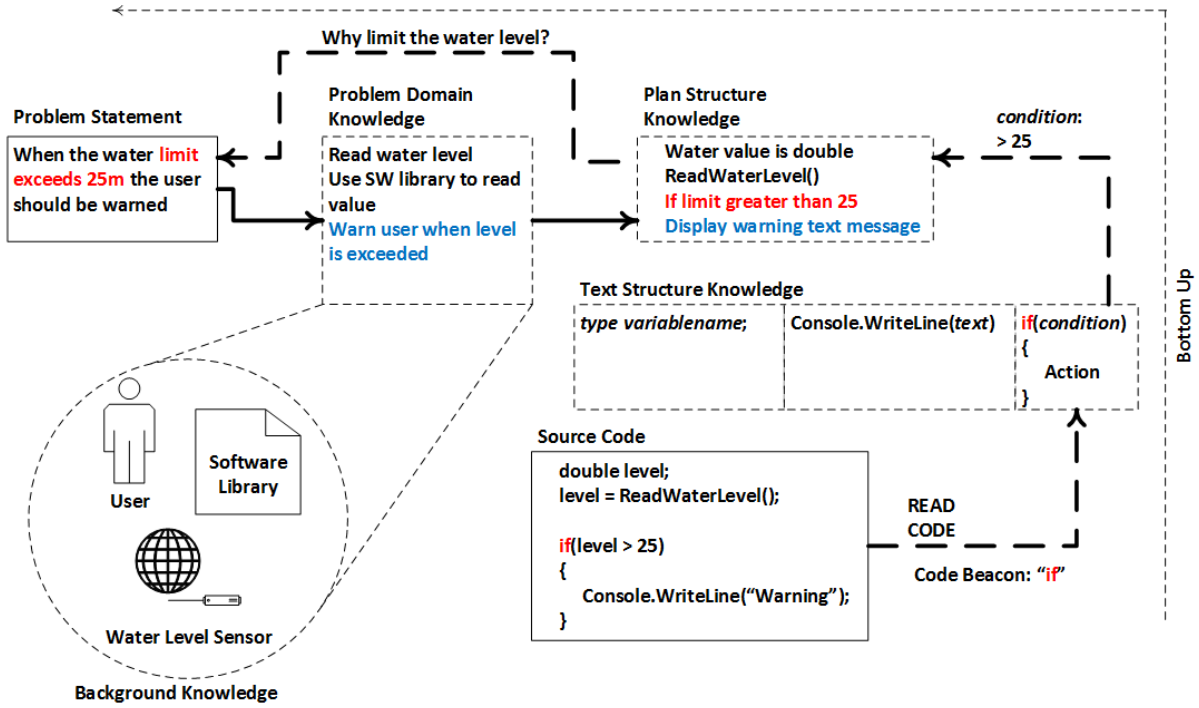


Figure 2-11 Cognitive Strategy for Reading Code

However, the reading strategies of novice and expert programmers differ. Novice programmers tend to read a program sequentially [128, 129], line-by-line, as if they are reading a book. As a result they fail to connect the "*pieces in terms of hierarchical structure, dynamic behaviour, interactions and purpose*" [130]. Thus, a novice programmer's strategy is more bottom-up concentrating on details first and general structure last [130]. By contrast experienced programmers use the control flow method of reading a program, in a top-down fashion following the flow of control of the executing program [130]. For example, they will step into a procedure when they encounter it and

when it returns they will continue to read from the location from which it was called [130].

Actually Pennington introduced the situation model in response to a second study that required the programmers to modify code. After the modification task there was an observable shift to increased comprehension of program function and data flow (Figure 2-12) but at the expense of control flow knowledge [101]. This suggested that programmers cross-referenced the source code with the real-world problem entities. In short, they understood the purpose of code and the data being passed, rather than focusing on an execution sequence of the instruction statements.

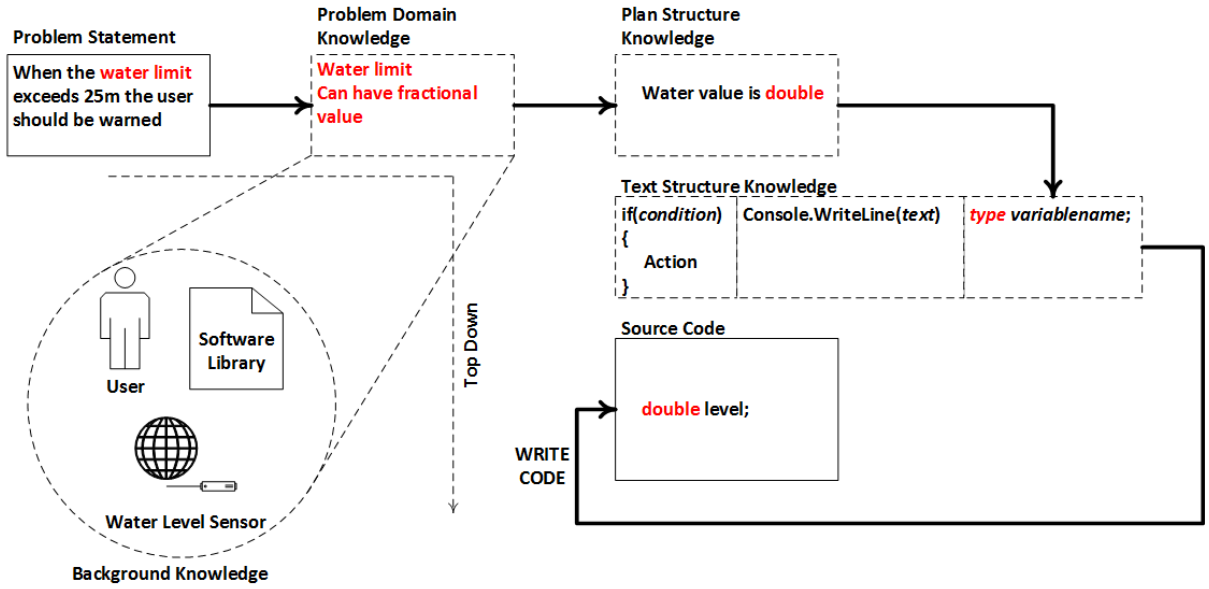


Figure 2-12 Cognitive Strategy Concentrating on Data Flow when Writing Code

As in the program model, when modifying code the situation model is created bottom-up. The situation model is matched with the program model and used to build high-order plans in the program model [104]. *Cross-referencing* [131] is required to relate abstraction levels to each other "*by mapping parts to functional descriptions*" [104]. For example, by identifying the purpose of a code segment (control-flow) and hence clarifying some aspect of the functionality (goal/functional abstraction) of the program. Pennington puts it more succinctly as:

> "*…construction of the situation model depends on construction of the textbase [program model] in the sense that the textbase defines the actions and events that need explaining*" [101]

Cross-referencing is essential to build a mental representation across abstraction levels [104] and must also be related to information foraging [126]. Thus, the situation model relates entities and functions in the problem domain to source language entities [120]. Given that plan knowledge bridges the gap between the problem and program domain, it must play an important role in the cross-referencing strategy [92] during software modification (Figure 2-13). A high level of comprehension was observed when programmers cross-referenced frequently between the program and situation models [120]. The mapping from problem to solution often requires analogical thinking [132].

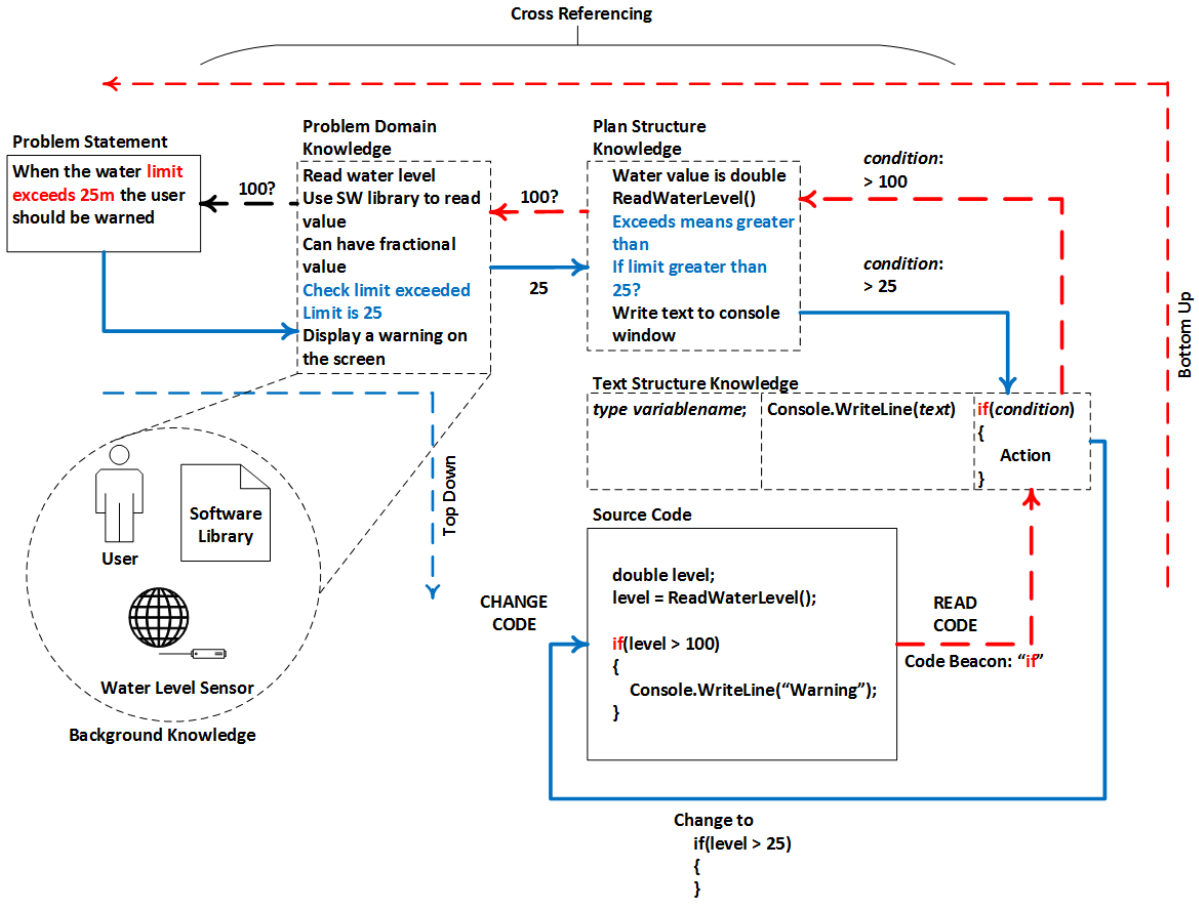


Figure 2-13 Cognitive Strategy for Modifying Code Showing Cross-Referencing Behaviour

This cross-referencing process causes an increased cognitive load that depends on the "*cognitive fit between the mental representations and the external representation*" [133]. Green *et al* [133] also observed that, if a programmer has a mental model of the control flow of a problem then the data flow will be harder to follow; likewise if they are thinking iteratively then recursion will be harder [133]. Unsurprisingly, for novice programmers this "*close tracking*" process is an issue [31].

The need for different strategies when reading and writing code, may account for the mixed results obtained when trying to correlate the ability to read code with the ability to write it. For example, a study by Sheard *et al* [134] using 79 undergraduate and 41 postgraduate programming exam scripts, showed that there was a correlation between ability to read code and the ability to write code. A correlation between reading, tracing and writing code has also been demonstrated by Lopez [135], while other studies have found none[136]. The BRACElet project [137] suggested that:

> "*...students who cannot read a short piece of code and describe it in relational terms are not intellectually well equipped to write similar code. We are not advocating that students must first be taught to read code ..... but we do advocate a mix of reading and writing tasks*" [137]

Given that writing code clearly involves a higher cognitive load, Lister *et al* [4] even suggest that the demands placed on novice programmers should be tailored so that the ability to write programs should only be required of "A" grade students whereas "C" grade students should focus on reading code. This implies that the mental load required to develop a situation model from a problem domain and to cross-reference it with the code being written is too much for some learners. Although reading and understanding code is an important aspect of the learning process, teaching only higher grade students to write code would significantly delay the development of programming skills of many members of the class. A far better approach would be to provide exercises that gradually increase in difficulty and the required cognitive load.

Evidence also suggests that novices [120, 138] have a limited program model focus, that prevents them from "*seeing the forest for the trees*" [137]. Soloway [139] even found that the cognitive load imposed by a complex problem may interfere with learning, even when a solution is eventually achieved.

Von Mayrhauser *et al* [140] found that programmers switch between a number of these comprehension models and therefore proposed an integrated code comprehension model known as the Integrated Metamodel. As an example, a programmer may recognize a beacon for a sorting algorithm, which leads to the hypothesis that something is being sorted causing a jump to the top-down model. They generate sub-goals and search for evidence to support them, but if they find unrecognized code they return to program model building using a bottom-up approach. Similarly, Gilmore concludes:

> *"… the evidence suggests that a programmer's choice of strategy is influenced by his/her knowledge, the programming task, the program representation and the program's complexity "* [92]

However, we can conclude that evidence points to there being two processes required to successfully program. Firstly, program implementation plans related to Pennington's text-based knowledge allow programmers to "see" simple generic software constructs when reading code and provide a kind of mental template when writing code. Thus we can conclude that building up knowledge of these implementation plans is fundamental to learning to program. Mayer [116] recommends a similar process and identifies different types of "memory chunks" required to teach BASIC. Secondly, an understanding of the problem domain must be developed and cross-referenced with the code (text base). This cross-referencing load is significantly higher when writing code, as the programmer must constantly ensure the code being written meets the requirements of the problem. This confirms the importance of good problem solving skills, but the importance of cross-referencing supports the argument for the involvement of an "intermediary" skill that enables a programmer to translate a problem solution to code by recognizing how elements of the problem may be implemented i.e. Pennington's plan structure knowledge. For example, recognizing that storing a set number of values requires an array, while an arbitrary number of values may be stored in a list.

From the grounded theory analysis conducted, Figure 2-14 shows the importance of perceptual learning through the "memory chunking process" and how the "mental plans" memorized by programmers become their understanding of how the code works i.e. they construct a "program model" to recognize code and write code using a "template" like approach. However, the task of reading code and the task of solving a problem and writing the code for the solution are different skills. The latter relies on an understanding of the problem and being able to infer a solution that can be coded. This requires the ability to develop an understanding of the problem domain (the situation model), and the ability to continually cross–reference between it and the program domain i.e. between the program and situation models. This cross-referencing process increases the cognitive load on the programmer. For learning it is important to adjust this cognitive load until the students develop the appropriate knowledge to reduce the problem search process. For most learning activities, this is typically achieved by providing appropriate examples and is known as the worked example effect [141, 142].

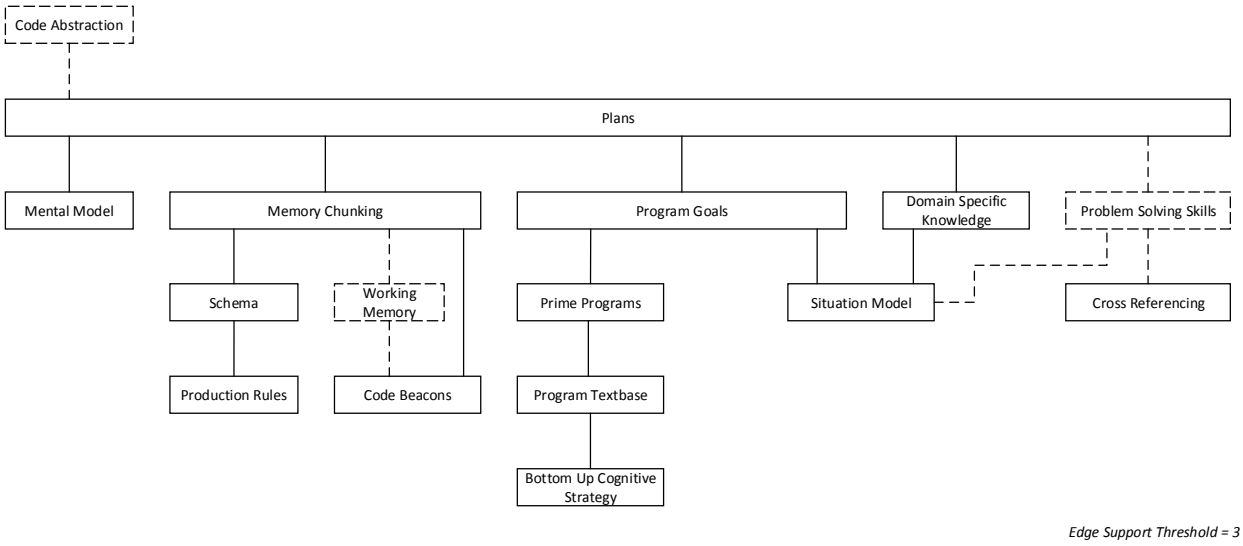


Figure 2-14 Overview of Software Comprehension Concepts

2.4.6 The Nature of Expertise

The differences between novices and experts can be explained in terms of increased knowledge in a particular domain.

> "*… peoples level of expertise in such domains, be it in chess, mathematics or physics plays a crucial role in how they represent problems and search for solutions*" [24]

Experts are able to ignore surface dissimilarities and concentrate on structural similarities allowing them to quickly identify a solution. In other words, they have a knowledge that is highly organized around domain principles so they can rapidly extract the solution relevant structure. Whereas novices tend to be bound to surface features of the problems that may be irrelevant to the solution [24]. It appears that experts are able to "chunk" information into meaningful blocks which they can quickly recall and apply later [24] known as perceptual learning. This difference has been found in numerous domains including chess [143] and computer programming [98].

> "*In effect, experts often see the solutions that novices have yet to compute*" [24]

Experts concentrate on the purpose of the code (*what* it does) and form abstract representations, while novice programmers focus on *how* the code works and form concrete representations [137, 144]. An expert uses larger more abstract chunks than a novice and their schemata are larger and better connected [122]. Given novice programmers have fewer, smaller, more concrete and fragile schemata than experts [122], they are not as effective or may not evaluate their code as it is being implemented because they have fewer choices and fewer design rules to draw on [122].

Fix *et al* [145] looked at the difference between the mental representations of code developed by novices and experts . In this study 20 expert programmers were asked 11 questions designed to assess their comprehension of a Pascal program. The conclusion was that experts' knowledge exhibits five abstract characteristics not seen in novice programmers, which are:

1. It is hierarchical and multilayer: When debugging, experts read the code in order of execution, starting with the main program, then the procedures called by the main program, then the procedures called by the procedures and so on until the last procedure is read. [128]. Thus, they build a hierarchical understanding of the program [128]. Experts are more able to build this hierarchical structure [145].
2. It contains explicit mappings between the layers: Experts are better able link specific segments of the source code to the program goals.
3. It is founded on the recognition of basic patterns: Experts are better at using complex programming plans [101] in developing code than novices who are restricted to simpler plans.
4. It is well connected internally: Code for implementing a plan or goal may be spread throughout the program, so experts tend to pay special attention to this code e.g. by designing good interfaces between modules. Experts tend to concentrate on and remember this detail more than novices.
5. It is well grounded in the program text: Experts are better able to recall locations of code and finding information they have seen before e.g. when debugging [128].

2.4.6.1 The Worked Example Effect

Learners who study worked-examples always perform better than those who have been required to learn by solving problems [59]. Termed the worked-example effect, this phenomenon was first demonstrated by Sweller and Cooper [141, 142] but has subsequently been demonstrated repeatedly for a variety of learners [146-148], a variety of materials [59] and especially in domains where algorithmic solutions are applied [149] e.g. programming. Empirical evidence has demonstrated that this is most important during the initial skill acquisition stages [150].

> "*For novices, studying worked examples seems invariably superior to discovering or constructing a solution to a problem*" [59]

For example, Sweller *et al* [151] conducted research into the role of worked examples in practice-based problem solving and found that students presented with practical exercises used novice approaches such as trial and error. However, if the students were given worked examples prior to such problem solving exercises then they used more efficient strategies and focused more on the structural aspects of the problems.

Studying worked examples reduces the cognitive load on working memory by limiting the scope of the problem and effectively directing attention (i.e. working memory resources) to the understanding of the rules and structure of the solution [59]. This is the basis of acquiring problem solving schema [152]. Obviously it is assumed that these worked examples do not require heavy cognitive load [59], and are designed to reduce intrinsic and extraneous cognitive loads to allow the learner to concentrate on activities that develop schemata [153] i.e. maximize the germane cognitive load [154]. Although, studies [154-156] have shown learners do not spontaneously develop such germane cognitive activities simply by reducing extraneous cognitive load.

As a learner gains expertise the advantages of worked examples diminish [59] because they have to integrate and cross-reference redundant information with their existing knowledge schemata [157].

> "*…when learners are sufficiently experienced so that studying a worked example is, for them, a redundant activity that increases working memory load compared to generating a known solution*" [59]

This reduction in cognitive load as expertise increases is known as the expertise reversal effect [157]. It has also been shown that this reversal effect is due to the learner's cognitive load differences rather than any overall motivation differences [158]. Since novice programmers lack the necessary knowledge to prevent unproductive problem solving searching, they require more guidance to reduce this cognitive load to give them time to develop the required knowledge and understanding. An interesting result of this research is that the level of guidance provided should be tailored to the level of expertise of the learner [157], which suggests that it should be reduced as the learner's expertise increases. This reduction of the support provided to the learner is known as fading [159] and is integral to scaffolded learning which is discussed in Section 2.8.4.

2.5 Problem Solving Skills

Gestalt psychologists investigated problem representation while Newell and Simon conducted research with various collaborators into how searching in the problem space may work [24]. It is now recognized that problem solving and insight is an integration of the ideas of Gestalt and Newell & Simon's [24]. These areas of research are too large for a detailed study to be included but a brief discussion of some of the concepts is warranted as programming problems are still "problems" that must be solved.

Newell and Simon [160] developed theories of problem solving based on the parallels between human and artificial intelligence. The solver's representation of the task is known as the "*problem space*", and the problem is solved by searching for a path through it that joins the initial state to the goal state [24].

This problem space consists of "*1. A set of knowledge states (the initial state, the goal state, and all possible intermediate states), 2. A set of operators that allow movement from one knowledge state to another, 3. A set of constraints and 4. Local information about the path one is taking through the space (e.g. the current knowledge state and how one got there)* " [24]

They discovered that solver's reduced the search process by relying on a number of *heuristic* strategies. Of these heuristics, the most important was mean-end analysis [24] which describes the process by which people devise a solution to a problem by establishing sub-goals to achieve the final goal [24]. Unfortunately, these heuristics are very specialized and people only rely on them until they gain experience and/or sufficient knowledge [24]. Therefore, we will focus on the work of Gestalt.

As an example of a Gestalt phenomenon, functional fixedness refers to the tendency to see an object as being used to fulfill a particular purpose and ignoring the properties that might allow it to be used for a dissimilar purpose. This problem has been noted in mathematics [161], programming [31, 162] and software design [163]. In McCaffrey's opinion [164], functional fixedness presents an enormous barrier to coming up with new ideas. Wertheimer [161] contrasted students that displayed "*reproductive thinking*" with those that displayed "*productive thinking*" and the ability to deduce the general principle and apply it in a different scenario. This is the basis of abstract thinking. The cognitive process by which a person learns information about one particular object or scenario and

then applies it to another is known as "*analogy*" [165]. Gick *et al* [37] studied the use of analogous transfer of knowledge, that is, the use of analogy to teach abstract thinking in solving problems. In psychology, the failure to transfer the general principle as described above, is known as "*inert knowledge*" [51]. The direct use of analogy in teaching has been applied to both the teaching of mathematics [166, 167], the *"notional machine*" behind the execution of code [168] [30, 169], parallel programming [170], algorithms [165] and variables [171] with some success. As Muller [28] notes:

> "*Analogical reasoning is an essential practice in the computer science domain; software solutions for recurring algorithmic and design problems are developed and utilized in various contexts. Therefore, realizing similarities between problems and reuse of previously solved problems are mandatory*"

One potential problem was the difficulty of providing suitably familiar analogies that the students can identify with [171, 172]. Lui *et al* [7] also note that:

> "*Computer programming is all fabricated that finds few parallels in the physical world and we believe that most analogies could potentially cause problems in some students*"

They do not specifically provide evidence for this claim, but instead of analogy their solution is to use multiple examples to refine and test the students' mental models [7]. Another note of caution has to be raised, analogous transfer allows abstract concepts to be taught through concrete examples making formal reasoning easier, but it is not a replacement for a rigorous formal approach to teaching programming [165].

Before continuing, a number of other terms used in psychology need to be defined. The "source problem" is an example previously seen by the solver and the problem to be solved is known as the "target problem". Mapping refers to finding corresponding components of one body of knowledge with components of another [173]. This mapping may take place between components (concepts) at the same level of abstraction i.e. when comparing the human heart with a water pump [173] or between a concrete component (concept) and a general schema (mental model built by the solver) i.e. the human heart and the abstract concept of a pump [173]. It may also occur during induction of schemata from examples i.e. learning the abstract sense of a pump by comparing an example of a heart and a water pump [173]. The source problem may also be referred to as a concrete problem/example [90], to signify the specific nature of the problem as opposed to the abstract solution required.

Holyoak [174] characterized analogous (analogical) problem solving as having four basic steps:

1. Constructing mental representations of the source and the target problems
2. Selecting the source as a potentially relevant analog to the target
3. Mapping the components of the source and target
4. Extending the mapping to generate a solution to the target

These steps may interact in many ways and it is not a strictly linear process, so some preliminary mapping might be required before selection. Mapping is also referred to as *establishing a structural alignment* [175] between the source and target problems by which *inferences* can be made [176]. The resulting alignment consists of an explicit set of correspondences between the sets of components (representational elements) of the analogs, with an emphasis on matching relational predicates (or relations) e.g. the goal. Once aligned, candidate inferences can be made between the implicit schema extracted from the source (also known as the *base* [175]) and the target problem.

In this sense, programming worked examples are intended to provide the novice programmer with the opportunity to learn the abstract principle from a number of concrete examples of its application.

Gick *et al* [173] found that the best approach to ensuring that the learner was able to abstract the underlying principle or schema (mental model) was to present at least two analogs. Induction of the schema could also be significantly improved by explicitly presenting the abstract solution. Often programming is taught by presenting a number of worked examples that require the use of one or more software constructs, with the assumption that the construct will be learnt and that knowledge will be transferred to other problems. For example, it is common for a "for loop" to be presented as a code snippet as shown in Figure 2-15.

```
int i;
for(i = 0; i < 10; i++)
{
    printf("The index value is %i\r\n", i);
}
```

Figure 2-15 Code Snippet for a for-loop

If we take the view that analogous transfer of knowledge is just the process of learning an abstract principle, then the work of Gick *et al* [173] suggests that the process of presenting a single example of using a software construct may not be sufficient.

In teaching by analogy, Spencer *et al* [177] identified a potential problem. If there is a delay or a change of context between the presentation of the examples and the problem to be solved then the learner may fail to induce the abstract principle. By implication [174], the test subjects established a relationship between the example and the problem simply because one was immediately followed by the other (known as the *demand characteristic*). However, these findings have been contradicted [174].

When completing examples or solving problems, the common abstract principle must be identified and applied. To determine the abstraction, differences between the worked examples or between the worked examples and the target problem must be ignored. In this context, there are two types of differences, "*surface dissimilarity*" and "*structural dissimilarity*" [174]. The term "structure" is a reference to underlying abstract principle. Therefore, structural dissimilarity refers to the differences between the contexts of the examples which seems to imply that a different principle is involved ("*an alteration in the causal relations in the two situations*" [174]) giving rise to a structure-violating difference [37]. For example, displaying the names of five people and displaying a user specified number of product names may suggest two different types of loop. Whereas, the term "surface" refers to the more superficial specific wording of the problem statement itself [37]. Thus, surface dissimilarity describes a difference that has no effect on the application of the principle ("*a change in a feature that does not influence goal attainment*" [174]) and is a structure-preserving difference [37]. For example, counting five people and counting five products both suggest a for-loop with a fixed count of five. Knowledge transfer can be significantly impaired if either the surface or the structural similarity are reduced [174]. Good surface similarity leads to moderately to highly elaborate solutions while surface dissimilarity typically leads to poorly elaborated solutions [178]. A number of studies have shown that learners tend to focus on the superficial details of a problem i.e. structure dissimilarity [179-181]. Some research [176] has shown that "*undeleted*" [182] surface dissimilarities are sometimes used to recall a previously learned solution and may support or compete with structural similarity. Mistakes are made when mismatches in superficial surface aspects win over solution-

relevant structural similarities. Without hints, the transfer of knowledge can be limited [183]. Although hints can be provided to improve surface similarity, they have no effect on improving structural similarity [174]. Therefore, it is very important to ensure strong structural similarity when constructing analogs [183] i.e. *structural isomorphism* [174].

Chi *et al* [180] investigated the role of self-generated explanation in "good" and "poor" students studying worked examples in mechanics. They discovered that the students:

> "….. *representations of the principles and other declarative knowledge introduced in the text will differ depending on the degree to which their understanding of the principles is enhanced during their studying of examples ….*"

In other words, the "good" students were able to focus on structural rather than surface features of the examples. That is to say, for well-structured domains like mathematics and physics learning from worked examples is very important but only for students that can explain the rationale for each step in the solution, the *"self-explanation effect".* [180, 184]

In a teaching context, the similarity between the learning and target contexts is defined as *near* and *far transfer* [185]. Near transfer refers to the use of knowledge in a context which is quite similar to the learning context. Far transfer occurs between contexts which are quite different [186]. The difficulties of transferring knowledge may be caused simply because skill and knowledge are specialised or localised to a particular context [74]. For example, Soloway *et al* [127] found that expert programmers had "*strong expectations about what a program should look like*" and when these expectations were not met, even in quite innocuous ways, their performance dropped drastically. This transfer of knowledge is often particularly difficult for novice programmers, their knowledge of program instructions (e.g. if and while) can remain inert during programming even when there is hardly any gap to transfer across [31]. Dunbar concluded that learners must be given extensive training, examples or hints [179]. Thus we can conclude that to teach abstract concepts a "scaffolded learning" approach is required.

Perkins *et al* [74] emphasized the classification of the transfer process itself [187]. They [74] defined two types of knowledge transfer known as "*low-road*" and "*high-road*" that roughly equate to the transfer of skill versus the transfer of knowledge from one context to another. Low-road transfer reflects the automatic triggering of well-practiced routines where there is a great deal of perceptual similarity between the learning context and the

problem context [74]. High-road transfer requires conscious reflective thinking to abstract the skill or knowledge from the learning context to allow it to be applied in another [74]. Thus, surface and structural similarities are sought [74, 188]. For students, identifying the abstract principle is difficult [37]. Low-road knowledge transfer problems arise when there is little surface similarity between the contexts to allow the relationship to be automatically recognized. As for high-road transfer, when comparing contexts, the student may be neither able to determine the abstract principle nor apply it because they are unable to break the "*patterns free of their accidental associations*" [74]. For example, in programming, the novice programmer starts with little programming experience so it is not a natural or automatic skill [74]. High-road transfer requires the abstraction of general problem solving principles from a programming context and relating them to the "real-world" problem at hand [74]. However, most programming courses focus on building programming skill and make little effort to build bridges between the programming and problem domain [74].

Perkins *et al* [74] suggest two techniques for enabling transfer, "*hugging*" and "*bridging*". "Hugging" means teaching low-road transfer by minimizing the surface dissimilarities between the learning and initial problem contexts. "Bridging" means providing conditions that help to mediate the process of abstraction and connection between contexts, so that the student is not required to spontaneously achieve transfer.

Bassock *et al* [182] investigated the role that interpretation plays in analogy and problem solving. They discovered that people have an interpretive bias, and they concluded that structural inferences triggered by object attributes are quite common in problem solving and are likely to affect transfer.

> "*In general , our claim is that when people are presented with a problem involving several entities, they reason about the situation described in the problem using knowledge about the way in which these entities typically interact with each other (e.g. children eat cakes, cakes do not each children). As a result, they abstract interpreted structures that include the reasons why certain entities play certain structural roles.*" [182]

Interestingly, they also noted a similar bias in a mathematical problem generation study [189], and that the process of "*semantic alignment*" leads to selective and sensible application of abstract formal knowledge e.g. dividing apples among baskets makes more sense than dividing baskets among apples. However, they also note:

> "*….our results strongly suggest that when application of formal rules conflicts with people's semantic and pragmatic knowledge, people who have good understanding of formal rules may prefer arriving at logically invalid but reasonable conclusions to arriving at valid but anomalous conclusions*" [189]

Chrysikou [190] describes another possible mechanism by which a solver's domain specific knowledge may be brought to bear on the solution through ad-hoc categorization. In this context, a category is a set of entities or examples from the solver's own experience that can be selected by concept. A concept refers to the information in working memory that is used to represent a category during the analysis of a problem. Chrysikou hypothesized that when faced with a problem, a solver categorizes the problem elements then constructs a set of goal-derived categories. These goal-derived categories may be either well-established from previous experience or ad-hoc. "Ad-hoc" in the sense that these categories are dynamically created "on the spot", from a combination of elements from well-established or taxonomic categories learned from exemplars from previous experience. By training solvers to spot alternative goal-derived categories, solvers can overcome the tendency to avoid transferring strategies from one task to another without additional explicit instruction. Chrysikou proposes that by putting the subjects into the right state of mind for creative problem solving, this method can overcome functional fixedness and thus boost creativity [191].

2.6 The Relationship between Problem Solving and Programming

From the grounded theory analysis, the relationship between problem solving skills and programming is shown in Figure 2-16. Drawing on the previous discussion of software comprehension, the four key concepts are abstraction, domain specific knowledge, plans/memory chunking (perceptual learning) and the need to continually cross reference between these concepts. This figure also illustrates that a characteristic of expertise is the development of domain specific knowledge that novice programmers initially lack and the problems that novice programmers often face when analyzing a problem statement. Namely, being fixated by the concrete surface dissimilarities instead of focusing on the structural similarities of the abstract principle required to solve the problem. Novice programmers also need to learn the prime programs [101] (software constructs) until they become automatic and almost unconsciously recognized (perceptual learning) when reading and writing code, minimizing the cognitive load imposed in constructing the program model and allowing more attention to be spent on cross referencing when writing code.

"The tendency to focus on details such as the syntax structure manipulation when writing programs is a hindering factor in schema formation. Often, the implementation of an algorithm takes attention away from the special ideas that one should learn from an algorithm development process and may prevent the formation of a schema" [28]

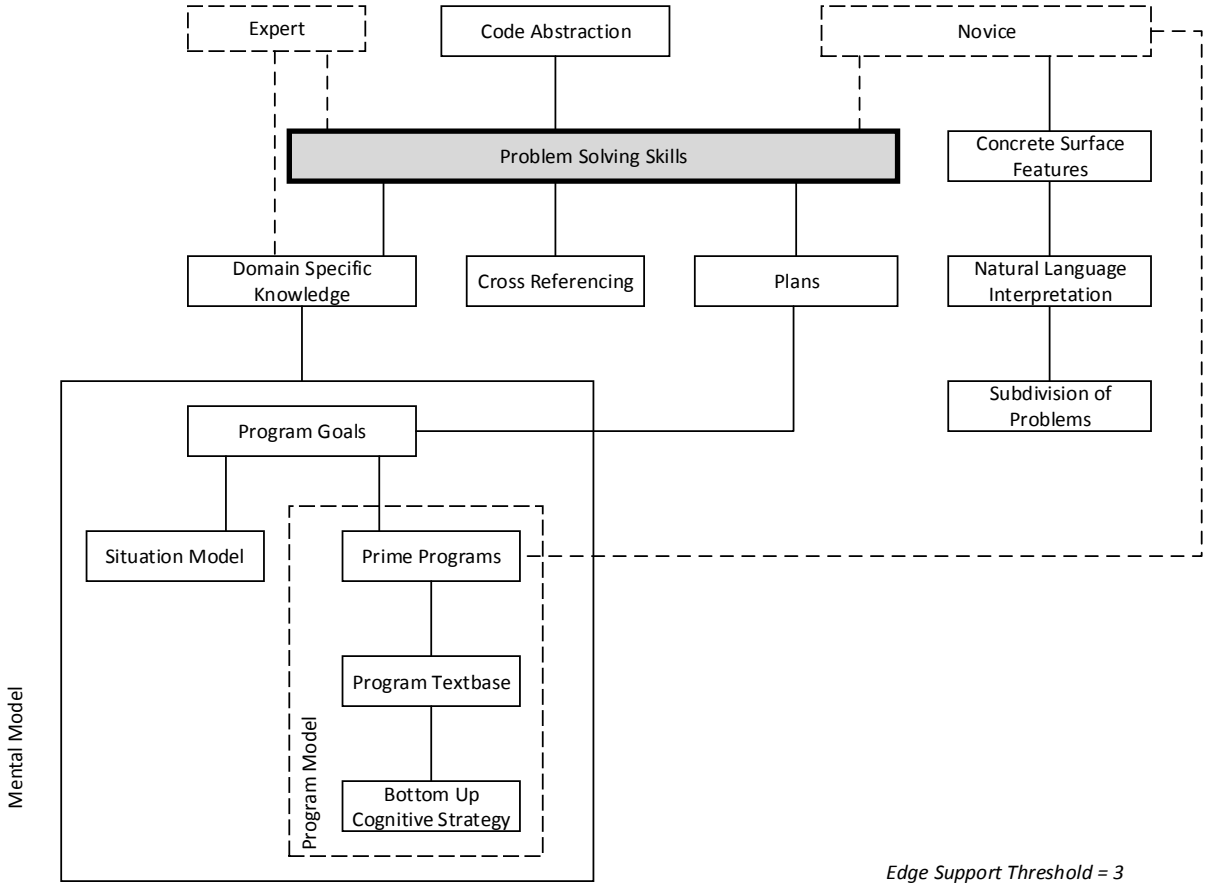


Figure 2-16 Overview of Problem Solving Skills and Software Comprehension Concepts

When defining problem solving in the context of programming, it is often described as the process of taking a mental plan which is in familiar terms and converting it into a program [192], that is to say there is a mapping between the problem domain and the program domain [110, 133]. Green *et al* [133] define a powerful corollary:

"...it is not easy to deal with entities in the program domain that do not have corresponding entities in the problem domain."

Pane *et al* [192] studied the language and structure of non-programmers' solutions to programming problems and noted that the "*mismatch between the way programmers*

think about a solution and the way it must be expressed in a programming language" [192] creates difficulties for both novice and experienced programmers alike [192].

In reviewing the work of Hoc *et al* [193] and Green *et al* [133], Pane *et al* [192] note that many bugs and misunderstandings are caused by a poor "*closeness of mapping*" between these domains i.e. when the distance between these domains is too great. There are a number of separate and distinct issues that may define this mapping process including the notional machine, the interpretation of the problem statement and the nature of problem solving in a programming context.

2.6.1 The Notional Machine

Ben-Ari [22] suggests that a first year computer science student has no effective model of a computer, at best this model is "*is limited to the grossly anthropomorphic 'giant brain'*" [22] or the idea that there is a 'hidden mind' within a programming language that has some intelligence [194].

Sleeman *et al* [195] found "*even after a full semester of Pascal students' knowledge of the conceptual machine underlying Pascal can be very fuzzy*" [22].

For example, the concept of a variable can be difficult for students to learn [195]. Unlike mathematical variables, programming variables have a type and misunderstandings arise when novices attempt to treat program variables like algebraic variables [171, 196]. In addition, novice programmers find it difficult to identify the type from a value given in a natural language problem [197].

Although an understanding of the machine level i.e. the actual hardware of the computer, is not required by a novice programmer [116] a misunderstanding of the limitations it imposes on code execution can be a source of confusion. Thus, an abstract model of the general functions and features of a computer is required, which Schulte *et al* termed the notional machine [120]. Mayer[116] refers to this as the "*transaction level*" of knowledge, where a transaction represents some program "operation" that is applied to an "object" at a some "general location". Here, operations include MOVE, FIND, CREATE and DESTROY, and are applied to an "object" specified as a number, pointer or program line at a "general location" such as a memory address, a file or the screen. Knowledge of the notional machine can be taught by analogy [116], or by simplifying the notional machine and making its processes and "parts" more visible through some form of simulation [168]

[198]. However, care must be taken when teaching by analogy. For example, to teach variables we may use the box metaphor to help visualise the concept, but students may come to believe that a variable can simultaneously contain two values [22]. The students have constructed a consistent concept, it just happens to be non-viable for successful programming [22].

2.6.2 The Situation Model and the Problem Statement

Perhaps the greatest problems faced by novice programmers is building the situation model i.e. extracting the pertinent information required from the problem statement written in natural language.

What can be said about the relative difficulty of natural language problems? This is obviously a difficult question to answer, but if the challenge of a problem is in "building a picture" of the potential solution then perhaps the difficulty is reflected in the mental model required. Johnson-Laird [199] investigated people's competence in deductive reasoning and proposed that building a mental model took three steps:

1. They construct a mental model assuming the premises of their argument are true
2. They develop an informative conclusion that is true based on the model they have constructed
3. They check for an alternative model where the same conclusion is generally found to be false. If no alternative model is found then the conclusion is accepted.

In a later investigation [200], it was found that the harder the task was the greater the number of such models that needed to be constructed in order to obtain the correct conclusion. Thus, in solving a programming task the more mental models that a programmer has to construct then the more difficult it can be said to complete [201].

In reading a problem statement, two difficulties faced by novice programmers are [193]:

1. A shift from value to variable processing
2. Elaboration of a representation of the procedure control structures of which beginners are not necessarily aware in usual problem solving situations. Mayer *et al* [75] identified two key skills:
    a. translate a word problem into an equation or answer (*problem translation*)
    b. predict the outcome of a procedure or set of directions that are stated in English (*procedural comprehension*)

When developing a code, the programmer must identify any quantities or equations required from the natural language description of the problem which can be difficult [75] and these quantities need to be represented as variables [193]. Secondly, a procedure expressed as a natural language statement must be identified from the text and translated to a programmable form that can be executed. Essentially, programming requires the execution of a series of instructions that implement such a procedure. Mayer *et al* [75] demonstrated that a related skill is procedural comprehension, the ability to follow a series of steps in natural language to determine their outcome. To demonstrate its importance in learning to program, they conducted an experiment using two groups of 23 randomly selected students, in which only one group was initially given pre-training of skills consisting of predicting the output of 60 problems stated in English e.g.

1. Put the number 3 in Box A
2. Add 5 to the number in Box A; put the result in Box B
3. Write down the number from Box B
4. Stop working on this.

Both groups were then asked to read a manual on BASIC programming and to predict the output of 100 BASIC example problems. For example, a typical problem might read:

```
10 LET A = 3
20 LET B =  A + 5
30 PRINT B
40 END
```

It was found that the students that received the initial training learned BASIC faster than those that did not.

> "*A straightforward conclusion is that procedure comprehension is a component skill in learning BASIC, and this skill can be taught to novices*" [75]

Novice programmers face a range of additional problems including failing to recognise the important elements of the problem statement by trying to directly translate text from a natural language into a programming language [202]. They introduce distortions in their programming syntax when their programming knowledge is lacking [193], and these errors reveal semantic difficulties in "*transforming well-known contents into quite different contents expressible in the new language*" [193]. This phenomenon is also seen in translation between natural languages [193]. Likewise, they will often attempt to use natural language semantics when translating natural language specifications into a programming language [192, 203]. However, a programming language defines constructs in ways that are not compatible with natural language constructs [192]. For example, 'then' is interpreted as 'afterwards' instead of 'in these conditions' [204].

Another area where natural language can give rise to problems is in Boolean logic. Pane *et al* [192] found that novice programmers tended to create mutually exclusive sets of rules or used a general case followed by an exception. The Boolean AND was used instead of OR and NOT was treated with lower precedence. Spohrer *et al* [117] also noted that natural language lead to novice programmers into making mistakes. For example, an English statement such as "Retry action if event is not A, B or C" may be interpreted as using a Boolean OR but the "not" here should convert this into an AND operation. Novices find the intended scope of the NOT operator to be ambiguous [205]. A study was also conducted to investigate logical thinking and formal reasoning [206], based on the work of Epp [207] and Herman *et al* [208]. In natural language OR is used exclusively or inclusively but in mathematics it is always used inclusively [207]. A similar issue arises when using *if-then* and quantified statements [207]. Herman *et al* [208] noted that:

> "*...the ability to translate natural language specification to Boolean expressions...is both important and difficult for students to learn*"

2.6.3 The Program Model and Problem Solving

Perkins *et al* [209] define the term *fragile knowledge* to describe the incomplete or fragmented knowledge a student possesses that is not sufficient to allow them to produce a complete solution [31]. They subdivide fragile knowledge into partial, inert [51], misplaced and conglomerated knowledge [31]. *Partial knowledge* refers to fragmented information caused either by the student having never been given the

opportunity to learn the missing information or because they have forgotten it [31]. *Inert knowledge* is information the student already possesses but fails to recall [31, 51]. *Misplaced knowledge* designates *"circumstances where a student imports command structures appropriate to some contexts into a line of code where they do not belong*" [31]. *Conglomerated knowledge* signifies situations where students join disparate elements together in the code in a "*syntactically or semantically anomalous way in an attempt to provide the computer with the information it needs*" [31]. Perkins *et al* [31] concluded that novice programmers difficulties were characterized by a "*fragile knowledge exacerbating a shortfall in elementary problem solving strategies*". Therefore, programming should not be viewed as an opportunity to develop general problem solving skills since these skills are required to program [92].

In programming, inert knowledge is in part reflected by a failure by the student to apply a "*critical filter*" to eliminate candidate solutions [31] and mistaken strategies [92]. A typical novice programmer's mistaken strategy is to insist on using syntactic features taught recently when simpler structures taught in the past would suffice [92]. This results in a "*neglected strategy*" [31] i.e. a failure to apply a known general problem solving strategy. However, it must also be recognised that novice programmers encounter problems when the solution requires the use of programming constructs that they have never seen [193]. A programmer must read a program to determine what it is doing while modifying it, known as "*close tracking*" [31], to apply a critical filter to the code and identify the required strategy [31]. A novice programmer's fragile knowledge can prevent this process [31] resulting in inert knowledge [92]. When a student gains a skill it tends to be bound to the initial learning context and they are unlikely to transfer it to a new context on their own [31] i.e. novices tend to be bound to surface features of the problems that may be irrelevant to the solution [24]. Clearly this relates to structural similarity [174] and analogous transfer of knowledge [37]. One technique to enable this transfer is for the student to use self-monitoring strategies [31]. Students can also learn to deploy skills and knowledge if they are explicitly supported or "*scaffolded*" [210] i.e. scaffolded learning.

2.6.4 Divide and Conquer

In solving a large problem one approach is to decompose it into a set of sub-problems [211]. The term *decomposition* refers to two different concepts, the *process* of sub-dividing the problem into more "*manageable units*" [211] and the *product* of this process. For example, a programmer may hack the code without following any structured

programming methods and then re-arrange it so that it looks like it was developed by following structured design principles. In this case, the *product* (the program) may be properly decomposed but an undesirable decomposition *process* [211] was used. For Berendsen *et al* [211], every sub-problem has a specific goal and a number of standard solutions or plans that can be used to solve it. The plans they refer to are clearly program implementation plans [18, 104] as they "*consist of lines of code which belong together to achieve a particular goal*".

2.7 Taxonomies of learning behaviours

2.7.1 Bloom's Taxonomy

Educational psychologists succeed in classifying the thought process in a number of dynamic levels (or categories) referred to as the "Cognitive Domain of Bloom's taxonomy of Educational Objectives" [212] as summarized in Table 2-1.

KNOWLEDGE	To acquire, to recall, to identify, to recognize (knowledge; of specifics, of dealing with specifics)(knowledge of universals and abstractions)
COMPREHENSION	Translation, interpretation, extrapolation
APPLICATION	To apply, to relate, to transfer, to use
ANALYSIS	To discriminate, to distinguish, to organize
SYNTHESIS	To constitute, to combine, to specify, to propose
EVALUATION	To validate, to argue, to appraise, to reconsider

Table 2-1 Blooms Taxonomy

The complexity increases and becomes more abstract [213] as the learner progresses through the levels from gaining Knowledge to being able to evaluate that knowledge. In 2001, the taxonomy was revised [42] by a team of cognitive psychologists and a number of changes were made to the original model. These changes were made to provide a better fit to learning outcomes which are now framed in terms of subject matter and a description of what will be achieved with or by modifying that content [213].

In the revised taxonomy, the noun and verb aspects were divided into a knowledge dimension and the verbs formed the basis of a cognitive dimension. The cognitive dimension table is very similar to the original taxonomy table, but with the levels renamed using verbs so "Comprehension" becomes "Understanding". "Synthesis" was renamed "Creating" and exchanged places with "Evaluating".

Table 2-2 illustrates this revised taxonomy, including relevant quotations [42] and characteristics.

REMEMBERING	"*retrieving relevant knowledge from long-term memory*" Recognizing, Recalling
UNDERSTANDING	"*determining the meaning of instructional messages, including oral, written and graphic communication*" Interpreting, Exemplifying, Classifying, Summarizing, Inferring, Comparing, Explaining
APPLYING	"*carrying out or using a procedure in a given situation*" Executing, Implementing
ANALYZING	"*breaking material into its constituent parts and determining how the parts relate to one another and to an overall structure or purpose*" Differentiating, Organizing, Attributing
EVALUATING	"*making judgements based on criteria and standards*" Checking, Critiquing
CREATING	"*putting elements together to form a coherent or functional whole; reorganizing elements into a new pattern or structure*" Generating, Planning, Producing

Table 2-2 Revised Blooms Taxonomy

In general, as in the original taxonomy, the layers increases in complexity as you move from "Remembering" to "Creating", although in the revised taxonomy there is some overlap as the scope of some categories has increased [213]. It should be noted that there is a sequential progression between the levels, from remembering to understanding, to applying, to analysing, to evaluating and finally to creating. If you are unable to remember any information there is nothing to understand and without understanding there can be no effective application of that knowledge to obtain the desired result. With no results there is nothing to analyse or evaluate and since nothing of consequence is learnt it becomes impossible to create new ideas or solutions. The levels are heavily interrelated and interdependent in this way.

Applying this to programming, a learner is required to "Remember" the software constructs and the programming keywords (syntax) and "Understand" how they can be used to "Create" a program. Scott [214] investigated using Bloom's taxonomy for constructing programming tests, but hypothesized that many problems experienced by students are caused by programming being taught through demonstration which is a comprehension level activity i.e. the lecturer provides an example that the student copies. A program assignment that requires a student to write a program, he argues, is a synthesis level activity that falls under the top two most complex levels of the taxonomy. Thus, the bimodal (or double bump) distribution often seen in the frequency versus test scores graph is caused because many students can answer questions at the lower levels of taxonomy while only a smaller percentage are able to answer them successfully at the

higher levels. He therefore recommends that instruction should start at the lower level before moving on to the higher level.

> "*In most learning there are "novices" and "experts" as well as the spectrum in-between. The critical differences between the expert and novice, given the same innate ability, is knowledge base, the operational level of cognition and the transition time from one level of cognition to the other.*" [215]

However, research by Lahtinen [216] suggests that students may perform quite well at high levels of the taxonomy even when they have problems at the lower levels. Thus, for programming the sequential progression between the Bloom's levels may not hold true.

Although the final objective of teaching programming is that all students will be able to write programs, Scott [214] recommends that tests be structured to assess all the levels of the taxonomy. Students that progress faster should be given credit for doing so, but well-structured tests allow the lecturer to monitor the progress of all students. Lister *et al* [4] go even further and suggest that programming assignments should not be marked according to a norm-referenced marking scheme but according to a criterion-referenced grading scheme where each grade is associated with explicit criteria based on Bloom's taxonomy. This would prevent weaker students that do not necessarily reach a competent programming standard from failing too soon and may prevent stronger students from feeling insufficiently challenged. Although these approaches may reduce initial failure rates, they fail to address the fundamental problem which is providing sufficient graduated exercises to nurture and support the development of programming skills. In effect, changing the marking scheme can be seen as a way of disguising or hiding the learning difficulties, rather than improving teaching to better support learning.

2.7.2 Structure of Observed Learning Outcomes (SOLO) Taxonomy

An alternative to Bloom's taxonomy is the Structure of Observed Learning Outcomes (SOLO) taxonomy [217], which "*represents a more qualitative way to classify cognitive processes*" [218]. It defines five levels (Table 2-3) and as in Bloom's, the cognitive load increases as one level builds on another. Depending on the course, the students may only need to complete a certain number of the stages.

Prestructural	students are simply acquiring elements of unconnected information, which have no organisation and make no sense
Unistructural	obvious connections are made, but their significance is not grasped
Multistructural	a number of connections may be made, but the meta-connections between them are missed, as is their significance for the whole
Relational	the student is now able to appreciate the significance of the parts in relation to the whole
Extended Abstract Level	the student is making connections not only within the given subject area, but also beyond it, able to generalise and transfer the principles and ideas underlying the specific instance

Table 2-3 The SOLO Taxonomy

As the student progresses through these levels [217, 219], their *cognitive capacity* increases so that more information must be extracted and applied. There is an increasing need for the student to *relate* the content to the intended results and it becomes more important to reach *consistent* conclusions that bring the results to a *closure*. Originally, the levels of SOLO were based on the age of the learner with the lowest level for the youngest learner. However, Shuhidan *et al* [218] argue that since all learning is about acquiring new knowledge, the levels of cognition can be used to describe this process regardless of age.

The SOLO taxonomy is not a model of cognitive development [137], but it can be used to analyse or develop assessment strategies that advocate a mix of assessment exercises [137]. SOLO represents a more qualitative way to classify cognitive processes [218]. As part of the BRACElet project [137], 108 students from two institutions were asked to explain "in plain English" a sample of code (Figure 2-17) and their responses were classified by the BRACElet group members according to the first four levels of the SOLO taxonomy.

In plain English, explain what the following segment of Java code does:

```
bool bValid = true;

for (int i = 0; i < iMAX-1; i++)
{
      if (iNumbers[i] > iNumbers[i+1])
            bValid = false;
}
```

Figure 2-17 An "explain in plain English" Question [137]

To be classified as a relational response a student had to understand that the code checks whether the array has been sorted and to be classified as a multistructural response they had to describe how the code worked (often line-by-line) without recognizing its purpose was to check the sort. If students gave answers which fell into both categories i.e. they described the function of each line and the overall purpose of the code, then the answer was classified as a relational response. Students seeking an abstract representation of the concrete code would find it natural to give a relation response (*what* the code does), while others would focus on the individual lines of code (*how* the code works) and not the relationship between them. However, the BRACElet group [220] also found that when applied to classifying exam responses of 14 students, the SOLO ratings could only be applied with moderate levels of consistency.

When classifying programming students' results against SOLO [134, 218, 220], it has often proven necessary to supplement the original SOLO levels as shown in Table 2-4. These additional sub-levels might aid consistency in categorization of student work [220], but they add a level of complexity that was not in the original taxonomy.

Not attempted or totally wrong [218]	The answer is blank or totally wrong
Prestructural	Substantially lacks knowledge of programming constructs or is unrelated to the question [220]
Unistructural	Provides a description for one portion of the code [220]
Multistructural	A line by line description is provided of all the code (the individual “trees”) [220]
Multistructural Error [220]	A line by line description is provided for most of the code, but with some minor errors
Multistructural Omission [220]	A line by line description is provided for most of the code, but with some detail omitted.
Relational	Provides a summary of what the code does in terms of the codes purpose (the “forest”) [220]
Relational with Extra [134]	Additional information provided
Relational Error [220]	Provides a summary of what the code does in terms of the code’s purpose, but with some minor error.
Extended Abstract Level	Novices able to make connections beyond the scope of the question and able to transfer knowledge to a new situation [218]

Table 2-4 SOLO Levels with Additional Sub-Levels

2.7.3 Software Comprehension, Perceptual Learning and Teaching

Given that neither the Bloom’s nor the SOLO taxonomies adequately model the process of learning to program, we need to reconsider the role of software comprehension and perceptual learning in light of these limitations. Understanding how programmers build mental models of the software they are developing should enable learning material and teaching approaches to be adjusted to take advantage of this knowledge. However, as Gilmore [92] explains, the main issue with such knowledge-based theory of expertise is that the focus of these theories is the process of acquiring knowledge about programming, not knowledge about how to program. Learning to program is not just about building semantic knowledge about programming, but also requires practice to develop episodic memories of how to apply it [91, 92].

In evaluating the software comprehension models, Schulte *et al* [120] derived goals and content that could be applied to teaching and learning programming. The goals were:

1. Develop unconscious / automated chunking strategies or skills (perceptual learning).
2. Skills to effectively navigate the mental representation, and to be able to map it and navigate to the corresponding external representation [145].
3. Skills in reading program code. Most software comprehension models emphasize the importance of understanding the program based on the program code (i.e. reading). Perhaps it should be explicitly emphasized in education as well.
4. Ability to extract different types of information from program text.
5. Ability to develop holistic understanding.
6. Ability to cross-reference different key elements

Furthermore, from the analysis the following content was suggested:

1. General orientation: What programs are for and why they are important.
2. The notional machine: an abstract model of the machine executing the code. [Note, this has already been studied by Mayer [116] who developed a layer between assembler and BASIC that represented the building blocks from which programming statements could be made.]
3. Notation: Syntax and semantics
4. Structures: Abstract solutions to standard problems including plans, beacons, discourse rules and patterns.
5. Pragmatics: Skills of planning, developing, testing and debugging.

Table 2-5 also presents a summary of the most interesting conclusions derived from the software comprehension models themselves.

Experts perform better when programs are built using plans i.e. structured programming
Experts use cross-referencing between the program and situation models, while novices focus on one or the other
Expertise requires a developed knowledge base
Experts are more likely to map aspects of function (goals) and execution to concrete code
Finding ways of being more explicit about the domain may perhaps help novices to more naturally draw the linkages between top-down programming and situation models.
A read-to-recall task caused novices to focus on the program model, while a read-to-use task that required novices to modify and re-use code enabled them to eventually develop a situation model. This observation was based on advanced students who were novice object oriented programmers [221].
In teaching and learning program execution sequence, micro-sequences focus on comprehending a program as individual examples e.g. implementing a sort algorithm. Macro-sequences focus on a course.

Table 2-5 Observations based on the Software Comprehension Models from the work of Schulte *et al* [120]

Schulte analyzed the models of software comprehension and constructed an educational model entitled the Block model [222]. This work was based on the earlier work by Kintsch [223] on text comprehension, who described a cyclical process [223] where comprehension begins by reading the text and identifying the atoms from which program statements are recognized thus forming the blocks. Inferences are then made about the relations between those blocks within the holistic macrostructure [120]. The Block Model consists of a matrix where columns represent the dimensions of software comprehension and rows represent the hierarchical levels of comprehension (Figure 2-18).

Hierarchical Levels of Comprehension				
	Macrostructure	Understanding the overall structure of the program text	Understanding the "algorithm" of the program	Understanding the goal/the purpose of the program (in its context)
	Relations	References between blocks e.g. method calls, object creation, accessing data, …	Sequence of method calls "object sequence diagrams"	Understanding how subgoals are related to goals, how function is achieved by subfunctions.
	Blocks	"Regions of Interests" (ROI) that syntactically or semantically build a unit	Operation of a block, a method or a ROI (as sequence of statements)	Function of a block maybe seen as subgoal
	Atoms	Language elements	Operation of a statement	Function of a statement. Goal only understandable in context
		Text Surface	Program execution (data flow and control flow)	Functions (as means or as purpose), goals of the program
	Duality	*Structure*		*Function*
		Dimensions of Program Comprehension		

Figure 2-18 The Block Model [222]

Schulte *et al* revised the Block model [120] to incorporate knowledge base, including semantics, goals, plans, efficiency knowledge, domain knowledge and discourse rules. However, this revised model left open the question of how this knowledge could be taught.

In the Block model, when the capacity of the short term memory is reached at the end of a line of text or "*perceived program block*", the information (program statement) needs to be transferred and integrated into working memory so that short-term-memory is freed for the next cycle [120]. Hence, short term memory is used to store the information recently acquired from the code and working memory to integrate it with appropriate information previously learnt [120]. The abstract mental representation of the program is created stepwise from the perceived material as each perceived program block is read [120]. Each level of comprehension becomes more abstract and "*independent of the*

perceived information" requiring more conscious attention and organization [120, 223]. The Block model distinguishes between three general types of knowledge:

- Text Surface, a representation of the actual code being read [120].
- Program Execution distinguishing program text from standard text [120] and the understanding of execution is important for learning to program [222]. In studying how experts and novices debug code, studies [128, 129] have shown that experts read the program in order of execution while novices read it from beginning to end like a piece of prose [129].
- Function, the purpose or goals of the program [120, 222].

The three dimensions are split between structure and function, resembling the separation of the program model and the situation model and reflecting the dual nature that source code plays in comprehension [120].

> "*By inferences and additional domain knowledge, a situation model is also constructed which includes comprehension of the function, the goals and purposes of the program*" [120]

Understanding the goals relies on inferences and on knowledge extracted from the code [120]. In teaching programming, the problem is balancing the teaching of structure and function [120]. Schulte [120, 222] recommends avoiding detaching the teaching of function from the teaching of structure and vice-versa. As comprehension increases it is assumed that the reader makes fewer errors in extracting information, integrating it and activating relevant prior knowledge [120, 222]. Experts are able to perform more of these processes:

> "*…automatically or unconsciously so that cognitive resources are freed for more complex and intentional processing [223]"* [120]

This implies that perceptual learning plays a significant role in developing coding ability. The term *program block* implies a memory chunk of some kind, and expertise can thus be defined as the unconscious activity of recognizing and applying the previously learnt atoms and blocks.

The Block model allows different learning paths to be taken through it by selecting, rearranging and omitting the cells in the matrix as required [120], although the core issues must still be taught. Thus, it does not directly define a pedagogical approach for teaching programming.

2.8 Teaching Approaches

Pedagogical study is a very active research field and a large volume of data has been accumulated over a number of years. In the area of teaching programming, a number of approaches have been attempted with mixed results [1, 7, 118]. An analysis of the benefits and drawbacks of these approaches is important, to determine if any potential modifications may be made and studied.

2.8.1 Constructivism

Students are more likely to transfer thinking skills if they are motivated to use them i.e. if, the rewards are clear and they can apply them in their own lives [188]. The idea that students will be more motivated and gain a sense of ownership of a problem the more realistic the scenario it depicts, is termed *authentic learning* [224]. Traditional teaching tends to simplify learning material to make memorisation of the content easier, at the cost of removing the inherent complexity associated with authenticity and denying the student the opportunity to develop "*associations between concepts and reflective mega-cognitive processes*" [224, 225]. This "*cognitive authenticity*" [226] is one of the central tenets of constructivist learning theory. Situated cognition [227] implies "*knowledge and conditions of its use are inextricably linked*" [224]. Constructivism essentially theorizes that an individual's knowledge is actively constructed and learning is an adaptive process [224] that results in the formation of mental models [228]. Effective learning in constructivism requires the construction of viable schema (mental models) that can correctly explain reality [7]. Learners actively construct knowledge while striving to make sense of the world, but this knowledge is based on personal experiences, goals and beliefs.

> "*It is the individual who imposes meaning on the world, rather than the meaning being imposed on the individual*" [224]

Thus, constructivism suggests that knowledge cannot simply be transferred from the teacher to the learner "*we can teach, even well, without having students learn*" [224], and instead promotes the active development of student knowledge over passive absorption from textbooks and lectures [228]. The theory also proposes that knowledge construction exists at many levels of abstraction [229], the first being sensory-motor experience (or perceptual experience) [229] with external objects and consequent abstraction of properties from them [230]. There is a continuum of levels of abstractions[229], abstractions built on abstractions starting with sensory-motor

experiences [229], where abstract concepts are learnt as a result of the operations performed [224].

Traditionally, learning material is pre-specified and sub-divided or modularized for delivery, whereas constructivists prefer teaching environments in which knowledge and skills emerge naturally [224]. Students are empowered to select what and how they learn, thus allowing different students to study different things [224]. Greater emphasis is put on the students' prior knowledge and self-reflection [224, 231]. To achieve the best transfer of knowledge a concept must be addressed from a wide range of learning contexts [224].

In constructivism, teaching involves striving to understand the student's mental model and then attempting to guide them to the correct theory [22]. However, since each student builds recursively on their own knowledge, they will each develop a slightly different understanding of the concepts. Given these mental models are individual to each student, it is not uncommon for a student to create a consistent model that is at variance with the correct model i.e. they have developed an *alternative framework* [22]. These variances from the accepted standard knowledge are known as *misconceptions*.

> "*A constructivist would view a misconception not as a mistake, but as a logical construction based on consistent, though non-standard concepts, held by the student. Misconceptions form the prior knowledge that is essential to the construction of new knowledge*" [22]

Constructivism requires instructional environments that aid the student's reflection by challenging the student's misconceptions [224]. Thus, teachers are interested in developing students' reflection skills rather than their recall of teaching material [224]. A student's ability to explain and defend decisions is important for the development of meta-cognitive skills and self-reflection [224]. Hence, assessment takes the form of evaluation and discussion of learning activities [224, 232]. Constructivist instructional environments should be:

> "*…student-centered, student-derived, collaborative, supported with teacher scaffolding, and authentic tasks and based on ideas of situation cognition, cognitive apprenticeship, anchored instruction and co-operative learning* "[224]

A second tenet of constructivism is that meaningful learning occurs when the student develops strategies to solve problems. This is related to *active learning* since the student is encouraged to be active in the learning process instead of passively absorbing the

learning material. Constructivists provide problems that can be solved in multiple ways and leave students define their own problems even if they may struggle to solve them [224, 229].

Obviously, the main issue with constructivism is that it is a learning theory not an instructional-design theory [224]. If constructivism avoids settings learning outcomes then the teacher cannot ensure that a common set of outcomes are met [224, 233] nor plan instructional activities nor predict how learners will learn [224]. Without defined outcomes it becomes difficult to make students accountable for their work [224] and set standards to assess the meaningfulness of the learning [224]. However, traditional instructional-design theories can fail to recognize that the goals of the learner will determine what is learnt not the learning outcomes [234]. This failure to engage the learner, too often resulting in the focus being on "*passing the test or putting in their time*" [234].

A pure discovery learning approach advocated by constructivism, leaves students free to construct the wrong knowledge and develop the wrong skills [224, 235]. Other students may demand more structured learning [224, 235] and not all students benefit from such a free approach to learning [224, 236]. The assumption that pure discovery learning with minimum guidance is an effective teaching approach has been repeatedly shown to be flawed [237]. A number of research papers have been published in teaching LOGO programming using pure discovery learning [238-240], all have shown that guided discovery learning produces the best results. An explanation for this phenomenon is suggested by Kirschner *et al* [59], who considered the role of cognitive psychology in constructivist learning activities and concluded that they engaged the learner in considerable searching of problem spaces for problem-relevant information [59]. Such search activities place a heavy load on working memory but do not promote the accumulation of knowledge in long term memory [59]. Even over extended periods of time searching can produce little alteration of the long term memory [241]. In discussing cognitive load theory, Paas *et al* [153] define three types of cognitive load: intrinsic, extraneous and germane. Intrinsic cognitive load is related to the difficulty of the material being studied [154] and was originally viewed as a base load that was irreducible by instructional design. Extraneous cognitive load is imposed by the instructional material and interferes with learning [153]. Many conventional instructional procedures impose

this load, because they are designed without taking into consideration cognitive load [153] e.g. searching the problem space for problem-relevant information [153]. Germane cognitive load is the result of cognitive activities that are relevant to schema acquisition and automation [153]. This load enhances learning and is influenced by instructional design. All three loads are additive and cannot exceed the working memory capacity of the learner [153]. An instructional design that reduces extraneous cognitive load should increase learning by allowing more cognitive resources to be dedicated to germane cognitive activities [153].

Cognitive load theory suggests that [59]:

> "*…free exploration of a highly complex environment may generate a heavy working memory load*"

As a result of high cognitive load, discovery based learning can produce poorer results than worked-examples practice [242].

2.8.2 Moderate Constructivism

The problem with traditional non-constructivist approaches to teaching is their failure to recognize that for a number of subject areas, including computer science, teaching static bodies of knowledge fails to deal with the fluidity and dynamism of these disciplines [33]. Is there a compromise between guided and unguided learning? Loosely there are two forms of constructivism: *radical constructivists* suggest that every reality is unique to the individual, while *non-radical/moderate constructivists* suggest that there is a shared reality based on social constraints placed on the individual [224]. This second, more moderate form of constructivism is more pragmatic and opens up the possibility for a clearer instructional-content design [224]. Karagiorgi *et al* [224], describe the following assumptions:

- Mental models are constructed as the result of experience
- Each individual's mental model may be different but the structure is the same
- Knowledge can be pre-defined yet still be applicable across multiple domains
- Teaching authentic tasks is desirable but de-contextualized abstraction should also be taught
- Instruction strategy and learning material are somewhat independent
- Fundamental instructional transactions can be adapted for a diverse number of contexts
- There are strategies which are applicable to all students
- Learning should be active but not always collaborative
- Testing can be integrated with learning objectives but separate assessment of achievement is also possible

Technology also opens up the possibilities of exploring a freer learning approach:

> "*Multimedia and the Internet are also alternatives to the linear structure and facilitate data gathering techniques, supportive of constructivist learning principles…microworlds and virtual reality simulations could simulate authentic learning while the Internet in general and Web Quests as innovative teaching strategies in particular could offer multiple representations of reality* " [224]

However, care must be taken not to confuse *active learning* with *active teaching* [237]. It is easy to fall into the trap of labeling different forms of teaching strategy as either passive or active. Activities such as reading books and attending lectures, become viewed as passive, whilst activities such as group discussions, and interactive games, are seen as

active forms of learning [237]. The constructivist teaching fallacy is to assume that constructivist learning can only be achieved through active teaching methods [237].

2.8.2.1 Action Learning

The main proponent of action learning is Revans [243], and it was originally developed for organizational learning. Action Learning primarily focuses on increasing the ability of a participant to solve problems, by increasing their and their organisation's ability to learn in a rapidly changing environment.

> "*In any epoch of rapid change, those organizations unable to adapt are soon in trouble, and adaptation is achieved only by learning – namely, by being able to do tomorrow that which might have been unnecessary today, or to be able to do today what was unnecessary last week. On the basis of the assumption that managers learn best by taking action and reflecting on the action, the following method of learning can be put forward.*"[243]

However, a number of researches have adapted this approach for education [244, 245].

Vat [245] used this approach with software engineering undergraduates, who were arranged into informal study groups to investigate e-Commerce. Translating Revan's work for his students with interpretations from problem-based learning,

Vat developed the following guidelines:

a. Students should be encouraged to see themselves as managers able to plan their time and judge the complexity of the problems that can be handled
b. They should be made aware that they do not possess enough prior information to solve the problems at the start of the project
c. They should be challenged to find solutions to often ill-structured problems
d. Students must identify, locate and use appropriate resources, and ask questions "learning issues" about various issues related to the problem(s). "*These learning issues help the students realize the knowledge they require, and thus focus their learning efforts and establish a means for integrating the information they require.*"[245]

A formula that is frequently quoted for action learning [245] is:

$$L = P + Q + R$$

Learning = **P**rogrammed Instruction + **Q**uestioning + **R**eflection

Programmed instruction includes the text books, lectures and other learning material from which the participant/student obtains knowledge. Through questioning (or feedback), new insight can be obtained in to what is not yet known. By questioning it is possible to determine whether information already exists or is relevant. Reflection implies making sense of the facts obtained and trying to understand the problems. Hence, the equation can be interpreted as planning the actions based on constant feedback and reflection as the learning process continues.

For example, Vat [245] often confronts students with unfamiliar problems forcing them to ask questions and "unfreeze" their underlying assumptions. As the students modify their assumptions, they begin to create new mental models causing them to reassess the learning material they possess (Programmed instruction), to question and to gain new insight.

Peterson [244], used action learning to see if it could help university students taking a course in Systems Analysis and Design for Business Professionals, to bridge the gap between the skills learned in the classroom and the skills demonstrated in employment.

> "*...the application of action learning concepts to information technology education seems particularly appropriate as a means of demonstrating to students that the intent of the application of technology is to solve business problems, not to create technical solutions that are in search of a problem.*"[244]

The objectives of Peterson's [244] research were two fold, to provide a hands-on learning experience to make students better at systems analysis and to create partnerships between them and other interested third parties (i.e. employers, in this case non-profit institutions). The approach taken provided a live, performance-impacted experience of the problems as they occurred in the "real-world". To assess the course, the students were given a number of work steps and deliverables that they had to meet. Regular meetings were held to discuss performance, deliverables and to plan activities for the following week. All of the teams were able to complete the work but the reports produced reflected the difficulties that were encountered and were less "clean" than those produced from simulated scenarios. However, action learning may not always be as motivational as expected. As part of Peterson's [244] study the students were required to conduct a number of interviews to develop a set of requirements. Although some students recognized the benefits of this approach, others retained a less professional

approach seeing it as just another form of assessment. In particular, these students continued to leave work to the last minute.

> "*Apparently, the traditional educational model has fostered an attitude of passivity that will not easily be overcome.*"[244]

2.8.3 Constructivism and Programming

In applying constructivism, Lui *et al* [7] identified two hazards that novices face when learning to program. Firstly, constructing new knowledge relies on existing knowledge that the student may not have already correctly constructed and secondly the student may have used incorrect knowledge as the basis for constructing new knowledge. Furthermore, weak students may also lack abstract thinking and the abilities to build schema from abstract ideas [7].

Many novice programmers have no appreciation of the notional machine executing their code [22], and the lack of an effective consistent model is a major problem. For without, it there are no misconceptions from which new knowledge can be constructed [22]. The computer forms an accessible ontological reality i.e. a correct answer is easily accessible and successful performance depends on a "*normative model of this reality [being] constructed*" [22]. Errors in the student's mental model can demotivate them, since programming gives immediate and "*brutal feedback*" [22] i.e. "*alternative frameworks cause bugs*". Programming pedagogies must consider concepts and techniques that can minimize or alleviate harsh or terse feedback.

Ben-Ari [22] describes a number of phenomena that occur in computer science that may be explained by constructivism (Table 2-6).

The construction of CS concepts is haphazard because sensory data from class must be integrated into a student's existing framework that is too superficial.
Frustration and perception that computer science is hard is due to the fact that models must be self-constructed from the ground up.
Autodidactic programming experience is not necessarily correlated with success in academic CS studies. These students, like physics students, probably come with firmly held constructions that are not viable for academic studies.
The reality feedback [brutal feedback] obtained by working on a computer can be discouraging to students who prefer a more reflective or social style of learning.

Table 2-6 Ben-Ari's Phenomena of Computer Science Education [22]

A number of conclusions may be reached from this analysis. Courses, help files and tutorials must focus on the mental model and not limit themselves to "*behaviorist practices of the form 'to do X, follow these steps'*" [22]. The model of the computer must

be explicitly taught including CPU, memory and I/O peripherals, although it may also be taught as a model computer [246] or notional machine [202]. Programming exercises should be delayed until the student has developed a mental model of the computer [22]. Premature attempts to program produce a "try it and see what happens" approach which delays the development of the correct mental model [22]. Delaying the use of programming exercises reduces the time available for practice. A more viable approach would be to introduce the notional machine concepts during programming instruction, and enable the relationship between both the machine and program mental models to be established in the same context: one reinforcing the other for maximum retention and refinement. Group work should be used to develop social interaction and to reduce the brutality of human-computer interaction, with a focus on student reflection [22]. However, group working may give rise to other problems, since programming is essentially an individual process rather than a social process.

2.8.4 Scaffolding

The term *scaffolding* comes from the work of Bruner [159] who believed that teachers or more capable peers should provide conceptual, procedural, strategic and metacognitive support to students [247]. For example, children learning a language require a social interaction framework [248] where the teacher provides content that pushes the child to just beyond their current limits (Vgotskii's Zone of Proximal Development (ZPD) [249]) but in a very well-known context with predictable routines. The predictable routines, such as the teacher reading a book together with the child, provide a structure within which expectations can be continually raised [250]. The approach is equivalent to an apprenticeship where the master craftsman provides a scaffold to enable the apprentice to perform the task and facilitate the apprentice's learning when the master is not available [251]. Collins *et al* [252] coined the term "*cognitive apprenticeship*" where the skills being scaffolded have a more cognitive nature. In synthesizing the descriptions of this apprentice scaffolding, Guzdial [253] identified 3 types of required support: the master communicates a process to the apprentice, the master watches the apprentice and provides feedback and finally the master occasionally requires the apprentice to articulate key concepts. Yelland *et al* [254] caution that the scaffold must be modified to accommodate the learner's perspective. Applebee *et al* [255, 256] described five criteria for effective instructional scaffolding: ownership of the learning event, appropriateness of

the instructional task, supportive instruction, shared responsibility and internalization [250] Table 2-7.

Ownership of the learning event	The instructional task must permit the student to make their own contribution as the activity progresses. Although the task may be initiated by the teacher, the student must be allowed to develop the topic as an independent researcher [257].
Appropriateness of the instructional task	A task must build upon the student's knowledge but must ask questions that cannot be solved without further help [257].
Supportive instruction (Structured Learning Environment)	The student should be motivated but requires additional skills to complete the task that are just beyond their current knowledge (ZPD). Instructional conversation [258] may be the most effective approach, which may include the student creating a blog [257]. A structured learning environment will provide a natural context for the activity chosen by the student[256], not in the context of schooling but one that is meaningful to the student while presenting them with useful strategies and approaches to the task [250]. Scaffolding strategies include [247]: eliciting student interest in the task maintaining student direction reducing complexity highlighting important problem features helping student's to manage frustration modelling expert processes eliciting student articulation
Shared responsibility	Tasks are solved jointly between the student and the teacher, so the teacher's role becomes more collaborative than evaluative [250]. The teacher's role changes "*from testing prior knowledge to assisting in developing new understanding. The teacher is no longer waiting passively for the project to be completed and handed in*" [257] .
Internalization (Transfer of Control)	As the student learns they internalize the procedures, routines and patterns of learning [250] and the amount of interaction with the student may increase [250]. The teacher must recognize this and replace the initial scaffolding (i.e. they should be faded out) with different scaffolds and a different type of teacher involvement [257]. For Applebee *et al* [255], the learning process consists of a gradual internalization of routines and procedures from the social and cultural context of the learner [250].

Table 2-7 The Five Criteria for Effective Instructional Scaffolding

To promote transfer of responsibility, it is argued that [159] scaffolds must be faded (i.e. removed) as the students gain the skills they require. Identical arguments have also been made for computer-based scaffolding [259]. However, there is some evidence [247] that such fading on a fixed schedule produces worse results than using scaffolding that did not fade. If students learn less when fixed fading is used then it is not possible to be sure that transfer of responsibility has occurred, which makes the use of such fading less relevant [247]. Although, for computer-based learning user controlled fading might have some advantages [247]. Jackson *et al* [260] found that high school students did indeed turn off

supportive content as they developed their expertise. These two types of fading are termed adaptive scaffolding (internal decision based) and adaptable scaffolding (user controlled) respectively [253]. If students are fading the scaffolding themselves, it is important that the teacher understands how students are using this feature [259]. Computer tools can provide supportive scaffolding through hints which may be "passive" in that they are activated by help buttons [259]. Reflective scaffolding may also be supported by providing a notepad window where the students can enter their thoughts. Such computer-based systems can only be developed to aid predictable student difficulties, whereas a teacher can react to a variety of difficulties as they arise [247]. However, tools can allow for a more personalised learning experience that is more sensitive to the students demands [259] and may also "*promote peer interactions*" [259]. Examples of these systems include open learning environments (OLEs) [261], that focus on the individual's learning experience and provide experience-based problem solving activities in the form of hands-on concrete realistic experiences relevant to the problems posed by the OLE [261]. Metacognition is also supported through ongoing assessment requiring learners to interpret and evaluate their answers [261]. By providing a diverse set of tools, on-line databases and other learning support features, OLEs promote inquiry and discovery [261]. The notion of scaffolding in tools tends to be extended to include the use of prompts or hints to aid student learning [259].

Shute conducted a study [76] that also looked at the effect of priming by providing a hint system built into an Intelligent Tutoring System. Initially the study concluded that hint-asking was a sub-optimal behaviour but following regression analysis this was refuted. Instead, Shute found four categories of behaviour:

i. *Productive*: Made few errors and asked for few hints. Benefited from working out the solution themselves. Higher outcomes.
ii. *Hint-abusers*: Made few errors but asked for many hints. Lower outcomes.
iii. *Counter-productive*: Made many errors but asked for few hints. Floundering, did not understand but refused to ask questions. Lower outcomes.
iv. *Hint-users*: Made many errors and asked for many hints. Needed help and asked for it. Higher outcomes.

Thus, hint priming was found to be a benefit for those that clearly needed it and were prepared to use the hints. Those who, for some reason, chose not to use the hints did worse than any other category.

The level of detail that should be provided by prompts (i.e. hints) given in middle school science was studied by Davis [262]. Davis found that in comparing generic and specific prompts, students displayed more reflection when the prompts were generic as opposed to directed prompts [259].

However, it should be noted that there is some evidence that the use of such prompts or hints neglects the important features of scaffolding such as "*ongoing diagnosis, calibrated support and fading*" [259].

Scaffolding approaches have been used successfully to teach software design in introductory computer science courses [263]. On these courses, the students worked on three assignments, a small assignment that introduced the design concepts, a slightly larger assignment where they worked in pairs and finally as teams of three students assigned a difficult and unique project. These projects were examples of authentic assessment tasks which mimicked real-world situations. Care was taken to ensure intra-group and inter-group cooperation while still allowing for individual accountability. The instructor took the role of a customer, allowing the students to elicit the requirements of the project, and also acted as the manager to keep the project on track. Version control software was used to foster team ownership and encourage frequent integration of the code. Instructors acted as coaches not lecturers and in setting the exercise care was taken not to lead the students too strongly towards a particular solution.

> "*While a more traditional lecture-format course in software design can be effective, an open-ended cooperative learning framework more effectively promotes learning and the positive benefits of instructional scaffolding and authentic assessment*" [263]

However, scaffolding does not always yield the expected results. Thomas *et al* [264] investigated using object (instance) diagrams on paper as a scaffolding mechanism to help students trace code in multiple-choice questions. Although they found that students who drew diagrams were better able to understand the code and object referencing, it seemed neither to help them nor to encourage them to use the technique themselves [264]. These diagrams were influenced by the work of Hegarty [265] in cognitive modeling

of dynamic systems. The approach assumed that the students would have decomposed the system into simpler components, then they would create a static mental model by retrieving information and encoding spatial and semantic relationships for these components. Beginning with some initial conditions they would infer the behaviour of these components one-by-one in order of the "*chain of causality or logic*" to mentally animate the model. This animation process required prior knowledge (e.g. of the rules that govern the behaviour) and spatial visualization processes.

Therefore, construction of the cognitive model required five stages which would not necessarily have been conducted sequentially:

- Decomposition of the system into separate simpler components
- Construction of a static mental model by making representational connections
- Making of referential connections i.e. integrate information from different types of content e.g. text and diagrams.
- Hypothesizing lines of action i.e. identify the chain of events
- Construction of a dynamic mental model by mental animation by making appropriate inferences

So why did Thomas *et al* [264] fail to produce a good scaffold for learning? They concluded that perhaps providing the object diagrams may have removed the first steps of Hegarty's model instead of allowing the students to build the diagram and hence animate it themselves [264] and this "*led to the conjecture that providing students with a specific diagrammatic abstraction of the code was not helpful because the self-development of such abstractions is intrinsic to developing an understanding of code*" [266]

This illustrates the difficulty of determining an appropriate level of scaffolding.

2.8.5 Problem-Based Learning

Scaffolding is not a stand-alone practice, instead it is used to support instructional approaches such as problem-based learning [247] e.g. when scaffolding was originally applied to education it was to support children's problem solving abilities [159]. The goals of problem-based learning are to promote deep content learning, problem solving abilities and self-directed learning abilities [267]. These goals are achieved by the explicit teaching of problem solving strategies using a hypothetical deductive method of

reasoning, and by presenting learning materials in an authentic context (authentic learning) [59]. However, Kirschner *et al* [59] found that there was no evidence to support the findings that problem-based learning produced any benefits because the high working memory load diminished the ability to learn the solution schemata. Hmelo-Silver *et al* [268], provided counter-arguments that using scaffolding with problem-based learning does indeed produce better results. Some caution must be taken when considering research in this area. In a meta-analysis, Belland *et al* [267] found that much of the research in this area failed to provide interpretable reliability and dependability coefficients. However, any improvement in a student's problem solving skills should improve their grades and prepare them to succeed in future courses [33]. Before continuing, it should be noted that:

> "*The real demonstration of understanding is application and retention. Thus being able to follow the analysis of the problem and design the solution is not indicative of being able to retain and apply required knowledge.*" [33]

There is a large body of supporting evidence showing that the most fundamental issue in learning to program is the need for good problem solving skills [30-34]. In most teaching the emphasis is on syntax and semantics, which creates an artificial separation of the problem solving activity and the translation of the solution to a programming language [34]. This approach is also reflected in programming textbooks that:

> "…*present the subject from a language construct view, ignoring the fundamentals not only of design methodology but also of problem solving concepts*" [34]

Deek *et al* [33] found that by changing the ordering of activities in a programming class and making the class session problem-driven, produced significantly better results than a traditional method of teaching. A typical class session took the following form:

1. **Present the problem**: the instructor presents a problem designed to require the use of the new course material to be studied
2. **Formulate the problem**: Develop an initial understanding of the problem by verbalisation and visualisation e.g. make a drawing, talking or answering questions. "*Developing a precise model of the problem is completed by elicitation and organization of all relevant information and the elimination of irrelevant information.*"
3. **Plan solution**: Develop an appropriate solution strategy with the aid of the instructor, subdivide goals into sub-goals.
4. **Design solution**: Organise and refine components of solution strategy, and define specifications to be translated to code.
5. **Walkthrough the algorithm**: Prepare to map the algorithmic solution to code by reviewing each line of the algorithm and selecting the exact language construct required.
6. **Present the syntax**: Check that the solution meets the goal of the lesson
7. **Implement**: Complete the program and execute it.
8. **Test**: Provided tests to verify code for each algorithm and the overall solution

More recent research by Rane-Sharma *et al* [269], adopted a similar approach by encouraging students to plan, design and translate in mandatory writing sessions before delivering the solution in a computer session. They concluded that the methodology was effective in improving student skills but found that their own approach lacked an explicit mechanism for helping students to "translate" a solution to a program.

Deek [34] proposed a pedagogical framework to simultaneously teach both problem solving and programming known as the dual common model. This model consists of six stages comprising multiple tasks. This framework necessitated a change to the assessment based on three distinct categories: process, product and subjective evaluation [34]. Process includes the software development and cognitive skills, whilst the product is the solution as an outcome of the problem solving process. In this context, quizzes are also categorized as a product. Furthermore, additional assessment criteria were included

such as self-evaluation to allow for student reflection through the students' observations and self-reporting. Interestingly, they [34] also assessed the students' attitude and motivation through observation and maintenance of monitoring records including attendance, quality of coursework, and submission of homework on time. Deek found that this approach leads to better evaluation of student performance and to greater student satisfaction [34]. In addition, students were able to transfer learnt skills to other software design methodologies and across other knowledge domains [34].

2.9 Overview of Teaching Related Decisions

As well as choosing an appropriate pedagogy, programming instructors also have a number of additional decisions to make related to the programming language and methodology required. Given that abstraction and working memory are two important components of programming, it is important to manage the number and scale of abstract mental models so they are learnt gradually and the "brutal feedback" [22] presented to the novice programmer is a little friendlier.

It has been shown that the mental models required by object oriented languages are too difficult for novice programmers to learn. Ma et al [198] investigated the mental models constructed by students of object oriented programming. The students were given a program containing object variable declarations, instance creation, and object reference assignments, and were asked to describe it. In addition, the students were asked to complete a number of multiple choice questions where they were asked to predict the results of executing a set of small programs. The set of pre-defined answers mapped onto a number of possible mental models. While 63% of the students had a viable mental model of value assignment, only 17% held a viable mental model of object reference assignment. Even using a combination of visualization and cognitive conflict only improved this figure to 50% of the students. Object oriented programming is considerably more abstract than procedural programming [21], and an "object first" approach to teaching leads to higher cognitive load.

Abstraction is essentially a way of forgetting 'detail' [22]. Object oriented programming is built on the premise that code can be more efficiently written by creating abstract solutions to problems. For example, software design patterns [270] often explicitly rely on concepts such as polymorphism and interfaces [21] to allow for the substitution of objects representing a variety of different 'unknown' or as yet unwritten future classes to

be incorporated. This level of abstraction is arguably only important when the scale of the problems to be solved demand it. For novice programmers, the problems they can reasonably solve are far simpler and the solutions required are far less sophisticated. Therefore, the power of the object oriented methodology is only truly expressed when the size or difficulty of the problem to be solved is beyond that of any learner.

From a cognitive load perspective, novice programmers should be taught procedural programming first and then progress to object oriented programming once their programming and problem solving skills have developed sufficiently. This may also be reflected in the choice of programming language, a language such as Java "forces" advanced concepts to be learnt at too early a stage [271]. Scripting languages such as Python which have simpler syntax may be adopted as an alternative [272].

A similar argument arises when considering a hybrid teaching approach that incorporates elements of software design with learning to program. The ability to design software implies a level of programming knowledge that novice programmers lack. Formal design approaches are used by experienced programmers to express the architecture of a solution to a challenging or large scale problem. For novice programmers, these types of problems are not be suitable and would impose too great a cognitive load.

Given that problem solving is a crucial skill, a better approach is to teach problem solving in a programming context. A number of fundamental problem solving techniques can be taught, with an emphasis on solving the problem before attempting to code it. One caveat: a common approach is to ask novice programmers to produce flow charts to describe their code. However, as previously discussed (Section 2.1), flow charting is just another expression of the flow of control in the program. Typically novice programmers use flow charting as a method of documenting existing code rather than as a method for solving problems. Instead, a less rigid and more informal approach is recommended. The divide and conquer principle should be given more emphasis with the focus on identifying and simplifying the problems that need to be solved. This may involve a top-down or a bottom-up analysis, but the aim is to simplify the problems that need to be addressed. In fact, a typical first problem might be identifying how to represent information. For example, in a game of Tic-Tac-Toe a fundamental problem is determining how to represent the symbols and the grid.

Once novice programmers have the skills required to solve problems using this approach, they are more equipped to progress to using formal approaches. The objective is to motivate and to support the development of novice programmers as they gradually learn to tackle larger problems and to build applications.

2.10 Summary of Literature Review

This literature review has been modified and restructured over a number of iterations to reflect the grounded theory approach undertaken. From this analysis, the main theme that emerged was abstraction and the central role that it plays in learning to program. Other important themes were problem solving and the mental models constructed by programmers. These mental models are stored in the programmer's memory, and in considering cognitive psychology, the potential impact of working memory on the learning process was also identified. Further consideration of the structure of the mental models, expanded the investigation into the field of software comprehension and in particular the focus became the concept of "*program model*" as an abstract representation of the code and the relationship of "*plans*" to program goals. Effectively, expertise can be defined as the extent of the plans learnt, the inherent ability to identify those plans through the associated code beacons in the code text, possessing the problem domain knowledge required to construct a viable situation model and the ability to map the program model (text and plan structure knowledge) to this situation model.

Problem solving skills became the second major theme to emerge, and its relationship with programming was analysed to determine the causes of difficulties experienced by novice programmers. Primarily, these difficulties were related to the inconsistencies in the situation model constructed by them from the natural language problem definitions and their own fragile knowledge inhibiting the mapping of this model to the plans and goals of the program. Again, these processes are related to creating abstractions of the "real-world" entities presented in the problem definition.

Taking these themes into account, a number of teaching approaches were considered from the perspective of developing the mental abstractions required by programming and the development of good problem solving skills. Benefits and drawbacks of these approaches were identified and scaffolded learning was discussed. Principles from these pedagogies form the basis of the action research conducted and described in Chapters 7, 8 and 9.

In the following Research Methodology chapter, the rationale for the research process is discussed and supporting arguments are provided.

3 Research Methodology

Research methodology refers to the application of a set of research methods to a field of study [273] to provide structure and organisation to the process of knowledge discovery i.e. it is a procedural framework within which the investigation is conducted [274].

> "*Essentially, the procedures by which researchers go about their work of describing, explaining and predicting phenomena are called* [the] *research methodology*" [275]

Research itself is divided into two types, secondary and primary research. Secondary Research is the study of the existing body of research and aims to categorize and analyse this research to inform and obtain supporting evidence for the primary research to be undertaken [276, 277]. Primary research in its most basic form is the collection and analysis of new data to discover new knowledge [276]. This type of research requires the data collection, data analysis and interpretation of results/findings to form a conclusion that advances the current body of research for a specific topic or area. This research addresses the difficulties higher education students face in learning computer programming, and will take the form of a series of experiments conducted through a number of tasks or assignments.

3.1 Grounded Theory

Grounded Theory is defined as theory which has been:

> "*systematically obtained through social research and is grounded in data*" [278]

Grounded theory provides a systematic method applied over a number of stages, to "ground" the theory or relate it to the reality of the phenomena under consideration [279].

> "*Grounded theory methods are inherently logical, which is often a factor that many researchers find attractive*"[3]

In grounded theory, the data is first collected and then the ideas and concepts are extracted from an analysis of this data. Originally proposed by Glaser *et al* [15], the three key principles are emergence, constant comparative analysis and theoretical sampling [280]. Instead of starting with hypothesis, concepts or ideas, these should *emerge* from the data itself. The *constant comparison* technique, determines accuracy, establishes empirical generalization, specifies a concept, verifies theory and generates theory [15]. To determine accuracy, evidence of *incidents* (elements of data i.e. occurrence of a concept)

are compared with one another, by constant comparison e.g. by comparing an incident with data from other organizations. Thus, the limits of the general concept can be established and any variations from this general concept that exist may also be discovered [280]. This approach can identify data that confirms the existence of categories and propositions. In this context, a *category* is defined by multiple incidents which can be assigned a common meaning.

> "*As concepts emerge and are named these are compared to other incidents in data, leading to the definition of properties of a category. As such, there is a constant iteration between naming and comparing data incident to data incident, and data incident to concepts, in the light of a category.*"[280]

Throughout this process, researchers should avoid pre-conceived ideas, and allow the analysis to produce the results: the theory should generate itself [15]. *Theoretical sampling* is defined as [15]:

> "*the process of data collection for generating theory whereby the analyst jointly collects, codes and analyses data and decides what data to collect next and where to find them, in order to develop a theory as it emerges*"

Decisions such as when to sample the data should not be taken at the start of the research. Instead theoretical sampling is the process of "*identifying and pursuing clues*" [3] as the research progresses. Decisions such as when, how and the sizes of data samples that should be used, must be directed by the emerging theory and theoretical sampling should continue until each category is fully identified (i.e. saturation occurs, when a point of diminishing returns is reached) [280]. The sampled data can consist of field notes and memos generated from literature review, observations, interviews, and other forms of primarily qualitative data [3]. As discussed by Matavire [280], although often only associated with qualitative data, quantitative data may also be included in the process.

Fundamentally, the approach is to maintain an archive or database of data. Often, this is in the form of field notes and memos [3] which are similar to diary entries, and identifying variables (categories, concepts and principles). In Grounded Theory, "*Codes*" are shorthand used to identify repetitive occurrences and similarities in patterns extracted from the data [3] and are given a name or label. Categories are formed by grouping related codes that illustrate a higher level concept.

Analysis involves three distinct processes *open*, *axial* and *selective* coding, which although presented sequentially, overlap with the researcher moving between processes as dictated by their research. As stated by Glaser [281], coding is:

> "...*the analytic process through which data are fractured, conceptualized and integrated to form theory*"

Open coding is the process of using the data to identify codes, properties, dimensions and categories. A property is a characteristic of a category which defines the category and "*gives it meaning*" [281]. Birks *et al* [3] give an example of a category "walking the dog" that demonstrates the idea of properties and categories:

> "*Properties of this category might be 'time', 'enjoyment' and 'energy'. Each of these properties can be dimensionalised; take for instance 'time'. Participants might identify the time they take walking the dog varies from short to long and they are influenced by the weather in making this decision.* "

The key activity in this process is the production of field notes and comparison of data. Given that theories are built from their constituent concepts, during this phase the analysis must identify and name these concepts. Through the constant comparison method, data incidents from various sources are compared and contrasted to reveal discrete nameable concepts. These names are derived from the data and are referred to as "*in vivo*" (within the living [data]) codes [281].

When categories are at a more advanced stage of development, the *axial coding* process looks for relationships (connections) between categories and sub-categories by investigating their properties and dimensions.

This is "*axial*" in the sense that the coding occurs around the axis of a main category. A sub-category, attempts to answer questions like who, where, when, why and how about a main category [280, 281].

For this analysis, Strauss [281] suggests a paradigm model (Figure 3-1), although it should be noted that there is some disagreement with this approach [282]. In the paradigm model, causal conditions (categories) influence or give rise to the main category (or phenomenon) which will result in certain consequences. Glaser [282] defines a contextual condition as:

> "...*a condition of the overriding scope, under which a set of related categories and properties occur*".

Intervening conditions serve to limit the impact of causal conditions, while actions and interactions arise as a result of the phenomenon.

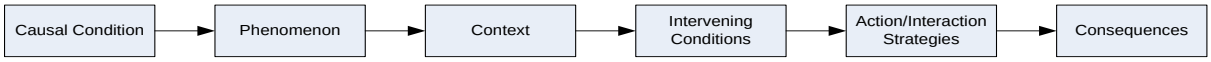


Figure 3-1 The Paradigm Model [283]

Selective coding seeks to identify a single core/central category to which all other categories can be related, and in most cases these categories should possess indicators pointing to it. The core category should be able to explain any variation and contradictory evidence [280, 281]. Naresh [283] describes the process as creating a simple descriptive narrative about the central phenomenon of study and using this storyline as the core category. A final step [281] involves validation by a high-level comparative analysis, adding missing detail and trimming excess categories. The resultant theory is a set of propositions or a running theoretical discussion [15, 280].

It is worth noting that there are alternative grounded theory approaches, the two main ones being derived from the work of Glaser [282] and another from the approach defined by Strauss and Corbin [281]. For example, the axial coding procedure and paradigm model are all adopted by the Straussian approach, which also promotes defining a research question before entering into the research.

3.2 Action Research

The relationship between lecturer and student is one where the lecturer must try different teaching methods to nurture and develop the students' abilities. A new teaching method is tried, the results evaluated and the lecturer reflects on the effects/success of the approach.

Action research is the process by which the researcher/practitioner studies the problems they encounter in order to evaluate the decisions and actions they take. It involves an individual taking action to improve what they do in practice (i.e. their work), conducting research to evaluate whether the actions they took improved their practice and documenting their actions and beliefs [16].

There are different forms of action; of particular interest in this study are *social action* and *educational action*. An action taken to influence others demonstrates *social intent* and is known as social action [273]. This includes the actions that people take as a result

of how they perceive they are viewed by others. Educational action includes social action, and attempts to influence people's thinking in order to improve their lives [273]. In the context of this research, this might include collaborative tasks where learners work together in pairs or groups with the social intent of fostering a positive attitude in their studies as they strive to meet some common goal. Learning becomes about developing a shared understanding of the concepts and ideas by bonding together through shared experience to develop skills and knowledge.

The 'research' in action research, is about taking action and analysing the effects of that action. Why take an action, what was the effect of that action and what was the significance of the effect produced? In traditional research, the researcher investigates a research topic from a more remote perspective with a view to creating a general theory that can then be applied and replicated in other scenarios [273]. Action research is associated with specific situations/environments and the purpose is to increase knowledge in that specific area and share that knowledge. Since the experiences are unique to the subjects of the research this may not be generalised or applicable elsewhere [273].

> *"Action research combines theory and practice (and researchers and practitioners) through change and reflection in an immediate problematic situation within a mutually acceptable ethical framework. Action research is an iterative process involving researchers and practitioners acting together on a particular cycle of activities, including problem diagnosis, action intervention, and reflective learning.*" [16]

By mutually acceptable framework, Avison *et al* [16] means an agreed framework that avoids conflict between researchers and practitioners or between practitioners and practitioners e.g. where somebody could lose their job or fail as a result of the research.

As described by Avison *et al* [16], the framework proposed by Lau [284] consists of four dimensions:

- The type of action research (such as action learning.).
- The tradition and beliefs implied by its assumptions
- Research process, role of researcher.
- Style of presentation adopted

> "*Action research is one of several qualitative research methods used in the field of information systems. Such qualitative research is important for studying complex,*

multivariate, real-world phenomena that cannot be reduced for study with more positivist approaches." [285]

In action research, the researcher is encouraged to try out a theory on practitioners, evaluate the results, modify the theory and repeat the process [16]. Each new modification strengthens or corrects the theory, until it meets the needs of the practitioners. Action research may be proactive or reactive i.e. it may either seek to find problems to solve or it may be used to solve existing problems [286]. For example, this may involve proactively trying a new approach and measuring its effectiveness which may result in another approach. Alternatively, it may involve reacting to a problem, collecting data to diagnose it and creating a plan to improve the existing approach.

"*The key assumptions of the action researcher are that social settings cannot be reduced for study by outside investigators and that action brings understanding leading to insight. One must keep in mind that it is these key assumptions that make action research uniquely different in form and structure from more traditional research conducted for the sake of research alone...*" [286]

Action research may use a qualitative research approach, a quantitative research approach or a mixture of both depending on the research being conducted [286]. Unsurprisingly, this approach has become important in the study of teaching methods [287, 288].

"*Data in the form of observations, classroom test scores, student artefacts, standardized test scores, discussion responses, and informal conversations are abundant, and all may inform practice.*" [286]

For example, a researcher who wishes to improve a course may collect data about student progress before and after implementing any changes in the form of test results. In addition, the researcher may wish to discover if students respond positively to the changes so the students may be asked to complete a survey that could then be statistically analysed. This would clearly be a quantitative study. A researcher may wish to establish whether the changes aided the teaching of the course, in this case the data may be interviews, discussions or recorded lectures. This is a qualitative approach but the research has similar goals.

Craig [286] recommends the collection of at least three data sets to allow for triangulation i.e. enough data sets giving similar results to allow for confirmation of any findings.

In a review of information systems studies in 1997 [284], Lau identified four classes of action research: action research, action science, participatory action research and action learning. These can be summarized as:

a. *Action research* focuses on the problems or issues from the practitioner's view point and conducts experiments to resolve those problems i.e. a process of change and reflection.
b. *Action science* emphasises the resolution of the conflicts between the theories espoused and applied by participants
c. *Participatory action* research emphasises the participant's collaboration in the research by involving them as both subjects and co-researchers.
d. *Action learning* focuses on programmed instruction, questioning and reflection. Programmed instruction takes the form of activities such as reading textbooks and attending lectures.

Baskerville [289] describes the action research cycle in five phases (Figure 3-2):

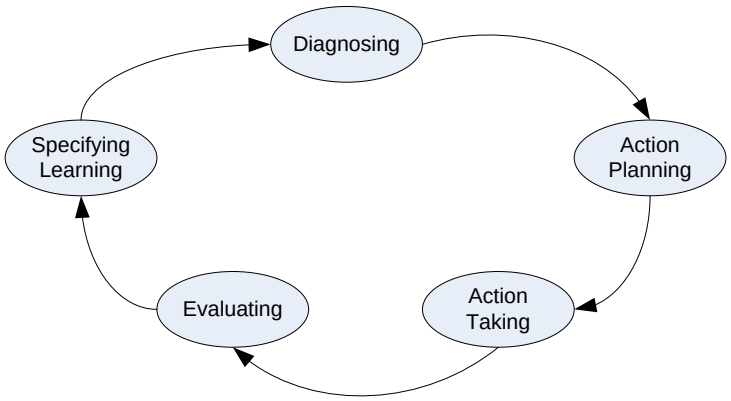


Figure 3-2 Five Stages of the Action Research Cycle

During the *diagnosis* stage, the problems are identified, the problem domain is described and a working hypothesis is derived. Having identified the problems, a set of actions are drawn up in an *action plan* to attempt to resolve those problems/issues by determining the required state and the alterations required to achieve it. These *actions are taken* by both the participants and researchers. During this phase, a number of intervention strategies may be adopted. On completion of the actions, the results are evaluated and their effects are *evaluated* to see if they solved the problems and/or if they met the theoretical expectations. Finally, as part of the ongoing *learning process*, the results are

analysed to determine what was learnt. The knowledge gained can be used by three audiences [289]:

- For "double loop" learning i.e. restructuring the organisational norms to reflect the new information obtained
- As a foundation for diagnosis during the next loop
- As important knowledge for the wider research community

The researcher keeps looping around the action research cycle until the problems have been solved or it becomes clear that they cannot be resolved.

Broadly speaking, most research is about proposing a theory, testing the validity of that theory through feedback (including experimental and observational results), analysis to determine its original contribution and a discussion of what has been achieved Figure 3-3.

Figure 3-3 Generalized Documentation of Research

In action research, what the researcher learns about their practice is the result.

> "*No one else does your practice, so no one else can claim they know it with the authority of your own experience. This is your original claim to knowledge. ....you will be judged on the quality of the action you took, whether you tried to enable others to learn for themselves. You will not be judged, however, on whether you succeeded.*" [273]

The focus of this research is not to demonstrate that practice has been improved but instead to demonstrate the validity of the claim to have improved practice through testing and to be able to show its significance. In traditional research, "theory" is seen as a set of propositions whereas in action research *living theory* is the personal theory of practice i.e. "*You do and live your theory through your practice.*" [273].

McNiff's [273] concept of a living theory transforms the way in which the research is performed and documented. McNiff talks about traditional research where the researcher stands outside the research field as E-theories (external) and the researcher studying their own practice as I-theories (individual). This extends to the documentation of the research, where tradition dictates the use of the passive voice whereas *living theory* requires the use of the first person voice in "I or we" stories to reflect the individuality of the work. It must be noted that, as computer scientists, the supervisors of

this project are strongly opposed to the use of first person voice and this practice will not be adopted for the research presented in this thesis.

In education, such research can gain *catalytic validity*, by improving the learning ability of students though enabling practitioner researchers (lecturers) to improve the way that they teach. An action research report makes a claim to knowledge, tests its personal and social validity, and demonstrates its significance by meeting standards from both practitioner and researcher perspectives. In action research the inclusion of *personal validity* means that the researcher must outline the values that they work by and how these have been met, as well as obtaining critical feedback from others (*social validity*). Such a report will be judged on the description of the actions taken, reflection on those actions and analysis of the results [273] (Figure 3-4).

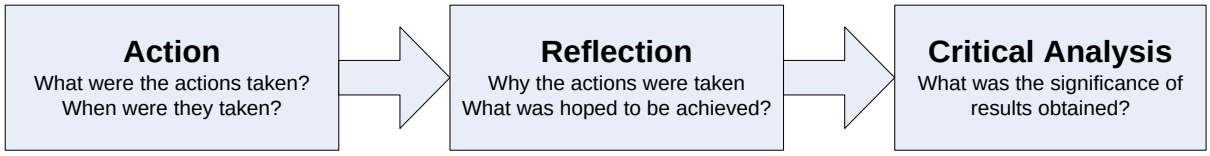


Figure 3-4 Evaluating an Action Research Report

An example of action research used in education, can be seen in research conducted in Israel [287] into the introduction of a new Computer Science curriculum to high schools. The approach was to encourage the teachers to conduct their own research, and share their research findings. It was felt to be important they had personal engagement with the actions taken instead of just reading the available research. Teachers were placed in teams and asked to produce final projects; two of the projects were presented in the paper. Team A retrospectively sought to classify the problems experienced by students in programming arrays. After identifying student difficulties, Team A produced a remedial task that sought to correct the student's misconceptions. Team B set themselves two goals, to identify students' beliefs about arrays for problem solving and to determine their understanding of the use of arrays through the use of a questionnaire. The conclusion from the final report was that teachers were able to integrate action research into their teaching effectively and that it proved to be a very valuable teaching tool.

In a wider context, action research use has been growing in the field of software engineering as evidenced by an initial survey by Santos *et al* [290] which saw an increasing number of papers being published in this area between 2005-2009. These

papers applied action research in a wide range of domains from management to software construction, although the largest grouping was in process implementation and change. The majority were qualitative in nature based on observation and interview, but a number were also quantitative using software metrics. Interestingly, the authors observed that 30% of these papers were inspired by the methodology but did not strictly follow it.

> "*This means that there is a need to improve rigor in action research (AR) studies if we want that AR investigations form a solid ground for further research and industrial applications in software engineering (SE).*" [290]

Given the rapid changes in technology, action research can be seen as an important methodology for studying information system development. Teaching programming is clearly related to both education and IT, therefore action research provides a good methodology for both studying and modifying the practice of training programmers. However, as a methodology it tends not to promote the active development of new theories. What is required is some integration of grounded theory within action research to allow for the development of new hypotheses which can then be used to adapt the training provided.

3.3 Grounded Action Research

Baskerville and Pries-Heje [285] proposed a mixed method approach known as Grounded Action Research that improves theory development in standard action research.

> "*In particular, we discovered that theory development is one area where action research methods can be made more powerful. ... Our approach to improving this rigor involves merging some of the techniques of grounded research with the theory formulation steps in action research.*" [285]

They contend that despite the iterative nature of action theory, the theory development in each cycle is not well defined and could be better served by using techniques from grounded theory to allow an emergent theory to be developed.

> "*The reason why the grounded theory units of analysis are particularly well suited for integration with action research is because they are suitable for holding data collection, analysis and theory formulation in a reciprocal relationship.*" [285]

However, they also note that action research with its emphasis on performing actions narrows the research field and prevents full use of the constant comparative technique. Typically, action research starts with an identified question/problem which suggests some

predefined concepts and categories. Grounded research in this context involves modifying or replacing these core concepts as the research progresses. Other techniques, such as theoretical sampling, are of limited use in action research.

The five phases of the action research cycle: diagnosing, action planning, action taking, evaluating and specifying learning, now incorporate techniques from grounded theory [285]. During the diagnosis phase, field notes are gathered and analysed using open, axial and selective coding to identify the initial core category and hence define the working hypothesis from which actions can be planned. When planning actions, care is taken to ensure that the actions are designed to bring about the required aim(s). While the actions are being taken field notes are also made, particularly regarding the effects of each action. During the evaluation phase, these and the previous field notes are reconsidered to increase the understanding of the results obtained. Also, axial and selective coding of the old and new notes should determine a new category or storyline for the process. If the results are not as required, then the new storyline becomes the start of a new diagnosis phase and the cycle repeats. The cycle completes when the categories reach saturation.

Multi-Grounded Action Research [291] is a related methodology that has been applied to information systems development method (ISDM) research. The fundamental difference is that the evolving theory is also used to direct the data collection and analysis, resulting in an internal, external and empirical grounding. Internal in that it reconstructs and describes the background research conducted, external in that it is concerned with the relationship of developed knowledge and other theoretical knowledge and finally empirical because it emphasises the importance of applying developed knowledge in practice. The application of knowledge may take the form of analysis, design and implementation or test and evaluation [291]. It is a 'canonical' action research method [292], thus the research takes place over a number of cycles of diagnosis, planning, action taking and reflection.

3.4 The Research Process

A Grounded Theory approach was undertaken to analyse what would normally be referred to as the "background research" to determine the main facilitative and inhibitory factors associated with programming performance. For this thesis, the project supervisors made the production of an initial literature review a formal requirement entailing

modification of the grounded theory approach undertaken. Producing a literature review pre-empts the research itself instead of allowing the research field to show itself repeatedly to the "neutral" researcher [293].

> "*Tradition often dictates that there be a priori conceptualisations of the research problem through extensive literature review, and well-designed research designs before data gathering. This is especially true for post-graduate students who are required to produce a detailed literature review before research commences as a course deliverable. These traditions are at odds with the emergent nature of grounded theory methodology*." [280]

However, there is some evidence [3, 293] that adopting a grounded research methodology and producing such a review are not necessarily incompatible. To take a holistic approach to a research field requires the researcher to read a significant quantity of cross-disciplinary literature [293]. The important aspect of grounded theory is to "*maintain theoretical sensitivity through constant comparison*" (e.g., constantly comparing incidents to incidents, incidents to concepts, and concept to concept) of this literature through memo writing [293]. In this sense, the literature review provides a motivation for the research [293]. In practice, the background research conducted and the development of the literature review itself were found to be effective in promoting the development of memos.

One difficulty presented itself, which was how could the results of the methodology be presented in a meaningful way, without duplicating the content of the literature review in giving meaning to the emergent theories. A compromise was reached that involved restructuring the literature presented in this thesis to form a distillation of the whole research context. Thus, it demonstrates the hypotheses, concepts and ideas that finally emerged from the application of the methodology in situ with the discussion of the research sources from which they originated.

Grounded Theory memos can take many forms and are considered dynamic documents [3], with 'active' memos being 'closed' as theories are constructed. In this study, memos were maintained in a series of text files containing comments, typically one or two lines in length, and associated references created during background research. Later, an alternative approach was adopted and these notes were written up in more detail in a Word document to allow the research papers to be referenced using EndNote. Where appropriate and relevant, these notes were then incorporated into the literature review.

To perform open coding, the “memo” text was transferred into an application called NVivo, using its memo database. In some cases, research papers were also directly imported into NVivo. Often these papers used different wording or phrasing that required interpretation, and in these cases it was easier to create memos than to import the papers directly. Sometimes importing the Adobe PDF files was not very useful (e.g. where papers had been photocopied) and they were effectively graphic images, making memoing the only option. This type of detailed memoing is to be anticipated [3]. In NVivo, coding involves identifying and naming “Nodes” which are then associated with the text in the memos and other external sources. The nodes themselves may be arranged in a tree structure to show their relationships (Figure 3-5), thus allowing initial themes to be developed providing a selective coding mechanism.

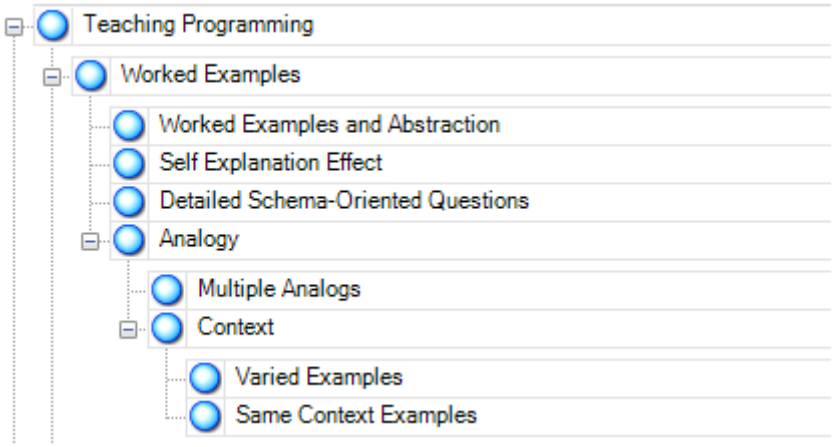
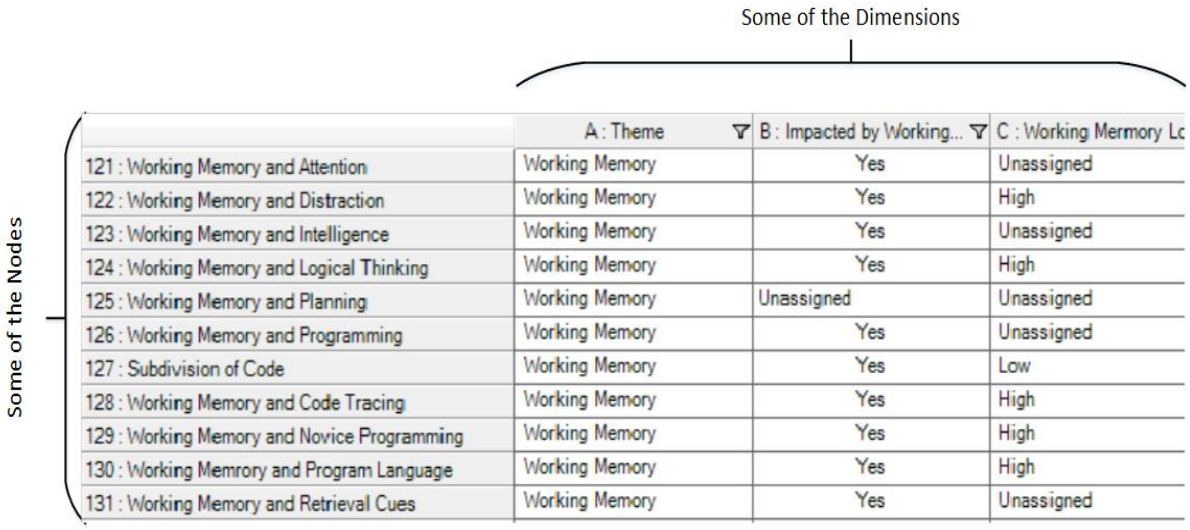


Figure 3-5 A Fragment of the NVivo Tree Structure Arranged to Show the Node Relationships

Traditionally in axial coding, dimensions would be established through analysis of common words/phrases or meanings from transcriptions of interviews or questionnaire data. As an alternative, NVivo also allows dimensions to be associated with the nodes previously identified. In this thesis, the dimensions were developed by reading the memos and identifying common impactors or implicators e.g. where problem solving was implicated as a benefit or causal factor, and associated with nodes. Primarily, these were Boolean dimensions chosen to reflect common concepts with the aim of identifying relationships between them. Typically, nodes were associated with both a number of memos and, as a consequence, to a number of these dimensions (Figure 3-6). This enabled a detailed and structured approach to be applied to the analysis of the research papers read throughout the course of the research process.



	A : Theme	B : Impacted by Working...	C : Working Memory Lo
121 : Working Memory and Attention	Working Memory	Yes	Unassigned
122 : Working Memory and Distraction	Working Memory	Yes	High
123 : Working Memory and Intelligence	Working Memory	Yes	Unassigned
124 : Working Memory and Logical Thinking	Working Memory	Yes	High
125 : Working Memory and Planning	Working Memory	Unassigned	Unassigned
126 : Working Memory and Programming	Working Memory	Yes	Unassigned
127 : Subdivision of Code	Working Memory	Yes	Low
128 : Working Memory and Code Tracing	Working Memory	Yes	High
129 : Working Memory and Novice Programming	Working Memory	Yes	High
130 : Working Memrory and Program Language	Working Memory	Yes	High
131 : Working Memory and Retrieval Cues	Working Memory	Yes	Unassigned

Figure 3-6 Dimensional Analysis in NVivo

45 dimensions were produced through this analysis. Although, given that the quantity of research papers read was only in the hundreds, for statistical analysis purposes the support for each dimension was considered quite low. Data mining of a text base is often used in grounded theory to determine the categories and aid selection of the central category. NVivo offers a number of data mining tools including cluster analysis. However, cluster analysis tends to produce less meaningful results when the volume of data is low [294] and the data provided is highly multivariate [295] causing the data points to becoming increasingly "sparse" (also known as the "*the curse of dimensionality*" [296]). In these cases, it is better to use simple graphical techniques [295]. An evaluation of the graphical approach employed is found in Chapter 4.

In addressing how students learn to program, some of the techniques from action research were adopted to study how they learnt through a number of tasks and experiments. The action research cycle was clearly appropriate in this context, but with some reservations concerning the documentation approach required to apply this methodology rigorously (specifically the use of first person voice). As already stated, grounded theory has an advantage in terms of developing a theory of how people learn or think when programming and provided a strong structure in which to explore the research topic. Consequently, a mixed methodological approach was adopted including a mixture of quantitative and qualitative data collection techniques. Where possible, a quantitative approach was used to obtain data from software metrics relating to student performance.

Although the rigorous and systematic application of the methodologies is important, some compromises were accepted. The grounded theory methodology seeks to determine a central category or single resultant theory, but it was considered unlikely that a single principle or concept would emerge to explain all programming difficulties and that a single resultant solution would be found. Instead, the mixed methodology adopted would be more correctly described as a variable analysis, since the objective was to determine the driving variables that limited programming ability. In the end, the abstract nature of programming did emerge as the most dominant variable but it cannot be said that identifying a single specific concept was the initial overall aim. In line with grounded theory, a number of experiments were conducted to test and confirm aspects of the initial literature review. New teaching approaches were adopted based on the concepts that emerged.

4 Grounded Theory Analysis

The data entered into NVivo consisted of 205 individual nodes and 45 dimensions, each specified using nominal values. The large number of dimensions produced was a by-product of the memoing and coding process, which led to a number of possible variables that needed consideration. Grounded theory itself encourages a wide exploration of ideas. Given that the codes or "data instances" were produced by hand and not by automatic software collection, a small volume of data was to be expected. The final analysis produced 237 nodes. In developing concepts, classification using decision trees such as Automatic Interaction Detection (AID), Chi-Squared Automatic Interaction Detection (CHAID), Classification and Regression Tree (CART) may also be used, and methods exist for displaying these trees [297]. Tools such as Waikato Environment for Knowledge Analysis (WEKA) [298] also enable visualisation of decision trees. For example, visualisations of such trees have been presented in a health research [299]. For this thesis, the WEKA J48 tree (based on C4.5 [300]) was used to analyse the data, but even with cross-validation the results were poor (36% correctly classified instances) and the visual trees produced were difficult to interpret. In order to classify data decision trees effectively, the data is subdivided into small groups giving rise to sparse data points, and therefore it is reasonable to conclude that these results were affected by the same problems as seen in clustering [296] i.e. the "*curse of dimensionality*". An alternative approach also considered was Graph-Based Data Mining [301] using the Subdue system, which uses a search guided by the Minimum Description Length (MDL) heuristic to search iteratively for repeated patterns that can be compressed to produce more abstract patterns. This iterative approach can be used to cluster the input graph, with the patterns forming a cluster lattice with each pattern defined in terms of one or more previously discovered parent patterns. For concept learning, SubdueCL [302] requires both positive and negative examples in a graph format. However, this algorithm again mines for patterns and is not specifically intended to visualise the relationships between nodes in the graph. Indeed, no tool could be found to visualise the data produced.

Given that the data mining techniques used did not provide useful results and that there was a lack of a meaningful approach to visualise the concepts, an application was written to produce a graph that could be used to visualise the data. Two passes through the database were required. On the first pass the significance (frequency) of the dimension was calculated and these were ordered with the most frequent dimension first. The

algorithm defines a dimension as a name/value pair resulting in different values in the data being represented as separate dimensions e.g. “Impacted by Problem Solving YES” and “Impacted by Problem Solving NO” were treated as separate dimensions. Furthermore, an initial filtering process ignored the “Unspecified” value since this value was used to indicate that there was no significant link between the data record associated with the grounded theory code and the dimension. By ordering the dimensions, the least significant nodes in the graph appear at the bottom where there are fewer links to them and allow them to be culled more easily. Thus, the initial approach is closely related to that of the FP-Tree. Let $D = \{d_1, d_2, \dots d_m\}$ be a set of dimensions where each dimension is a unique name/value pair $d_i = d(N_i, V_i)$, the database $DB = \{R_1, R_2, \dots, R_n\}$ is a list of database records and $R_i(i \in [1..n]) = \{v_1, v_2, \dots v_m\}$ is an instance of a database record consisting of the values for each dimension where $n = |DB|$ and $m = |D|$. Each value in the database is mapped to a dimension $f: v_i \rightarrow d_i(N_i, V_i)$, although unassigned values for the dimension can also be ignored.

As a comparison, assume the database contains the following dimensions A, B, .. F given in the database shown in Table 4-1.

Frequency	Dimension (D)
2	A
5	A, B
3	A, B, C
3	A, B, D
1	A, B, D, E
4	A, C
1	A, D
1	A, D, E, F
1	B, E
2	C
1	C, D
1	C, D, E
1	D, E

Table 4-1 Example of Dimension in Records Contained in a Database

The first pass through the database produces the resultant dimension frequency table shown in Table 4-2.

Dimension	Overall Frequency
A	20
B	13
C	11
D	9
E	5
F	1

Table 4-2 Dimension Frequency List

Given DB, the Dimension Frequency List $DFList = \{f_1, f_2, \ldots f_m\}$ where $m = |D|$, and $f_i = f(d_i, fc_i)$ where fc_i is a frequency count associated with the dimension. The $DFList$ is sorted such that $f_i >= f_{i+1}$ given $(i \in [1..m-1])$.

During the second pass, the graph or tree (acyclic graph) was constructed by reading the dimensions from the most frequent to the least, matching them with data values from the database record and adding or updating the appropriate nodes in the graph. The application was constructed to visualise the data based on two hypotheses:

1. The importance of a link within a branch relies on the significance of the nodes it links together
2. The importance of a node relies on the significance of the link joining it to another node

The first hypothesis was related to the Frequent Pattern (FP) Tree [303], and produced a wide shallow tree structure sometimes with multiple nodes representing the same dimension. In line with the FP-Tree approach, a frequency count was associated with each tree node and was incremented when a data line followed the same path through the tree. Having completed the second pass, all nodes below a set significance level were culled, in a similar fashion to the frequent pattern growth method [303]. Using the database shown in Table 4-1 and creating the tree with the support threshold ξ set to 3 produces the result shown in Figure 4-1. The frequency count for the dimension is spread across nodes on multiple branches of the tree corresponding with the support for that "pattern" within each branch. Notice that the relationships B--C and D--E are culled

because neither is deemed frequent enough in any branch. As a result E is also culled despite its overall frequency in the tree suggesting it could be of interest.

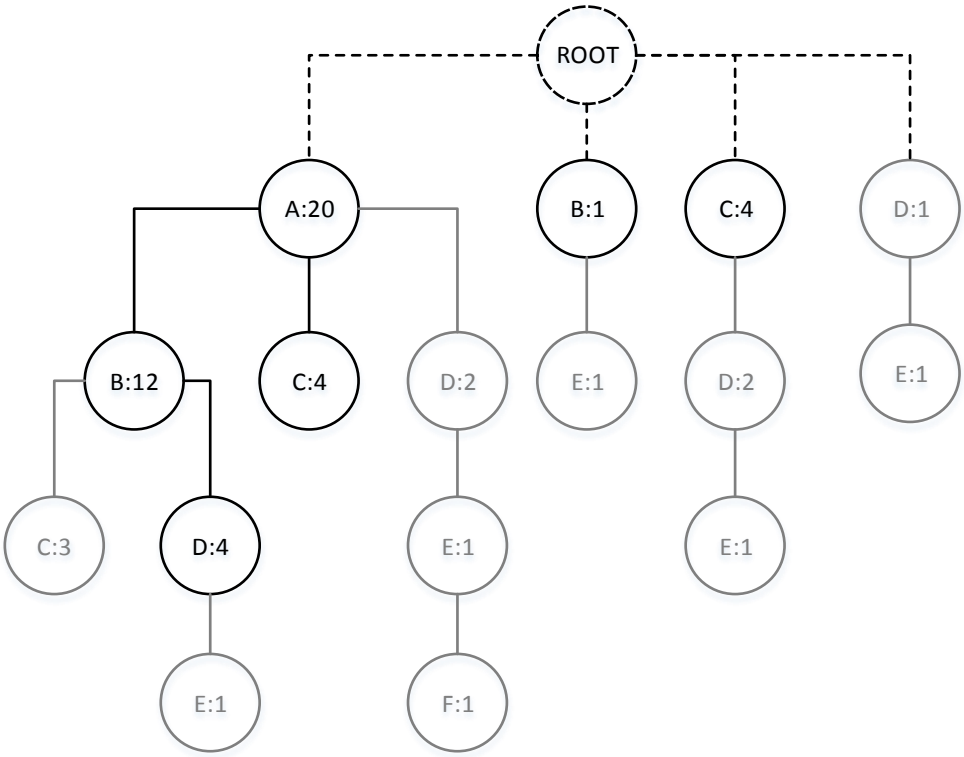


Figure 4-1 The FP-Tree (shaded nodes and edges fall below the threshold and are removed)

Because of this "spread" of the incidences of a dimension across the tree, the significance (frequency) of the node tended to be very low and they were culled more aggressively than required. An alternative approach was also tested to reduce this aggressive culling. Instead of maintaining a frequency count for each node, because each node represented a dimension, the overall frequency of that dimension, calculated in the first pass through the database, was used to determine whether the node should be culled. For data visualisation purposes, we wish to see the relationships between the most significant dimensions that emerged from the research rather than to cull them from a branch because they are deemed less important in that particular context. Therefore, it is reasonable to allow the overall significance of the dimension represented by a node to outweigh its significance in a particular branch. As expected, this alternative approach did avoid the previously aggressive culling problems but did lead to another problem with orphaned nodes being created. The intention was to visually inspect the "patterns" produced by these graphs, to determine node relationships but the width of the graph

still made visualising the relationships unclear. Given these problems, the diagram produced was of limited use for cross-referencing but this approach was taken no further.

Applying the second hypothesis, a graph was constructed between all the dimensions but only one instance of a dimension was added. The use of nodes and edges to specify information is related to Semantic Networks [304] and semantic classification in natural language processing [305]. A frequency count was associated with each edge in the graph to enable the construction of a weighted graph which is commonly used to find the shortest path [306] between nodes. Where a database record duplicated a path through the graph, this edge frequency count was incremented. Thus an edge-weighted graph $G(n, e)$ consisting of $nodes(n)$ and $edges(e)$ was created, with each node being a unique representation of a dimension d, thus $f: d_i \rightarrow n_i$ where $i \in 1..m$ and $m = |D|$ producing $N = \{n_1, n_2, n_3 \dots n_m\}$. Each edge represents a relationship between two nodes $e(n_i, n_j, w)$ where $i \neq j$ and w is the edge weighting. Duplicate paths through the graph follow the same nodes and edges, with the weight representing the number of incidents of the same edge being followed by those paths. Let E be the set of these edges. Given *n* is the number of nodes, then the minimum and maximum number of edges is given by $n - 1$ and $n(n-1)\ /\ 2$ respectively. A database record R_i represents a path P_i through the graph, where $P_i(n, e) \subseteq G$ where $i \in 1..p$ and $p \leq |D|$. Thus, the $Nodes(P_i) \subseteq N$ while the $Edges(P_i) \subseteq E$. Each database record R in DB is mapped to one path P in the graph, thus $f: R \rightarrow P_i$ where $i \leq |E|$. In mapping, R to Pi, the values of each dimension v are sorted in $DFList$ order such that $\{v_i: fc_i, v_{i+1}: fc_{i+1}, \dots v_m: fc_{m+1}\}$ where $fc_i \geq fc_{i+1} \geq \cdots \geq fc_{m+1}$. Thus nodes with the least significant edges will be added at the "bottom" of the graph making it easier to read. No assumption can be made that the graph will be complete.

Having constructed the graph, the edges that have frequencies that fall below a support threshold ε are removed. Therefore, $P_i(\varepsilon) \subset P_i$ where $|P_i(\varepsilon)| \leq |D|$ and the least significant edges and potentially the least significant nodes are removed. The graph produced is easier to read because it shows the relationships between nodes more clearly and better reflects the strength of those relationships. The result is also a narrower and deeper graph for visualisation purposes. It was never the intention to "mine" data from this graph, but to provide a means of visual inspection that allows human insight into the relationships between the codes discovered in grounded theory analysis. Taking the

database shown in Table 4-1 and constructing the graph with the support threshold ξ set to 3 creates the graph shown in Figure 4-2. The edges between A—D and E—F are culled but the relationships between B—C and D—E are still represented. This is important because there is clearly a relationship that needs to be shown between these nodes. The culled edges and nodes are not displayed, and the root node may also be excluded when displaying the graph Figure 4-2b. The relationship between A--B—D—E is now revealed. A similar relationship between A—B—D is also visible in Figure 4-1.

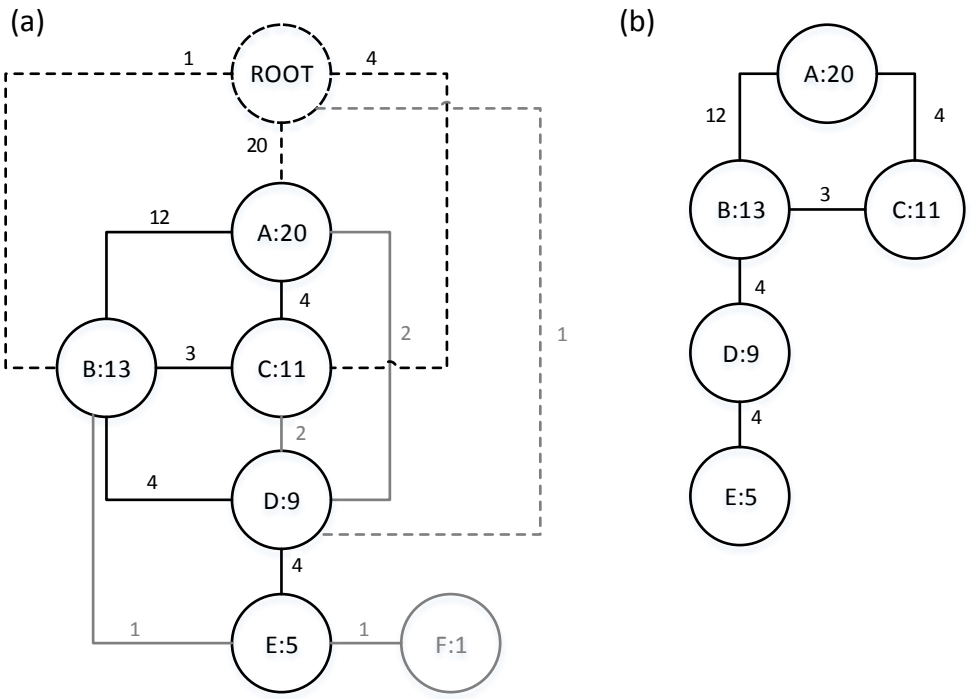


Figure 4-2 Example Weighted Graph

The graph produced does not strictly allow a relationship "between" edges to be visualised. For example, say three nodes X, Y and Z are joined by two edges X—Y and Y—Z. It does not follow that a relationship between X and Z exists, since a data record X, Y and Z may never have existed. All that is required for such a graph to be drawn is two data records containing the pairs (X, Y) and (Y, Z) in some combination.

The original data and the FP-Tree results were consulted to confirm or to review multiple relationships such as these. In practice, for information analysis this is not typically a problem because of three mitigating factors:

1. If the relationship X—Z without the occurrence of Y is significant, it will be represented separately in the graph.
2. If the relationship X—Z only occurs in the presence of Y, then Y is a significant factor in establishing this relationship. In this case, since the edges are not directional, both relationships may be read from the perspective of the common node Y i.e. Y—X and Y—Z.
3. The nodes are sorted by support (frequency), for example, let X be the highest and Z the least supported node. Nodes with higher significance are more common and generic than nodes with lower significance which are more unique and specific because they occur less often. The graph is drawn top-down starting with the most significant nodes. Therefore, the edge from X—Y may be read as the relationship between a main concept and sub concept, while the edge Y—Z may be read as Z being a specific trait or characteristic of Y. The use of taxonomic hierarchy in graphs is common and found, for example, in Knowledge Graphs based on the Simple Knowledge Organization System (SKOS) standard published by W3C [307, 308].

Each record in the database R contained a field that contained the name of the grounded theory code to which it was associated. A modification to the diagram was tested to display these names. On creating of path P_i a special code name node was attached to the last node in the path that allowed the name to be displayed. This resulted in a more cluttered diagram, but to some extent it did enable the relationships between nodes to be traced back to their origin (data record and associated Grounded Theory code).

The data produced by memoing, not unexpectedly, produced fairly low frequency counts as can be seen in Figure 4-3. This figure shows the relationship between the dimension or node in the graph, its support (frequency of occurrence) and the edges associated with it (note all edges “from the root” are not included). The dimensions run along the X-axis with the support frequency on the Y-axis. For reference, two dimensions are highlighted, “Impacted by Abstraction” which has the highest support and “WM Load Implied” which has the lowest support. As can be seen from the figure, edges with higher support are

associated with the nodes with higher support, suggesting that common themes or concepts emerge from the left of the graph. In the mid-range, concepts are being developed and supported by a number of more specialised relationships so the number of edges increases but the support for the edges declines. To the far right there are very few if any relationships to consider. In the context of analysing memos, even edges that have a low significance may actually be important because they may have been well established by authoritative research sources. Therefore, changing the edge significance threshold was regarded as just a mechanism for viewing the graph at various levels of detail.

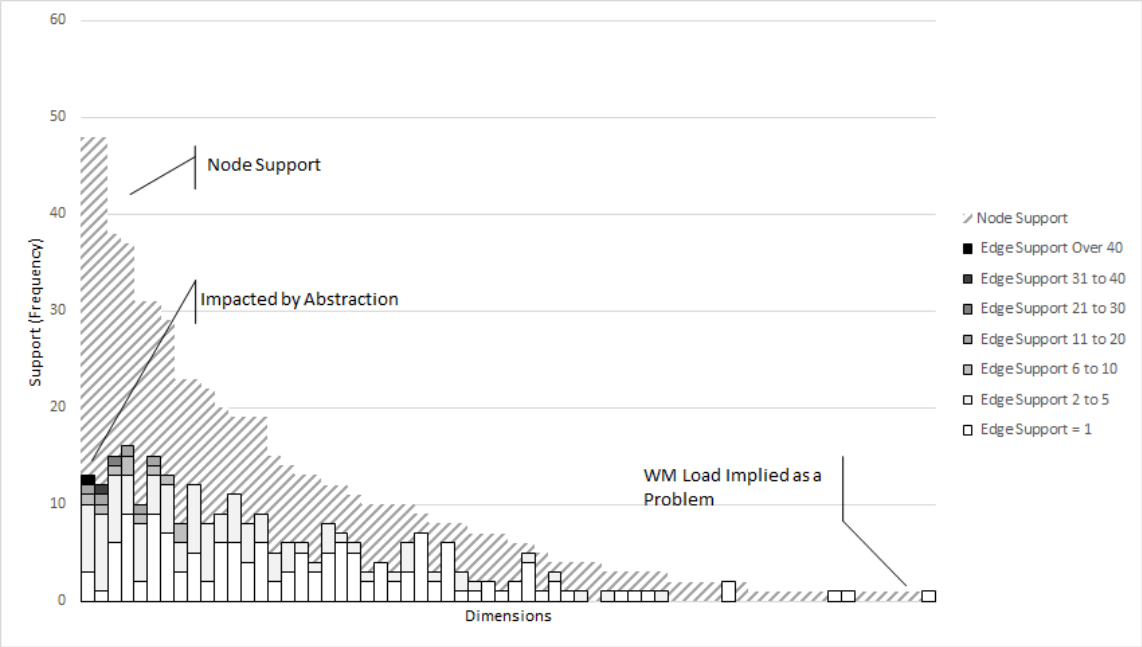


Figure 4-3 The Node, Number of Edges and their Associated Support (frequency)

In practice, for this thesis thresholds between 3 and 6 provided the best results. Figure 4-4 shows the result of changing the edge support threshold. For illustration purposes, the nodes remaining at a threshold value of 6 have been shaded grey, but the graph at this threshold is also shown in Figure 4-5(a). Raising the threshold removes links and nodes reducing and simplifying the graph produced, while the main concepts remain present.

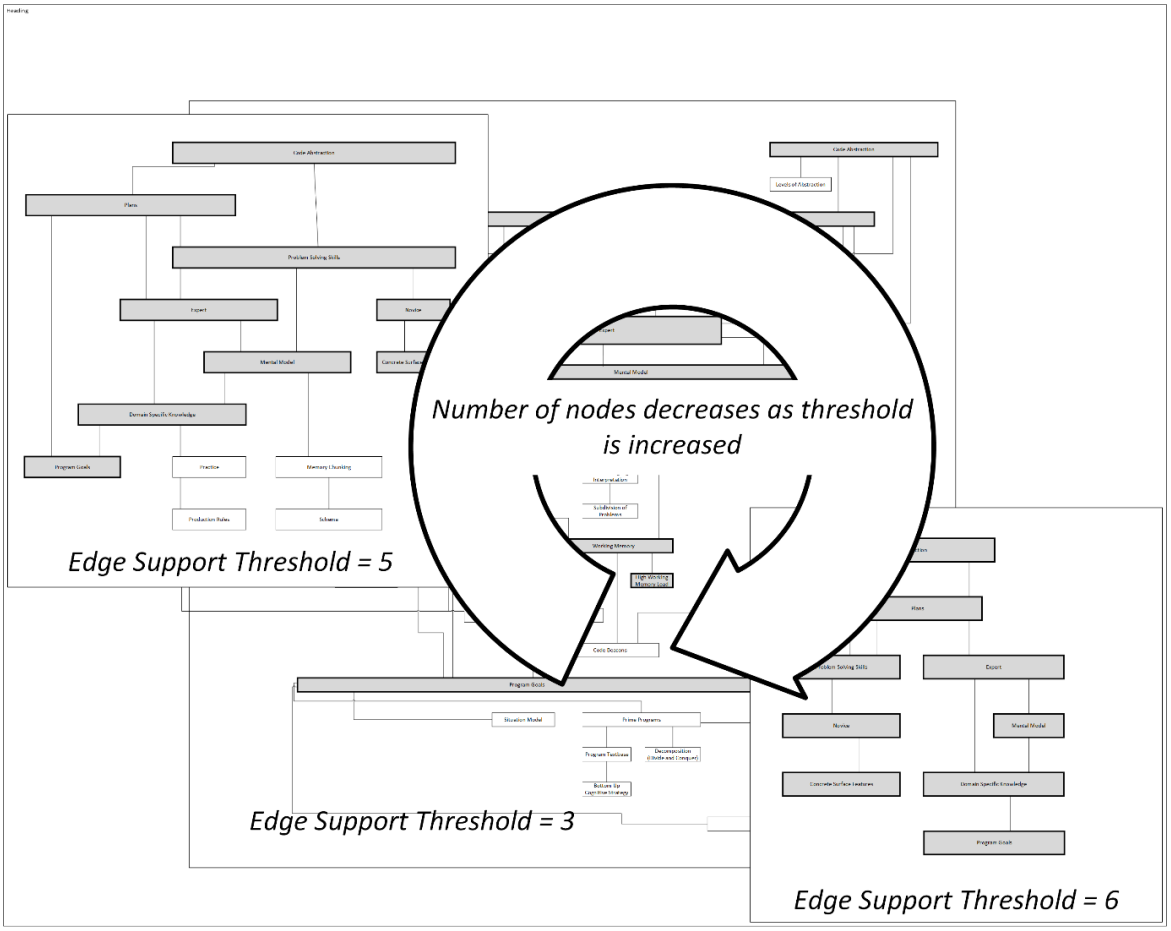


Figure 4-4 Results of Changing Edge Support Threshold

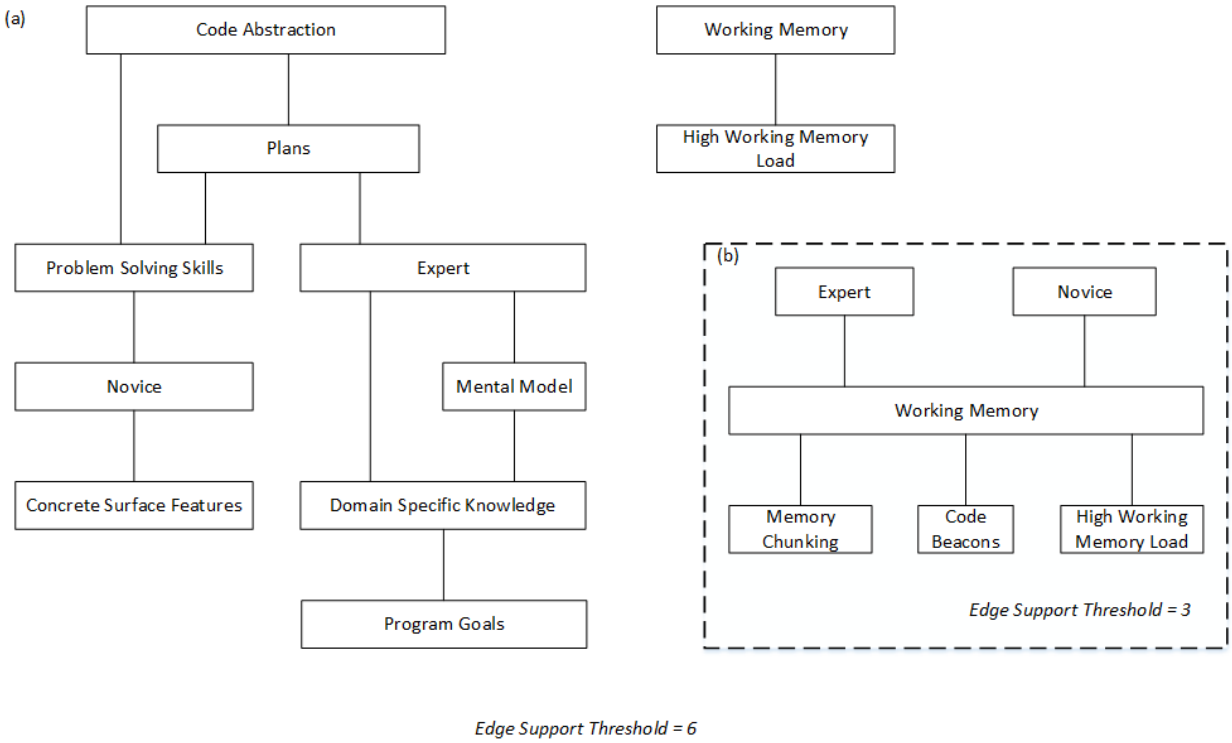


Figure 4-5 Graph produced with edge threshold value of 6

The main theme (or in grounded theory terms, the resultant theory) identified in Figure 4-5(a) is that programming success is significantly related to code abstraction, and this figure also suggests abstraction is primarily associated with code plan knowledge and problem solving skills. These topics have already been discussed in more detail in the literature review (Chapter 2). Two other interesting observations are that novice programmers struggle with problem solving and fixate on concrete surface features of a problem. Briefly, Figure 4-5(a) suggests that novice programmers are unable to abstract the generic ideas and principles required to solve problems and are distracted by the "irrelevant" specific details of the problem. For example, they may fail to identify that a linked list is required when an arbitrary number of values must be stored. Experienced programmers are able to leverage the domain specific knowledge they have acquired and hence construct a more complete mental model of the solution. Code plan knowledge also plays a part in problem solving and experienced programmers have a better understanding of the programming constructs and techniques required to solve programming problems. A small sub-graph (Figure 4-5b) demonstrates that working memory also plays a significant role in novice learning and programming expertise. When constructing these graphs, the terse nature of the dimension naming process inevitably means that some interpretation is required. Note that the names of the nodes in the figures in this thesis have been renamed to simplify the presentation.

Selective coding was achieved by reviewing the memos in light of the relationships identified in these graphs. The primary concept determined from this research was the role of abstraction in problem solving. As a result, the original literature review was rewritten and restructured to reflect the concepts and relationships identified using grounded theory with abstraction being the main theme running throughout. By cross-referencing between graphs at different support thresholds, the primary concepts and relationships were extracted into separate graphs for the purpose of discussion.

Abstract thinking encompasses many concepts in many fields of research including general problem solving through analogy, abstract thought, logic and pedagogy. Grounded theory analysis provided a framework within these concepts and their relationships could be approached in a holistic way without preconceived ideas.

4.1 Research Phases

The research presented in this thesis was driven by the grounded theory analysis, and was conducted in three phases: initial exploration of potential factors that might be associated with and predict student performance, further exploration of concepts as they developed through the axial coding process and implementation of new teaching approaches addressing the concepts identified.

The initial research was conducted over three years and involved first year programming students completing a number of worksheets (Chapter 5). Each worksheet was marked and scored against a number of metrics. The analysis of the metrics was conducted using a FP-Tree data mining approach to identify possible patterns of behaviour. The main conclusion from this research was that a lack of problem solving skills was the principal characteristic associated with poor programming performance.

During the axial coding process, additional research was conducted into the role of working memory in programming (Chapter 6). In a two year study, students were tested using Raven Matrices to measure their working memory and given programming tests. The results showed that working memory did have an effect, especially in the initial phase of exposure to programming.

In developing a new teaching approach, two dominant concepts were directly explorable. Firstly, given problem solving skills are important in developing programming ability, an approach was explored in which problems were presented in a coding framework using a scaffolded problem based learning approach (Chapter 8). This approach also sought to overcome the potential problems of lack of domain specific knowledge and motivation to solve more arbitrary problems by providing a context that led to solving larger and more meaningful problems. A different teaching style was also adopted that emphasised the "divide and conquer" principle. Secondly, the grounded research suggested that experienced programmers have better mental models and code implementation plans that novice programmers lack when developing code. To address this issue, a "plan/prime program" based approach to teaching was adopted (Chapter 7). This involved sub-dividing each programming construct into its most fundamental structure and providing a series of exercises, reinforced by a number of tests, to enable students to build the required mental models and plans more efficiently. It was hoped that continual

testing would promote memorisation of the models/plans and potentially overcome working memory issues that might cause learning difficulties.

5 Identifying Common Indicators of Programming Success during Continuous Practice

One of the fundamental issues with teaching programming is measuring and analysing a student's performance. A number of papers [309-311] have been published identifying metrics that might be used to measure and hence monitor the students' ability as they progress through a course. In analysing the quality of novice programmers' work, Mengel *et al* [310] and Jackson [312] identified a number of features for analysing student programs in an automated way. These features were "Correctness" determined by the output of the program and how closely it conformed to the requirements set by the tutor. "Style" including module length, identifier length, comment lines and indentation. "Efficiency" was a measure of the CPU time taken by the student program compared to the tutor's program and finally "Complexity" was measured by using McCabe's Cyclomatic Complexity metric [313]. To simplify the marking, the meaning of *efficiency* was expanded to include programming skill and understanding. Programming skill was defined as the ability to approach a problem logically and for larger problems this was expanded to include a measure of the closeness of their solution to a "best" solution. Rohaida *et al* [314] as discussed in [315] , suggested a system of measuring complexity focusing on object oriented programming that selected the McCabe's Cyclomatic Complexity [313], Number of Classes [316], Number of Properties [317], Attributes Complexity [318] and Operation Complexity of Classes [318] metrics. This approach was adopted for scoring the object oriented worksheets.

There have been a number of studies aimed at determining the factors that differentiate successful students from failing students using data mining techniques. Examples include the collection of data about various student characteristics such as gender, age, study type, place of residency and the grade obtained, through questionnaires and applying decision trees to determine characteristics of successful students [319, 320]. When a web application acts as the medium by which the course is taught, another approach was to extract a number of online metrics for analysis. For example, one such study looked at the total correct answers (success rate), getting a problem right on the first attempt, total attempts, time spent on problem, student participation in communications versus working alone, reading the supporting material before attempting an exercise, submitting a lot of attempts without reading supporting material in between, giving up on a problem

and time of the first log on (i.e. when they started the exercise) [321]. The results were analysed using genetic algorithms and it was found that the two most important characteristics were the success rate and the total number of attempts. However, none of these data mining studies has addressed the possibility of predicting performance through analysis of potential coding performance indicators.

5.1 Methodology

To allow continuous assessment of student progress throughout the academic year, a first year programming course was taught through a series of six worksheets. This research consisted of a two year study consisting of 104 students and a confirmation trial consisting of 89 students. Each worksheet contained lecture notes covering a number of concepts. Concept was immediately followed by an associated tasks designed to assess the student's understanding and ability to apply the concept. A number of metrics were identified that could act as indicators of programming ability. The worksheets were assessed against each of these metrics, with each metric given a Likert score: 0 for poor, 1 for average and 2 for good. An exception was made when measuring complexity, any solution of high complexity was given the value 2 and during analysis this was considered a poor result. The students' grades were also included in the data mining process. A set of results was produced for each student which could be analysed both per worksheet as well as across the entire academic year.

Two studies were performed: the first collected data across two academic years and the second conducted across a third academic year was used as a confirmation study during which metrics were collected for just the first four worksheets. Before analysing the metrics, it was noted that the results obtained by 104 students over the course of a two year study, demonstrated the class polarization effect or bimodal distribution [4, 214] often seen in programming classes (Figure 5-1).

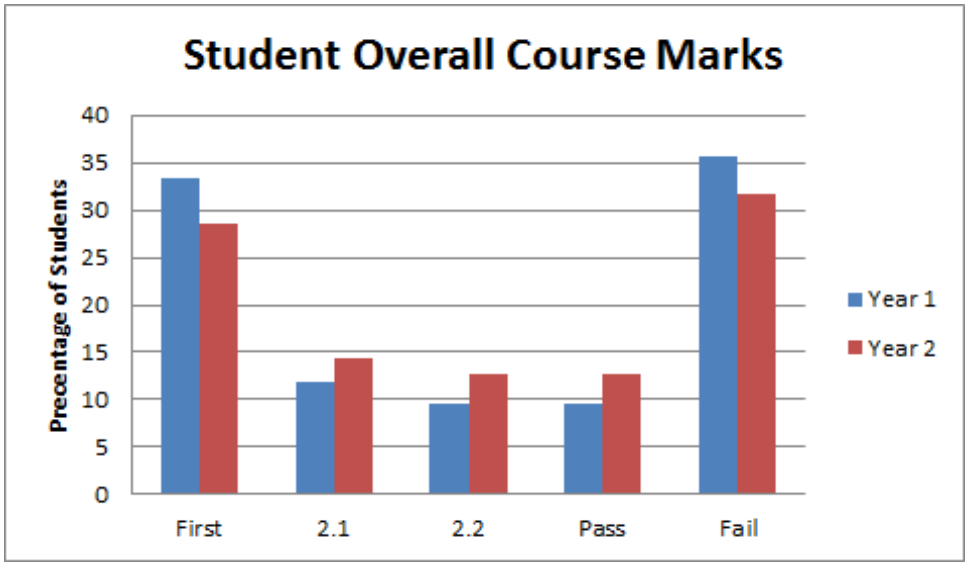


Figure 5-1 Overall End of Year Course Marks obtained by the Students

The metrics across all the spreadsheets were grouped and analysed according to the final grades achieved by the students in each worksheet. The marks awarded to the students across all the worksheets is shown in Figure 5-2, and this illustrates the progress of the student body throughout the year i.e. all the students tended to achieve poorer marks as the course became more challenging. In terms of these results, two large groupings of students were awarded either a First (>=70%) or a 2.2 (>=50% and < 60%) with the remaining students distributed more evenly. Therefore, the primary analysis of the metrics focused around these two groupings which were statistically the most relevant. The analysis of the data for failing students is less reliable due to a number of factors. Firstly, the low number of failing students (Figure 5-2) significantly reduces the accuracy of the analysis. Secondly, the disparity between the good and poor students increases across the worksheets, thus the earlier worksheets produce results which are too similar to allow discrimination between the student types (Figure 5-3). Finally, a number of the poorer students withdrew before completing the course and as a consequence complete data was not obtained for these students. A significant student dropout rate starts around worksheet 4 (Figure 5-3), which was submitted in January following the Christmas holidays, and continues through worksheet 5 and 6, where principles of object oriented programming were introduced in the second semester. However, a similar dropout rate in January has also been observed when programming was not taught in the first semester indicating that there are other underlying issues related to general academic study and computing skills which contribute to this problem. Nonetheless, the results show that an issue does appear to exist when considering students obtaining average marks, of around

2.2 (Figure 5-2), when moving on from the most basic concepts of object oriented programming to the more challenging and abstract concepts covered in worksheet 6. Identifying the cause(s) of these difficulties and formulating methods for overcoming them requires further study. Comparing Figure 5-1 and Figure 5-2, there appears to be an anomaly given the significant failure rate in the course and the low failure rate in the worksheets. The main cause of this phenomenon was non-submission of worksheets, as the distribution of the grades across the worksheets only varies slightly (Figure 5-3). In other words, the results demonstrate a filtering effect as the weaker students drop out throughout the academic year reducing the number of poor results in the later worksheets. This is evidence of the lower persistence of novice programmers [322], and may also be related to the "*brutal feedback*" associated with programming [22].

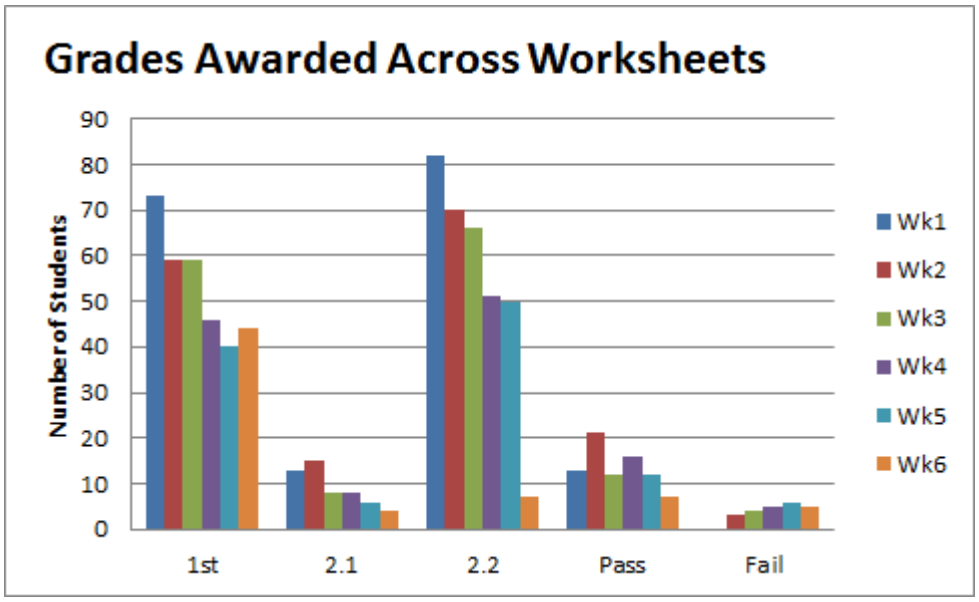


Figure 5-2 Grades Awarded to Students for Each Worksheet

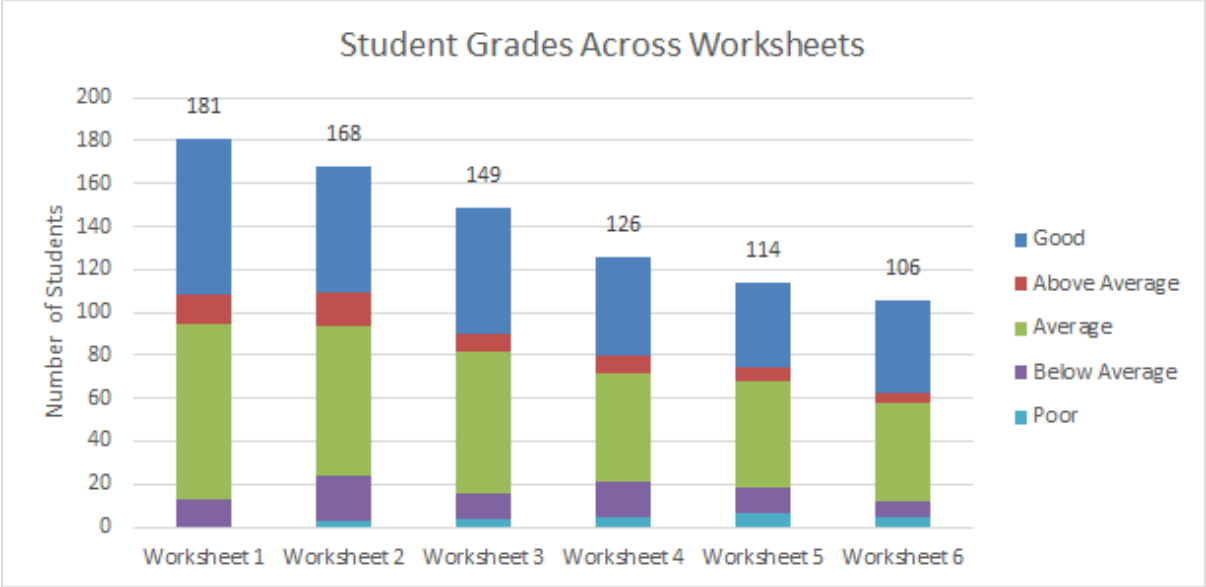


Figure 5-3 Student Numbers across Worksheets

5.2 The Worksheets

The implementation of the worksheets followed the Programming-First Model and Imperative-First approach as described in [323] i.e. with a focus on programming syntax and semantics. The programming fundamentals were divided into six key topics each with its own worksheet:

- Understanding Variables covering declaration, naming and data types
- Branch Statements including if, else and switch statements
- Iteration including the for and while loop statements, and the use of arrays and loops
- Functions including declaration and calling
- Classes covering the basics of object oriented programming
- Inheritance including the basics of polymorphic behaviour

Thus the course was divided approximately into procedural programming in the first semester and object oriented programming in the second. Each topic was taught over a number of weeks depending on the nature and complexity of the concepts, for example, the use of variables was covered over a three week period to allow significant emphasis to be placed on naming and data types. Both lectures and tutorials were delivered in a computer laboratory, to enable students to put into practice the principles being taught. The content of the worksheets was subdivided to follow the pattern of the lectures and it was intended that students would complete tasks associated with each lecture on a

weekly basis (see Figure 5-4). The worksheets were submitted at regular intervals throughout the academic year so that students were given feedback on their progress as early as possible.

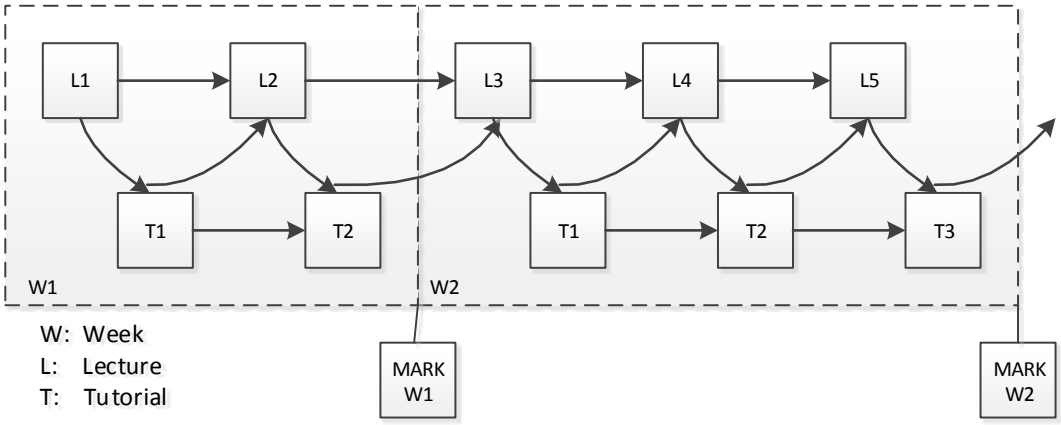


Figure 5-4 Relationship between Lectures, Tasks and Worksheets (derived from [324])

The tasks were chosen to be small and focused to make the marking effort manageable while providing an opportunity to provide sufficient feedback [325]. Langrich *et al* [324] investigated the types of tasks that are normally set given the significant marking load generated by providing frequent exercises. They concluded that exercises must be solvable, verifiable and manageable in terms of the effort for both the student and the tutor, and reasonable in the sense they "*must be sufficient to train the student in the necessary programming competencies*" [326]. Furthermore, they found that in comparing typical tasks set in Computer Science 101 Programming Fundamentals courses taken from textbooks or exercise units the "*…similarity between the tasks and the aim to train typical tasks of a programmer was remarkable*" [324]

Tasks could be classified from the tutor's view and the student's view. From the tutor's perspective they are divided into atomic or aggregate tasks. Atomic tasks consist of Open-Value tasks where the student has to apply a function to obtain a result for some given data, Close-Value tasks that consist of multiple choice questions, Specification tasks where an implementation must meet some set of requirements (the specification) which could be checked by automatic testing, and Tutor-Reviewed tasks which can only reasonably be checked by the tutor. Aggregate tasks consist of Complex tasks solved by a number of atomic tasks with no dependencies between them and Step tasks where the dependences between the atomic tasks require that they must be solved in a specific sequence.

From the student's viewpoint, tasks are categorised as Implementation/Correction of a solution to meet a given specification, Calculation of a result given some input or output value(s), Testing of a provided implementation and the declaration of functions to meet some specification.

In choosing the tasks for the worksheets, a variety of approaches were taken, such as flow charts and variable tables, but primarily the focus was on Specification tasks where the student was asked to implement a solution to a given problem. For example, a Console application where the user can select a menu option that will invoke a function to perform some required action as shown in Figure 5-5.

A client needs an application that displays a menu with the options:
1. Say hello
2. Say goodbye
3. Say hello again
4. Quit
However, the client wants the user to be able to type "one", "two" and "three" instead of the numbers 1, 2 and 3. Anything not matching these words should cause the program to quit

Figure 5-5 A Typical Worksheet Exercise

Wherever code was implemented, the student was required to provide the full source code and evidence that the code was working, which usually took the form of screen shots.

However, not all tasks required code to be implemented. In some cases, it was felt that the students would develop a deeper understanding through exercises that required them to develop related skills. For example, in considering variables, the students were required to identify the values they considered to vary from a written specification. To reinforce the process of identifying variables, their types, initial values and assigning appropriate variable names, the concept of a Variable Table was introduced (Figure 5-6).

Name	Type	Initial Value	Multiplicity	Description
productPrice	double	0	1	The price of each product
totalCost	double	0	1	The total cost of the products in the shopping basket

Figure 5-6 An Example of a Variable Table

Other tasks required the students to familiarise themselves with the flow of control by tracing the execution of code and documenting the changes in the variable values using a Trace Chart. On introducing branch statements, flow diagrams were used to enable the

students to visualize the change in the flow of control when conditions are introduced. In the first academic year that worksheets were introduced, tasks often required the students to provide a combination of a variable table, flow chart and code for the solution. However, in studying the student work, it was found flow charts added little to the students understanding of the code as they served only as a method of documenting the flow of control, a result also noted by Koppelman [26]. The work involved often proved a very onerous exercise for both the students' and the tutor. Thus, their use became more constrained later in the research.

The tasks themselves offered different levels of difficulty, becoming more challenging as the student progressed from one worksheet to another. Care was taken to ensure that simple tasks were always provided to introduce a new concept to the student. For example, when functions were introduced, the first task simply required the student to write the functions to solve some very simple problems Figure 5-7.

Write the following functions and write a test program to demonstrate how they are used.
1. SayHello: just displays a "Hello World" message, has no return
2. SayHelloToUser: passed the name of user and displays "Hello user", has no return
3. Sum: passed two numbers, sums them and returns the result
4. AddToTotal: passed two parameters the first being the current total, the second the new value to add to it and returns the new total
5. Average: passed current total, the number of values added to the total to calculate the average and returns the average value

Figure 5-7 A Task Introducing Function Declaration

This gradual approach to increasing the difficulty of tasks was important so that the students felt challenged but could still solve them [316].

In order to supplement the worksheets, the students were also required to investigate a number of the concepts covered in lectures and provide their own examples to illustrate those concepts. The objective of this assessment component was to require them to perform additional reading and attempt new approaches to applying the concepts. This also served to alleviate the problem of more advanced students being slowed down and becoming frustrated [324], because they were able to be more creative and develop code that interested them.

5.3 The Performance Metrics

A number of possible performance metrics were identified, some of which were object oriented specific. However, the initial range of worksheets covered only procedural

programming and to simplify the marking it was decided to apply the same set of metrics across all of the worksheets excluding the object oriented metrics as necessary. For novice programmers' work, it was felt that the Cyclomatic Complexity metric would be a sufficient measure of complexity for both procedural and object oriented exercises. Table 5-1 shows a series of metrics that were identified to analyse a student's ability across all the worksheets. Although mostly drawn from the metrics previously discussed, some were added by extrapolation from software analysis [326] or from pedagogical research [325].

Correctness	*Code correctness*	The number of errors in the student's program.
	Code completeness	How much of the task was completed or how many of the requirements were met?
	Testing completeness	How thorough was the student's test coverage.
	Testing validity	How appropriate were the student's tests?
	Syntax	How syntactically correct was the student's program(s)?
Style	*Annotation*	How descriptively and accurately has the student documented their code?
	Adherence to conventions	How strictly did the student adhere to the coding standards taught by the tutor?
Efficiency	*Performance*	How well does the program perform in terms of CPU cycles? If the task was too simplistic this measure was not used.
	Best Solution	How close to a good solution was the student's work? If the tasks were too short this metric was not used.
	Understanding	How well has the student demonstrated their understanding of the concepts and techniques? This was measured using both coding and/or a written analysis.
	Problem Solving	How logically has the student approached the problem?
Complexity	*McCabe's Complexity Analysis*	How complex was the program? Only used when the program size would justify it and was used to determine if the student had over complicated a solution. Low complexity was considered good.

Table 5-1 Programming Features and Metrics

5.4 Analysis of Metrics

Figure 5-8 shows the results obtained across all the metrics for students gaining both good and average marks, which have been normalised to account for slight variations in student numbers due to withdrawals and non-submissions. What becomes immediately clear is the larger diversity of issues present in the profile of the average student compared to the good student. Also, for average students, the number of incidents of poor metric values is far higher across the majority of the metrics. By the final worksheet,

incidents of poor code correctness and completeness are insignificant for the good students but remain persistent problems for the average student. Other areas of concern for average students are the understanding of coding principles, the ability to produce an implementation that approximates to the "best solution" and problem solving skills (Figure 5-9). .

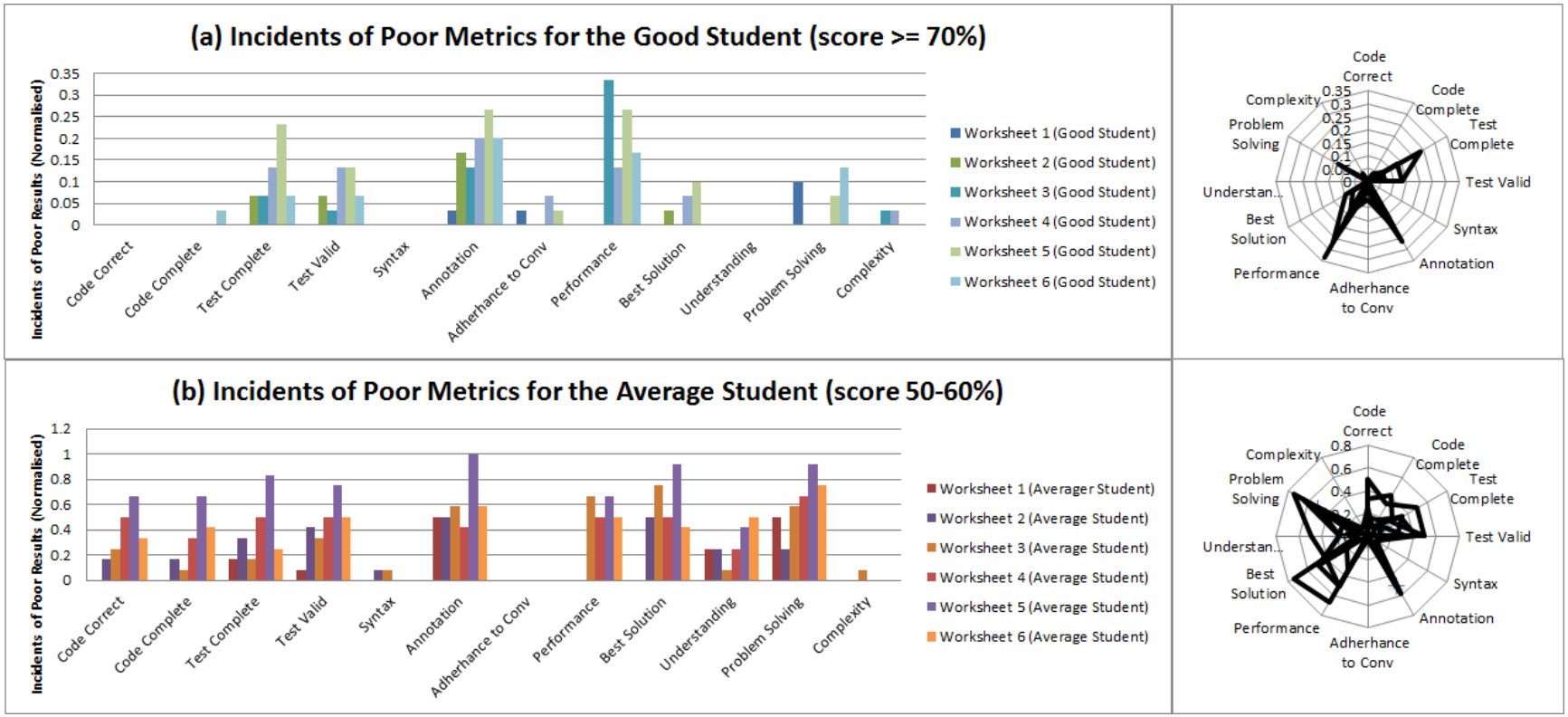


Figure 5-8 Analysis of the Metrics for Good and Average Students

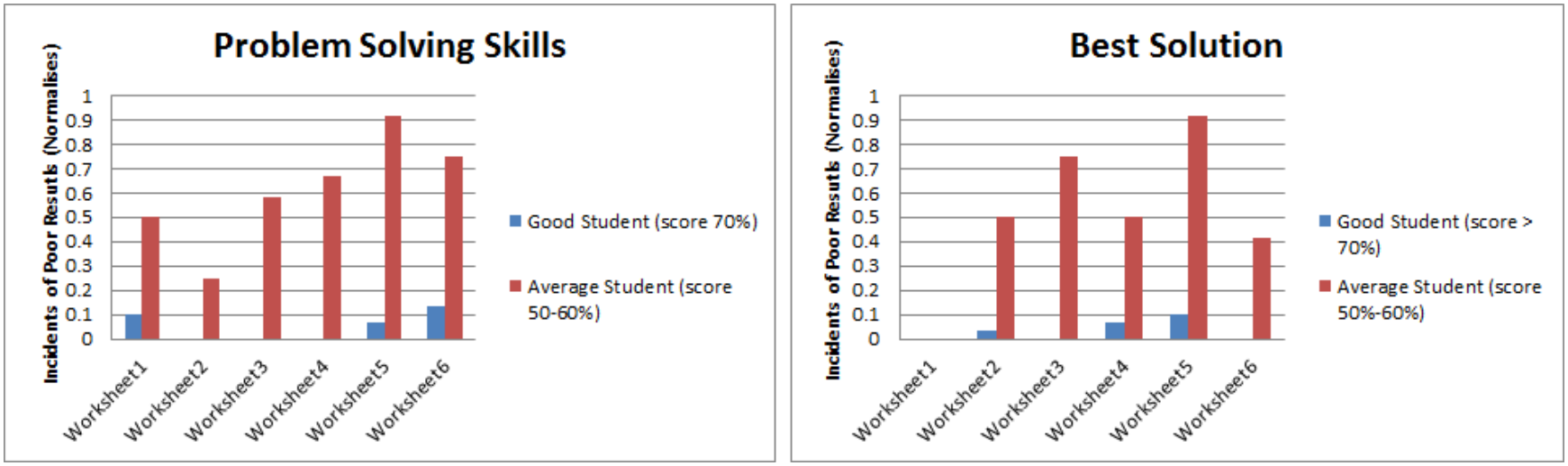


Figure 5-9 Analysis of the Best Solution and Problem Solving Metrics

Analysing the best solution and problem solving metrics in more detail (Figure 5-9), we see that for the average students the lack of problem solving skills seems to be an issue throughout the academic year. Unsurprisingly, when it comes to developing a good solution they also lag behind their more able peers and have far more difficulty in producing working complete code as the course progresses. (Figure 5-10).

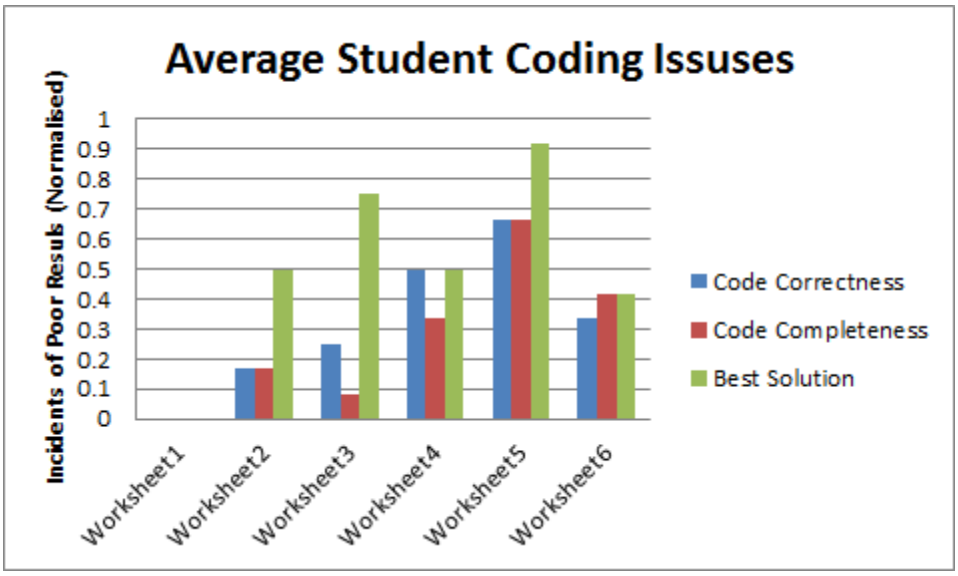


Figure 5-10 The Average Students Coding Performance

5.5 Seeking Common Success or Failure Factors Using Pattern Analysis

Having identified a number of metrics associated with programming performance and used them to obtain a dataset from the worksheets, it was possible to mine the data in an attempt to determine whether a common set of factors was associated with successful student performance. Pattern Analysis was chosen to seek sequences of metric values (patterns of items) that had statistical significance and might allow early prediction of a student who may have problems.

Typically, this form of data mining occurs in Market Basket Analysis where customers' purchasing habits are analysed by developing association rules based on the products they placed in their "shopping basket". An association rule takes the form "$A_1 \wedge \ldots \wedge A_m \rightarrow B_1 \wedge \ldots \wedge B_n$" where A_i (for $i \in \{1, \ldots, m\}$) and B_j (for $j \in \{1, \ldots, n\}$) are attribute-value pairs. Usually, this is written as $X \Rightarrow Y$ and has the meaning "the database tuples (i.e. rows) that satisfy the conditions in X will probably satisfy Y". Large quantities of rules are generated and a number of interestingness measures are applied to eliminate those which are considered to be uninteresting. A common measure is support (or prevalence). Given that

A and B are sets of items and D is a set of transactions then P(A $\cup$ B) the probability of both itemsets A and B occurring in set D can be written as a percentage of the transactions in the database (Equation 1):

$$\text{support}(A \Rightarrow B) = \frac{\#\_tuples\_containing\_both\_A\_and\_B}{total\_\#\_of\_tuples\_in\_D} \qquad (1)$$

The problem of deriving association rules is formally described [327] as follows: -Let J = { I_1, I_2, …., I_m } be a set of results. Each result tuple T in D is a set of results such that T $\subseteq$ J. A result tuple T is said to contain A if and only if A $\subseteq$ T. Association rules take the form X $\Rightarrow$ Y, where X $\subset$ J, Y $\subset$ J, and X $\cap$ Y = $\varnothing$. In this research, an item became a metric value , for example, “Code Completeness [Good]” became an item as did “Code Completeness [Poor]”. For mining purposes, the typical technique is to convert each item to a corresponding unique ID with a support count.

Common approaches for analysing this type of data include those based on the Apriori algorithm [328] or those based on tree algorithms such as the Frequent Pattern (FP) Tree algorithm [303] which forms the basis of this research. Both these methods apply constraints to remove itemsets considered to be uninteresting. A constraint C_{AM} is said to be anti-monotone if for every itemset that satisfies C_{AM}, every one of its subsets also satisfies C_{AM}. Given an itemset X, a constraint C_{AM} is anti-monotone if

$$\forall Y \subseteq X : C_{AM}(X) \Rightarrow C_{AM}(Y) \qquad (2)$$

If C_{AM} holds for X then it also holds for any subset of X. In its simplest form, an anti-monotone constraint is a support count that counts the number of occurrences of a given itemset against some threshold value.

A significant disadvantage of Apriori based algorithms is the generation of large numbers of candidate itemsets that must be eliminated. The advantage of the FP Tree algorithm is the generation of frequent itemsets without candidate generation. However, it requires two passes of the database. Firstly the database is scanned to construct a list of frequent 1- itemsets (i.e. a set containing one item) that are ordered in terms of frequency from highest to lowest. A second scan orders the items in each item tuple accordingly, each item becomes a tree node and branches of the tree are built so that items with the highest frequency appear at the top of the tree.

Formally, let I = {a_1, a_2, …., a_m } be a set of items and DB = { T_1, T_2, …., T_n } be a database of item tuples where T_i($i \in [1..n]$) is an item tuple which contains a set of items in I. The support of a pattern (or itemset) A is the number of item tuples containing A in DB. Given a predefined minimum support threshold ξ then A is a frequent pattern if its support is no less than ξ. In the FP tree, each node consists of an *itemid, count* and *node-link.* The *itemid* identifies the item the node represents, *count* is the number of item tuples used to add (or support) this node and the *node-link* links to the next node of the same *itemid* or null.

Having constructed the FP-tree, it must be mined to generate all the frequent patterns using the FP-growth algorithm. FP-growth takes each node in the tree and searches for patterns containing that node which conforms to the anti-monotone constraint that takes the form of a support threshold. For analysis and prediction purposes, these frequent patterns or n-itemsets (i.e. a set containing n items) can be compared with future item tuples to identify matching itemsets.

To make predictions, for example the pattern of behaviour associated with high grade students, separate trees were generated for each grade value. This required a scan through the item tuples looking for tuples containing the required grade and then passing the matching tuples to the FP-Tree and Growth algorithms.

Data mining usually requires many thousands of records to build up statistical confidence in the results being mined by the algorithms. A clear limitation of this study, and indeed of any application of data mining as a pedagogical tool, is the lack of such large quantities of data. Given the limited dataset, this research should be seen as a pilot study intended to determine if this approach could lead to identifying different patterns of behaviour between students at different grade levels. However, given that the factors being measured need some interpretation, it was argued that any observed patterns could be determined at lower levels of data because more human analysis was required i.e. a much smaller number of “interesting” patterns are important since we are not attempting to find sub-patterns in large itemsets. Even with relatively small amounts of data, the number of patterns generated ran into hundreds and support thresholds had to be modified accordingly to remove the less supported/less interesting itemsets.

5.6 Results of Mining Worksheet Data

When comparing the itemsets generated by students gaining good, average and poor marks for each worksheet, an important distinction must be made, students obtaining a high overall final grade, such as a First, could still obtain a poor result for an individual worksheet. Thus the analysis was not based on the student's final grade but on the individual results for each worksheet. The support counts for the itemsets generated for students achieving good grades were significantly higher than for those achieving lower grades. Since the majority of students would be expected to pass a course, the patterns produced were overwhelmingly positive i.e. predictors of success rather than failure, and this made analysis of the causes of failure less predictable. Hence, the "fail" results need be treated with some caution. The cause of failure could of course also be predicted by the lack of the success predictors. To improve the analysis of the results, the 5-itemsets were studied alone because the support for 6-itemsets became much lower and sufficient interesting metrics were generated at this point. Care was taken to ensure that the 5-itemsets reflected the values seen in the 2, 3 and 4-itemsets which in any case would have been combinations extractable from the 5-itemsets. Furthermore, the support threshold (ξ) was adjusted for each grade (awarded for each worksheet) to eliminate less supportable itemsets. The threshold had to be reduced to study assessment marks/grades at the lower end of the scale due to the reduction in the data available, since assessments are intended to enable students to learn and not to fail them unnecessarily.

The study analysed the worksheet data from two perspectives. Firstly, by comparing the metric scores obtained by all the students across the worksheets to determine the difference between good and poor performance in completing the tasks (Figure 5-11). Secondly, the data was subdivided based on the student's overall final grade to reveal differences in overall performance since it was anticipated that student's performance was likely to vary across the worksheets i.e. the later worksheets covered concepts that were likely to be challenging for more of the students (Figure 5-12). Not all the metrics appeared in the mined itemsets because the itemsets in which they occurred had lower support values, and had been culled when they failed to meet the support threshold i.e. they were not considered significant predictors of performance.

5.6.1 Analysis of Results

Figure 5-11(a) shows the results of analysing the data across the worksheets without reference to the final grade obtained by the student. The results show that students achieving good marks for a worksheet, wrote more complete and correct code than the other students who produced poor code and often failed to complete it. The same is true for testing completeness and validity.

However, it is also true that problems with code syntax are insignificant, from Figure 5-11(c) we can see that this in itself was not a predictor of poor performance. This suggests that students across all the grades are able to correctly apply the syntax but their main difficulty appears to have been related to obtaining a solution in a form that could be coded. We can conclude that if a student understood the problem and its solution then they were capable of writing the code. To determine the quality of the code produced, although somewhat subjectively, the closeness of the code produced by the student was compared to the solution(s) that the lecturer considered to be the "best" solution. It was found that poor students were unable to produce code of a good standard because they lacked the skills to produce well thought-out code, but the metric itself was not a predictor of good performance. We can conclude that not all students achieving good grades produce good code, they just produced more working code. Likewise, code complexity was also not a predictor of performance.

There is some evidence that annotation of code may be a predictor of performance, with poorer annotation being produced by both the average and poor student. Good commenting should be emphasised, otherwise students often fixate on the code and treat documentation as an afterthought. This may also be related to a good understanding of the solution, given it is likely that students are more able to document their code when they have a thorough understanding of it. A related metric measured the ability of the student to abide by a specified coding convention, but this was found not to be a predictor of poor performance. Essentially, a number of the poor students were able to follow a convention even though their code was poor.

Figure 5-11 clearly demonstrates that while the students gaining good marks mainly show good/average problem solving skills, lack of problem solving skills was a clear indicator of poor performance. The students were asked to provide written reports describing the concepts discussed in class and to provide their own examples to illustrate these ideas.

However, another key indicator of poor performance was the lack of understanding, and this included documenting the concepts and ideas covered in class. This form of assessment in itself now appears to play less of a role in supporting the development of the weaker students' programming skills than first thought.

The students obtaining average marks were of interest because they were capable of producing complete and correct code, but also often demonstrated a number of issues with applying algorithms and logic to a new task. Even providing a formal structured design approach using variable tables, code trace charts, flow diagrams and UML class diagrams did little to resolve these problems.

5.6.2 Relationship between Final Grade and Worksheet Metrics

To associate the worksheet results with the final grades achieved by each student during the mining process, extra items were added to the worksheet data representing the grades First, 2.1, 2.2, Pass and Fail that they obtained. Figure 5-12 shows the results of mining data for students with respect to their overall grades. Between students achieving a good grade and an average grade, these results show a significant drop in correct and complete code with rises in metrics including poor annotation and variance from the best solution, but the most significant change is a large spike in problem solving difficulties i.e. problem solving is a significant predictor of coding difficulties. As previously noted, the data obtained for poor students is less statistically relevant. Therefore, the analysis of their results is for illustration and comparison purposes only, but Figure 5-12(b) and (c) do seem to confirm that annotation and problem solving difficulties are predictors of coding problems. The overall conclusion from this analysis appears to be that the dominant predictor of poor performance in coding is poor problem solving skills.

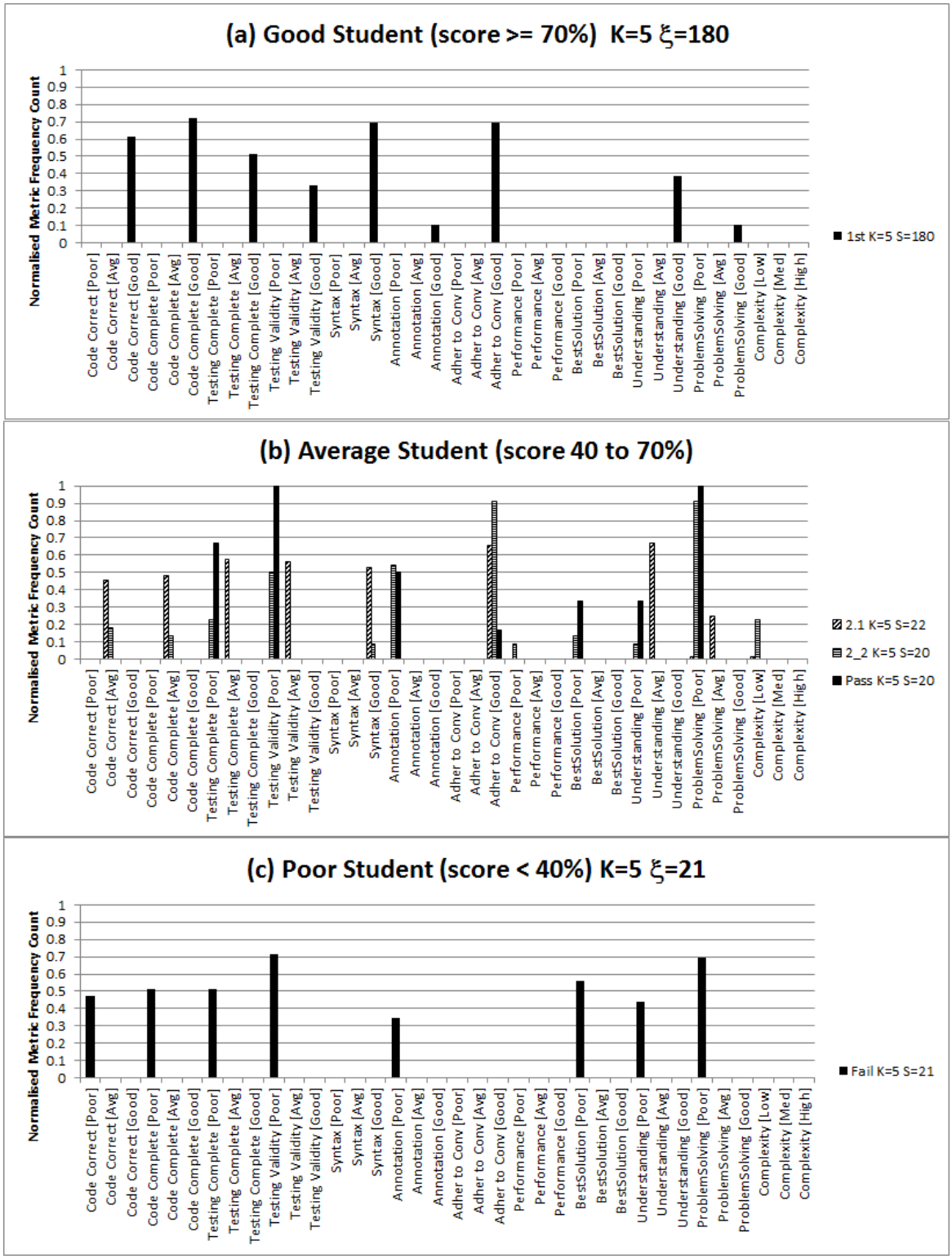


Figure 5-11 Comparison of Metric Frequency Counts within Itemsets at K=5 for Students Gaining Good, Average and Poor Grades in a Worksheet

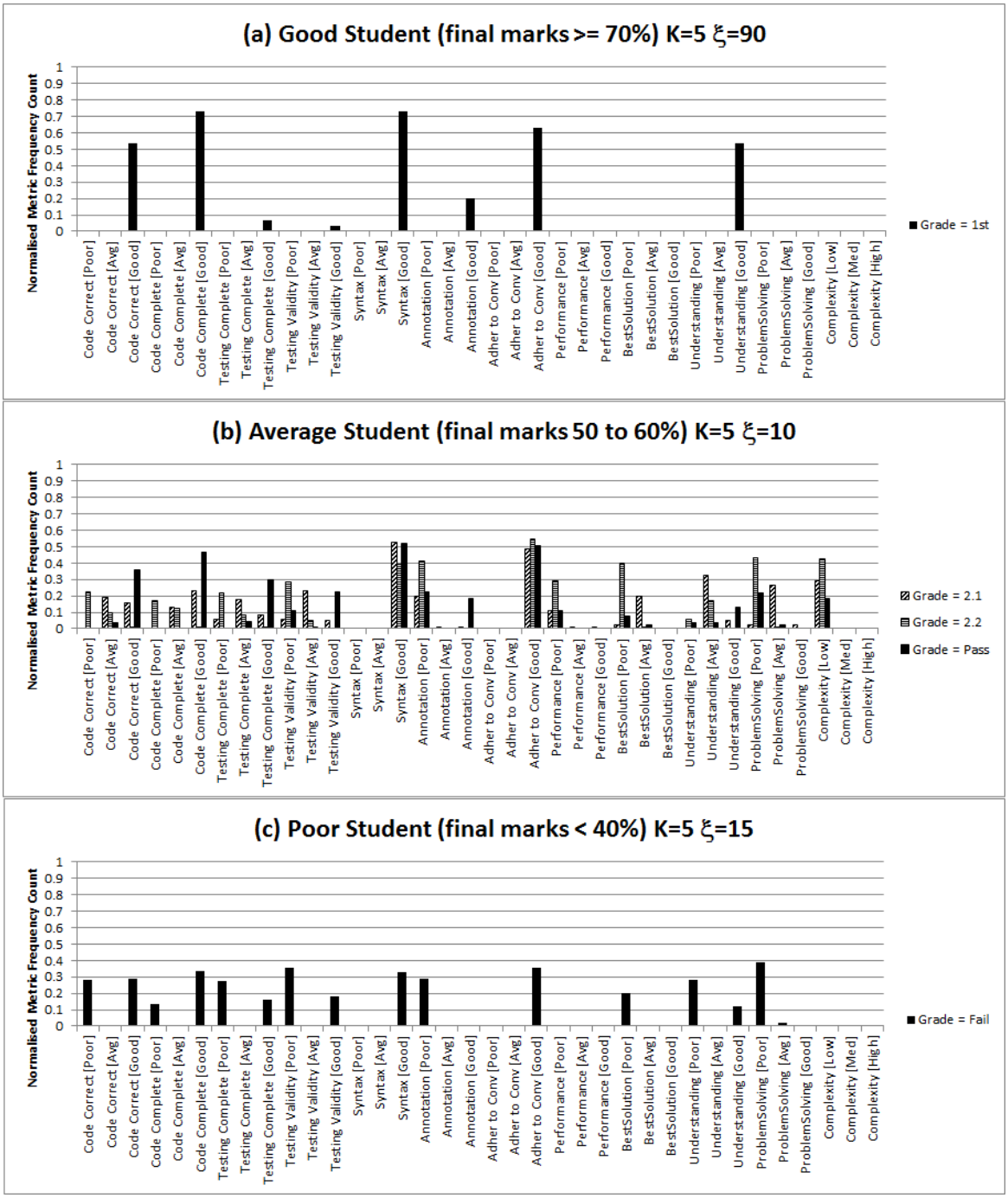


Figure 5-12 Comparison of Metric Frequency Counts within Itemsets at K=5 for Students Obtaining Good, Average and Poor Final Grades

5.7 Confirmation Trial

A confirmation trial was conducted consisting of 89 first year students. The course was taught in the same way with the same worksheets but given the problems previously noted with dropout rates, only the first four procedural coding worksheets were used in this study. Figure 5-13 and Figure 5-14 show that both the final results obtained by the students and the grades awarded for each worksheet are broadly in line with those obtained in the main study. The same polarization of results can be seen and the marks awarded for the worksheets show preponderance around the First and 2.2 levels. Figure 5-14 also shows that only a relatively small amount of data is available to study failing students, again making specific analysis of these students much less statistically relevant.

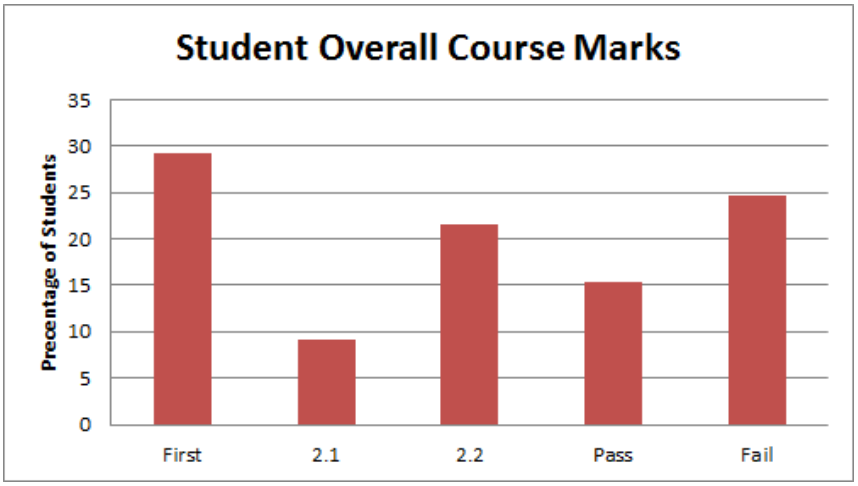


Figure 5-13 Overall End of Year Course Marks Obtained by the Students in Confirmation Trial

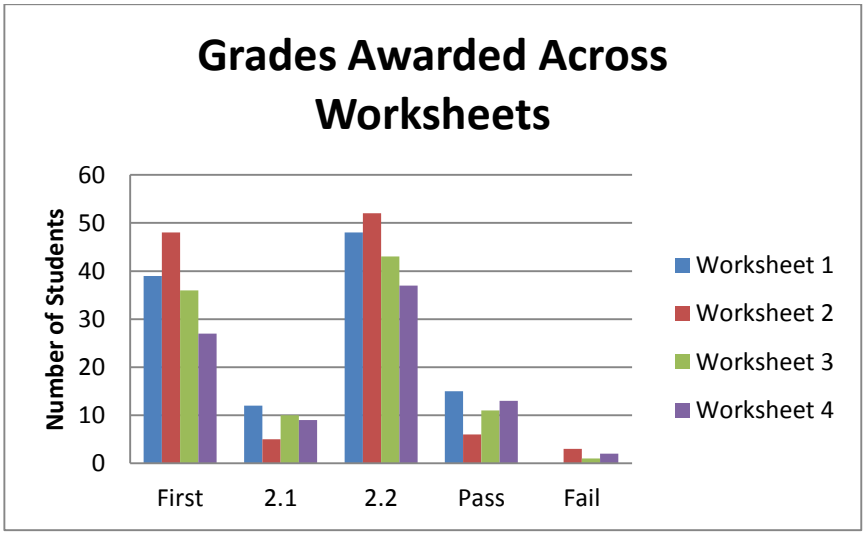


Figure 5-14 Grades Awarded to Students for Each Worksheet in Confirmation Trial

Similarly, the differentiation in problem solving skills between the good and average student can also be seen in the analysis of the results (Figure 5-15). It is therefore possible to conclude that the main study and the confirmation trial produced similar results.

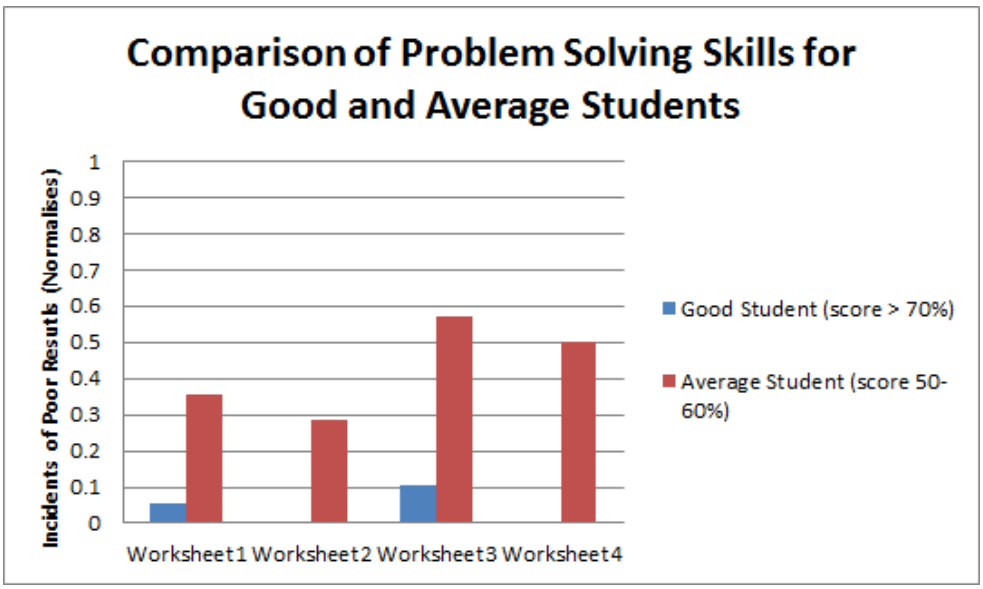


Figure 5-15 Analysis of the Problem Solving Metric in Confirmation Trial

The FP-Tree data created in the main study was used to mine the corresponding frequent patterns using a Market Basket Analysis approach. The objective was to determine if the pattern matches obtained in the main study i.e. the predictions, corresponded to those obtained for each grade in the confirmation trial. Each student profile (set of metrics) for each worksheet was analysed to obtain a set of matches. A large number of matches were obtained, and the longest matching patterns (K = 5) with sufficient statistical support were chosen for review. Due to the variety of matches produced, this approach was not found to be suitable for predicting a student's final grade with any accuracy.

5.8 Conclusions

The three purposes of this research were to promote continuous practice through a set of worksheets containing a variety of exercises, to investigate whether metrics for analysing student code could be used to predict good or poor student behaviours, and to determine whether such behaviours could be used to predict student performance. Given the bimodal distribution of the students' final results it seems that carefully structuring the course content and providing associated exercises in the form of worksheets, does not in itself give novice programmers adequate support and fails to overcome the inherently unforgiving nature of programming. In analysing the metrics, market basket analysis was applied to obtain patterns of metric values associated with students at different grade

levels. Although the patterns of behaviour between grade levels did look different, no specific patterns of behaviour were found to be associated with good or poor student performance. A result that was also confirmed in a separate trial that demonstrated that the approach had no significant predictive power. The items (metric values) in each pattern were counted to produce summaries showing the most significant metric values associated with each grade level i.e. the metric values associated with the most patterns (or common behaviours). Using these summaries, the students' behavioural differences could be reviewed between grade levels at both the individual worksheets stage (Figure 5-11) and the overall final marks stage (Figure 5-12). This analysis revealed that the key metric associated with programming success was problem solving.

Although both the main study and the results from confirmation trial demonstrated that problem solving was one of the main causes of programming difficulties, neither suggested the root cause of poor problem solving abilities in the context of programming. It may be related to a lack of deductive, logical reasoning ability or fluid intelligence (gF). Alternatively, it may just be a problem of lack of practice or student motivation to learn programming. Should a lower gF measurement be associated with poorer programming ability, then it would be possible to measure this at the beginning of a programming course and potentially identify students that are likely to struggle.

6 Predicting Potential Programming Success

A study was conducted to determine if programming success could be underpinned and predicted by providing an accelerated learning course in computational thinking prior to the start of the academic year, at the conclusion of which a number of tests were performed. The study involved 168 students entering the first year of their programmes of study and 4 members of academic staff over two years. These students were of a mixed range of ages and drawn from the full range of computing programmes offered by the School. The aim of the computational thinking course was to focus on problem solving skills, in the context of creating Python programs to solve a range of problems.

On completion of the computational thinking course, the students were required to complete a programming test and a Raven Advanced Progressive Matrices (APM) Set II test. The APM test was chosen instead of the Standard Progressive Matrices, since APM is targeted at adults of a higher-level educational ability [329]. There is also some unpublished evidence referred to by Raven [329], that the APM test administered without a time limit is a good predictor of computer programming success. This might be due to the need for similar levels of attention to detail, checking and persistence required for success [329].There is a strong correlation between APM and fluid intelligence gF [70], and gF is related to problem solving skills [12],[62],[63],[17]. Since the deficiency of problem solving skills has been identified as a key factor in programming success, the relationship between gF and programming skills needed further investigation. Therefore, this study sought to determine whether the Raven test results could be used to predict student programming performance.

6.1 The Testing Methodology

The programming test took the form of a two hour time restricted test (see Appendix 1) and was subdivided into four sections consisting of (i) the analysis of a natural language problem and conversion to procedural steps, (ii) appreciation of code design, (iii) understanding of programming logic and (iv) the ability to write code. Question 1 was a variation of the classic “Making a Cup of Tea” exercise (Figure 6-1) and was used to assess the student’s ability to interpret a natural language problem.

Produce program for a new robot intended to create hot drinks. The robot is capable of following simple, tea and coffee-oriented commands precisely, but has no understanding either of the process, or the fundamental principles which underpin it (e.g. that a kettle requires power). The robot has access to the following items:

- Kettle (initially unplugged)
- Tea bags
- Jar of ground instant coffee
- 1L carton of milk
- Unopened bag of sugar
- 1 metal tea spoon
- 1 large mug
- Access to a sink for water and an electrical socket for power

Figure 6-1 Assessment of Natural Language Reasoning and Structured Logical Thinking

The instructions specified that the student should produce a set of instructions for the robot to successfully make a cup of milky coffee with one teaspoon of sugar. Further guidance stated that each instruction should be on a new line, written in a logical order with no steps missed. An additional component of this exercise allowed a user to specify the required drink option before the robot made it. It was intended that this exercise would test the ability to logically define the steps required to solve the problem, and determine the student's ability to think procedurally and logically. There was no requirement for the student use any formal language in detailing the solutions.

Question 2 was a flow chart exercise (Figure 6-2) used to assess the student's appreciation of code design. Although there were some reservations in using flow charting [26], it was felt that determining the students' understanding of "flow of control" was sufficiently important to overcome any objection and any disadvantages could be reduced by simplifying the exercise.

a) *Says "Hello" to the user at the start.*
b) *Asks the user how many addition operations they would like to perform.*
c) *Loops the number of times requested by the user.*
d) *For each loop, takes two new numbers from the user, adds them together and outputs the result*

Figure 6-2 Extract from Question 2 Flow Chart Requirements Presented to Student

Question 3 was designed to test the student's ability to interpret requirements and develop the logic for a program. The students were allowed to skip elements of the problem they found too challenging and the answers could be provided in Python, pseudocode or simply outlined in English. The objective was to determine the mental model constructed by the student and their ability to express it in a logical manner. They were told not to concern themselves with implementing a complete solution, as explained in the statement shown in Figure 6-3.

> "*You should aim to implement as many of these requirements as possible within your Python solution. Focus on the logic of the program, and do not worry unduly about syntax. If you feel a requirement will be too difficult to implement, ignore it and focus on the others.*"

Figure 6-3 Extract from Question 3

Finally, Question 4 was a coding exercise provided to assess coding ability, by requiring students to write programs in Python to draw various shapes using Turtle. This was subdivided into 3 component parts that involved increasing levels of difficulty, with the level of difficulty stated in the exercise.

The answers to these tests were subdivided between the academic staff members and marked separately to avoid any marking bias and were cross checked to confirm consistency in marking.

In addition to the coding test, the marks awarded at the end of the academic year for programming assignments completed by these students were also obtained. Since these students were studying a range of programmes taught by a number of members of staff, the coding related assignments were taught using a range of different approaches and languages. Where multiple assignment results were obtained for a student, the mean was taken to produce an overall assessment grade. Hence, any bias that might have occurred by teaching using a specific language, approach or marking scheme was negated.

The APM test was conducted following the procedure outlined in Raven Manual 4 [330]. The student group were told in advance that they would be required to complete the test. APM Set I was used to familiarise them with the thought process required to solve the problems. The first two items of the Set I test were used as examples to ensure familiarity with the test procedure. The students were then given unlimited time to

complete the remaining tests in Set I followed by the tests in APM Set II. After scoring the tests, the results were compared with the UK reference norms Tables APM 12, APM 13 and 14 [330].

6.2 Results of Programming and Raven Tests

The results of these tests are shown in Table 6-1 and Figure 6-4, and have been subdivided into data bins covering different score ranges. Excluding Question 1, which required an interpretation of a natural language problem statement, these results show that a significant number of students were awarded marks under 40% across all the remaining questions and the Raven test. However, this analysis does not reveal whether a specific group of students gained poor marks across all the tests. Variations in the student counts shown in Table 6-1 are due to students not answering those particular questions. Furthermore, a lower number of assignment results were available for study due to reasons that included students leaving their studies, changing course or failing to submit.

	Student Totals in Each Score Range						
Marks	**Raven Full Test Result**	**Code Test Results**					**Assignment Result**
		Q1	**Q2**	**Q3**	**Q4**	**Result**	
Over 90%	13	18	25	22	38	9	0
80-89%	32	35	18	14	33	22	21
70-79%	16	36	13	23	5	22	18
60-69%	23	37	19	26	18	31	16
50-59%	17	24	11	11	8	25	27
40-49%	4	11	27	8	15	18	22
Under 40%	61	6	55	64	51	44	15

Table 6-1 Results of Coding and Raven Test (Final Assignment Mark also shown)

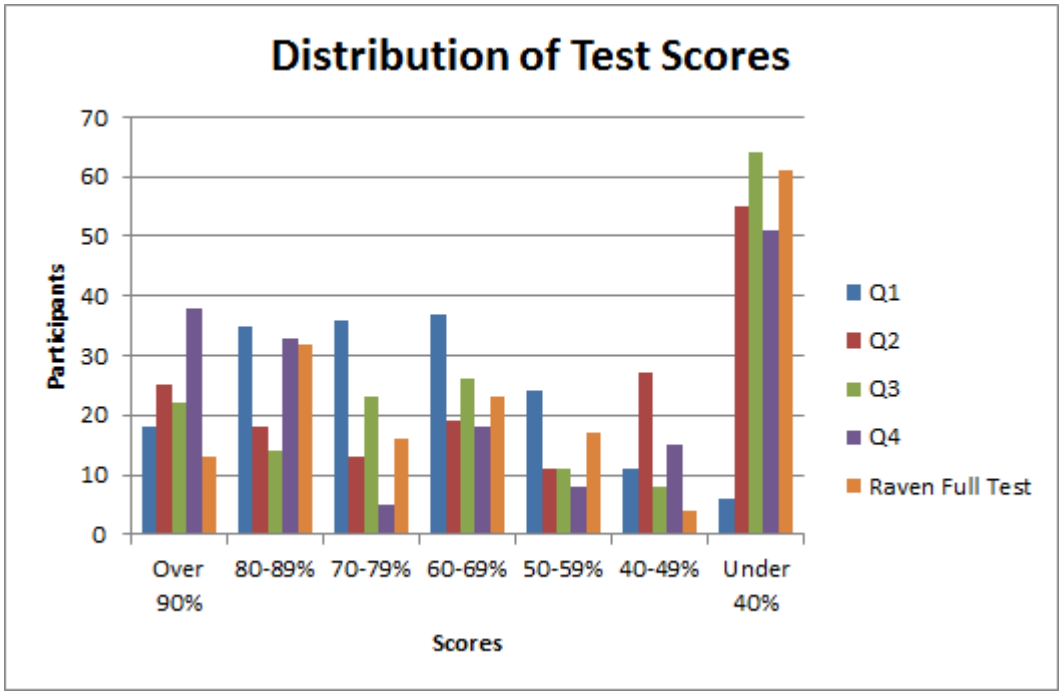


Figure 6-4 Distribution of Test Scores

In a typical programming course, results would be normally classified as first class in the range 70% or over, 2.1 in the range 60% to 69%, 2.2 in the range 50 to 59, 3rd in the range 40 to 49 and fail at below 40%. Binning the test scores according to these classifications gives Figure 6-5, which demonstrates the familiar bimodal distribution. These results are particularly evident in questions 3 and 4 which dealt specifically with the coding of a solution. Thus, the scores from these questions alone may have been the major contributors to the effectiveness of the testing that was conducted.

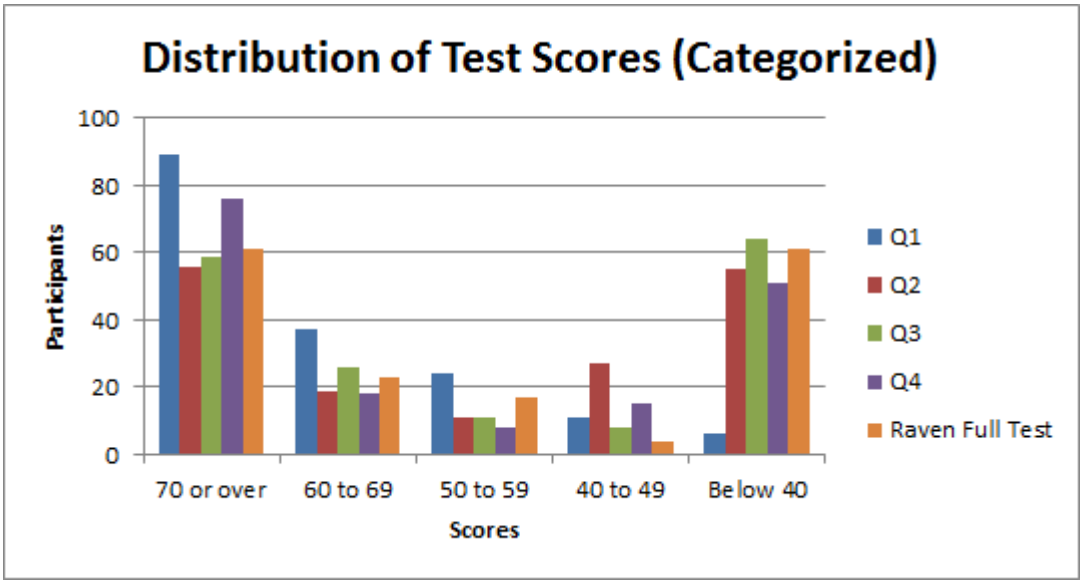


Figure 6-5 Distribution of Test Scores using Larger Bin Size

6.2.1 A Comparison of Code Test Results with Final Assignment Marks

Although the coding test was time restricted, the assignment was completed by the students over the duration of the academic year and this gave them an opportunity for self-study and practice. An analysis was performed to determine whether there was any significant change in the participants' performance after a year of study. Since assignment marks were not available for all students, the sample size had to be reduced to 118 pairs where both the code test and the corresponding assignment marks were available. To identify which matched pair tests could be performed, it was necessary to assess whether the distributions of both these sets of marks followed a normal distribution. SPSS provides two tests for normality, the Shapiro-Wilk and the Kolmogorov-Smirnov tests. The Kolmogorov-Smirnov test has been found to be less powerful [331] and will be ignored. Furthermore, the SPSS documentation recommends that these tests are only applied when the sample size is less than 50 [332, 333]. Normality should therefore be assessed visually [331]. Inspection of the results suggests that neither the distribution of the code test marks (Figure 6-6 and Figure 6-7) nor the distribution of the assignment marks (Figure 6-8 and Figure 6-9) appear to be normally distributed. Indeed, the plots suggest a normal distribution around 60% with a bimodal distribution with peaks around the 40% and 90% marks. Therefore, some caution must be exercised in any discussion of statistical results reliant on a normal distribution.

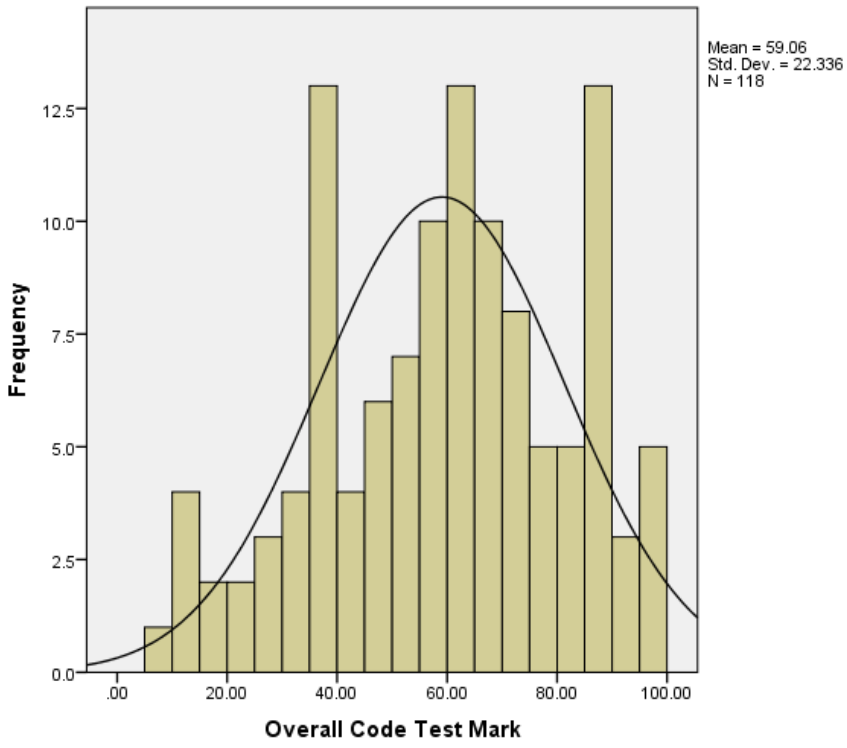


Figure 6-6 Distribution of Overall Code Test Results

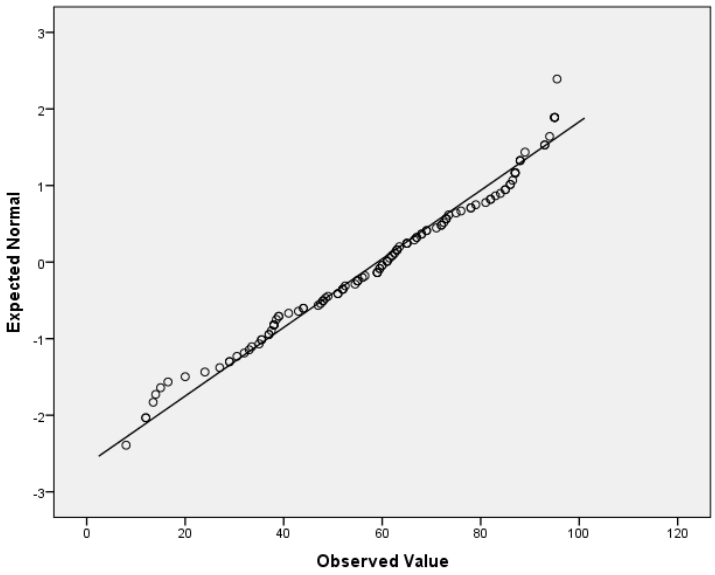


Figure 6-7 Normal Q-Q Plot of Overall Code Test Results

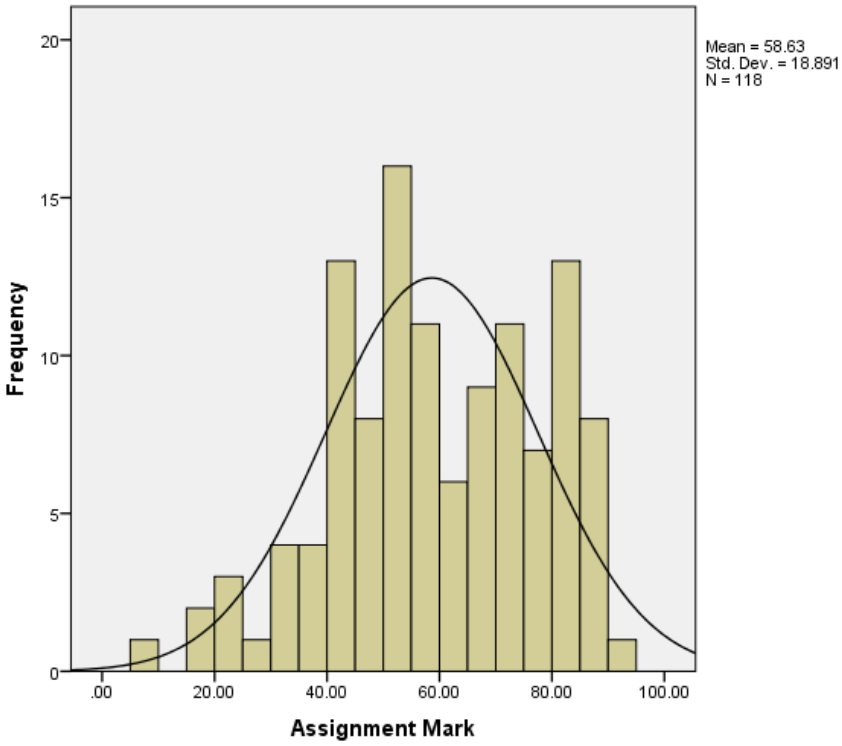


Figure 6-8 Distribution of Assignment Marks (Excluding Zeroes)

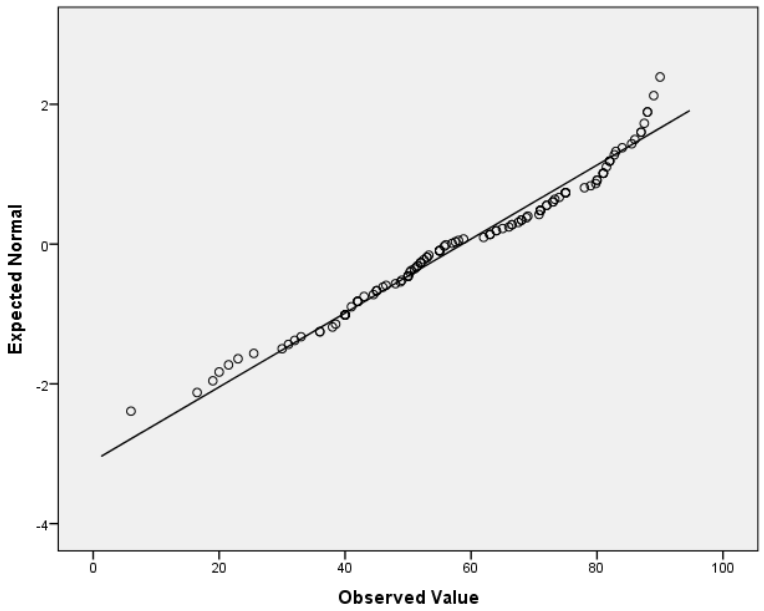


Figure 6-9 Q-Q Plot of Assignment Results (Excluding Zeroes)

For completeness, the Shapiro-Wilk test results were also recorded as:

	Statistic	Degrees of Freedom (df)	Significance (p)
Overall Code Test Results	0.970	118	0.010
Assignment Results	0.971	118	0.011

The Shapiro-Wilk test gives a p value below 0.05 that enables us to reject the null hypothesis that the values are normally distributed and confirms the original visual observation. Again, it must be noted that the Shapiro-Wilk test is considered to be more appropriate for small sample sizes (N <=50, in their seminal paper [332], Shapiro and Wilk only simulated data with a maximum N of 50).

Table 6-2 shows the comparative statistics for both these distributions, and it is clear from these results that the mean, median and standard deviations are very close. The Skewness and Kurtosis values show that both curves have the data slightly skewed to the left (the left tail of the distribution is slight longer) and are slightly flatter than the normal distribution. Therefore, the results do show very similar characteristics.

Distribution	Mean	Median	Std Dev	Skewness	Kurtosis (excess)
Code Results	59.06 Std Err: 2.05	61	22.336	-0.284 Std Err: 0.223	-0.662 Std Err: 0.442
Assignment Results (Excl. Zeroes)	58.63 Std Err: 1.739	56.5	18.89	-0.296 Std Err: 0.223	-0.532 Std Err: 0.442

Table 6-2 Comparison of Distributions of Code and Assignment Results

Skewness and Kurtosis values of -0.284 (std err of 0.223) and -0.662 (std err of 0.442) respectively, giving z-scores of 1.27 and 1.5 respectively, both of which are less than ±1.96 suggesting the distribution is normal [334]. However an alternative approach suggests that a normal distribution requires Skewness and Kurtosis values to be within the ±1 range and less than three times the associated standard errors. Both of which are also true in our case. Similar results are obtained for the distribution of the assignment marks, with Skewness and Kurtosis values of -0.296 (std err of 0.223) and -0.532 (std err of 0.442), giving z-scores of 1.33 and 1.2 respectively. Therefore, although there is some evidence of normal distribution of marks, the safest conclusion must be that both these distributions are not normally distributed and the most appropriate methods for analysis must be non-parametric.

The distribution of the differences between the two related groups (Figure 6-10) is quite symmetrical in shape but there are a number of outliers at the extremes. This potentially rules out the possibility of using the Wilcoxon matched pairs signed rank test (although the result of this test was Z=-0.113 with p=0.910).

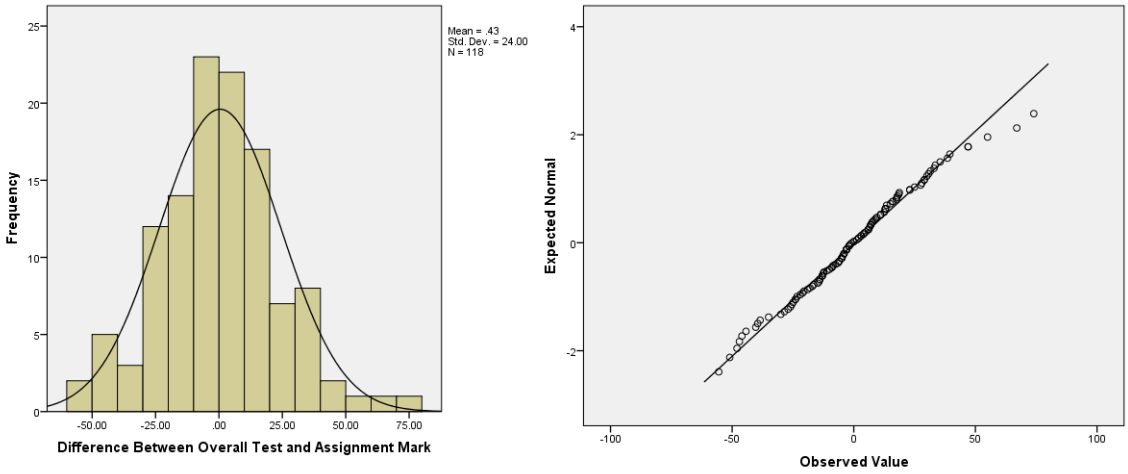


Figure 6-10 Distribution of the Differences between Overall Code Test and Assignment Results

Therefore the sign-test was chosen, a test that does not rely on the data following a normal distribution. The result of this test (Table 6-3) were Z=-0.093 and p=0.926 indicating that there was not a statistically significant change in the results post teaching of programming. 57 students did worse than the test suggested and 59 did better, with a mean of 0.43 and a standard deviation of 24. This suggests that the overall code test marks mirror the performance of the students over the academic year.

Assignment - OverallTest	Negative Differences[a]	57
	Positive Differences[b]	59
	Ties[c]	2
	Total	118

a Assignment < OverallTest
b Assignment > OverallTest
c Assignment = OverallTest

Table 6-3 Sign Test Results

In analysing the individual results for each question in the test, the sample size needed to be varied. Not all the students completed the coding questions and some also failed to attempt the Raven test (Table 6-4). Little information can be gained from considering students that did not attempt both these tests, and they were therefore excluded. The sample size used when comparing coding tests and the Raven test was adjusted accordingly.

Total student in study	171
Non-Attempts at Coding or Raven Tests	
Q3	16
Q4	8
Q3 and Q4	16
Assignment	51
Raven	1
Sample Size Adjustments	
Q3 Excluding non-attempts	139
Q4 Excluding non-attempts	147
Assignment Sample Size	120

Table 6-4 Non Attempts and Effect on Sample Sizes

To further evaluate the effectiveness of the code test in predicting the programming ability of the students, a k-Means Cluster Analysis was performed (Figure 6-11), and was found to produce 5 clusters (k=5) with the centroid values shown in Table 6-5.

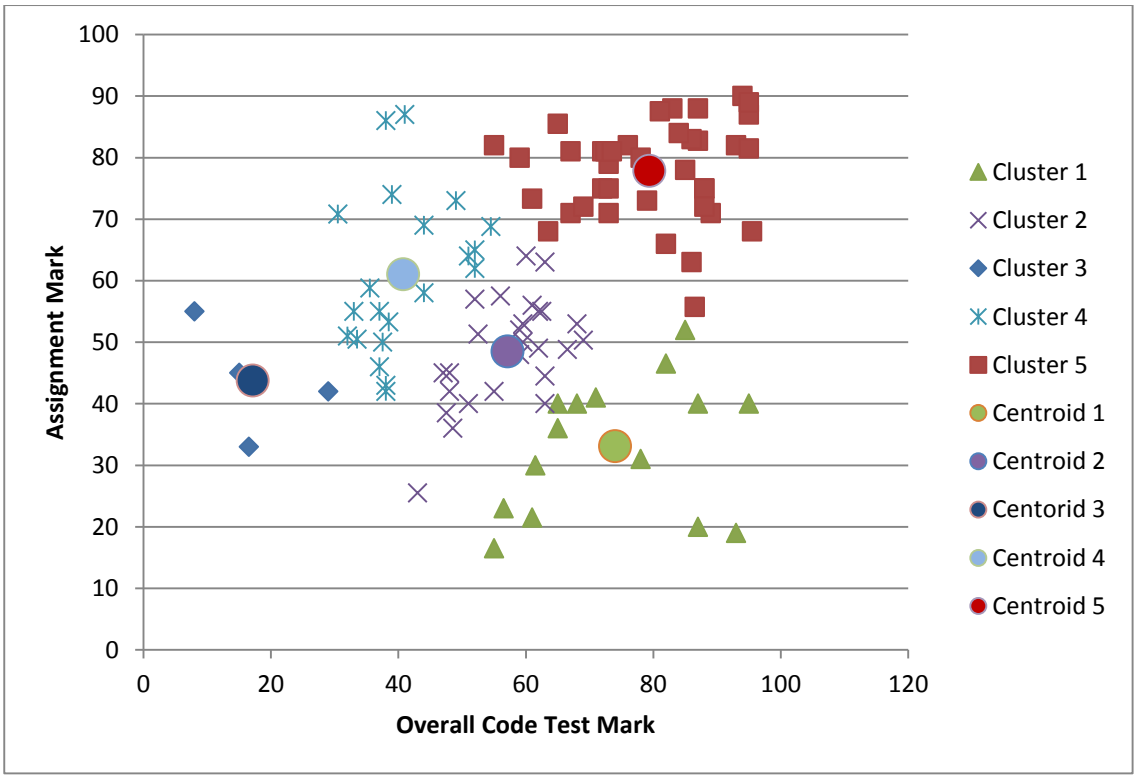


Figure 6-11 Cluster Analysis of Code Test and Assignment Results

	Centroid		Cluster Size	
	Q Overall	Assign		
Cluster 1	74	33	15	Performed worse than predicted
Cluster 2	57	48	26	Average performance in both
Cluster 3	17	44	4	Performed poorly in both
Cluster 4	41	61	21	Performed better than predicted
Cluster 5	79	78	38	Performed well in both

Table 6-5 Centroid Values from the Cluster Analysis of the Overall Code Test Results With Respect to the Final Assignment Marks Obtained

Cluster 5 shows that students gaining high code test marks also gained high assignment marks. Cluster 3 shows that students achieving lower marks in the code test were also achieving lower assignment marks, although this is a very small cluster most likely due to norm-referenced assessment. The remaining clusters, Cluster 1, Cluster 2 and Cluster 4, represent the average student and show students performing below, in-line with and above expectations respectively. In terms of size, Cluster 5 is the most significant and this indicates that these students maintain their advantage, followed by Cluster 2 showing average performance in both tests. Together, these account for 62% of all students. This analysis suggests that the code test is able to most accurately predict performance for students that gained good or average test results.

6.2.2 A Comparison of Code Test and Raven Matrices Test Results

The inclusion of Raven Matrices tests in this study, allowed further analysis of the code test results. Firstly, Figure 6-5 shows that the distribution of Raven scores is broadly in agreement with the test scores obtained. However, this figure does not specifically demonstrate the relationship between each student's results and their Raven APM scores. This relationship is shown in Figure 6-12, generated using the APM 14 Norms, and again this appears to show that there is a correlation between the Raven test scores and mean code test and end of-course assignment marks i.e. they have a monotonic relationship, with higher Raven scores being associated with good coding ability. The data binning seen is due to the Raven scoring methodology (see Table APM13 and 14[330]). Applying linear regression to the means of the data bins gives a coefficient of determination (R^2) value of 0.6743 and 0.6012 indicating moderate significance.

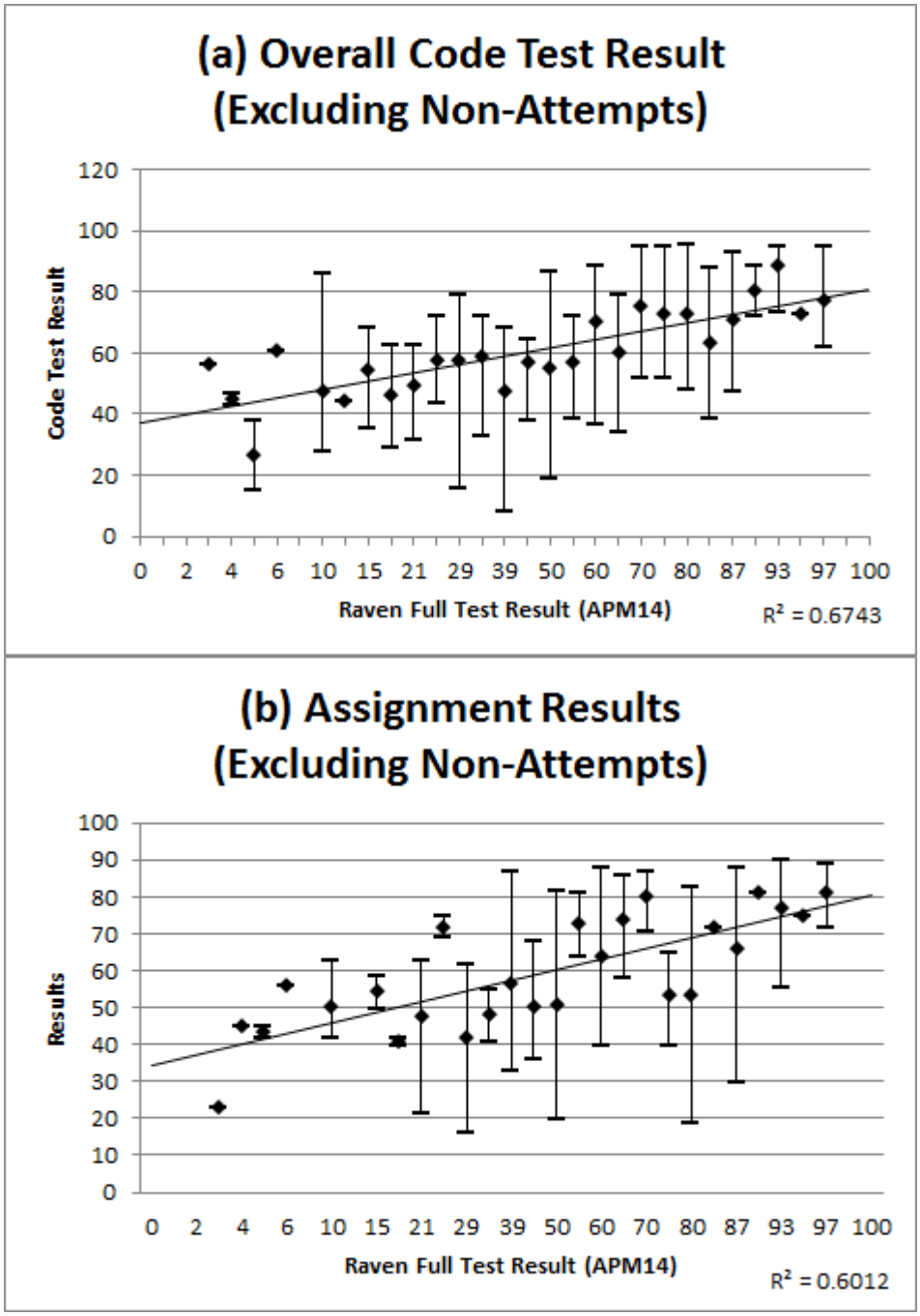


Figure 6-12 Comparison of Raven APM Scores to Student Results using APM 14 Norms

However, by applying the APM 13 Norms we obtain a less finely detailed scoring method which reduces the number of data bins produced and the effect of the outliers. These results (Figure 6-13) show a much stronger correlation between the Raven Matrices test scores and student results. Again, analysing the linear regression of the mean values of these data bins we get the R^2 values of 0.9794 and 0.8202. Thus, we can conclude that Figure 6-12 and Figure 6-13 both show some correlation between code results and Raven test results. However, Figure 6-13 (a) shows this relationship becomes more linear when the size of the data bins is increased and the effect of outliers on their mean values is

reduced. The slight reduction in correlation observed when comparing Raven's test results with assignment marks may be explained by:

1. the lower number of students submitting assignments
2. students rewriting their code over an extended period of time giving them the opportunity to obtain a result, although with a less efficient working process

Despite these factors the results still demonstrate a correlation: indicating that students with a high Raven's test score retain their advantage. However, the use of Raven's for predicting an individual student's future assignment marks may be affected by the amount of time and effort they are willing to commit to overcome their working memory limitations.

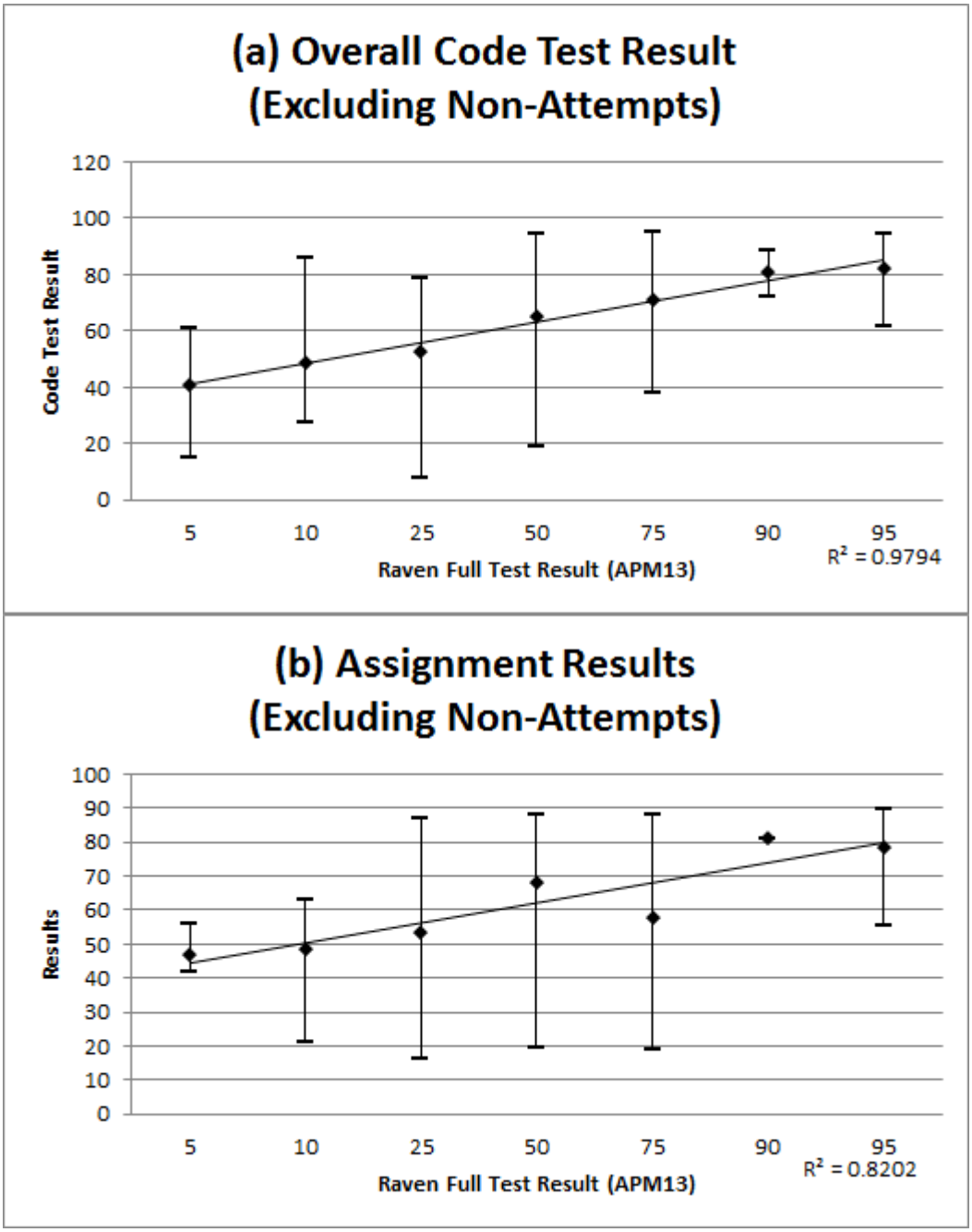


Figure 6-13 Comparison of Raven APM Scores to Student Results using APM 13 Norms

Neither of these APM tests produced a normal distribution (Figure 6-14). Therefore, tests using this data must be non-parametric.

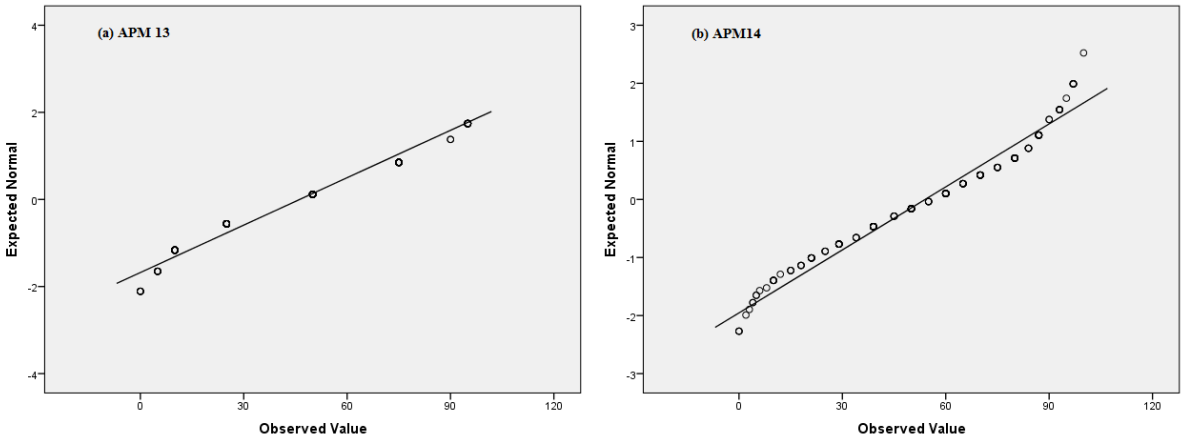


Figure 6-14 Normal Q-Q Plots form Raven APM13 and APM14 Tests

To determine the effectiveness of Ravens APM13 in predicting coding ability, the overall code marks (excluding zeroes) were categorized from Under 40% to Over 90% as shown in Table 6-6. Using the chi squared test, to compare the overall code test result with the Raven score for each student gave Pearson Chi-Square value of 66.954 with a significance p=0.001 with sample size N=164, where the number of degrees of freedom (df) is 36. The results produced by this test are shown in Table 6-6 and a graphical representation of this table is shown in Figure 6-15 (see Table APM13 and 14[330] for Raven scoring).

	Raven APM13 Score %						
Test Marks	5.00	10.00	25.00	50.00	75.00	90.00	95.00
Under 40%	3	9	11	12	4	0	1
40-49%	2	3	3	4	5	0	0
50-59%	0	2	8	8	6	0	0
60-69%	1	4	10	8	5	0	3
70-79%	0	0	5	9	3	1	3
80-89%	0	1	0	9	9	1	2
Over 90%	0	0	0	1	4	0	4
Sample Percentage	3.7%	11.6%	22.6%	31.10%	22.0%	1.2%	7.9%

Table 6-6 Table Produced by Two Way Chi Squared Test

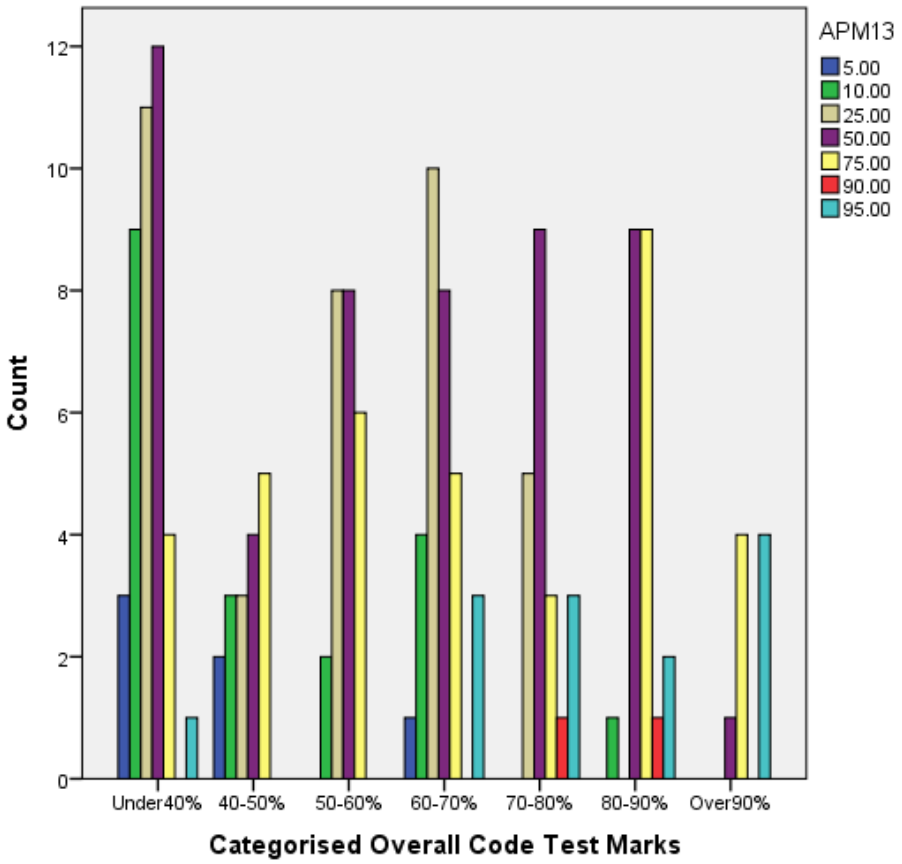


Figure 6-15 Graphical Representation of Table Produced by the Two Way Chi-Squared Test

The Phi and Cramer's V test results gave values of ϕ=0.639 and V=0.261 respectively with the identical significance value of p=0.001. However, the Phi (ϕ) test value is only suitable for situations where both the variables under consideration (i.e. test result and Raven score) have exactly two possible values resulting in a table with the dimensions 2x2 (or df=1). By contrast, the Cramer's V test is suitable for variables that produce more than two possible values (as in Table 6-6) and an unequal number of values, resulting in tables of varying dimensions (including unequal rows and columns). The Cramer's V value indicates the strength of the relationship as follows:

> "*A small effect size is one that is greater than 0.1 but not more than 0.3, a medium effect size is one that is greater than 0.3 but not more than 0.5 and a large effect size is greater than 0.5.* " [335]

It should be noted that these values are widely cited [336] and derived from Cohen's work [337] but there is some dispute over the categorization of the effect into small, medium and large [338]. Furthermore, the value of df for the Chi-Square test and df for Cramer's V are different. For Cramer's V it is referred to as df*= min(rows-1, columns-1) giving df* = (7-1) = 6 and this value reduces these limits [339]. Although no table data could be found for df*=6 the values for df*=3 are small=0.06, medium=0.17 and

large=0.29 [339] (Table 6-7). Thus, the Cramer's V result of V=0.261 suggests a medium strength relationship exists between these variables.

df*	Small	Medium	Large
1	0.10	0.30	0.50
2	0.07	0.21	0.35
3	0.06	0.17	0.29

Table 6-7 Effect Sizes for Cramer's V [339]

However, there are two caveats. Firstly the relatively small sample size and the distribution of the students across the Chi-Squared table produces a low number of students in each cell (often less than 5), which limits the reliability of the test. Secondly, some sources [340] indicate that the larger the dimensions of the table, the lower the reliability of the Cramer's V test, as an artefact of the type of variable used. To resolve these problems, the marks and APM scores were re-categorised into 3 simple categories below 50%, 50 to 69% and over 70% for the marks, and below 50%, 50-74% and over 75% for the APM scores (reflecting the APM13 norms [330]). Repeating the chi squared test, to compare the overall code test results (excluding zeroes) with the Raven score for each student, gave a Pearson Chi-Square value of 26.403 with a significance p=0.000 and a degrees of freedom (df) value of 4. The Cramer's V test result gave a value of V=0.284 and a significance value of p=0.000. With df*=(3-1)=2. From Table 6-7 we see that V is greater than 0.21 and can again conclude that we have a medium association between the code marks and the Raven scores. Figure 6-16 shows the same bimodal performance often seen in coding assignments and tests, with lower Ravens scores associated with poorer performance and the opposite for higher Raven scores. Students obtaining Ravens scores in the mid-range (50-75%) obtained a broad range of test marks suggesting that other factors determined their success or failure in coding.

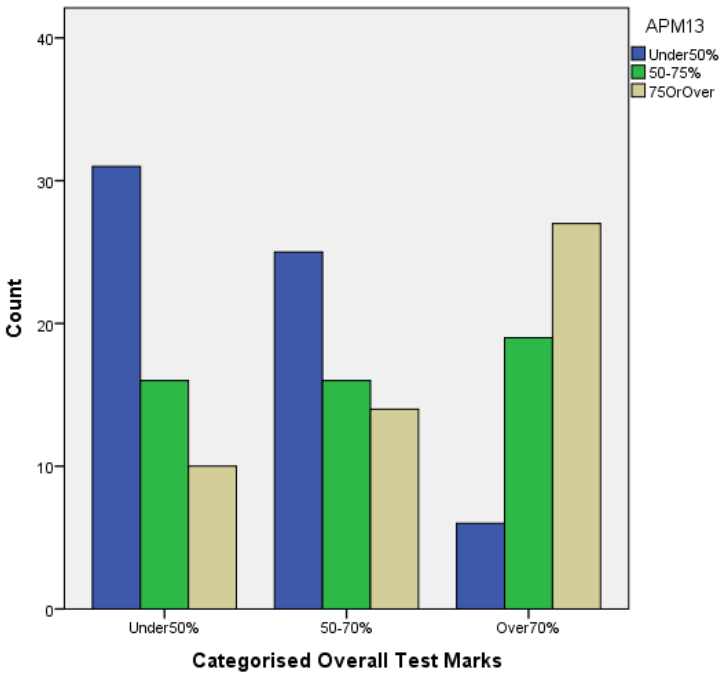


Figure 6-16 Graphical Representation of Table Produced by the Two Way Chi-Squared Test for Reduced Categories

As shown in Figure 6-17, this bimodal distribution can also been seen by comparing the Raven Matrices scores with the overall code test results, and offers a potential explanation for this effect seen in many programming courses. Given the relationship between working memory and programming has now been established, it seems probable that students with poor working memory begin their programming studies with an inherent disadvantage over those with a better working memory. As a result, the cause of many weaker student's problems manifests itself in the form of poor problem solving skills, a characteristic much less associated with good programming students.

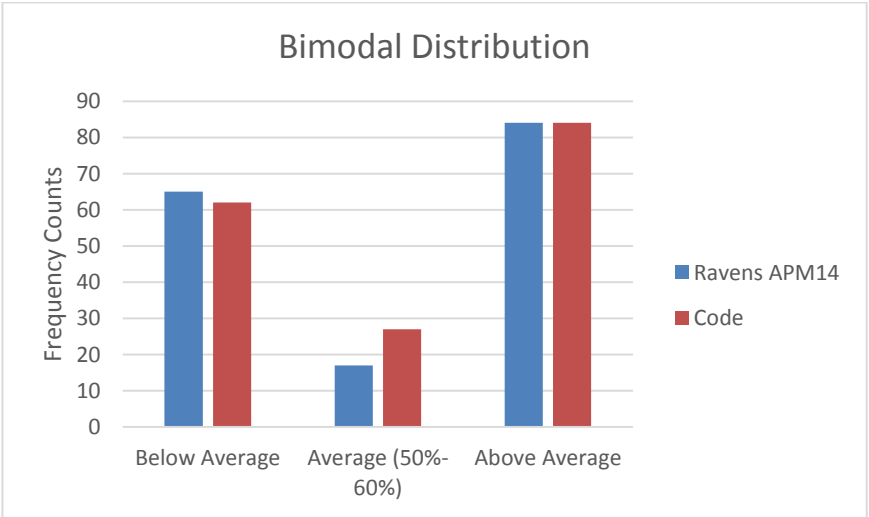


Figure 6-17 Comparison of Bimodal Distributions of Raven vs Code Test

6.2.3 Analysis of Individual Questions

The individual test results for each question and the correlation of the individual questions in the code test to the Raven Matrices scores for the students (APM13 Norms) is shown in Table 6-8.

	Correlation R^2
Q1	0.7078
Q2	0.9672
Q3	0.8392
Q4	0.9184

Table 6-8 Correlations Obtained for Individual Questions in Code Test

Question 2 and 4 show the highest correlations. Question 2 required a greater understanding of the flow of control of a program and the production of a flow chart, and gave the clearest discrimination of participants' reasoning ability with an R^2 value of 0.9672, while in question 4 the focus was primarily on coding and gave an R^2 value of 0.9184. When seeking to discriminate between participants based on their programming ability, the flow charting and pure programming tests provided the clearest predictors. Furthermore, as shown in Figure 6-18, for question 4 the ranges of marks awarded decreased as the Raven scores of the students increased indicating that higher working memory capacity becomes more strongly correlated with higher coding ability. Students with lower working memory capacity, produce a larger variety of results suggesting that, at least for some students, this inherent weakness can be overcome and that it does not necessarily preclude them from successfully programming.

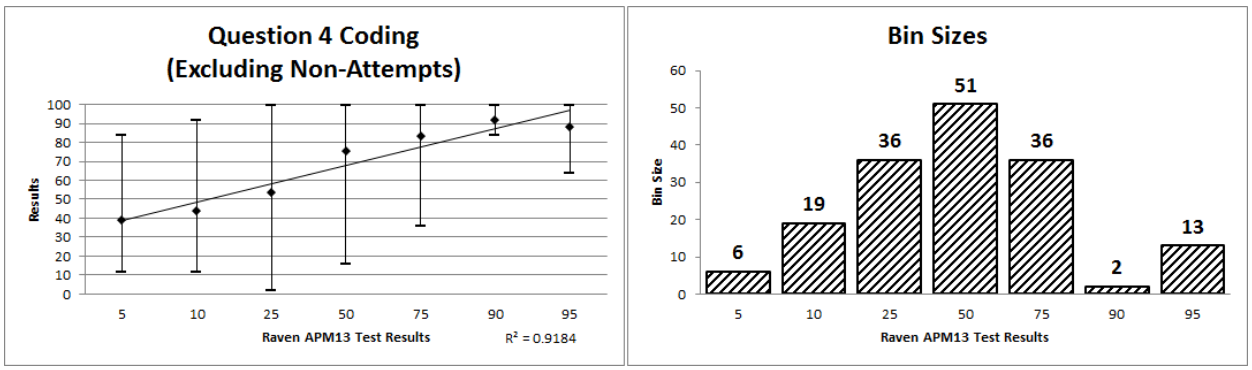


Figure 6-18 Results for Question 4 in Code Test Correlated Against Raven Matrices Scores

Overall, we can conclude that the individual questions from the code test have successfully measured student performance, and a number of component parts comprising the "programming thought process" have been tested. However, not all of these parts are equally important and future testing could be more targeted.

6.3 Conclusions

Undergraduate students studying computing took a short course on Computational Thinking prior to starting their main studies. In a previous study (Chapter 5), an analysis of programming metrics established that problem solving was a key discriminator of programming success. To further investigate the role of problem solving in learning to program, a measurement of students' fluid intelligence (gF) was taken using Raven Matrices and compared with the results of a coding test. A correlation between these results was established, confirming that Raven Matrices were a predictor of initial programming success. This result has significant implications. Firstly, both the code test results and the Raven Matrices scores showed the familiar bimodal distribution. Therefore, at least to some degree, problem solving and coding skills are inherent and a lower Raven Matrices score implies that these students will initially have more difficulties learning to program. A longer term study may show that these difficulties can be overcome through practice and by gaining more programming experience. A second important observation was that Raven Matrices themselves are primarily a measure of working memory, consequently this result also supports the relationship between working memory, problem solving and programming established in the grounded theory analysis. Finally, the Computational Thinking course was intended to provide an accelerated learning and motivational experience for students to underpin their first year of study. Although it might have succeeded in motivating students, the academic results were not improved by the course. Therefore, the benefits of running such a course in the future would be in enabling prediction of the students that are most likely to experience difficulties. This may allow ongoing support to be better targeted at weaker students.

7 Pattern-Based Learning in Programming

The grounded theory analysis indicates that one of the key skills required to be able to successfully program is the ability to deal with abstraction. One way we may choose to view abstraction, is through software comprehension [19, 123] and the associated mental patterns or "*plans*" required.

In software design, the use of patterns has become well established following the publication of the seminal work "Design Patterns: Elements of Reusable Object-Oriented Software" [270]. The authors of this work are often referred to as the Gang of Four (GoF). A number of books have covered these patterns but two stand out by attempting to present these patterns in a novel fashion. Freeman et al [341], presents each pattern through a series of simple design steps using sketches and quick quizzes to reinforce the ideas behind each pattern. Each chapter covers a single pattern and has to be worked through systematically to fully appreciate it. Laraman's book [342], is more formal and emphasises that the reader must learn both General Responsibility Assignment Software Patterns (GRASP) and the GoF patterns. These ideas are presented by ongoing case studies that are followed throughout the book. While Freeman focuses specifically on the patterns, why and how they are used, Laraman is more concerned with identifying when the patterns should be applied. For Laraman, developers "*build up a repertoire of both principles and idiomatic solutions*" which when identified as providing a solution to a problem are then named, and may be called a pattern [342].

> "*It [naming] supports chunking and incorporating that concept into our understanding and memory*" [342]

Naming the pattern is very important. Inexperienced programmers lack the knowledge to name the principles they are using, making it harder for them to communicate what they are doing and to learn new ideas.

> "*Software development is a young field. Young fields lack well-established names for their principles – and that makes communication and education difficult*" [342]

A number of approaches have been taken to teaching design patterns, examples include Warren [343] and Astrachan et al [344]. Warren is interesting because he suggests that design patterns should be taught as part of problem-based learning since the patterns must be learnt and applied in solving a real problem. Learning a pattern in isolation is ineffective because students lack the design experience to see its importance. On the

other hand, Astrachan et al [344] emphasizes learning good design through using patterns and learning to implement them through a number of exercises (the Applied Apprenticeship Approach [345]). They observed that the majority of introductory textbooks:

> "*…are driven by the syntactic details of a specific language rather than general methods for solving problems and designing programs*" [344]

Patterns expand the students' design and development vocabulary. The "*strength, purpose and abstractness"* of design patterns that makes them effective also serves to reduce their accessibility. This can be particularly problematic for new learners [344]. However, they did advocate teaching simpler programming patterns based on the work of Wallingford [346].

Although the use of patterns is common in software design, it is not as common in teaching programming. Rist demonstrated that novice programmers tended to work backwards from the program goal through code to the solution plan, while experienced programmers tended to work top-down i.e. they develop a solution plan first [121]. Thus, Wallingford reasoned, if students are taught suitable plans as abstractions they would have the required schema to work in a more top-down manner [346]. To reduce the abstraction, he defined a set of programming patterns. Five such patterns were identified, as shown in Table 7-1.

He also investigated creating similar patterns in an object oriented programming [346]. The course was taught with an emphasis on patterns over syntax and semantics. An example, from Wallingford's revised course outline one topic read:

> "*Alternative-actions: implementation options; embedding alternative actions in the process-all-items-pattern; choosing between the selection patterns*" [347]

<table>
<tr><td>Process-One-Item
Get data to be processed
Process the data
Output the results</td><td rowspan="2">Process-All-Items
Prepare for processing
Loop over (process-one-item):
Get next item
Process the item
Prepare for next item
Perform closing action</td></tr>
<tr><td>Guarded-Action
If guard-condition is satisfied
Take action</td></tr>
<tr><td>Alternative-Actions
Select appropriately from the following
Condition1, take action 1
….
Condition n, take action n</td><td>Process-Items-Until-Done
Prepare for processing
Loop until done:
Get next item
If appropriate
Take action
Prepare for next item
If needed
Process last or found item</td></tr>
</table>

Table 7-1 Wallingford's Programming Patterns [347]

An example of a Counting pattern is shown in Figure 7-1.

Pattern	Counting
Problem	Need to count the number of items in a collection of values
Algorithm	Initialise counter to 0 While there are more items, Process the item Increment the counter

Code	count:=0;
	/* Get the first value */
	while(value <> STOPPER)
	{
	/* Process the value */
	count := count + 1;
	/* Get the next value */
	}

Figure 7-1 The Counting Pattern

Using patterns in this way guided the novice programmers' use of the language constructs and encouraged the reuse of software components [346]. When faced with a problem, weaker students can recognise the need for a pattern and start working with a "*chunk*" of meaningful code [346].

In related work, Haberman et al [27], developed pattern oriented instruction (POI) as a pedagogical approach with the main goal of developing algorithmic problem solving skills. Patterns (schema) are examples of "expert solutions" that can be applied to create algorithms to solve problems. In POI, students learn to program by solving problems that are organised around patterns rather than programming constructs. The complexity of the problems is gradually increased to enhance "*assimilation and formation of cognitive schema*."[27]. POI involves three main processes: Pattern Recognition, Black-Boxing and Structure Identification. Pattern recognition involves abstracting the pattern from the context of the problems i.e. it "*… relates to the realization of similarities between analogical problems.*" [27]. Black-Boxing or chunking, is concerned with encapsulating code so that it can be reused within a number of problems. Structure Identification involves subdividing a problem and requires high level abstraction and the development of a solution plan. POI stresses abstraction processes, not "*abstraction products*" the simple following of recipes for solutions [27]. Muller [28] looked at the relationship between analogous transfer of knowledge and POI. A course was developed that embraced the concept of abstracting a pattern from various examples. His results showed that the students were more able to subdivide problems and that they tended to remember patterns rather than look them up. However, in analysing their results, Haberman et al [27] noted three difficulties experienced by the students:

- Although they obtained the correct solution they also wrote more unnecessary code.
- They found moving between abstraction levels difficult when problem solving
- They failed to recognise how the component parts of the solution could be integrated to complete the solution. (However, Muller's [28] work in analogical reasoning suggests a solution.)

An interesting problem recognised by Muller [28], was that all the tested students were distracted by irrelevant surface features in the problems description which were often caused by incorrect comparison of the current exercise against a previously completed exercise. The surface features made it harder to recall the solution from memory [348], although the more competent the solver the quicker they recovered from this error [348]. However, Muller [28] found that students taught through POI tended to be able to recover more quickly from these errors.

Wallingford [346] noted that teaching patterns may limit the students' ability to create their own unique solutions i.e. it may *"…inhibit the better students' development of their own problem solving schema*". He thought this might be alleviated by providing a wide enough range of exercises, that allowed students to modify and combine patterns [346].

However, could creative problem solving and pattern based pedagogy be made compatible by adopting this approach in the teaching of programming constructs?

7.1 The Proposed Abstracted Construct Instruction Pedagogy

Both Pennington [101] and Rist [19, 123] suggest that programmers construct a series of plans that allow them to negotiate their way through code and latterly make necessary modifications. These plans are remembered in memory "chunks", and expertise is therefore obtained by learning numerous such abstract chunks which can then be applied across a range of problems. In POI these chunks are taught as patterns which tend to consist of a number of steps, they provide a procedure for solving a simple problem and moves away from the more software construct oriented approach of teaching programming. This is consistent with Pennington's concept of plan structure knowledge [101]. In the following proposed Abstracted Construct Instruction (ACI) pedagogy, the emphasis is on Pennington's prime programs [101] or software constructs and the "instruction" focuses on abstractions of these. These abstractions will be referred to as abstracted construct patterns (ACPs), and they are more aligned with Pennington's concept of text structure knowledge [101]. The text and plan structure knowledge together form the programmer's knowledge of the source code, so there is some overlap in these concepts and consequently between POI and ACI.

In discussing plans, Rist [122] uses the concept of "*slots*" [104, 115] when translating these plans to concrete code. Understanding what these "*slots*" represent is crucial in this translation process and is the concept that underpins ACI. Naming of an ACP is important, to ensure that there is an established point of reference between the student and teacher. Multiple exercises are required to reinforce both the memory of the ACP and the understanding of the purpose and correct usage of each "*slot*". This is a significant distinguishing feature of ACI over other approaches. It is never assumed that completing one or more examples using a construct means that students will learn its appropriate usage. Instead, the abstraction is taught and simple exercises focus on appropriate elements of the pattern allowing the students time to deduce the semantics for

themselves. Thus, ACI adopts the principles of Analogous Transfer of Knowledge [37] i.e. learning a concept through multiple applications of it while minimizing structural dissimilarity.

The overall structure of the course was constructed around the development of ACPs and is shown in Table 7-2.

		Abstracted Construct Patterns	Additional Instruction
Concept	***Variables***	• Variable Declaration Pattern • Variable Assignment Pattern	Notional machine Quantities to variable name association
	I/O	• Input Text Pattern • Output Text Pattern • Input Number Pattern	
	Variable Calculations	• Variable Assignment Pattern (with focus on assigning variables to variables)	Variable identification from problem statement BODMAS
	Branches	• Branch Pattern • Branch with Alternative Pattern	Notional machine Word to branch condition association Implied logic Logic order
		• Nested If Pattern	
		• AND Condition Pattern • OR Condition Pattern	Number line
		• Switch Pattern	
	Loops	• Conditional Loop Pattern • Counting Loop Pattern	
	Arrays	• Array Creation Pattern • Array Write Pattern • Array Read Pattern	Notional machine
		• Array Counting Loop Pattern	Mixing Patterns: Array Counting Loop and If Patterns
		• Multidimensional Array Declaration Pattern • Multidimensional Array Write Pattern • Multidimensional Array Read Pattern	

		• Multidimensional Array Counting Loop Pattern	
	Functions	• Generic Function Pattern • Procedure Call Pattern • Procedure Declaration Pattern • Procedure Declaration Pattern (with arguments) • Function Call Pattern • Function Declaration Pattern • Function Declaration Pattern (with arguments)	
		• Nested Function Call Pattern	Basic problem solving techniques

Table 7-2 Course Outline based on ACI

ACI does not disregard all other teaching techniques, such as promoting an understanding of the notional machine [120] that is executing the code. On the contrary, understanding the basic operation of the "machine" makes the abstractions in the ACPs easier to accept and understand.

The core principles of ACI are:

1. Teach software constructs as abstract patterns.

For example, declaring and initializing a variable are taught as two separate ACPs Figure 7-2.

Variable Declaration Pattern	**Variable Assignment Pattern**

• type *variablename*;
 	
• type
  ○ int
  ○ double
  ○ string
  ○ bool
	• *variablename* = value**;**
	
	• value:
	  ○ Whole numbers -2,-1,0,1,2…etc (int)
	  ○ Floating point numbers -2.1,-1.3,0.0,1.33,2.45, etc (double)
	  ○ Strings "Hello", "Goodbye", "1", etc (string)
	  ○ Boolean true, false (bool)
	  ○ *variablename*

variablename = any name following variable naming rules	

Figure 7-2 The Abstract Construct Patterns for Variables

Both the ACPs are given names, “Variable Declaration” and “Variable Assignment”, followed by the abstract pattern itself and examples of values that can be provided for the “*slots*” or variable parts of the pattern. A deliberately pared down selection of values was provided to limit the amount of information that needed to be remembered. To allow the students to infer how these ACPs could be used, a series of examples were provided. For example, the first set of exercises are shown in Figure 7-3.

type name; name = “fred”;	*type* flag; flag= false;
type number1; number22 = 22;	*type* number3; number3 = 4.57;
type number2; number2 = 2.34;	*type* address; address= “22 Oak St”;

Figure 7-3 Initial Exercises in Variable Declaration and Initialisation

These problems require the student to deduce the variable type from the given variable name and the value being provided. Hence, the student forms their own understanding of the meaning of “type”. The principle of reinforcing the development of mental schema by constant repetition by use of multiple examples is based on the work of Lui *et al* [7].

2. The simplest forms of abstract patterns should be used

When teaching constructs, there is a tendency to present its use and hence fail to help the learner construct a mental model of the individual component parts of the syntax. In ACI, the array is taught as three ACPs: Array Creation, Array Read and Array Write. Only later, once these patterns are understood, are counting loops integrated with them to form an Array Counting Loop. The emphasis is on simplifying the patterns and giving the learner the opportunity to correct any misunderstandings of them as early as possible. For example, when reading an array, the index into the array must fall within a legal range of values: this concept is introduced naturally as a consequence of exploring the Array Read and Array Write patterns. As a consequence, it becomes much easier to identify potential learning difficulties because these typically arise when multiple patterns are merged. Furthermore, one result of naming the ACPs is that it makes communicating hints much easier.

Students should be dissuaded from modifying the pattern where possible, since systematically applying the same pattern makes it easier to remember. As an example,

when teaching counting loops, students should be dissuaded from modifying the Counting Loop pattern as they may be tempted to do if they are asked to count down rather than up.

3. Provide multiple exercises to reinforce application of the abstract pattern

All the fundamental constructs were taught using named ACPs and the number and variety of exercises provided were tailored to develop the students' mental model (schema) of the construct. For example, the first exercise for applying branch statements is shown in Figure 7-4.

Enter price Display "Price is greater than £302.20p"

Figure 7-4 An Exercise for Using a Branch Statement

The presentation of the exercise is deliberately quite terse to minimise confusion that might be introduced by the surface features of the problem statement itself. A danger with providing exercises is that they may become too complicated and interfere with the learning of the ACP. Initial exercises, at least, should prompt the recall of the ACP itself and emphasis the correct approach to converting the abstract pattern into a concrete form. Thus, in the previous question, the "condition" becomes "price > 302.20" in a branch statement. Providing numerous exercises, such as these, may be considered rote learning but as noted by Lui et al [7]:

> "*Rote learning is often criticized, but in the case of weak students memorizing key programs and program segments can consolidate viable knowledge construction*"

4. Graduation of exercise difficulty

Although each ACP is introduced with a targeted set of exercises, for useful program construction these patterns must be integrated to form solutions to more interesting problems. Therefore, once the learner has understood an ACP then more exercises can be introduced that require previously learnt patterns to be used to complete them. However, the number of patterns required to solve a problem should be strictly controlled to avoid interference effects caused by difficulties recalling previous patterns, identifying the required patterns and over complexity. Where learning to integrate a new pattern with previously learned patterns is concerned, exercises should make the solution as easy to identify as possible. The guiding rule is not to create exercises to develop

problem solving skills, but to aid recall of the patterns themselves. Of course, there is some overlap with POI in that students can be expected to recall a particular approach they have previously encountered e.g. reading through an array in reverse.

5. Promote Memorisation of Patterns

Although students will be given or may take their own notes, ACI must specifically address the memorisation of patterns and their meanings. Therefore, reviewing notes and searching for patterns when completing exercises must be seen as poor practice and strongly discouraged. In this regard, four approaches were taken to modify student behaviour:

a) At all times during class, memorisation of the patterns was given particular importance.
b) Although students were encouraged to take notes, they were asked not to consult any notes from previous classes and to immediately attempt to commit to memory the current pattern being considered.
c) A set of exercises was only provided in class, although additional exercises were made available for home consumption. This enabled the progress of the students to be more closely monitored than would otherwise have been possible.
d) A number of unannounced informal in-class tests were given to the class at regular periods. The students were aware that tests would occur but not when or what they would contain. In addition, it was clearly explained to the students that these would not be marked and the purpose of the tests was for the individual student to assess their own progress. Thus, students were asked to score their own work. All the students submitted the written work and code for the tests for research purposes.

6. ACI and Problem Solving

Although the development of problem solving skills is not the main purpose of ACI, a Generic Function Pattern is introduced to promote the association between function and a specific problem being solved (Figure 7-5). The pattern contains three components, the "ProblemName" which must describe the problem being solved, the information required to solve the problem (if any) and the type of result produced by solving the problem (if any).

resulttype ProblemName (*information to solve problem*)

Figure 7-5 The Generic Function Pattern

Introducing functions through this pattern immediately introduces the principle of higher level abstraction of a problem. In effect, calling functions tells the "story" of how the problem is being solved. Excluding the Nested Function Call pattern, the exercises provided focused on solving a single problem through the creation of one function. Such an example is shown in Figure 7-6, where the student must write the function, but to do so, they must also infer from the problem description both the return type and the arguments required to solve the problem.

returntype **AddThreeNumbers**(*args*) Pass in three whole numbers, sum them and return the result.

Figure 7-6 An Exercise in Writing a Function

To teach the Nested Function Call pattern, the exercises involved subdividing simple problems into multiple functions. Subsequent exercises promoted reusing functions from previous solutions. Although the problems themselves are not complex and are designed to teach learners to call functions from other functions, the result is that they learn to identify reoccurring solutions. It should be noted that many hints were given since coding the solution was the challenge and not solving the problem per se. Two consecutive examples demonstrate the approach (Figure 7-7):

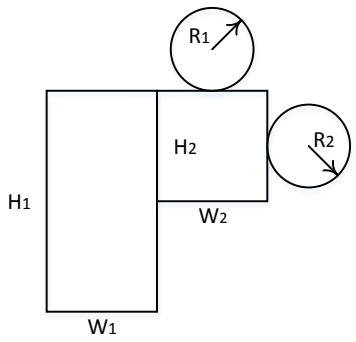
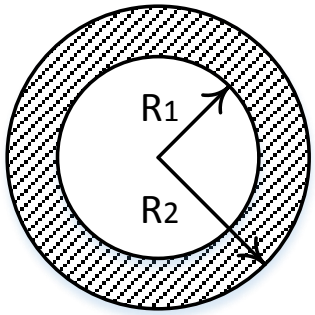
Exercise 1	**Exercise 2**
Write a function to calculate the area of the following shape using function to calculate the area of each different shape. The user must supply values for each dimension.  	Write a function to calculate the shaded area of the following shape. The user must supply values for each dimension. 

Figure 7-7 Two Exercises for Demonstrating the Use of Nested Functions

In Exercise 2, the learner is expected to reuse the function for calculating the area of a circle in writing the function for calculating the shaded area of the shape. Such exercises require the learner to subdivide the problem, name the functions after the problem is identified, write the functions, test the functions and combine them in an overall single function representing the complete solution. These are all fundamental skills required to solve much larger, more complex problems. Thus, ACI does provide the underpinning necessary for learners to move forward to solve much more interesting and complicated problems.

7.2 Teaching Problem Solving Skills

ACI was the approach taken in teaching the first semester of the course. During the second semester, the course evolved into an investigation of problem solving techniques and their application to programming problems. To assess the effectiveness of ACI itself, the POI approach was not adopted. The primary aim was to determine if ACI enabled students to solve problems for themselves without limiting the creativity of the solutions that they may develop. Thus, a set of techniques were taught and exercises were provided to develop the students' appreciation of their use. These techniques were developed by observations of past students' programming behaviour during class, and were a modified superset of those introduced by Spraul [349] which were themselves derived from common techniques such as those described by Kirkley [350]. The techniques used in this study are listed in Table 7-3.

Technique	Description
First solve the problem on paper	Do not try to solve the problem by typing at the keyboard
Use what you know	Remember the things you have been taught and think how they might be applied
Restate the problem	Make the restatement of the problem constraints generic (or abstract)
Subdivide the problem	Always break the problem into parts
Solve the easiest problem first	Always solve the most heavily constrained or the most obvious component of a problem first Start by coding/designing that which "you know how to do" before doing anything else
Generalise the solution	Try to make the solution as generic as problem so that it can solve a range of problems rather than just one
Change the game	If you cannot solve the problem you are given, rewrite the problem to make it simpler Concentrate on a simpler, reduced version of the problem by adding or removing constraints Learn from solving the simpler problem
Work the problem (General skills)	Do not attempt to solve the problem by typing at the keyboard, solve it on paper first Identify what you do not understand or do not know Break the problem statement down e.g. highlight key words and phrases Experiment Choose part of the problem and try a brute force approach to coding solution Try different input values or combinations of input values Look for relationships between entities such as values, inputs and outputs. Draw diagrams to represent the problem
Testing	Always test the solutions to check that they solved the problem.

Table 7-3 Problem Solving Techniques

The following describes a number of common observation of students' programming behaviour during class over a number of years.

1. The typing race

Typically once introduced to programming, students will seek to solve all exercises by immediately typing at the keyboard. This leads to "code thrashing" where a student rewrites the same section of code multiple times attempting to find a sequence of program statements that gives the solution. It also leads to students writing irrelevant and fundamentally incorrect code. To reduce these problems, they must be encouraged to view the implementation step as a translation of a pre-determined codeable solution. To prevent incorrect coding and as a memory aid, the solution should be documented prior to coding.

2. Failure to retain or apply existing knowledge

When trying to solve a programming problem students appear to forget what they have learnt (known as inert knowledge [31, 51]), or fail to map their knowledge to the problem facing them. This failure in mapping, related to the "*closeness of mapping*" [192] takes multiple forms and results in the student:

a) Failing to recognise features of the problem description that implicitly require the use of an ACP. An example of this is where an exercise requires the user to enter multiple values and then to display all the values entered. The student may fail to recognise that "multiple values" implies the use of an array.
b) Being distracted by surface dissimilarities between problem descriptions [174] and consequently failing to recognise that the same ACP is required. For example, where one exercise might ask for the user to be able to "enter ten numbers" and another exercise might ask for a "list of ten names". Here, the use of word "list" in English is given increased relevance by the student and not interpreted as just storing multiple values (names in this case). Thus, they fail to see this as just another array.
c) Selecting an incorrect approach to the problem, and hence introducing an "unsolvable" problem, either immediately or sometime later. For example, if the exercise requires the user to enter an arbitrary number of values, the student may apply a counting loop. Applying a counting loop immediately opens the question of how many times the loop should be repeated, which of course, is unanswerable.
d) Identifying a solution to a problem but being unable to adapt the ACP to translate this solution to code. A common exercise used in teaching problem solving using an array, is to require a solution that involves adding an offset value to an array index within a counting loop. For example, an exercise that requires the production of the Fibonacci sequence up to a value of n (where n >= 2) will require array values to be summed in pairs. The solution is very similar to the Array Counting Loop Pattern but applies the Array Read Pattern within it to read both the current and the next array value. Two potential approaches involve incrementing the count by 2 or calculating the number of pairs required and setting the count limit to this value. While the latter does not break the standard

pattern, the former approach does and this structural dissimilarity may cause problems later. For this reason, it is recommended that students be dissuaded from changing the ACPs where alternative solutions are possible. In this case, the count limit value can be calculated and the required array indices computed from the count.

Mapping failures such as these can set up a cascade effect. A failure to recognise that an array is required will lead to a failure to recognise that in order to use an array, the number of values to be stored in it must also be known. Such initial mistakes can be hard for a novice programmer to correct.

To aid students, the mapping process was made as explicit as possible by creating a table with two columns, one with a list of the features from the problem description, named sub-problems that they knew how to solve, or anything not understood such as words, phrases or sub-problems and the other containing brief headings and subheadings for the topics covered during the course. An example of the initial headings that might be used in the right hand column is shown in Table 7-4, but students were encouraged to personalise this list. In the left hand column, if they did not understand words such as “list”, “print” and “before printing”, they were instructed to include them all.

Problem Information	Existing Knowledge
Any information extracted from the problem definition or related to the potential solution.	Variables ▪ Must have a value ▪ Must be declared ▪ Types (int, string, double, bool)
	Branches ▪ Must be a question to ask ▪ If, else ▪ Switch (one value to be tested)
	Loops ▪ Must repeat ▪ Counting loop count (starting at 0) ▪ Conditional loop (loop when condition is true)
	Arrays ▪ More than one value ▪ Must know the number of values required ▪ Must create array using 'new' ▪ All the same type ▪ First element starts at 0 ▪ Last element is arrayLength – 1 ▪ Use counting loops and arrays together
	Input/Output

Table 7-4 Map of Current Student Knowledge

Once the left hand column contained enough information, the process of mapping between the problem space and the student's knowledge domain could begin. This process consisted of simply drawing arrows from the left hand to the right hand column and took a number of iterations as the student mapped between the two. As more questions arose, they were written in the left hand column, and links started to be made between these questions and the known quantities. An example of the anticipated results following a couple of reviews of a question might resemble Figure 7-8.

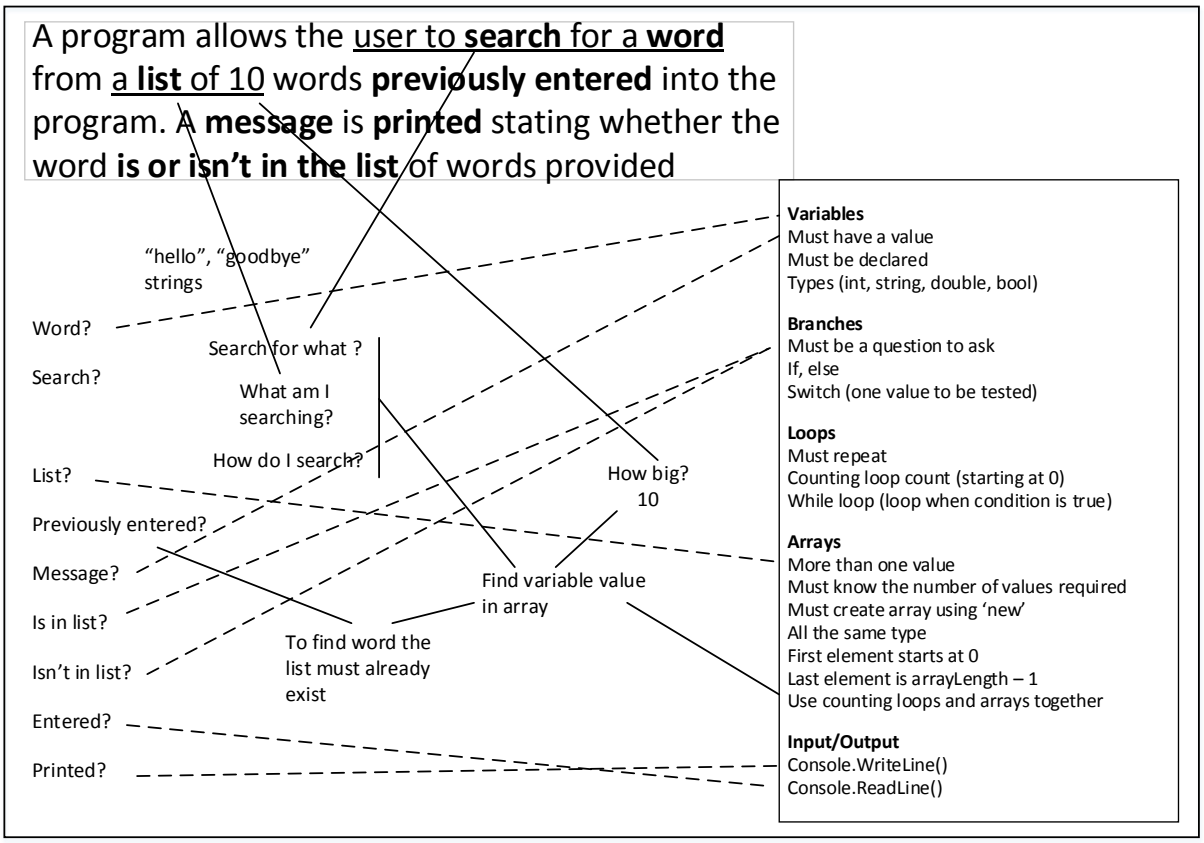


Figure 7-8 A Presentation Slide Illustrating the Mapping between Student's Knowledge Domain and the Problem Space

3. Understanding the problem

Often, if a student fails to understand a problem, they will attempt different strategies to understand more about it. Of course, the first fundamental step is to identify the requirements, procedures and constraints. But what if this is still not enough for the student to make progress? Two techniques suggested by Spraul [349] are to restate the problem in "your own language" and to make the description of the constraints involved as generic as possible. To illustrate this idea, Spraul [349], uses the classic puzzle known as *The Fox, the Goose and the Corn puzzle* and shows how it can be solved by rewriting the operations and making them generic. He argues that this approach allows you to gain insight into how a problem can be solved and enables transfer of the solution across related problems.

4. Divide and Conquer

The standard problem solving approach is to subdivide a complicated problem into a set of simpler problems and then solve these instead [350]. For programming, this approach

is particularly apt [349], [211] and should be continually emphasised. In teaching functions using the Generic Function Pattern, this process is embedded immediately in the mind of the learner since they are continually being forced to identify and name the problem they are solving each time they write a function. From the outset, ACI systematically teaches the learner how to write and call functions/procedures from the perspective of deciding the information required to solve the problem, and whether the solution produces a result. The alternative approach of writing code and searching for replication of code discourages the learner from seeing a program as a set of well-defined solved problems. As a result, ACI encourages the student to see the main function of a program as the place where functions are called to tell "the story" i.e. the function names and the sequence in which they are called should produce a main function that reads like the problem description. Learners are not dissuaded from creating one line functions if their purpose is clear.

5. Take the easy path

Having created a set of the sub-problems to solve, the next step is to rank the problems from easiest to hardest which then becomes the order in which they should be solved [349]. This approach prevents the learner from becoming too focused on the parts of the problem they initially do not know how to solve. A commonly observed problem is students failing to make any progress on a solution despite aspects of the problem being quite straightforward. Firstly they become disheartened and secondly they fail to learn more about the problem i.e. solving part of the problem may provide additional information that they are missing.

6. Abstract solutions are best

When creating functions, the learner should seek to make them as generic as possible so that they can be applied through a range of problems [28, 349]. Although, the Nested Function Call pattern and associated exercises demonstrated the reuse of functions across problems, generalising those solutions was not necessarily the main emphasis. Thus, additional exercises were provided to give more opportunity for this abstraction.

7. Change the game

Spraul [349] described an approach that overcame the seemingly challenging problem of drawing half a square over a number of steps, by reducing the problem each time with

modifications to the problem constraints. This principle of modifying a seemingly intractable problem into one that is easy to solve in order to glean more information, formed a major principle in the second semester. Sometimes this was referred to as a "brute force" or "hard coding" approach. An example would be repeating a line of code ten times instead of using a loop to swap array values. For the assignment, the students were requested to provide evidence of this form of experimentation and incomplete solutions were accepted providing they demonstrated a potential path to the solution i.e. evidence of a reasoned approach.

8. Always Test

A common novice programmer mistake is to assume that their solution must work, because they find it harder to spot programming bugs and to make a good hypothesis of how the code works when reading it [351]. Even worse, they often add new bugs when searching for errors [351] because of these poor assumptions. This is related to perceptual learning [351], [143]. Coders who think about testing are more likely to write correct code, therefore learners must be encouraged to frequently test their code [352].

7.2.1 Incomplete Solutions are Acceptable

It was important to remind students that the process undertaken to solve a problem was more important than implementing a complete solution. A complete solution, constructed with little thought or design, is of little interest. The student tendency of seeing programming as a kind of typing exercise has to be broken. Instead, the stages of solution development were emphasised during instruction and actively promoted in all exercises and assignments.

7.3 Methodology

The research was conducted in two phases. Firstly ACI was used in teaching a first year introductory programming course of ten students for 10 weeks during the first semester. Secondly, during the 10 weeks of the second semester, problem solving skills were taught to two separate focus groups consisting of four students each. The first group were drafted from the original ACI cohort and the second, for comparison, was created of four students that were drawn from a separate course taught through a worksheet approach with no exposure to ACI. These smaller focus groups allowed for closer observation and monitoring of the students, and the results were recorded for later discussion. Results

were gathered through observation, testing and interview. All observations were conducted over a series of two hour sessions held once a week. A series of unannounced tests was also given to all the students, with the objective of assessing their progress and testing their recall of the patterns. For comparison purposes, an initial test was given to both focus groups at the start of the second semester to compare their relative programming performances before the problem solving element of the course began.

Finally, at the end of the course, all members of both focus groups were interviewed to determine if their attitudes and approach to programming had changed. The ACI focus group were also questioned about the role and influence of ACI in their learning.

7.4 Observations during ACI

The teaching of ACI was divided into two sessions, the first introducing the ACPs and contextualising their usage with some examples. During the follow up session a range of exercises, typically around 6 per ACP, were provided and the students were closely observed and questioned during this time with particular attention paid to those in the focus group.

One of the first exercises given to the students asked them to identify quantities or values and to categorise them into one of four groups: integers representing whole number values, doubles representing floating point values, strings representing text and Booleans consisting of a true or false value. It is known that novice programmers find it difficult to both identify variables [75] and their type [197] from a value given in a natural language problem. Interestingly, all the students assigned the values such as postcode (or sort code) and telephone number to the set of integer values. Even when the nature of these quantities was further explored, by considering specific values, some students went on to assign them to the next set which consisted of double values. In normal life we may refer to numbers when we are talking about identifiers e.g. pin number, but this is both misleading and incorrect in a programming context. This may also explain a second commonly observed misconception that numbers entered at the keyboard automatically become numbers when read in a program. Key values are ASCII, and an entered number is a string unless converted. This mistake was observed in 3 of the 4 students in the initial test conducted on the non-ACI focus group despite having completed a full semester of programming.

Not surprisingly, when asked to simply identify what they thought were the variables from a problem statement, most of the students struggled to produce a coherent list of names. The exercises produced for the course were kept short and often terse to minimise this type of problem.

Another exercise asked students to identify words that could be associated with conditions within branch statements, such as greater than, less than and equal to. Again, natural language gave rise to problems e.g. the use of “same” for equals and “exceeds” for greater than. Specific training had to be provided, in the form of exercises that required branch statements to be written where the problem description used various different natural language terms to indicate the appropriate condition required.

Novice programmers see variables as being “unique”, that is to say, they see a variable as a “use once entity” which results in them trying to create multiple variables where only one is required. As an example, suppose the user should enter two numbers and the program should add and display the sum of both. A novice may create three variables, where only one or two are required. This in itself is not a problem but it may start to fuel the misconception that a variable cannot be reused. Later, when an exercise requires a running total of an arbitrary set of entered numbers students will often store all the entered values in an array. Exercises can be designed to minimise these problems.

Another area where difficulties were observed was in misapplying natural language logic to a program. To observe the influence of natural language, an exercise was created to intentionally produce the incorrect answer if the logic of the problem statement was explicitly followed step by step. The problem description is shown in Figure 7-9.

Given a temperature under 100 check

- when pressure is below 56 just display “pressure is too low”
- when pressure is 23 or under just display “warning pressure is falling too low”

Given a temperate at 300 or more check

- when pressure is above 182 just display “warning pressure is rising too high”
- when pressure is above 239 just display “pressure is too high”

Figure 7-9 Number Range Condition Test

Of the 10 students 9 incorrectly implemented the solution by simply following the natural language procedure as stated and 7 of the students were unable to determine the cause of the problem. Even after the cause of the error was explained using a number line

diagram, the majority of the students preferred to stick with the natural language sequence and apply a Boolean AND operation to isolate pressures above 23. The principle of using branch statements to test number ranges, and having an appreciation of the ranges of numbers excluded and included by applying conditions, is a key to understanding branching in programming. Therefore, a range of additional exercises were designed specifically around this principle to provide students with the opportunity to gain a better understanding of the number line.

Modifying an ACP should not cause novice programmers many problems, especially a simple ACP such as the Counting Loop. However, when an exercise asked students to count down rather than count up, all the students initially failed to obtain an answer. They had to effectively create a completely new pattern that looked similar the original ACP, but in the process, the range of possible solutions lead to a great deal of confusion. The alternative was to leave the ACP untouched and seek to use the count value to generate the required alternative range of numbers. This method, of course, lead to other misunderstandings, but had two benefits. Firstly, thinking about using the count to create new values introduced an idea that could be expanded later to solve multiple additional problems, and secondly using a POI approach, the solution could be named the Counting Down Loop and taught as a new pattern. To reinforce this concept, a set of exercises were created using a Counting Loop to display various ranges of numbers and pairs of numbers calculated from the count. However, many of the students still had problems adapting the counting loop to reverse the count even after completing a number of these exercises.

Arrays were taught as four separate patterns, Array Creation, Array Write, Array Read and finally in combination with a Counting Loop to form the Array Counting Loop. Exercises were provided as each of these patterns were introduced. For example, the array creation exercises required the students to just create an array of the appropriate type and size. Likewise, the exercises for writing and reading were introduced separately: firstly the writing pattern required values to be manually set to fill the entire array (no loop) and secondly the read pattern required the values to be read out again and used in some form of calculation. When the exercises focused on an individual pattern, the errors were mainly due to unfamiliarity with the individual pattern. This was by far the most difficult concept for the students to grasp but the problems mainly occurred when they had to combine the patterns. To write any useful code using arrays requires the integration of

patterns, and it appears that novice programmers struggle remembering and applying closely related patterns. Furthermore, processing arrays requires the implementation of an Array Counting Loop which is an integration of the Counting Loop with the array patterns. The use of a variable to select an array element, specifically the count variable in this case, is the primary cause of these issues. As the course progressed, the array patterns faded faster than any other ACP, which could be related to interference effects [353] due to the need to learn multiple patterns together over a short duration.

7.5 Observations during Problem Solving

Two focus groups were observed while completing problem solving exercises. These groups were drawn from the student body taught using ACI and from another course taught using traditional worksheets.

When observing the students, it became clear that one of the limitations of the study was to overestimate the knowledge base of the students. It became apparent that although the exercises appeared to be quite straightforward, many of the students struggled with them. For example, some students had little knowledge of geometry and found questions based on circle area and circumference more challenging than expected. A number of exercises involved arrays, to give students more practice in applying them and because questions involving arrays and string manipulation tend to be more varied. One exercise that caused problems required the user to enter a word and asked that a program be written to reverse the letters in the word to form a new string. This involved reading backwards through the array using a Counting Down Loop. The two problems observed were misunderstanding how the count could be used with the array to read each element, and then how to store the reversed word. One student thought of swapping the letters around in the character array containing the word itself, but most created a second array and were then unable to copy letters between them. Others simply failed to use the array correctly and made little progress.

The problem with setting suitable coding examples is not new. In the 1980s, an assignment that was set for students became well known as the "*The Rainfall Problem*" because it neatly demonstrated the difficulties novice programmers had with solving programming problems. The assignment said:

> "*Write a program that repeatedly reads in positive integers, until it reads the integer 99999. After seeing 99999, it should print out the average*" [354]

Surprisingly, this seemingly straightforward problem could only be solved by 14% of the novice student programmers and even 30% of the most advanced failed to solve it [354]. This result has been repeated in numerous studies such as [32, 355, 356]. This also illustrates the challenges of setting appropriate exercises for novices when those exercises are being created by teachers with high programming expertise.

However, ACI is not specifically aimed at teaching problem solving skills and observational results between both focus groups were comparable, as borne out by the test results. The ACI focus on functions as self-contained solutions to problems received very good feedback from the focus groups.

7.6 Student Test Results

To assess student progress throughout the course, five unannounced tests were given to the students, copies of which are provided in Appendix 2 with the average marks shown in Table 7-5. At the beginning and end of the second semester, comparison tests were given to both the ACI and non-ACI focus groups to allow comparison of their relative problem solving and programming abilities to determine whether ACI had affected the development of these skills over the academic year.

Test	Description	Average Mark
First Semester		
Variables Test	Covered declaration and calculations using variables. Input/Output of values was also indirectly tested.	83
Branch Test	Covered the use of branch if and else statements. Also re-tested variable declaration and input/output.	69
Loop Test	Built upon the previous two tests but introduced the conditional and counting loop.	68
Second Semester		
Comparison Test	Used at start of semester two for comparison of progress of both focus groups. Covered previous test content.	69
Functions and Problem Solving Test	Covered problem solving through subdivision into functions.	66

Table 7-5 Structure of Student Testing and Results

Although the first test has a slightly high average, the marks across the tests (excluding zeroes) demonstrate that the students had a good understanding of the concepts and there is no significant decline in marks between tests over the course of both semesters.

As well as scoring the tests, the type and counts of the number of errors were also recorded to establish the causes of errors and any student misconceptions. In addition,

the errors were mapped against the ACPs to identify any weaknesses in the students' recall or understanding. As anticipated, most students completed many of the questions with no issues. Where students ran out of time, providing incomplete solutions, these were ignored in the error analysis unless sufficient progress had been made to allow conclusions to be drawn from them. The types of errors made by the students and the distribution of the error counts across the tests are shown in Table 7-6.

	Variables	Branches	Loops & Arrays	Comparison	Problem Solving	TOTAL ERRORS
NUMBER OF PARTICIPANTS	*10*	*10*	*6*	*9*	*8*	
Wrong Type	0	0	0	2	0	2
Wrong value assigned to variable	6	0	0	1	0	7
Incorrect calculation of percentage value	3	4	0	0	0	7
Lack of domain knowledge	5	4	0	0	0	9
Branch logic error (condition incorrect)	NA	1	0	1	0	2
Branch logic sequence (unrequired Boolean operator)	NA	3	1	3	0	7
Branch logic sequence error	NA	3	2	0	0	5
Branch logic error (missing else)	NA	0	1	3	0	4
Branch logic error (unrequired if in else statement)	NA	0	0	3	0	3
Over complicating solution	NA	3	0	3	3	9
Misunderstanding problem	NA	0	0	3	4	7
Loop Limit Error	NA	NA	1	0	0	1
Array Declaration Error	NA	NA	2	0	0	2
Lack of programming knowledge	NA	NA	5	4	3	12
Loop logic error	NA	NA	4	3	1	8
Forgot Counting Down Loop	NA	NA	0	3	2	5
Array read error	NA	NA	0	0	2	2

Table 7-6 Student Error Counts

7.6.1 Initial Assessment of Variable Knowledge

This first test (Appendix 2: Test 1) assessed the students' understanding of variables and consisted of three questions. The first question required the identification of variable types from variable names, and the kind of values that could be assigned to those variables. Even this fairly fundamental concept caused a number of problems, in particular, remembering the double quotes around string values. The second question involved a calculation using variables, in this case, the total cost including a specified percentage tax. The problems encountered in this question, stemmed from the students' lack of understanding of percentages and resulted in a number of erroneous approaches to coding the necessary calculations. Finally, the third question required the students to identify appropriate variables from a written problem description. No coding was required for this last question, and the students had no difficulty completing it.

7.6.2 Assessment of Program Branch Knowledge

Following instruction on branch statements using ACI, this test (Appendix 2: Test 2) consisted of three questions. The first of these reassessed the students' understanding of

variables. Again, four of the students demonstrated an inability to recall the solution to calculating a percentage value, despite having seen a similar problem on more than one occasion. This may support the case for using POI, since recall might have been improved by introducing the solution as a formal pattern. In the second question, the incorrect use of branch conditions to select number ranges was tested. 3 of the 10 students made an error in sequencing the branch statements thus incorrectly including ranges of numbers which should have been processed separately, while a further 3 students used a less than optimal solution by using Boolean ANDs to solve the problem. We can conclude that, in the mind of the student, the procedures contained in the natural language description of the problem supersedes programming logic i.e. the cleaner, more logical solution to this problem that can be derived from consideration of the number line. This was, in spite of encountering similar problems in previous exercises. The final question required finding the highest value of four numbers (no array or loop required). 4 of the 10 students either did not attempt it or did not complete this question, but for those who did provide a solution, the only issue of note was a tendency to slightly overcomplicate the answer.

7.6.3 Assessment of Loop and Array Knowledge

This test (Appendix 2: Test 3) consisted of five questions, the first of which was divided into four parts that tested fundamental understanding of variables, arrays and loops. In the second question, the contents of an array had to be displayed in reverse order. Neither of these questions caused the students any problems. In answering a question requiring an array search, 4 out of 6 students completed the problem, while a fifth student made a logic error that required the user to enter a value each iteration. When observed and questioned, this student believed that the solution was correct because when he ran the program he would enter the same value as the first entry in the array each time. A fourth question was similar to that in the previous test and required branch statements to check different number ranges but in this case, within a while loop to allow multiple checks to be made. For 3 students, the difficulty was in creating a while loop, while 2 students still had problems correctly selecting the ranges of numbers in a branch condition. The final question was only partially completed by one student and involved using a switch within a loop.

7.6.4 Assessment of Function and Problem Solving Knowledge

This test (Appendix 2: Post-Instruction Test 5) consisted of four questions, the first two assessed understanding and usage of functions and the last two assessed the students'

problem solving approach. It was not assumed that all students would complete the last exercise, but it was anticipated that they would use a systematic approach in attempting to solve it. The first question was a straightforward area calculation while the second required combinations of area calculations to find the overall area of a more complicated shape. These were based on previously covered exercises, with the objective of determining if the students would remember and apply the same approach they had previously encountered. Only one student had a problem with these questions, and did not attempt them because he felt they were "more difficult". These results suggest that the students are able to apply a similar solution to a problem only when the problem itself closely matches problems they have already solved. In the third question, numbers had to be stored in an array before being printed out in reverse order. The wording of the question stated that five numbers had to be entered and then "printed in reverse order" of entry. From this the student had to infer that the numbers had to be stored in an array, although the question does not require the values be reversed in the array itself. 3 of the students attempted to reverse the array contents, only 2 of whom were successful. The remaining students used the Counting Down Loop. 2 failed to complete the solution: in one case by incorrectly implementing the loop and in the other by subtracting the count from the array element value rather than using the count to select the array element. Again, the former is a case of "*inert knowledge*" [51], while the latter is more probably a programming logic error as shown in Figure 7-10.

```
int[] numbers = new int[5];
int length = 5;

for (int i = 0; i < length; i++)
{
    Console.WriteLine("Enter Number >> ");
    numbers[i] = int.Parse(Console.ReadLine());
}

for (int i = 0; i < length; i++)
{
    Console.WriteLine("{0}", numbers[i] - length - 1); ← should be numbers[i -  length – 1]
}
```

Figure 7-10 Example of Student's Incorrect Use of Array in Counting Down Loop

The final problem, question four, involved generating five randomised lottery ball numbers without duplicates. To help the students, they were given a function for creating a random integer value. This problem requires an array to solve it, but was novel to the students and had not been covered in any previous exercises. 3 students just selected five

random numbers without preventing duplication. Only 2 students fully completed the solution. One student solved this problem by building his own version of a counting loop using a conditional loop, while the second student decremented the loop count to cause another iteration of the counting loop. Actually, a third student was also close to solving it by searching all the previously chosen values using a separate tailored loop as each ball was selected.

In attempting both these problems, evidence of a problem solving approach being taken was clear and recognised by the subdivision of the problem into a number of self-contained functions that fulfilled a single responsibility. For example, the problem of displaying the contents of an array using a counting loop was commonly separated into a dedicated functions given names such as DisplayBalls and PrintArray.

Overall, the results from this test were mixed. The students clearly understood the role of functions in problem solving, and they were able to solve a problem when it was clearly related to a set of problems they had seen before. However, when the problem required combining concepts, as in question three, only 5 of the 8 students managed to solve the problem. Likewise, in question four, five of the students made some progress but the crux of the problem was to prevent the selection of duplicate numbers and this was ignored. For questions three and four, the students were asked to map their knowledge to the problem domain in writing as shown in Figure 7-8. Most found this difficult and their analysis was very brief (e.g. Figure 7-11a). Interestingly, only one student identified the importance of preventing the selection of duplicate numbers in question four (Figure 7-11b) and began developing a strategy to solve it.

	Students' Analysis (in their own words)
(a)	Generate random number Declare array of numbers 1 to 5 *[index 0 to 4]* Randomise selection of number
(b)	Have 5 balls in a sorted order One is selected at random Copy that value into new array Random new value **Check to see if it is already in new array**, if not, copy it into new slot, move to next slot *[slot here means array element]* Check to see if array is full, if *[it is]* break, Output lottery array

Figure 7-11 Student Analysis of Lottery Ball Problem

This reluctance to solve a problem before attempting to code it might explain why so few students were able to identify a potential approach to solve this problem. For example, in Figure 7-11b the student has decided that a new array is required to store the randomised ball value in a "new slot" i.e. a new position, and deduced that a search of this array is required to determine if the value already exists before storing it. In considering question three, the same student's analysis of reversing the display of the array contained the crucial observation that the count had to be translated into a count down and this can be seen in the code as shown in Figure 7-12. Actually, the array did not need to be reversed but as a solution to the problem it still works. This solution is also generalised so it can work for any size array.

i = 0 j = 4
1 3
2 2
3 1
4 0
	static int[] reverseArray(int[] numbers)
	{
	int[] reverseNumbers = new int[5];
	
	for(int i = 0; i < numbers.Length; i ++) ← Range of i is 0 to 4
	{
	int j = numbers.Length - 1 - i; ← Range of j is 4 to 0
	reverseNumbers[i] = numbers[j];
	}
	return reverseNumbers;
	}

Figure 7-12 Student Analysis of Reversing Array of Numbers

However, in Figure 7-11a, "Randomise selection of number" does not add to the student's understanding of the solution because there is a failure to map from the problem domain to the student's existing knowledge of the program domain.

A previous test (Appendix 2: Pre-Instruction Test 4) was given to both focus groups at the beginning of semester 2 to compare their relative performance before problems solving skills were taught to both groups. The average marks for this test was 62% for the ACI group and 81% for the non-ACI group. In this final test, the respective marks were 57% and 74% showing the gap had closed a little. On further inspection, the weakest participant (Student A) across both groups was a member of the ACI group (Table 7-7), and had significantly lower marks as a result of failing to complete a number of the questions. This student had issues performing under test conditions, although the assignment results he later achieved were comparable with the other students in the group.

ACI	
Student A	22%
Student B	84%
Student C	78%
Student D	44%
Non ACI	
Student E	95%
Student F	55%
Student G	91%
Student H	55%

Table 7-7 Final Test Marks for Focus Groups

Excluding the marks for Student A produced a fairer reflection of the ACI group's progress and produced a result of 69% showing that the gap had not just narrowed but had closed. An explanation for this is that ACI does not focus on problem solving skills whereas the non-ACI group had attempted a more varied range of problems in the more traditional teaching approach. However, by the end of the course the results for both groups were comparable demonstrating that ACI did not disadvantage the students over the full academic year. The potentially negative effects of a time-restricted test are students becoming overly stressed or running out of time by concentrating on a particular difficulty they are having with some aspect of one of the problems. However, these effects were minimised by carefully monitoring the students during the tests and emphasising that they were designed to enable the students to evaluate their own progress and would not count as part of the official course assessment. All the students engaged with the tests and the range of test results obtained suggest that these issues had little impact on the results of this research.

7.7 Student Interviews

At the end of the academic year, the students in the focus group were formally interviewed and transcripts were taken. A set of eleven questions were designed, of which five were directly related to ACI and therefore only answered by the 4 students in the ACI focus group. The student's prior knowledge is shown in Table 7-8.

ACI Focus Group	
Subject A	Some basic HTML and CSS.
Subject B	Limited programming experience.
Subject C	Limited programming experience.
Subject D	None
Non-ACI Focus Group	
Subject E	None
Subject F	Python coding at college
Subject G	Prior programming experience in JAVA
Subject H	Prior programming experience in Python and JAVA. Found OOPs too difficult.

Table 7-8 Student Experience Prior to Course

7.7.1 Analysis of ACI Interviews

The five ACI questions (Table 7-9) were designed to evaluate the students' experience of the ACPs and the process of learning them through small, tailored exercises. Therefore, these questions were only addressed to the students in the ACI focus group.

Question	
1	How did you find learning these concepts as patterns?
2	How did you find applying these patterns?
3	Did the style of exercises provided help you?
4	Was the number of exercises appropriate?
5	How much did you find the exercises reinforced your learning of the patterns?

Table 7-9 List of ACI Interview Questions

The students' reaction to the principle of learning through ACPs was overwhelmingly positive, all of the group referred to the need to memorise them and Subject A specifically mentioned that learning the constructs as patterns made it easier to recall them. In fact, the need to remember the patterns was a continual theme with Subject B feeling "betrayed by my own memory". Subject C compared ACI with the previous teaching approach they had experienced and felt that they had learned more "we learnt from the bottom up how to apply concepts and learned the ins and outs". We can infer that they felt that the ACPs provided a framework within which they could build their understanding i.e. "the ins and outs" of programming. The role of repeated application of ACPs in remembering and transferring knowledge across problems was also commented on by 3 of the students. The increase in difficulty of the exercises was noted by the group but the gradual nature of this increase succeeded in mitigated any potentially negative effects. For Subjects B and C, the exercises were easy at the beginning, although both felt they were still good practice. An interesting comment made by Subject C was that the exercises were getting harder as a result of more being incorporated in them. Since the

exercises being discussed explored the application of patterns rather than problem solving, we may conclude that the perceived difficulty of the exercises is related to the integration of multiple patterns. The same student also noted that the time taken to complete the exercises increased until they were completing far fewer exercises in the available time as the course progressed. A core principle of ACI is the provision of multiple exercises to reinforce an ACP: as the time taken to complete exercises increased so the number of exposures to the pattern decreased. With hindsight, even the limited "problem solving" may have served to distract the students from the purpose of memorisation and application of the patterns. For example, a student may have known how to swap the values in two variables but asking them to deduce that the same technique could be applied to swapping values in array elements may have taken them some time. Might this time have been better spent over a number of exercises examining exchanging array values in counting loops in various ways? The exercises do not need to have any real-world relevance, so they can be arbitrary and focus solely on the swapping of array values. This is potentially one benefit of POI, since the initial solution is provided and students practise applying it across a range of similar problems.

7.7.2 Analysis of Problem Solving Interviews

Both focus groups (8 students in total) were interviewed to assess their attitudes towards problem solving and programming following instruction in programming problem solving techniques. The questions (Table 7-10) were broadly divided into three themes, the first identified whether the students felt they had changed their approach to programming problems, the second addressed the nature of any difficulties they experienced and the third assessed whether they felt the pedagogical approach had been effective. In considering the nature of the difficulties experienced by the students, three of the questions (3a, 3b and 3c) were only asked if during the interview there was suggestion that the student might have had issues with their problem solving ability. These additional questions were intended to establish the stage at which these issues had developed. Did the difficulties arise when trying to interpret the question, in visualising a solution or were they specifically related to the process of translating a solution to code?

Question	
1	What would be the first thing you do when approaching a new programming problem?
2	What do you think of your overall approach to programming problems?
3	How would you describe your ability to solve problems? a: What kind of difficulties did you experience in understanding the problem? b: What kind of difficulties did you experience in solving the problem once you understood it? c: How difficult did you find coding the solution once you knew the solution?
4	Did you find solving programming problems helpful?
5	Did you feel you were reapplying underlying principles?
6	How much practice did you do outside of class?

Table 7-10 List of Problem Solving Interview Questions

The first two questions were intended to identify each student's approach to programming. Subjects F and H, stated that before the course, they would sit in front of blank screens when faced with a new problem they could not immediately solve. Both now felt more confident and more able to tackle problems. When asked what their approach to a new problem would be, 6 of the 8 students discussed subdividing a problem immediately while the remaining students talked about taking a step-by-step approach. Furthermore, the important role of functions in this process was also recognised by 5 of the students. For example, Subject F elaborated on developing functions instead of putting all of the code in the main. In this case, the student is breaking the code into separate problems and making the main function tell the story defined by the problem description i.e. the functions are named according to the sub-problems identified. Related to this, Subject H described creating functions that did only one thing: a consequence of ensuring a function only solves one problem reflecting the name it is given. Although Subject D reported feeling "scared" when first faced with a new problem, overall, the interviews demonstrated that the students felt more confident and had a better understanding of how to approach problems.

In evaluating their ability to solve problems (question 3) 5 students expressed varying levels of difficulty, the two most common difficulties being understanding the requirements of the problem (2 students) and coding the solution (4 students). Example comments include "…getting confused about what the problem requires", "Trying to figure out what is required was difficult" and "Knowing how to code the solution was the

hard part". Only Subject C felt that they had difficulties solving a problem once they understood the requirements of the question, and in this case the student felt that he needed to develop the ability to view problems from different perspectives. This suggests that students who recognise that they have difficulties see their problems being related to a direct translation from natural language problem description to code. However, the evidence from observation and testing indicates the core problem is an intermediate step involving developing a solution that meets the requirements of the problem and crucially, in a form that can be programmed. In short, they fail to recognise the importance of mapping between the problem domain and their existing programming knowledge. As an example, Subject B concluded that he was "…too eager to get programming to solve the problem first".

Questions 4 and 5 addressed the students' views on the pedagogical approach. In terms of solving problems, the main benefit that all 6 students identified was the number and range of problems they were provided with one student likening it to the process of learning mathematics. All agreed that exercises enabled them to transfer principles (and, of course, patterns) between exercises. Two students found this process more difficult, for example, Subject B found he was forgetting the patterns required and was "blinded by the problem". This, of course, is related to the mapping process but also acknowledges that without continuous repetition, programming knowledge fades over time even during the duration of the course. During the interview, the same student also expressed his appreciation of the unannounced tests because they helped him to monitor his own progress. A key purpose of these tests was to provide opportunities for students to test their memory, and to further promote the importance of memorising the ACPs.

The final question considered the students' approach to practice. Disappointingly, only 2 of the students engaged in regular practice, the remainder viewed the assignment as the opportunity to practice in their own time. Limited practice has two drawbacks, firstly, it allows the memory of previously learnt patterns to degrade over time and secondly it reduces the range of exercises the students are exposed to, thus constraining the development of the skills required to transfer those patterns between problems.

7.8 Conclusions

ACI acknowledges the importance of abstraction in programming and incorporates it in teaching at a very fundamental level through a series of patterns. These patterns adopt a

template approach that mimics the process by which programmers memorise syntax and thus learn to program. By splitting off and moving problem solving into a later activity, the students were able to focus on memorising, recognising and applying the fundamental abstract programming patterns. The provision of multiple simple exercises tailored to each pattern provided a more gradual gradient in difficulty that gave better support for the weaker students reducing programming's "*brutal feedback*" [22], and all the students demonstrated good recall of the patterns. The benefits of this concentration on memorisation through repetition and test, were noted by the students in their interviews. Two indirect benefits of this approach were that it enabled a better appreciation of the difficulties faced by the students and it allowed exercises to be targeted more carefully to those specific weaknesses. For example, one conclusion that came from observation was that mixing even simple patterns, such as in an Array Counting Loop, caused considerable difficulty. Even though the individual patterns were learnt separately and understood, the main obstacle became the relationship between them. In short, novice programmers found the interaction between patterns difficult because they failed to "see" the data flow or control flow they shared. In the Array Counting Loop, they failed to see how a variable could be both a count and an index into the array. Therefore, at least 3 exercises must be provided to reinforce such interactions. In setting an exercise, care was taken to simplify the problem description to avoid distracting the students from the key concept being applied. A number of misunderstandings were quickly remedied by teaching the students how to interpret natural language statements to enable them to extract the pertinent information. A process of starting with terse problem statements and slowly introducing more text was found to be very beneficial, since the students were effectively being primed to find the required underlying abstraction. An interesting observation was that there existed a contradiction between the difficulties that the students had in solving problems and their explicit belief that the problem was related to code translation. That is, they believed they knew the solution to the problem but could not translate it to code. This was best observed when the students were required to document, via a table, the mapping from the problem domain to their existing knowledge. This process implies that there is an intermediate stage in the development model, where the solution to the problem is ascertained in a form that can be translated to code. This provides support for Pennington's [101] theory that programming knowledge is divided into a situation and a program model, where the situation model represents the knowledge drawn from the

problem and a mapping process occurs between both models. The students found this mapping process very difficult, which leads to the conclusion that the students tended to build their understanding of the program from a simplistic situation model. Pennington's [101] program model was divided into text structure and plan structure knowledge, where plan structure knowledge represents an intermediate stage between the situation mode and the translation to code. The reluctance and difficulties experienced by the students in performing this mapping suggests that the problem may lie in this intermediate process, in which the situation model has to be converted into a programmable solution. The programmable solution does not represent code but represents a solution that can be translated to code. The students focused on raw information extracted from the problem and code patterns (syntax), but they failed in the mapping stage. In short, they failed to solve the problem before attempting to code it. This most clearly manifests itself when students take a "code thrashing" approach, where they rewrite the same piece of code multiple times until they accidentally implement the correct solution or give up.

The difficulties experienced by the students during this ACI instruction vindicated the decision to minimise the emphasis on problem solving. Problem solving is a key skill, but applying programming abstractions (syntax) themselves posed a sufficient enough problem during this initial exposure to programming. Therefore, it is clear that introducing problem solving at a later stage was the correct decision. Following a course in problem solving, the results achieved by the students given ACI instruction were comparable to those achieved by students given more traditional programming instruction. Therefore, we can conclude that the teaching of problem solving can be delayed without impacting student performance over the academic year. During problem solving instruction, some issues were identified in setting appropriate problems. These issues were, in part, related to assuming that students possessed the domain knowledge to solve them. This reflects the grounded theory analysis, where a significant component of what is considered expertise is possessing the necessary domain knowledge. This study suggests that accessing the students' domain knowledge may be a prerequisite before providing exercises requiring that knowledge. However, while recognising these issues, the overall response to this instruction was very positive with students commenting on the confidence it gave them.

In conclusion, familiarity with the programming concepts should precede a more formal approach to teaching problem solving skills. Identifying the appropriate level of challenge for a problem is difficult as it relies on a student's ability to interpret the problem and their existing level of domain knowledge. Simplifying exercises and focusing on pattern based learning builds initial confidence, and delaying the exposure to more challenging problems has no adverse effects.

8 Teaching Advanced Programming Problem Solving Skills for Programming

Given the importance of problem solving in programming, a study was conducted into a teaching approach that would enable students to solve larger more "open" problems that reflected real-world scenarios. The use of problem based learning is well established in some sectors of education but has not been widely adopted in programming. In this study, a modified and carefully structured approach was adopted to determine the potential benefits and drawbacks of scaffolding using a programming framework constructed by the student through a series of exercises. A programming framework is a software structure or set of components that allows a programmer to solve larger problems and to build applications [357].

8.1 A Structured Problem Solving Approach to Teaching Programming

Problem based learning was first proposed in the 1960s at McMaster University Medical School [358], to encourage students to work together in groups. It is intended to encourage the development of communication, problem solving and self-directed learning skills [359]. An open-ended problem is posed that requires the students to work together in collaborative groups with the lecturer taking the role of the "facilitator" of learning [360]. The process has been refined to six essential steps, namely, starting with the essential question, designing a plan for the project, creating a schedule, monitoring the students and the progress of the project, assessing the outcome, and evaluating the experience of the learners [361, 362].

> *"…problem based learning is any learning environment in which the problem drives the learning"*[360]

The problem is posed before the students are given any new knowledge and the students should discover through their own activities that they lack the knowledge to solve the problem. In so doing they should develop their inquiry and intellectual skills [360]. The main issue with this approach is neatly summed up by Michalewicz:

> "*Since problem based learning starts with a problem to be solved, students working in a problem based learning environment should be skilled in problem solving or critical thinking or 'thinking on your feet' (as opposed to rote recall).*" [360].

Puzzle based learning [360] is a complementary approach that emphasises the learning of problem solving skills while retaining the fundamental concept of learning through problem solving. It recognizes that developing problem solving skills requires practice and the review of solutions in order to study the principles and techniques required. It encourages reflection on what has been learnt and most importantly, an understanding of how the solution has been applied. Primarily, this approach has been used to teach mathematics [360]. In developing puzzles, four factors should be considered [360]:

1. Puzzles should be used to develop a general/universal principle that can be applied to a range of problems.
2. The easier the description of the problem is the easier the solution is to remember.
3. However, the problem should frustrate the solver so they gain a sense of reward when they solve it (the Eureka factor). This is also clearly related to "*effort after meaning*"[363].
4. Finally, the puzzles should be entertaining, perhaps by setting them in an interesting context (e.g. a game), to prevent the students losing interest.

8.1.1 What is a Structured Problem Based Programming Exercise?

Unfortunately, in programming, problems tend neither to take the form of one simple problem to solve nor one simple strategy to apply. The nature of programming requires the solution of a number of interrelated problems. What can be considered a problem depends on the ability of the student. One student might take on the challenge of implementing a linked list, while a weaker student may find simply identifying where a loop is required quite difficult. Therefore, neither the problem nor the puzzle based approaches is fully applicable in teaching programming students. The proposed alternative structured problem based learning approach, provides the student with a code framework constructed by the student themselves through a series of exercises. "Structured" in the sense that the exercises must be self-contained and presented with appropriate teacher scaffolding [159], such that when the solutions are integrated they form a complete framework that can then be used to solve much larger problems. In effect, this confirms the findings of Deek *et al*[33] that scaffolding the problem helps to present the problem and develops an initial understanding of the problem. The scaffolding may also begin the process of subdividing a problem and setting initial sub-goals. This approach recognises that large software projects require the ability to solve a

range of programming problems. Unlike pure problem based learning, the nature of programming primarily requires the student to develop the ability to discover the problems and to consider abstract generic solutions that can be translated to code. Thus, the key difference in programming exercises is that the main focus is on developing skills rather than gaining more knowledge. In this respect, the approach is similar to puzzle based learning. However, in pure puzzle based learning the emphasis is on making puzzles simple to state and reinforcing a general principle. However, Michalewicz does state that puzzles do not necessarily need to conform to both these criteria [360]. Nevertheless, programming problems tend to be too open-ended to be considered puzzles.

In applying the structured problem based learning approach to teaching programming, a number of issues must be addressed:

1. Poor Problem Solving Skills and Lack of Motivation

   The framework should be designed to emphasise the subdivision of problems. This means paying careful attention to ensure functions only solve one problem, and in Object Oriented Programming it means applying the Single Responsibility Principle (SRP) [364] even when this may lead to many additional classes. Problems may be set at more than one level, for example, students may be required to complete a class or just use provided classes. The emphasis should be on the student discovering problems for themselves, a skill that underpins this approach. Often students are demotivated when they see no purpose to the theory they are covering or the problem they are solving. Allowing the students to incorporate their solutions in the framework or enabling them to visualise how their work would be applied is of considerable benefit. A student is more motivated to solve a problem if they see a need for the solution because they discover it themselves. The framework should enable scenarios to be designed where this type of discovery based learning can take place and leads to the introduction of new concepts (aka problem based learning). As in pure problem based learning, they may not have the complete knowledge to immediately solve a particular problem they discover. In this case, the framework can be designed to enable them to work around this until the new concept is formally introduced. Alternatively, the framework could just act as a familiar environment in which a number of principles or concepts can be demonstrated. Teaching material should provide

enough detail for the students to begin developing a complete solution, and enable them to develop the ability to see related problems.

2. Copy Cat Syndrome

   One way for beginners to learn how to program is to follow how a problem is solved and the program is written [29]. Often this takes the form of a written tutorial containing code that they can copy. However, Scott [214] hypothesized that many problems experienced by students arise because programming is taught this way, and this does not help students to learn how to solve open-ended problems [29]. A carefully structured framework should provide considerable opportunity to challenge students to apply their skills and knowledge to complete aspects of it or solve specific problems.

3. Lack of Knowledge

   By carefully subdividing problems in the framework implementation and the liberal usage of SRP, problems can be cleanly isolated and the knowledge required to solve them can be well defined. Research [365] has also shown that students perform better when provided with a template in which to work rather than developing code from “scratch”. Thus, the framework should provide a structure to support the students’ learning.

4. Scalability of Problems

   Large programming problems, or more advanced software architectures such as a class library, are often quite complicated and take considerable time to teach. Subdivision of the architecture enables a series of simpler component parts to be considered. In addition, dividing the architecture allows the concept(s) used to be built up to a complete solution over multiple lectures. This also allows theoretical content to be integrated more closely with the practice. Not all the classes in the framework may need to be taught: if one class uses three others it may only be necessary to discuss the most important class to illustrate a concept. Students can be expected to investigate certain aspects of the framework as part of their self-study.

5. Separation of Theory and Practice

   Theoretical ideas are easier to contextualise and illustrate when the architecture of the code is designed to provide a clean separation between individual problems or implemented principles.

6. Lost Students

   Although the problems provided should be well defined in scope and appropriate teacher scaffolding provided, it must be anticipated that some students will implement an incorrect or incomplete solution that will require some remedial action. Therefore, a complete solution should always be provided at the end of each stage for student review.

8.2 Methodology

A study was conducted into teaching the principles and concepts associated with Javascript frameworks such as Backbone.js, Angular.js and Ember.js. A new framework was developed explicitly for teaching purposes and to enable problem based learning. In the study, a focus group consisting of 8 final year degree computing students were introduced to this framework over the course of several weeks, during which their progress was observed and at the end of which they were asked to complete a survey. A number of the questions on the survey were designed to confirm that the students' experience matched the observations made.

8.3 The Framework

The framework was developed to teach a class in advanced Javascript application development and was developed in object oriented Typescript and jQuery. Typescript was chosen because it provides type checking and is a good Object Oriented language, allowing a very readable class library to be created. In developing the framework, the emphasis was on applying the Single Responsibility Principle (SRP) [364], which states that a class should fulfil only one responsibility and should have one, and only one, reason to change [366]. This resulted in over 10 smaller classes and often meant rewriting existing Javascript libraries to simplify their structure. However, performance was not a major consideration as the primary emphasis was on readability. The approach was to construct as many elements of a number of classes as possible with the students, and set them challenges to complete the work. To make this possible, the size of the classes produced had to be as short as possible. Where this was not possible, the students had to be provided with the code for a class and instructions in its use.

A number of the main features of the available MV* frameworks were identified and the framework was designed to incorporate similar features. Again, the aim was not to build an efficient framework, but to develop a coherent design that allowed various concepts

to be demonstrated. A popular design pattern used in these frameworks is the Model View Controller (MVC) pattern [367], and consequently the implemented framework contained the concept of a Model that described the data, a View that displayed the data and a Controller that determined the view to be displayed and the data to be passed to it.

An event-driven architecture based on the publisher/subscriber principle (or broadcast design pattern [270]), enabled communications between the various component parts e.g. a Model generates an event when a data value is changed. This approach achieved the flexibility required to allow a variety of classes to communicate with each other and thus enabled a number of software architectures to be explored. This was key to enabling different problems to be identified and tackled independently.

The View classes in all the frameworks use standard HTML to display the content and any associated data values in the browser. A common approach taken by similar frameworks is to use a template engine, which interprets a set of special tags embedded in the standard HTML and replaces them with the required values rendered appropriately e.g. as rows in a table. This is a variation of the Template View pattern [368]. Handlebars.js was chosen for this framework because of its simplicity. A TemplateView class was provided to underpin this concept, from which students could derive their own View classes. Inheritance enabled the student to build upon solutions previously developed.

As an additional challenge, an XML parser was produced that enabled additional template tags to be parsed to register event handlers that could handle browser DOM events e.g. if a button was clicked. This would have enabled more advanced students to take on a more significant challenge, but in the end was never used because of time constraints.

Javascript frameworks allow for dynamic and interactive delivery of data to the user by using Asynchronous Javascript and XML (AJAX) [369] and Representational State Transfer (REST) web services [370]. An AjaxifiedModel class was provided, from which the students could derive their own class for reading appropriate data using AJAX. The data was retrieved in Javascript Object Notation (JSON) [371] format and students were introduced to the benefits and use of this format. Before teaching of the framework began, the students were introduced to AJAX and the dynamic exchange of data in JSON through a number of examples. The aim was to demonstrate that encapsulating the AJAX handling prevented the reinvention of similar solutions. It also made it possible to set the students challenges to complete aspects of the AJAX handling.

8.4 Instructional Scaffolding for Teaching Using a Framework

An important difference between pure problem based learning and structured problem based learning, is providing sufficient scaffolding so that a programming problem is approachable and engages the students' interest in implementing a solution which contributes towards the "bigger picture" in a carefully structured way. A balance has to be struck between frustrating the learner [360] and providing sufficient knowledge to ensure that they are able to solve the problem. Both direct and indirect learning approaches [372] were used in this study. Lectures began with direct instruction of the theoretical concepts to provide the background knowledge of the underlying concepts and to outline the goals of the lecture. It was important to begin the lecture with a discussion of the problem that needed to be solved and to clearly outline why it was important to solve it in the context of the framework. The lecture itself was interspersed with a discussion of code that demonstrated the ideas being outlined and this code sometimes acted as a partial solution. Once the problem, the reasons for solving it and the concepts(s) being covered were understood, the students were then asked to solve a related problem. This problem would either require them to complete a solution, or alternatively to apply the concept(s) to solving a similar problem. As the course progressed, the problems would make use of the elements of the framework already created by them. Thus, the challenges presented to the students grew in scale as the solutions drew upon their previously acquired knowledge. Typically, each solution involved developing or adapting one or two classes. This incremental acquisition of knowledge is a key characteristic of structured problem based learning. Learners can be divided into entity-theorists and incremental-theorists (Table 8-1). Entity-theorists "*believe their aptitude is natural fixed trait*" while incremental-theorists believe it is a "*malleable quality which is increased through effort*" [373, 374].

	Entity-Theorists	**Incremental-Theorists**
Goal of student?	To demonstrate a high coding ability	To improve coding ability, even if it reveals poor progress
Meaning of failure?	Indicator of low programming aptitude	Indicative of lack of effort, strategy, or pre-requisites
Meaning of effort?	Demonstrates low programming aptitude	Method of enhancing programming aptitude
Strategy when meets difficulty?	Less time practicing	More time practicing
Performance after difficulty?	Impaired	Equal or improved

Table 8-1 The Potential Influence of Different Theories of Aptitude[373]

The structured and incremental approach is important as it encourages an incremental-theorist attitude, which is an important component of any programming pedagogy [373].

One difficulty that may arise is that students can fail to understand aspects of the code or architecture, or simply forget the solutions they produced. Clearly the benefits of progressively building a mental model can be lost should this occur. To resolve this issue, a website was created containing pages that outlined the problem, motivation, and then fully documented the code covered in class. Pages containing the solutions were also added as the course continued, so that students always had access to a solution to review. In creating this website, a Cascading Style Sheet (CSS) file was used to highlight lines in the code and hyperlinking was used to associate text to code, code to text and code to code. These hyperlinks were also made between pages, allowing for easy reference to previous solutions and the text covering previous concepts. The aim was to minimise any distraction from the current work in progress by enabling the students to refer to any information they needed with the minimum of searching. For example, a student could link from a concept in the text to the method that implemented it in the code and follow the chain of method calls.

8.5 Survey Results

On completion of the short course, the eight students were asked to complete an online survey, the objective of which was to confirm in-class observations and assess the benefits/drawbacks of the structured problem based learning approach. All the questions (Appendix 3) were scored on a Likert scale of 1 to 10, excluding question 5 that addressed the topic of practice. These questions are shown in Table 8-2, and where appropriate the mean score has also been shown:

Question			Mean Score
REFFLECTION			
Q1	I recognise the importance of solving problems in programming		8.63
Q2	I find solving problems challenging		7.63
Q3	I find solving coding and solving problems interesting		8.00
Q4	I have learnt more by attempting to solve problems myself in class		7.50
Q5	In working on the exercises provided:		
	I spent very little time attempting them	Response	0%
	I would like to have spent more time attempting them		87.5%
	I was too busy or unable to attempt them for other reasons		12.50%
	I felt I dedicated enough time		0%
	I spent too much time		0%
INTERACTION			
Q6	Engaging in solving problems leads to more class interaction between students		7.88
Q7	Engaging in problem solving learning leads to more class interaction between students and lecturer		7.63
Q8	I felt I was solving problems with the lecturer		7.63
Q9	I found the class more interesting when trying to solve the challenges presented by the lecturer		8.25
TEACHING APPROACH			
Q10	I prefer to follow code or solutions, step-by-step, developed by the lecturer		6.25
Q11	Problem solving activities provide gave me a better understanding of the technologies or principles being taught		7.50
Q12	The context of the problem is important (I like to know why it is important to solve a problem)		8.25
Q13	It is more interesting to discover next problem(s) myself, as a consequence of completing a previous exercise		7.25
Q14	I prefer partially solved problems to new problems with no initial code provided		5.25
Q15	I prefer to learn new technologies or concepts by attempting to build my own solutions		7.13
Q16	I reviewed the completed solutions offered by the lecturer after attempting the problems myself		7.50
LEARNING MATERIAL			
Q17	Sufficient documentation was provided to attempt the exercises		7.13
Q18	Providing hyperlinks between the code in the documentation enabled me to follow the code more easily		7.63
Q19	The exercises provided a gradual increase in difficulty (allowing for the complexity of the concepts being taught)		7.38
CONFIDENCE			
Q20	I found this approach gave me confidence in my ability to develop my own learning skills		7.43
Q21	I will be more confident in studying new technologies in the future		7.86

Table 8-2 Structured Problem Based Learning Survey

In the following text, the question number and mean score are shown in brackets e.g. question 1 with a mean score of 8.63 is shown as (Q1:8.63).

The questions were subdivided into five separate sections: reflection, interaction, teaching approach, learning material and confidence. In the first section, the students were asked to reflect on their own personal skills and whether they felt that problem solving was a skill that they wished to develop further. This section consisted of five questions. In terms of student motivation, there was recognition that it was an important skill (Q1:8.63) and that solving problems helps to develop this skill (Q4:7.50). However, the answers to question 5 show that they all felt they should have dedicated more time to practice. Unfortunately, learners often claim that they lack time [373] for dedicated practice. There are a number of factors [373] that can prevent students from practicing. Given there was no continual formal assessment associated with each problem set within the course, it is possible that this contributed to the lack of engagement outside the class. As Gibb *et al* [375] have observed, learners "*often focus on topics associated with assessment and nothing else*". Therefore, the study has to be restricted to an analysis of the students' behaviour within class. The lack of practice may also be because "*learners start to believe an inherent aptitude is required to become a programmer*" [373] or becoming overly frustrated [373]. An element of frustration is inherent in the process of learning by problem solving [360] and this is reflected in the survey (Q2:7.63). However, student motivation to solve the problems has not been impacted by this (Q3:8.00).

Structured problem based learning should provide sufficient scaffolding to enable students to discuss problems coherently and the framework should provide an environment that promotes shared experiences. It would appear that there is broad agreement that with this approach, the level of interaction between students (Q6:7.88) and students with lecturer (Q7:7.63) was good. By observation, setting students challenges immediately prompts questions and the lecturer must play an important role in guiding the students towards the correct solution i.e. play the role of the "*facilitator*" of learning [360]. Again the students agreed that this approach enabled the lecturer, in this role, to be seen to be solving the problems <u>with</u> the students as opposed to solving them <u>for</u> the students (Q8:7.63). Setting problems did not negatively affect the relationship with the lecturer and confirms that the progressive incremental nature of the approach was successful in maintaining student interest (Q9:8.25).

A set of questions directly addressed the benefits of structured problem based learning. When asked whether they preferred to follow lecturer provided solutions instead of developing their own (Q10:6.25), the results were moderately in favour of solving the problems themselves. However, there was more agreement when this approach was related to learning a new technology or concept (Q15:7.13) and the students felt that the approach did help them to learn new ideas more effectively (Q11:7.50). This suggests that, at least within a lecture environment, students respond positively to the structured problem based learning provided they can see they are learning something new i.e. they are not just being asked to practice something they have already covered. Obviously, students should practice outside of class and they were aware of this (Q5). Interestingly, when it comes to developing a solution, providing an initial code skeleton to contextualise or aid the student in solving the problem may instead be acting as a barrier to learning (Q14:5.25) for some students. Figure 8-1 shows a comparison of each students response to questions 10, 14 and 15 which gauge the students' reaction to the amount of code that should be provided with a problem, ranging from a complete analysis of the solution (Q10:6.25), to a partial solution (Q14:5.25) or just a statement of the problem with no code provided (Q15:7.13). Clearly the results are mixed, although five out of the eight students preferred extending a partial solution or developing their own complete solution over constructing the full solution with the aid of the lecturer. These results demonstrate that with appropriate teacher scaffolding many students prefer solving problems themselves with minimal support. It also shows that even students that prefer to be led through a solution to a problem, are in most cases not alienated by the approach (Subject H being an exception). There is some evidence [4, 365] that students do perform better with a template rather than creating a complete solution on their own, but a comparison of the students' preference against actual performance was not conducted in this study. Of course, these results may also reflect the difficulty in providing appropriate scaffolding that meets the needs of each student without reducing the overall effort and the challenge that makes structured problem based learning interesting and motivational.

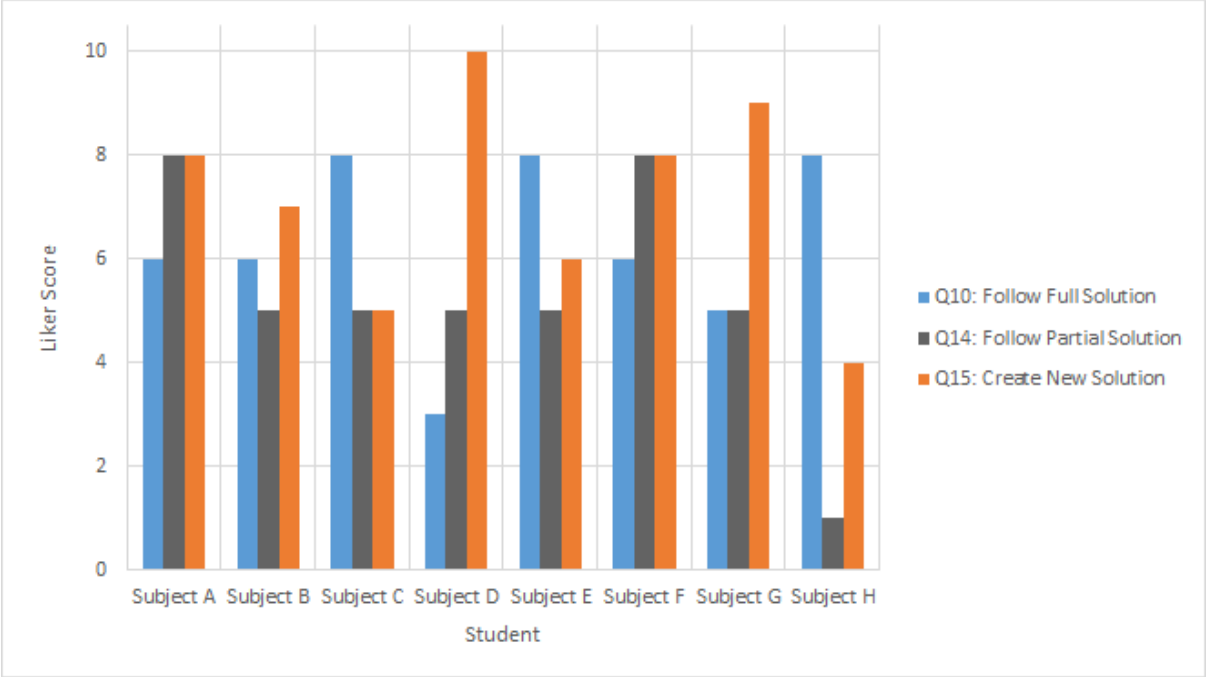


Figure 8-1 Comparison of Student Responses to the Presentation of Problems

During the course, it was observed that students found the reason for solving the problem i.e. not solving a problem for the sake of solving it, was an important element in motivating them. This observation became a key component of structured problem based learning. To determine whether these informal observations were correct, the survey contained questions that queried the importance to the student of the context of the problems they had solved. Likewise, to confirm another observation, the survey set out to find if they felt more motivated if they identified problems by themselves. The student responses confirmed both these observations. Contextualising the problem was very important (Q12:8.25) and the process of discovery also drew a positive response (Q13:7.25).

The final question about the teaching approach sought to determine whether providing correct solutions for students to evaluate against their own, was taken advantage of by the students. On average, response to this question (Q16:7.50) was good but an examination of the individual responses (Table 8-3) suggests that some students failed to engage with the process as actively as had been hoped. It is likely that the reasons for this are similar to those already discussed in considering question 5. However, students were observed reading through and making appropriate modifications during the lecture in order to progress to the next problem/concept. Unfortunately, it is also likely that some students simply copied the solutions. This is unavoidable, since in an incremental learning

approach it is important to give students an opportunity to solve their own problems and to keep up to date with current progress.

	Likert Score
Subject A	7
Subject B	8
Subject C	5
Subject D	10
Subject E	7
Subject F	6
Subject G	10
Subject H	7

Table 8-3 Student Response to Q16: Review of Provided Solutions

The learning material section of the survey, contained three questions that directly investigated the appropriateness of the scaffolding. Documentation was provided in the form of a website, and the response (Q17:7.13) was encouraging although three students gave a Likert score of 6. In true scaffolding, the scaffold must be faded [159, 259]. Fixed fading can lead to worse results [247]: instead, the student should be able reduce the scaffolding when they no longer require the support [253]. As previously discussed (Section 8.4), a website was produced with separate pages progressively covering the development of principles and concepts, setting a range of problems and providing solutions. One objective of this approach was to allow the student to reference material as required as a pseudo-fading method, but this may not have achieved the intended aim for all students. In evaluating the effectiveness of the pseudo-fading approach using hyperlinking, the students found this enabled them to navigate through the documentation in quite a natural way (Q18:7.63) and none of the students found the progressive challenge of the problems too difficult (Q19:7.38). Figure 8-2 shows each student's response to the individual questions, and although there is some variation, the majority of students found the material and the incremental learning approach to be very good.

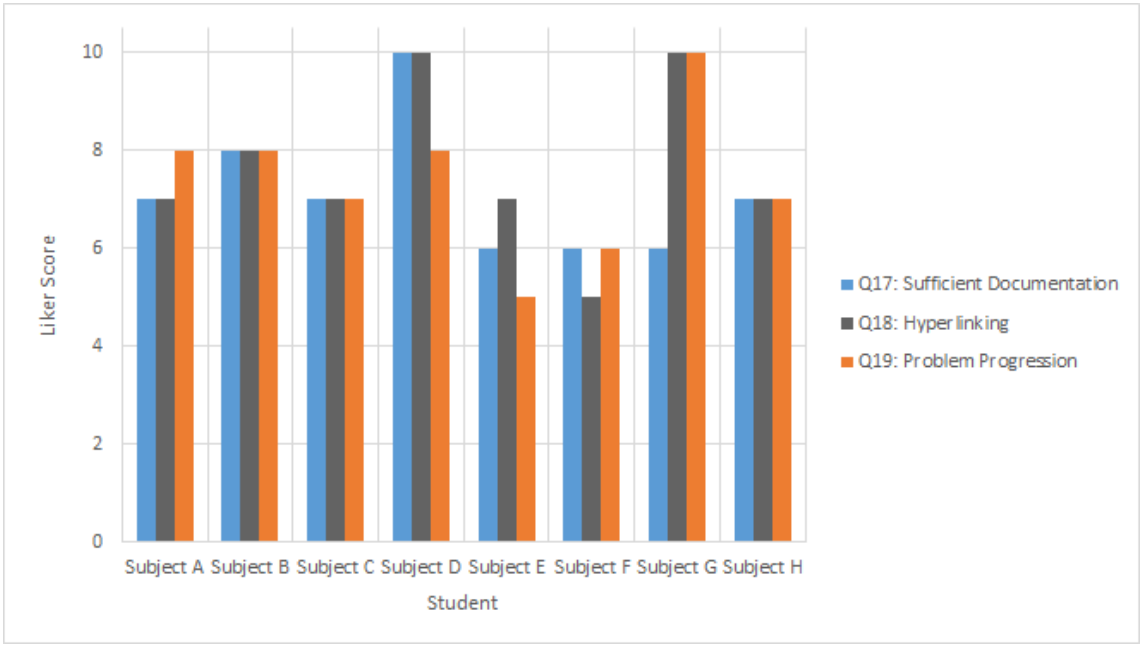


Figure 8-2 Comparison of Student Responses to the Provided Learning Material

In the final section of the survey, the students were asked to reflect on the overall effectiveness of the course in developing confidence in their skills. This is a very subjective measurement and must be treated with caution, but gives some indication of the success of structured problem based learning. In response to two questions, in the students' opinion they found the course gave them confidence in their ability to develop their own learning skills (Q20:7.43) and in learning new technologies (Q21:7.86). Figure 8-3 shows the individual student responses, the vast majority of which are rated 7 or above.

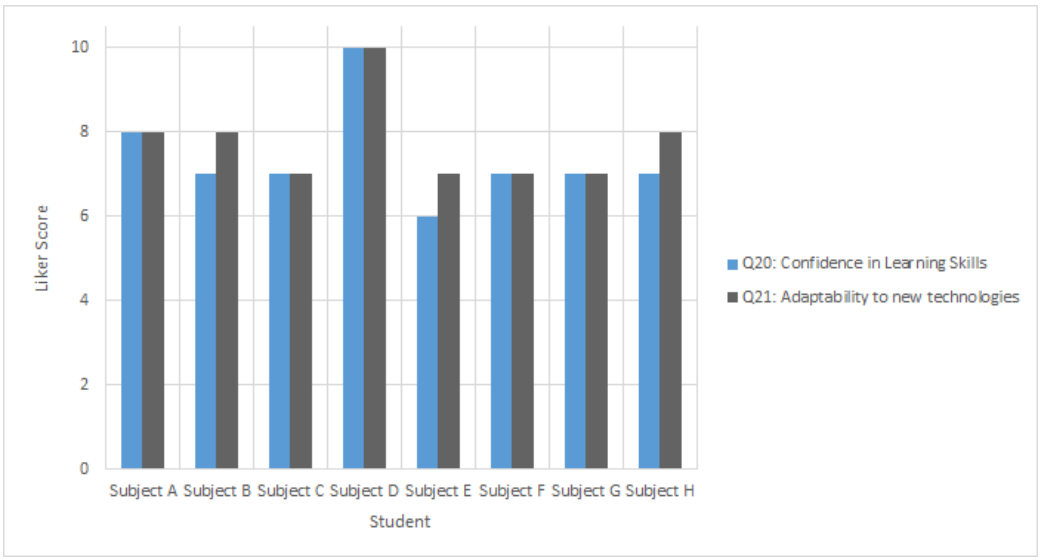


Figure 8-3 Comparison of Student Reflection

8.6 Conclusions

This research investigated the benefits of a Structured Problem Based Learning pedagogy using a programming framework to provide appropriate scaffolding within which problems could organically develop. One conclusion that arose very early in the study was that student motivation to solve a problem was dependent on how invested they felt in defining it and the relevance of the solution in real-world scenarios. In this respect, the programming framework and supporting material proved very successful in both providing a rationale for a problem and enabling students to identify future problems themselves. Solving a problem implicitly requires setting a challenge, and the capability of students to overcome that challenge varies, requiring a scaffolded learning approach. Two types of scaffolding are required: scaffolding of knowledge and scaffolding of practice. Scaffolding of knowledge was provided through a series of highly interlinked webpages providing descriptions of the concepts being introduced. Scaffolding of practice was provided through code either provided in context within the webpages or separately, which was intended to provide the students with a starting point from which to work. Both the documentation and the progression of difficulty were found to be of an appropriate standard. However, in scaffolding of practice, the survey revealed more mixed results. Most students felt that they wanted to develop their own solutions rather than follow a completed solution, demonstrating that the rationale and principles behind the pedagogy were correct. However, this was not true of all the students suggesting that either the level of personalisation of the scaffolding was insufficient to support those students or that they simply disliked the thought of solving any problems. In the latter case, it is likely that they would have reacted negatively to any form of problem based learning. Overall, the results demonstrate that the approach has a number of benefits in motivating and building student confidence, but the scaffolding approach needs further research.

9 Discussion of Action Research and Results

The initial teaching approach adopted worksheets as a means of fostering continuous programming practice. These were constructed such that the introduction of the syntax of a new programming instruction was followed by the associated example(s) that enabled the students to learn how to apply it. However, the grades achieved by the students showed the typical bimodal distribution. Clearly the effectiveness of the practice the students were engaged in was of limited value for weaker students and further research was required to establish a more effective means of practising. In endeavouring to determine why this should be, 12 metrics were identified and were scored at the end of each worksheet, to try to determine patterns of behaviour that were indicative of good or poor programming skills. As a result, problem solving skills were identified as the main, if not the only, important characteristic related to the performance of the students. This study also failed to address the reasons why students had poor problem solving skills and how better student performance could be achieved. In addition, poorer coding performance was particularly noticeable in worksheets 5 and 6, as shown in Figure 5-2 and Figure 5-10, which covered classes and class inheritance respectively. Clearly, Object Oriented Programming (OOP) is implicitly more abstract and this may provide further evidence in support of the Grounded Theory analysis findings. Following these results, for some degree schemes, OOPs was split into a new course and moved to the second year of the programme. The teaching of programming to first year novice programmers now concentrates solely on procedural programming and problem solving. This decision also moved the focus of the research away from OOP specific issues since for novice programmers this represents a higher-order level of abstraction.

From a personal perspective, it was important to investigate new pedagogical approaches that could address the issues identified and determine their effectiveness. Action Research is a methodology for a research process based on the development of one's own practice. In applying this methodology, results are considered to be what the practitioner learns about their practice. This iterative process involves taking action, reflecting on the actions taken and critically analysing the significance of the results obtained. In the mixed methodology adopted, Grounded Theory was applied, initially to enable the development of the literature review, and on an ongoing basis throughout each cycle of modification of practice. In adopting a mixed methodological approach, some compromises were required. The requirement for an initial literature review meant

that the natural emergence of theories associated with Grounded Theory was, at least initially, pre-empted. However, this was redressed by continuing research, experimentation and the re-structuring of the review on completion of the research. The focus on taking action may influence or narrow the research field, which could limit the use of the constant comparative and theoretical sampling approach associated with Grounded Theory. However, the mind of the learner is so complex that this never became a problem as the process of teaching is in essence a process of experimentation.

During the grounded research analysis, the two main theories which developed were that code abstraction and problem solving skills were the primary influences on the development of programming ability. Closely related to these theories were the effect of working memory on problem solving and Pennington's concept of the program model. In particular "plans" as mental abstractions of the code. A study was conducted to determine the influence of working memory on programming and confirmed that a relationship does indeed exist between programming ability and working memory. By comparing results from both a code and Raven Matrices test at the end of a short programming course, a correlation was found between working memory and programming. This offers an explanation for the bimodal distribution of results obtained. Furthermore, this result also suggests that some students are at an inherent disadvantage, at least initially, which requires them to dedicate more time to practice and/or requires a different teaching approach that focuses on minimising distractions i.e. examples and exercises need to be precisely targeted at learning single steps. In the grounded research conducted, a number of characteristics were found to distinguish between expert and novice programmers. Expertise involves building detailed mental models constructed from acquired domain specific knowledge which novices do not have. In addition, novices also lack the same level of problem solving skills and focus more on the concrete surface features of a problem because they are less able to identify abstractions. That is to say, they are more distracted by the natural language presentation of a problem and fail to recognise the applicable abstract programming concept(s). In a study investigating the range selection problem, it was demonstrated that students do tend to apply a natural language procedure literally rather than converting it to the correct Boolean logic. This illustrates that the concrete surface features of a natural language problem definition present problems for novice programmers seeking the underlying abstractions. The potential limitations of student memory combined with the

need to simplify problem definitions, implies that exercises have to be structured in order to promote recall with minimum distraction from the main concept being introduced. Therefore, in developing examples and exercises, these results highlight the need to make them short and direct to minimise irrelevant information foraging. Given working memory capacity limits the tolerance to distractions, significant levels of problem solving presented in an exercise may prevent later recall of the central concept being studied. Thus the challenge of solving problems can reduce the effectiveness of these exercises as a means of practising a specific concept. Practice involves repetition but not all practice is effective. To be effective, the overall aim of the practice must be subdivided into the specific component skills that enable it to be achieved and exercises must be designed to target these skills. This leads to the conclusion that programming is best learnt through a series of highly targeted short exercises, but leaves open the question of the nature and structure of these exercises.

Given the importance attached to abstraction in the Grounded Theory analysis, an approach was sought that would bring together the research fields of software comprehension and programming pedagogy to promote abstract thinking. The grounded theory analysis provided evidence for the relationship between the concept of “plan” knowledge and the mechanism by which that knowledge is applied to the reading and writing of code. Perceptual learning describes the process by which the load on working memory is reduced by learning patterns that can be quickly recognised, a defining characteristic of gaining expertise. Therefore, identifying these “plans” and developing them into patterns for easy memorisation and recall provided a route by which this expertise may be gained more quickly and less painfully. Each programming concept was introduced as one or more patterns and these patterns provide a fixed text structure representing the instruction statement(s). Students were taught to recognise and modify the elements of theses pattern that were dependant on the context in which they were used i.e. the problem being solved. POI is a related pedagogy that develops a series of patterns for solving problems but reduces the creativity by providing template like solutions. ACI was proposed as a new pedagogy that concentrated on the fundamental constructs in significantly more detail and crucially, introduced them as abstractions. Thus, ACI was developed based on the concept of pattern learning. An important feature of this approach was to name these patterns (or ACPs), as this both acted as an aide-memoire and provided a common point of reference between the teacher and the

learner. The exercises provided were short and concise, designed to vary different elements or combinations of elements within the patterns and were graduated in difficulty to allow the learning process to be carefully controlled, especially during the initial exposure to a new pattern. The aim of these exercises was not problem solving per se, but to promote the recall of an abstract pattern and its usage. Furthermore, it was found that difficulties experienced in array usage were due to interference effects caused by attempting to learn multiple ACPs simultaneously. In this particular case, it was not possible to avoid this situation because it is a fundamental of array syntax and semantics. However, this did provide evidence that one of the root causes of difficulty in translating a solution to code is the number and mixture of the ACPs required. This was observed even when, as in the case of arrays, these ACPs were not especially complicated. Thus, in creating exercises to focus on a specific use of an ACP, the number of additional ACPs required was minimised.

As already stated, problem solving skills are a critical component of programming and not supporting and nurturing these skills would be counterproductive. Therefore, ACI instruction was followed by a course covering problem solving skills in a programming context. To determine the effectiveness of both ACI and the subsequent problem solving course, two focus groups were created and tested, observed and interviewed. The first student focus group was drawn from a programming course taught using the ACI approach, the second focus group was drawn from a course taught in a more traditional worksheet approach. Dealing first with ACI, a number of interesting results were obtained (as discussed in chapter 7) and from a teaching perspective these were very encouraging, with the emphasis on recall and rote learning being favourably received by the students. Even the inclusion of unannounced in-class tests proved to be both motivational and beneficial. Furthermore, evidence gathered through the observation and testing also showed that the students were able to recall and apply the patterns. To investigate any potential drawbacks of ACI, both focus groups were taught problem solving skills in a follow-up course with testing before and after. From the results, it was clear that the non ACI group had better problem solving skills at the start of instruction but that this gap had closed by the end of instruction. Thus, we can conclude that ACI did not inhibit the development of these skills over the duration of a full academic year. In fact, abstracting and relating functions to problems in a clearly defined manner was a process that the students clearly identified as beneficial during interviews. There was also some evidence

to support this during the testing of the effectiveness of this approach in teaching functions. The overall results from the problem solving instruction were, by contrast, more mixed. A significant element of this instruction was aimed at building confidence in tackling problems rather than producing complete the solutions. In this respect, the problem solving instruction was very successful: all of the students described feeling more confident, whereas prior to instruction they had felt more fearful or might have suffered a mental block when faced with a new problem. All the students commented positively on the significant number and range of exercises provided during both the ACI and problem solving instruction phases. The main difficulty was assessing the appropriate level of difficulty of an exercise. In setting an exercise, the difficulty experienced by the students depends on their existing problem domain and programming knowledge, and their ability to map that knowledge in order to solve it. For example, it was assumed that basic geometry would be familiar to all students but this belief proved unfounded and resulted in an initial set of exercises being more difficult than expected. A very interesting observation was made by comparing observations of the students' coding approaches and the code they produced against their interviews following problem solving instruction. In attempting to distinguish between the difficulties experienced by the students in interpreting a problem description, the process of developing and coding a solution showed a clear misconception. On one hand, observation and testing showed that the main difficulty encountered was in understanding how the solution to the problem would work i.e. defining the solution, while the students themselves felt they possessed the solution but could not translate it into code. For example, if a problem required a student to find the highest value in a list then most of the students considered this to be a code translation issue rather than problem solving process. A novice programmer might read this as *some value being greater than another*, whereas an experience programmer sees this as *read each value from a list of values and compare it with the provided value*. An experienced programmer does the problem mapping inherently. Requiring students to explicitly solve the problem on paper first by mapping their programming knowledge to the problem description, revealed both a reluctance to perform the mapping explicitly and a casual attitude to its application. This failure by the students to recognise that programming is not a simple translation exercise, combined with a reticence to solve the problem before attempting to code it, explained a number of novice programmer difficulties.

Comparing ACI with a more traditional teaching approach is difficult given the variety of teaching styles and approaches employed. However, to form the basis of a comparison of the new pedagogical approaches proposed in this thesis with “old” approaches, some assumptions need to be made. These assumptions are:

1. A programming concept and the syntax for the associated instruction statement are formally presented
2. A number of worked examples are used to illustrate the use of the syntax, and the student infers how the syntax can be applied
3. The students is given a series of exercises in the form of a number of problems that require the use of the syntax, and problem solving is implicitly required to complete the exercises.
4. Exercises take the form of problem definitions from which the student must elicit the appropriate abstraction(s) required e.g. nested if statement.
5. The programming concepts are presented in a number of defined stages over a number of weeks

Given these assumption, Table 9-1 provides a comparison between the traditional and the ACI approach highlighting a number of benefits.

TRADITIONAL APPROACH	ACI APPROACH
Abstraction is implicit. The learner must develop their own abstract knowledge.	Abstraction is taught explicitly, the learner is taught to view the instruction syntax as an abstract text pattern with elements that vary.
	Simple exercises are provided to explicitly promote understanding of the meaning of the pattern and recognition of the variability of its elements.
The learner is expected to implicitly learn the syntax over time by solving problems.	Exercises are used to promote recall, the learner is actively encouraged to memorize the patterns. Learning by rote is encouraged.
The emphasis on syntax makes it harder to prompt the learner.	The naming of patterns makes interaction with learners easier, and the constant emphasis on recall means that students should be able to quickly understand the teacher's prompts.
The learner is expected to be able to deduce the correct usage of syntax from a natural language problem definition without training	The design of exercises for introducing a new pattern are simple and terse. For example, the learner is often required to just choose appropriate values to complete a pattern.
	Exercises are provided to support the process of mapping natural language to a pattern.
	Where natural language may give rise to misunderstandings, such as in the range selection problem, these are explicitly taught. No assumptions are made with respect to the learners' deductive reasoning skills.
Problem solving skills are an implicit requirement of many of the exercises presented to the learner. Typically, the learner is expected to be able to solve problems they have never seen before, or apply a solution in a different context e.g. applying a loop within an if-statement when they have only seen them used separately.	Exercises deliberately minimize the need for problem solving skills.
	Functions are explicitly taught as solutions to problems, rather than as opportunities to prevent code duplication. This may not be unique to ACI, but it is strongly encouraged in ACI as it provides a clear stepping stone into problem solving.
	Problem solving is taught separately following ACI instruction, although in the teaching of functions there is scope to blur this boundary at the end of ACI instruction.
The range and number of exercises is fairly limited, often due to the time taken by the learner to complete them.	Many shorter exercises are preferred over fewer longer exercises, and a number of exercises are provided that use the same abstract solution.
Practice tends to be more sub-divided into self-contained blocks. For example, a work sheet about arrays might provide exercises that require a counting loop to read through an array but may not provide exercises just on loops to aid recall.	The emphasis on memorisation requires continual testing of the learner's memory: this naturally entails testing of previous concepts across the course. The shortness of many exercises means that they take up little time, allowing more frequent testing. Testing in this context could just entail including exercises during a tutorial session.
Programming concepts are presented in a number of stages over a number of weeks.	Programming concepts are presented in a number of stages over a number of weeks. However, problem solving is taught much later,

	and therefore the exercises become longer and more difficult at a later stage than normal.

Table 9-1 A Comparison of the Traditional to the ACI Programming Pedagogical Approach

The level of problem solving skills that could be introduced at an introductory level was of course limited, and more challenging problems were faced by the students as they progressed. In particular, for final year degree students the expectation was that they by the end of the course they would be able to use code frameworks and tackle problems that reflected those encountered in industry. A key motivation for students when solving a problem is to understand the real-world purpose of the solution. Introducing students to large scale problems raises the issue of how to present those problems in a manner that challenges the students but without the students becoming too confused or intimidated.

In considering a number of pedagogical approaches, problem based learning was the pedagogy that seemed most appropriate for developing problem solving skills. However, the complex nature of programming precludes adopting such an approach without significant modification. Instead, a more nuanced, moderate constructivist and structured approach was adopted. A significant consideration in adopting this approach was the provision of appropriate scaffolding. Two forms of scaffolding were required, scaffolding of knowledge and scaffolding of practice. Scaffolding of knowledge entailed providing suitable documentation and instruction on the concepts being covered. Scaffolding of practice involved providing a code structure within which the students could implement their code and experiment with their solutions. The survey results suggest that although not all of the students engaged with this material to the extent that was anticipated, there were no particular concerns about the quality of this material. Thus, we can conclude that in terms of scaffolding of knowledge, the documentation provided was sufficient. Careful consideration was given to the integration of both scaffolds, and the documentation provided consisted of a series of webpages within which the content (including relevant code) was carefully hyperlinked to allow the students to trace between concept to code and code to concept. It was intended that the mapping between the situation and program models would be as straightforward as possible. One consideration when constructing the scaffolding was the amount of scaffolding of practice required, in other words, how much code should be provided and should the students be expected to understand all of the code? If pre-prepared code or even an

existing coding framework had been used, the scale of the problems to be solved would have been larger. Of course, an implicit disadvantage of providing significant amounts of code to students is the need for them to learn how to use it, which does not necessarily promote problem solving and indeed may form a barrier to learning. Hence, the approach adopted eschewed the provision of an initial framework in favour of the development of the framework itself. Thus, the problem solving took the form of stages in development of the framework, which also had the benefit of integrating the understanding of the principles and concepts of its application into the problem solving exercises. On completion, the students were able to apply the framework to build applications that mirrored real-world practices. This approach of careful scaffolding of knowledge and practice, combined with a staged approach to building solutions to tackle larger problems was the basis of the Structured Problem Solving approach. One observation made very early in the application of this pedagogy was that students felt even more motivated when they were able to discover problems that needed to be solved themselves. Thus, where possible the scaffolding was designed to give them the opportunity to "see" potential future problems that would need to be addressed in order to make progress, implicitly building the desire for a solution. Sometimes this also gave the students the opportunity to attempt their own solution, before moving onto the next development stage where the problem was more formally covered.

In general, this pedagogical approach was successful in building student confidence and the survey results also show that a number of students (Q20:7.43) felt they benefited from solving the problems through the framework. However, the results also indicate that this was not a universal opinion (Q14:5.25), implying that for scaffolding of practice, the balance between supporting the individual student's needs while maintaining an appropriate level of challenge was more difficult than anticipated. One possible solution would be to design a more carefully constructed fading system that would be able to provide scaffolding of practice that is more tailored to the individual. At some point, students need to be given complete solutions to enable them to review their own solutions and to allow absent students to catch up. Although an obvious potential drawback, on balance, this can be countered by close monitoring of the students and gauging their motivation.

The benefits of structured problem solving are summarised in Table 9-2.

Benefit	Description
Increased motivation to solve problems and better student engagement	Although students can learn concepts and principles through a series of exercises by exploring each one individually, student motivation to solve those problems increases when the solutions contribute to a much larger outcome. For example, students gain a better understanding of MVC by building their own MVC framework and exploring the concepts in the process.
Increased opportunity for problem discovery	The process of construction in stages provides the means by which problems can be discovered. This discovery process encourages students to seek their own solutions more readily.
Larger scale problem solving (real-world challenges)	To solve larger problems, it is necessary to develop applications based on a typically large existing code base. Structured problem solving seeks to build a solution to a large problem by solving a number of smaller problems over a number of stages. This reflects real-world software engineering practice.
Increased problem solving skills	The students are engaged in solving a range of problems using particular languages and technologies.
Better understanding of principles and concepts associated with a software framework	Instead of building a series of applications using existing code, the students focus on solving problems that require knowledge of the core principles and concepts.
Scaffolded practice	A software framework must be built or selected which allows incremental evaluation of the principles or concepts at a suitably granular level. This approach is different to teaching an existing framework by discussing a concept and then providing a worked example demonstrating it, because the students engage in solving a problem that is crucial to understanding the principle. For example, in MVC they may be required to complete the code for the View class to create an example using a view. Building a View class gives the student a better understanding of why such a class is required and how to use it rather than simply creating a subclass from an existing View class. Alternatively, if an existing framework is being used, the student may be required to experiment with a number of methods to solve a specific problem related to all views e.g. effectively create a fake view class.
Increased student confidence	Solving problems at each stage builds confidence in the use of the scaffolded framework in implementing applications and gives students exposure to real-world software development.

Table 9-2 Summary of Benefits of Structured Problem Solving

9.1 Suggested Structure for Programming Content within a Computer Science Programme

Figure 9-1 illustrates the suggested overall structure of the programming content in a Computer Science degree. A study was conducted (Chapter 6) in which an initial short Computational Thinking course in programming was given to all students prior to the

commencement of their studies, with the intention of accelerating their initial learning and to enable prediction of any potentially weaker students that may require more support. The results showed that it was possible to predict performance, but in the process, also demonstrated that the course had no direct benefit. Therefore, such a course is only recommended as a means of identifying students that may require support. Hence, it precedes other academic activities as shown at the beginning of the first year in this figure. A drawback of POI is that it potentially limits the creative problem solving required by programmers: one objective of ACI was to avoid this problem by delaying problem solving until the students had a good appreciation of coding. It was also thought that by developing an appropriate programming problem solving course, it would be possible to gain the benefits of good programming and problem solving skills without resorting to fixed patterns. The results obtained at the end of the problem solving course do bear out these initial beliefs, but in hindsight the POI approach has the benefit of reducing the initial difficulties and provides more scaffolding for weaker students. Therefore, POI is shown in Figure 9-1 as sitting between and overlapping with ACI and problem solving. OOPs and software design patterns are shown in the second year, but in practice it is common for at least some OOPs concepts to be taught in the first year. Here, OOPs is placed in the second year in recognition of the higher abstractions it represents e.g. inheritance and polymorphism. Furthermore, while structures like classes can be used to write short simple code, they only become fundamentally important when the problems being solved become large enough to warrant data encapsulation. Similarly, software design patterns are a natural extension of the programming patterns encountered in POI, as they represent solutions to well-known problems in software engineering and consequently are also shown in the second year. Structured problem solving is shown in the third year, and assumes that the students have developed an appropriate level of programming ability for application level development.

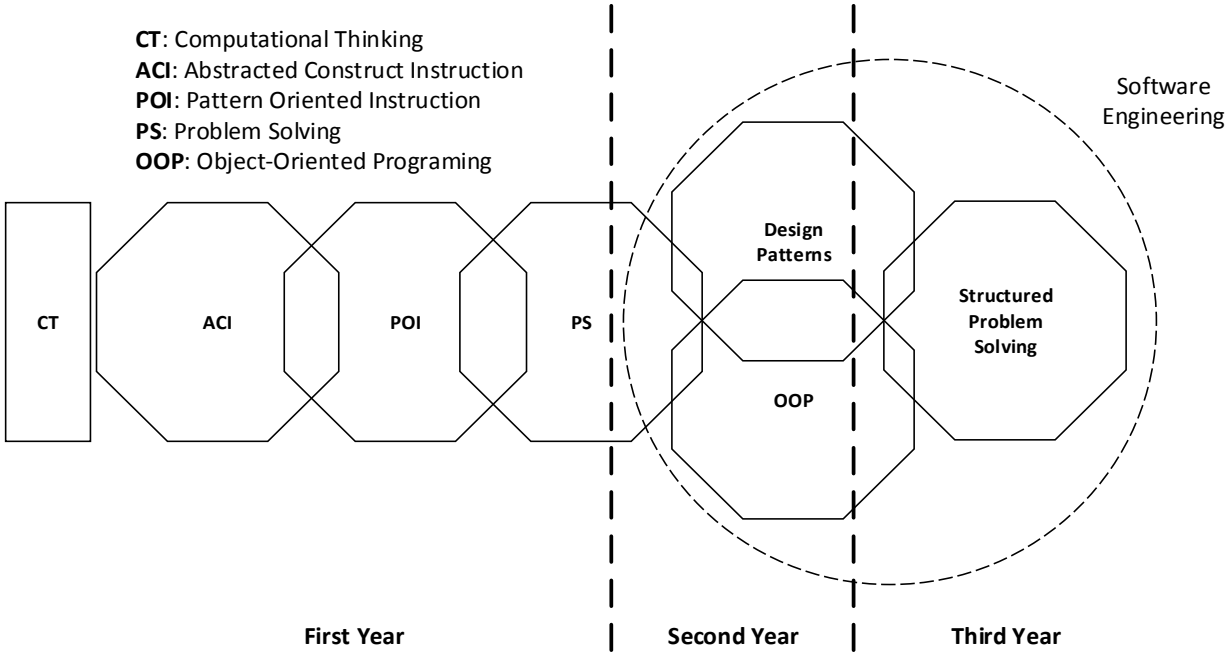


Figure 9-1 Suggested Overall Structure of Programming Content in a Computer Science Degree

10 Conclusions and Future Work

At the start of the research process, it was anticipated that there would be no single cause of novice programmer difficulties and no simple solution. Instead, the aim was to identify a number of causes and a number of approaches to alleviating these difficulties. Given the breadth and depth of the research available, the Grounded Theory approach was identified as the most appropriate research methodology around which the research process could be constructed. From this analysis, a number of significant factors emerged and were used to structure the literature review previously presented in this thesis. Primarily, the analysis demonstrated the importance of abstraction, cognitive load and problem solving.

A grounded action research mixed methodology was applied to the research. As a consequence of applying this methodology, two new pedagogical approaches were developed. Firstly Abstracted Construction Instruction pedagogy and secondly Structured Problem Solving for teaching more advanced problem solving. However, prior to these developments, a more traditional worksheet based teaching approach was used, with the purpose of encouraging continual practice. As part of the grounded theory investigations, a series of coding performance metrics were used to score each student across the worksheets with the objective of identifying any patterns of behaviour associated with categories of student grades. Market basket analysis was chosen for this analysis, but no significant pattern could be identified and none were found suitable for prediction of student performance. However, overall it was shown that problem solving was a key discriminator, confirming other research in the field as identified in the grounded theory analysis. In considering new programming pedagogies, problem solving is the key component that needs special attention. Little evidence could be found to suggest that promoting other characteristics such as enforcing a programming style, would improve programming ability.

The grounded theory analysis also suggested that a relationship between problem solving, fluid intelligence, working memory and programming may exist. To further investigate this relationship, tests were conducted using Raven Matrices to determine if any such relationship could be measured. A correlation was found, which for the first time provides an explanation for the relationship between problem solving and programming. Working memory provides a kind of mental notepad in which temporary

results are stored. Lower working memory capacity means that a novice programmer is able to process fewer ideas at the same time. As a result, such a novice is likely to find it more difficult to create a mental model and to map multiple elements from the problem definition to it. Furthermore, the bimodal distribution obtained also indicates that this inherent limitation is associated with weaker programmers.

The interesting conclusions from this result are that programming pedagogies must pay close attention to the role of cognitive psychology and the process of learning abstractions, and that lowering the cognitive load imposed will bring considerable benefits.

A further study is required to determine whether weaker programmers would be able to overcome working memory capacity limitations by adopting different strategies to reduce the demand on working memory. For example, by the simple expedient of making notes.

An important factor that contributes to cognitive load when developing code is the cross-referencing process between the code and the information available that defines the problem being solved. This load can be so considerable that even if the learner is able to find a solution to a problem, the effort required may lead them to forget the very abstract concept that they were intended to learn. To address this issue, the nature of “expertise” was considered. Clearly, expertise is also related to the amount of domain specific knowledge already possessed. However, beyond this, expertise is gained by memorising abstract patterns and being able to quickly recall them. This reduction in effort enables expert programmers to concentrate on extracting pertinent information and solving the problem at hand. A programming pedagogy should aim to accelerate the learning of fundamental patterns to enable the learner to mimic expert behaviour.

Furthermore, novices fixate on the concrete surface dissimilarities while experts concentrate on structural similarity. In the former, for example, a novice programmer may fail to see the underlying abstract principle required to solve a problem because the natural language used and/or the context are different to the original example in which that principle was introduced. Whereas in the latter, to be considered an expert, a programmer will have gained expertise by being previously being exposed to similar concepts and will have built up knowledge of the field in which the question is posed. This would include the general programming field, for example, understanding the implications of a “sort” or a “search”. Working memory determines the level of

distraction an individual can tolerate, multiple distractors in the surface features of problem can cause the underlying abstract principle to be missed. This implies that to ensure good levels of recall, the exercises provided should be brief and highly focused on the abstraction being taught.

ACI was developed around the core principle of teaching abstract patterns based on mental models used by programmers and minimising the cognitive load required. The aim of ACI is to encourage novice programmers to memorise the patterns so that they can be quickly recalled and applied. The cognitive load was reduced in two ways. Firstly by using terse exercises requiring minimum interpretation and secondly by recognising problem solving skills are vital but their teaching can be delayed to avoid distraction from the pattern being studied. Specific problem solving skills were taught separately to two focus groups: one of which was taught using the ACI approach and a comparison was made. The results were found to be comparable, indicating that the ACI group's problem solving ability was unaffected by this approach. ACI was demonstrated to be a very effective approach to supporting and developing programming ability, with all the students recognising the importance of memory and taking the correct approach to solving problems.

A secondary benefit of ACI, was the ability it afforded to micro manage the teaching of programming and to observe difficulties at a much more granular level. Observations following ACI instruction show that as well the natural language used in the problem definition causing problems, the students often fail to map the problem to their existing knowledge. Instead they attempt to solve the problem on the fly while coding. They spent little, if any, time planning the solution prior to coding and their strategy appeared to rely heavily on cross-referencing from the code back to the problem to see if the code "looked correct" in bottom-up manner. The phrase "looked correct" is used, since most students never tested their code during and sometimes not even after completing it. The general aim of their approach was that the solution would emerge as more and more code was developed. Of course, the main issue with this strategy was that sometimes the correct solution never emerged or that it would take significantly longer to emerge when multiple wrong decisions were made. This led to the students viewing their coding difficulties as code translation issues rather than problem solving issues. The correct approach was to solve the problem before attempting to code the solution, suggesting

that there is an intermediate level of knowledge and understanding which represents the solution to the problem. However, to obtain this solution the problem definition must be recontextualized into one that can be solved in code. In Pennington's [101] view, the situation model represents the information acquired from the problem definition and the intermediate level is represented by the plan structure knowledge within the program model. From this perspective, solving a problem involves mapping the pertinent information from the problem definition to the correct situation model and then mapping the situation model to the plan structure knowledge in order to create a programmable solution. In an attempt to teach this mapping process, a novel approach was used to aid the students in visualising the mapping of the problem definition to their existing knowledge with the intent of encouraging a solution first approach to coding. Although there was some evidence in the results to support this approach, most students found it difficult and typically the level of detail provided was far too vague. The students were also reluctant to apply it, even when they acknowledged its importance. Thus, the conclusion is that the mapping process is difficult and novice programmers prefer to perform this mapping process by writing the code and solving problems as and when they reach points from which they are not sure how to proceed.

One final observation from ACI is related to the complexity of the patterns. It was found that even when simple patterns (ACPs) were combined, the problems experienced by novice programmers became very significant due to the interrelationship between them. Where a multi-pattern required more than one new pattern to be learnt and applied simultaneously, as in the Array Counting Loop, these problems were magnified. A simple conclusion is that learning multiple patterns simultaneously is a significant barrier to learning. Separately, the array patterns are not difficult to understand or learn and were taught individually with a number of associated exercises. However, adding a counting loop to form an Array Counting Loop still caused students a great number of problems, in particular the interrelationship between count/index. The main conclusion is that when combining multiple patterns, the interrelationships between data and control flows causes considerable confusion in novice programmers. Further research is required into methods of improving novice understanding of these mixed patterns. If exercises require multiple ACPs, the exercises could be delineated by presenting them separately and perhaps by naming the specific ACPs required. Some combinations of ACPs could be presented as a new ACP, especially if they serve a particular purpose e.g. a search. In ACI,

it should never be assumed that novice programmers will automatically learn to nest ACPs in specific ways. In this respect, there is some overlap between ACI and POI, but teaching the use of constructs is still the aim.

10.1 Conclusions from Action Research

10.1.1 Teaching Using Worksheets

Initially, the main cause of student programming difficulties was considered to be related to lack of practice. Accordingly, programming concepts were subdivided across 6 worksheets, each containing a set of exercises that were to be submitted on a regular basis. Unfortunately, the results shown in Figure 5-1 and Figure 5-13 disappointingly demonstrated the same bimodal distribution common to many programming courses. In addition, the worksheets were used to obtain a dataset of results that could be analysed to determine potential indicators of success or failure. Although no such pattern could be determined, the results did show that problem solving was strongly associated with good programming ability. This result confirms the findings of a number of research studies. Therefore, the focus of the research switched to determining how problem solving skills could be developed and to what extent they may be inherent.

10.1.2 Accelerated Teaching of Computational Thinking

If problem solving skills are one of the key elements determining potential programming success, then to what extent are they inherent? Could providing an initial accelerated learning course prior to full-time study aid students by providing them with an opportunity to study fundamental concepts? Over a two year period, all first year computing students at UWTSD were required to complete a Computational Thinking course prior to the start of their normal studies. At the end of this course, they were assessed using a programming test and a Raven Matrices test to determine both their programming knowledge and their working memory capacity, this capacity being a good measure of problem solving ability. The results showed a correlation between the programming test and the final assignment marks obtained by the students. Thus, a disappointing conclusion that can be drawn from this correlation is that the accelerated learning process failed as a method for boosting initial learning. However, it did also demonstrate that it was possible to predict student performance prior to starting a programming course. Furthermore, it indicated an inherent component to programming ability and found that students with higher working memory capacity enjoyed an initial

advantage over other students. This conclusion reinforces the importance of problem solving skills in programming, but raises a number of questions related to addressing this weakness.

10.1.3 Abstracted Construct Instruction Pedagogy

The ACPs were reinforced by providing at least 3 exercises specifically designed to promote learning and the application of the abstraction. A core principle of the ACI pedagogy is that significant time must be devoted to encouraging the memorisation of the ACPs. In setting exercises, ACI encourages problem definitions that are concise, terse (almost bordering on “abstract”), repetitive and specifically targeted to promote recall. Results were obtained through testing, observation and interview, and it was found the emphasis on patterns and memorisation was beneficial.

Creating concise and terse definitions reduces surface dissimilarity and procedural comprehension difficulties that novice programmers often experience when reading natural language problem definitions. These difficulties were observed, and included miscategorising of values, the inability to identify the correct conditional operator and the number range selection problem.

In programming, problem solving requires a specific set of techniques which can be easily described but are difficult to master. ACI is not intended to develop problem solving skills, so approaches to developing these skills were also explored. For comparison, a focus group was also drawn from a cohort that had been taught using a more traditional worksheet approach. Both groups were tested before and after undergoing problem solving instruction. In the final test at the end of the academic year, the results for both groups were comparable indicating that the students given ACI instruction were not disadvantaged.

10.1.4 Structured Problem Based Learning Pedagogy

For more advanced final year degree students, the limited problems presented to students on the first year of a computing degree course provide insufficient challenge and do not prepare them for more real-world open problems encountered in industry. To investigate how these much larger problems could be presented to students, a structured problem based learning pedagogy was adopted. A cohort of final year undergraduate students were taught using a Javascript framework designed to explore a number of concepts associated with the development of AJAX enabled single page applications. It

became clear very early in the study that the students felt that there had to be a clear motivation for solving a problem, and an approach of gradually building a framework that could be used to create applications was a distinct advantage. In particular, this approach worked well when the next problem to be solved was "discovered" during completion of the previous exercise. Supporting teaching material was provided in the form of a series of webpages, and considerable effort was expended in ensuring hyperlinks were provided between all the significant elements in both the text and the presented code. These notes were also made available to the students online to enable them the view the exercises outside of class as well as the sample solutions. In providing a code framework, one issue identified was the quantity of the code that should be provided for scaffolding for practice. Some students preferred none at all, while some wanted to be provided with a complete solution that they could copy. By providing the correct scaffolding for knowledge and practice, the majority of students felt engaged and motivated. On completion of the course, the overall feedback from the students demonstrated the effectiveness of this approach in building confidence to develop their own learning skills and to adopting new technologies. Thus, this approach holds considerable promise for developing higher level student problem solving skills in programming courses, particularly if those courses are providing instruction on design patterns, algorithms or developing the core principles associated with a set of technologies and their applications.

10.2 Future Work

The small sample size used for analysis of ACI allowed a depth and variety of results to be obtained which would not have been possible in larger sample size. To consider further the effectiveness of this technique, a study should be conducted using a larger body of novice programmers with an evaluation of the benefits from the teacher's perspective. By allowing problems of novice programmers to be viewed at a very granular level, ACI affords the teacher the opportunity to intervene at a much earlier stage when programming difficulties begin to emerge. It would interesting to identify the type and nature of these interventions. It is anticipated that the type, structure and the nature of the abstract patterns may evolve as more is learnt about the difficulties novices experience using and combining them. More fundamentally, there exist many programming languages and some present more challenges than others. For example, the use of pointers in C and C++ can be a source of great confusion. From a teaching

perspective, analysing the interaction between teacher and learner, could yield better cognitive explanations of the difficulties faced by the novice programmer.

To ensure the findings are balanced, a cross institutional study should be conducted to evaluate the effectiveness of the approach applied in different programmes and institutions. It is likely that staff within the same institution have identified and developed similar strategies and viewpoints. Furthermore, the student profile and cohort may vary across institutions. A new study involving two or three institutions would enable an evaluation of the importance of these factors as well as further confirming the effectiveness of the approach.

The highly abstract nature of object oriented programming meant that the syntax, concepts and principles of this methodology were not addressed in the current version of ACI. Given that object oriented analysis, design and programming has become almost ubiquitous in the software development industry, novice programmers must be exposed to these concepts but only once they have acquired the necessary problem solving skills.

Further research is required into the best approach for applying ACI in developing the necessary OOP mental models. This would require the development of new ACPs, but would also need to take into consideration the difficulties students have in understanding fundamental concepts such as the difference between a class and an object. The difficulty here is not just related to retaining knowledge of the mental model but appreciating the benefits of translating entities that might be found in the real world into the appropriate abstractions. This is analogous to defining a database table and creating tuples in the database itself. Development of the database tables implies an analysis and mapping process from real world information. Class development is a similar process but is further complicated by the introducing of constructors and methods. The abstract nature of object orientation means that the benefits of this process can be very unclear for novice programmers. For example, database tables are created to enable sets of data to be stored in multiple tuples. Likewise, classes allow multiple objects to be created representing multiple entities of the same type but they are also used for many other reasons including data encapsulation and separation of program logic. Therefore, careful consideration is required to ensure that both the mental model of the class as a data structure and the notional machine model learnt by novice programmers are correctly aligned and understood.

A further difficulty arises because classes are actually programmer defined variable types. This very powerful feature means that programmers can create, store and pass objects of their own types. However, novices must be supported to enable them to make this mental leap and fully appreciate the impact this has on their current mental models. Passing objects into methods is a specific example where this concept can be perceived as simple to the teacher but can potentially result in great difficulties for the learner. Clearly, even more challenging concepts such as polymorphism require a firm understanding of these OOP fundamentals. From a novice programmer's perspective, many of the characteristics and features of object orientation represent a considerably bigger challenge than creating a database table and inserting rows into a database.

As already alluded to, ACI allows a micro-management of the abstract patterns being studied allowing learner behaviour to be studied in more detail. Therefore, two studies should be conducted to evaluate these challenges from both the teacher and the learner standpoint. These studies should provide more detailed knowledge and understanding of the most effective approach to teaching object oriented programming and lead to further developments of the ACI pedagogy.

10.2.1 Further Considerations Suggested by Related Research

Learning to program can be seen as equivalent to learning a foreign language. First you learn how to construct words, then sentences from the words using the correct grammar, then paragraphs from sentences, in a gradual process that develops writing skills. In programming, you first learn keywords and constructs which is similar to building the vocabulary and the fundamental rules of grammar where each construct has both syntax and semantics. These are the ACP patterns that must be memorised. With the grammar learnt, you can begin to apply the rules to construct meaningful sentences. In ACI, the novice programmer solves a number of exercises that explore different ways of using the ACP patterns. In a foreign language, the first sentences learnt are simple, but the length and complexity increases as your vocabulary expands and your knowledge of the rules of grammar increases. Likewise, in ACI the ACP patterns begin to combine so a Counting Loop becomes an Array Counting Loop, complete with exercises to reinforce the "rules of grammar". Next we combine sentences to begin to tell a short story by forming paragraphs. In programming, we solve problems by combining sequences and nesting ACP patterns within procedures and functions. The equivalent in programming to writing an essay is to write longer programs by creating multiple functions, using function calls to

define the complete solution. Problem solving instruction provides the tools for identifying the problems to be solved, like the chapter headings in a book, and by calling these functions the main function should tell the story. It is in the construction of the paragraphs, or the production of a generic solution to a set of common problems that additional work is required. POI [27] is a pedagogical approach that espouses the principle of teaching patterns that can be combined to form more complex solutions. It may restrict the creative thinking process, focusing on the construction of solutions through a building block approach, but it supports the development of problem solving skills in weaker students.

Therefore, the next step in the research is to integrate POI as the stage between ACI and the more "open" problem solving instruction. It may also be possible to allow the novice programmer to develop the initial solutions to the "new" pattern themselves by specifying the ACPs required in the problem definition. This process builds upon ACI, allowing the novice to use creative thinking skills while potentially giving them the same named building block as POI. Since ACI promotes the view of functions as solutions, this should not prove too onerous a task. It would be wise to set a time limit and provide a suitable solution for students who fail to find one for themselves. Multiple exercises need to be provided to explore various uses of the new pattern with the aim of improving memory recall. The merging of the ACI and POI stages needs further investigation, since there is some overlap but one should also build upon the other.

From the results obtained from the research into ACI, there is a suggestion of a contradiction between the cause of novice programmer difficulties and their beliefs and attitudes towards solving those issues. While the root cause of their difficulties is actually their inability to solve a problem, their implicit belief is that it is a code translation issue. The probable cause of this contradiction, is the existence of an intermediate stage that exists between "solving" a problem and solving it in such a way that it can be programmed. Pennington [101] identifies this division in the program model, which is separated into text structure knowledge representing the translation stage and plan structure knowledge representing the intermediate stage. Understanding the problem is, of course, also related to the Pennington [104] situation model which represents the extraction and mapping of relevant detail from the problem. Thus, "solving" a programming problem requires the programmer to construct both a situation model and

an associated plan structure knowledge. Psychologically, novice programmers appear to believe that they already have a perfect model of the problem as evidenced by their reluctance to engage in an explicit mapping process from problem space to their domain knowledge. In fact, they have created a simplistic situation model that prevents them drawing the appropriate inferences from which to construct the required plan structure knowledge. Support for this hypothesis can be found in research that contrasts the performance of experts and novices [150]. One robust finding from this research was that "*experts can sort problems into categories according to features in the solution, whereas novices can only sort problems using features in the problem statement itself*" [150]. For example, an intermediate level of planning might occur when a programmer is faced with a problem that asks them to "display the top 10 rated products". Experienced programmers might divide this into 3 separate problems "the products are stored on a list because we do not know how many there are", "the products must be sorted by their ratings" and "10 items must be displayed". These new sub-problems are still natural language but contain within them programming knowledge and cues such as "list" and "sort". This concept is related to "*information scent*" [126]. In intermediate planning the problem is not solved, for example, what is the "product" and how do the list and sort work together? As Green et al note [376]:

> "*Semantic knowledge is required for solving a problem but not for coding the solution in the specified language*"

Crucially, the original natural language problem has been re-contextualized into a set of problems that can be solved by a program. Although some testing was carried out, the full extent of this mapping process needs further exploration.

Another area where further research is required is the presentation of exercises to provide interleaved practice [377]. In block practice, students study problems of one type before moving on to the next topic. In interleaved practice, students alternate their practice between different types of problems. There is significant evidence that although students perform worse during practice, this is reversed when students whose practice was interleaved are subsequently tested [378, 379]. A plausible explanation is that the simultaneous exposure to multiple problem types helps students to discriminate between them by allowing them to be more readily compared i.e. the solution to the previous problem is already in working memory to allow the comparison to be made. Other evidence [380] suggests that interleaved practice is most beneficial when the student has

a certain level of ability achieved through block practice. In short, block practice followed by interleaved practice does not detract from the benefits of interleaved practice alone. Thus, when a new topic is introduced, it should be followed by block practice and then by an extra practice session that interleaves problems from previous classes [377]. Interleaved practice has been found to be ineffective in some studies, such as learning French vocabulary [381]. However, for ACI and POI, incorporating interleaved practice should not be a significant undertaking and may offer significant benefits in problem solving.

The limited size of the focus groups enabled closer observation of the participants and allowed the delivery and content of the course to be adjusted with minimum disruption. However, it will be necessary to expand this to a trial using the full student cohort to evaluate its effectiveness in a larger group. This should include different teaching staff and different programming languages to eliminate any potential undesirable extraneous influences, such as the ability of the teacher to inspire and motivate students.

In investigating problem solving in larger scale problems, one issue identified in applying a structured problem solving approach was that the scaffolding provided should have been implemented to allow for fading and the level of fading required deserves further study. The mixed results from the survey demonstrate that some of the benefits of the approach may have been lost because the scaffolding was not sufficiently personalised to the individual student. Obviously, this also has an impact on the interaction between the scaffolding of knowledge and practice which also deserves further consideration.

References

1. Sheard, J., et al., *Analysis of research into the teaching and learning of programming*, in *Proceedings of the fifth international workshop on Computing education research workshop*. 2009, ACM: Berkeley, CA, USA. p. 93-104.
2. Strubing, J., *Research as Pragmatic Problem-solving: The Pragmatist Roots of Empirically-grounded Theorizing*, in *The SAGE Handbook of Grounded Theory*, K.C. Antony Bryant, Editor. 2007, SAGE Publications Ltd.
3. Birks, M. and J. Mills, *Grounded Theory: A Practical Guide*. 2011: SAGE Publications.
4. Lister, R. and J. Leaney, *Introductory programming, criterion-referencing, and bloom.* SIGCSE Bull., 2003. **35**(1): p. 143-147.
5. McCracken, M., et al., *A multi-national, multi-institutional study of assessment of programming skills of first-year CS students*, in *Working group reports from ITiCSE on Innovation and technology in computer science education*. 2001, ACM: Canterbury, UK. p. 125-180.
6. Lister, R., *Ten years after the McCracken Working Group.* ACM Inroads, 2011. **2**(4): p. 18-19.
7. Lui, A.K., et al., *Saving weak programming students: applying constructivism in a first programming course.* SIGCSE Bull., 2004. **36**(2): p. 72-76.
8. Nyugen, D., Wong, S., *OOP in Introductory CS: Better Students through Abstraction*, in *OOPSLA'01*. 2001: Tampa, Florida, USA.
9. Or-Bach, R. and I. Lavy, *Cognitive activities of abstraction in object orientation: an empirical study.* SIGCSE Bull., 2004. **36**(2): p. 82-86.
10. Craik, K.J.W., *The Nature of Explanation*. 1967: Cambridge University Press.
11. Baddeley, A.D., Hitch, G.J., *Working Memory*, in *The psychology of learning and motivation. Recent Advances in Learning and Motivation*, G.H. Bower, Editor. 1974, Academic Press: New York. p. 47-89.
12. Engle, R.W., Tuholski, S.W., Laughlin, J.E., Conway, A.R.A., *Working Memory, short term memory, and general fluid intelligence: A latent variable approach.* Journal of Experimental Psychology: General, 1999(128): p. 309-331.
13. Letovsky, S., *Cognitive processes in program comprehension*, in *Papers presented at the first workshop on empirical studies of programmers on Empirical studies of programmers*. 1986, Ablex Publishing Corp.: Washington, D.C., United States. p. 58-79.
14. Frederick P. Brooks, J., *No Silver Bullet Essence and Accidents of Software Engineering.* Computer, 1987. **20**(4): p. 10-19.
15. Glaser, B.G., and Strauss, A., *Discovery of grounded theory.* Strategies for qualitative research Sociology Press, 1967.
16. Avison, D.E., et al., *Action research.* Commun. ACM, 1999. **42**(1): p. 94-97.
17. Klingberg, T., *The Overflowing Brain: Information Overload and the Limits of Working Memory*. 2008: Oxford University Press, USA.
18. Soloway, E., Adelson, B., Ehrlich, K., *Knowledge and Processes in the Comprehension of Computer Programs.* The Nature of Expertise, 1988: p. 129-152.
19. Rist, R.S., *Knowledge creation and retrieval in program design: a comparison of novice and intermediate student programmers.* Hum.-Comput. Interact., 1991. **6**(1): p. 1-46.
20. Dijkstra, E.W., *The humble programmer.* Commun. ACM, 1972. **15**(10): p. 859-866.
21. Sprague, P. and C. Schahczenski, *Abstraction the key to CS1.* J. Comput. Small Coll., 2002. **17**(3): p. 211-218.
22. Ben-Ari, M., *Constructivism in computer science education.* SIGCSE Bull., 1998. **30**(1): p. 257-261.

23. Adelson, B., *Problem solving and the development of abstract categories in programming languages.* Memory & Cognition, 1981. **9**(4): p. 422-433.
24. Holyoak, K.J., Morrison, R.G., *The Oxford Handbook of Thinking and Reasoning*. 2012, Oxford University Press. p. 413-432.
25. Corritore, C.L. and S. Wiedenbeck, *An exploratory study of program comprehension strategies of procedural and object-oriented programmers.* Int. J. Hum.-Comput. Stud., 2001. **54**(1): p. 1-23.
26. Koppelman, H. and B.v. Dijk, *Teaching abstraction in introductory courses*, in *Proceedings of the fifteenth annual conference on Innovation and technology in computer science education*. 2010, ACM: Bilkent, Ankara, Turkey. p. 174-178.
27. Haberman, B. and O. Muller. *Teaching abstraction to novices: Pattern-based and ADT-based problem-solving processes*. in *2008 38th Annual Frontiers in Education Conference*. 2008.
28. Muller, O., *Pattern oriented instruction and the enhancement of analogical reasoning*, in *Proceedings of the first international workshop on Computing education research*. 2005, ACM: Seattle, WA, USA. p. 57-67.
29. B.P.Hogan, *Exercises for Programmers: 57 Challenges to Develop Your Coding Skills* 2015: Pragmatic Bookshelf.
30. Mayer, R.E., *The Psychology of How Novices Learn Computer Programming.* ACM Comput. Surv., 1981. **13**(1): p. 121-141.
31. Perkins, D.N. and F. Martin. *Fragile knowledge and neglected strategies in novice programmers*. in *Papers presented at the first workshop on empirical studies of programmers on Empirical studies of programmers*. 1986. Ablex Publishing Corp.
32. Ebrahimi, A., *Novice programmer errors: language constructs and plan composition.* International Journal of Human-Computer Studies, 1994. **41**(4): p. 457-480.
33. Deek, F., Kimmel, H., McHugh, J. , *Pedagogical Changes in the Delivery of the First-Course in Computer Science: Problem Solving, then Programming.* Journal of Engineering Education, 1998. **87**(3): p. 313--320.
34. Deek, F.P., Hiltz, S.R., Kimmel, H., Rotter, N., *Cognitive assessment of students' problem solving and program development skills.* JOURNAL OF ENGINEERING EDUCATION-WASHINGTON-, 1999. **88**: p. 317-326.
35. Mostrom, J.E., et al., *Concrete examples of abstraction as manifested in students' transformative experiences*, in *Proceedings of the Fourth international Workshop on Computing Education Research*. 2008, ACM: Sydney, Australia. p. 125-136.
36. Inhelder, B., Piaget, J., *The growth of logical thinking from childhood to adolescence: .* 1958, London: Routledge and Kegan Paul.
37. Gick, M.L., Holyoak, K.J., *Analogical problem solving.* Cognitive Psychology, 1980. **12**: p. 306--355.
38. Bennedsen, J. and M.E. Caspersen, *Abstraction ability as an indicator of success for learning object-oriented programming?* SIGCSE Bull., 2006. **38**(2): p. 39-43.
39. Bennedsen, J. and M.E. Caspersen, *Abstraction ability as an indicator of success for learning computing science?*, in *Proceedings of the Fourth international Workshop on Computing Education Research*. 2008, ACM: Sydney, Australia. p. 15-26.
40. Baddeley, A., Eysenck, M.W., Anderson, M.C., *Memory*. 2009: Psychology Press.
41. Neisser, U., *Cognitive Psychology*. 1967: New York: Appleton-Century Crofts.
42. Anderson, L.W., Krathwohl D.R., Airasian, P.W., Cruikshank, K.A., Mayer, R.E., Pintrich, P.R., Raths, J., Wittrock, M.C., ed. *A Taxonomy for Learning and Teaching and Assessing: A revision of Bloom's taxonomy of educational objectives*. 2001, Addison Wesley Longman.
43. Engle, R.W., *Working memory and retrieval: An inhibition-resource approach.*, in *Working Memory and Human Cognition*, J.T.E. Richardson, Engle, R.W., Hasher, L., Logie, R.H., Stoltfus, E.R., Zacks, R.T., Editor. 1996, Oxford University Press: New York. p. 89-119.
44. Kitamura, T., et al., *Engrams and circuits crucial for systems consolidation of a memory.* Science, 2017. **356**(6333): p. 73-78.

45. Miller, G.A., *The magical number seven, plus or minus two: some limits on our capacity for processing information.* Psychological Review, Vol. 63, 1956. **63**(2): p. 81-97.
46. Miller, G.A., *The magical number seven, plus or minus two: Some limits on our capacity for processing information.* Psychological Review, 1956(63): p. 81-97.
47. Gobet, F., et al., *Chunking mechanisms in human learning.* Trends in Cognitive Sciences, 2001. **5**(6): p. 236-243.
48. Ryan, J., *Temporal grouping, rehearsal and short-term memory.* Quartely Journal of Experimental Psychology, 1969. **21**: p. 148-155.
49. Blinman, S. and A. Cockburn, *Program comprehension: investigating the effects of naming style and documentation*, in *Proceedings of the Sixth Australasian conference on User interface - Volume 40*. 2005, Australian Computer Society, Inc.: Newcastle, Australia. p. 73-78.
50. Norman, D.A. and T. Shallice, *Attention to Action: Willed and Automatic Control of Behavior*. 2002.
51. Clark, R.C., *Building Expertise: Cognitive Methods for Training and Performance Improvement*. 2008: John Wiley & Sons.
52. Hidi, S.E., *A reexamination of the role of attention in learning from text.* Educational Psychology Review, 1995. **7**(4): p. 323.
53. Corbetta, M., Shulman, G. L., *Control of Goal-Directed and Stimulus-Driven Attention in the Brain.* Nature Reviews Neuroscience, 2002. **3**(3): p. 201-215.
54. Lavie, N., Hirst, A., Fockert, Jan W. de, Viding, E., *Load Theory of Selective Attention and Cognitive Control.* Journal of Experimental Psychology General, 2004. **133**(3): p. 339-354.
55. Kane, M.J., Brown, L.H., McVay, J.C., Silvia, P.J, Myin-Germeys, I., Kwapil, T.R., *For Whom the Mind Wanders, and When: An Experience-Sampling Study of Working Memory and Executive Control in Daily Life.* Psychological Science, 2008. **18**(7).
56. Vogel, E.K., A.W. McCollough, and M.G. Machizawa, *Neural measures reveal individual differences in controlling access to working memory.* Nature, 2005. **438**(7067): p. 500-503.
57. Kane M.J., B., L.H., McVay, J.C., Silvia, P.J., Myin-Germeys, I, Kwapil, T.R., *For whom the mind wanders, and when: an experience-sampling study of working memory and executive control in daily life.* Psycholigical Science, 2007. **18**(7): p. 614-21.
58. Kane, M.J. and R.W. Engle, *The role of prefrontal cortex in working-memory capacity, executive attention, and general fluid intelligence: an individual-differences perspective.* Psychon Bull Rev, 2002. **9**(4): p. 637-71.
59. Kirschner, P.A., J. Sweller, and R.E. Clark, *Why Minimal Guidance During Instruction Does Not Work: An Analysis of the Failure of Constructivist, Discovery, Problem-Based, Experiential, and Inquiry-Based Teaching.* Educational Psychologist, 2006. **41**(2): p. 75-86.
60. Kyllonen, P.C., Christal, R.E., *Reasoning ability is (little more than) working memory capacity.* Intelligence, 1990(14): p. 389-433.
61. Engle, R.W., M.J. Kane, and S.W. Tuholski, *Individual differences in working memory capacity and what they tell us about controlled attention, general fluid intelligence and functions of the prefrontal cortex*, in *Models of working memory: Mechanisms of active maintenance and executive control*, A. Miyake, Shah, P., Editor. 1999, Cambridge University Press. p. 102-134.
62. Conway, A.R.A., Cowan, N., Bunting, M.F., Therriault, D.J., Minkoff, S.R. B., *A Latent Variable Analysis of Working Memory Capacity, Short-Term Memory Capacity, Processing Speed, and General Fluid Intelligence.* Intelligence, 2002. **30**(2): p. 163-83.
63. Halford, G.S., Cowan, N., Andrews, G., *Separating Cognitive Capacity from Knowledge: A New Hypothesis.* Trends in Cognitivie Science, 2007. **11**(6): p. 236–242.
64. Jaeggi, S.M., Buschkuehl, M., Jonides, J., Perrig, W.J., *Improving fluid intelligence with training on working memory.* Proceedings of the Natural Academy of Sciences, 2008. **105**(19): p. 6829–6833.

65. Su¨ß, H.-M., Oberauer, K., Wittmann, W.W., Wilhelm, O., Schulze, R., *Working-memory capacity explains reasoning ability—and a little bit more.* Intelligence, 2002. **30**(3): p. 261–288.
66. Cattell, R.B., *The measurement of adult intelligence.* Psychological Bulletin, 1943. **40**(3): p. 153-193.
67. Schraw, G. and J. Nietfeld, *A further test of the general monitoring skill hypothesis.* Journal of Educational Psychology, 1998. **90**(2): p. 236-248.
68. Prabhakaran, V., Smith, J.A.L., Desmond, J.E., Glover, G.H., Gabrieli, J.D.E., *Neural substrates of fluid reasoning: an fMRI study of neocortical activation during performance of the Raven's Progressive Matrices Test.* Cognitive Psychology, 1997. **33**(1): p. 43-63.
69. Spearman, C., *"General Intelligence", Objectively Determined and Measured.* American Journal of Psychology, 1904. **15**: p. 201-293.
70. Carpenter, P.A., Just, M.A., Shell, P., *What one intelligence test measures: a theoretical account of the processing in the Raven Progressive Matrices Test.* Psychological Review, 1990. **97**(3): p. 404-31.
71. Billing, D., *Teaching for transfer of core/key skills in higher education: Cognitive skills.* Higher Education, 2007. **53**(4): p. 483-516.
72. Papert, S.A., *Mindstorms: Children, Computers, And Powerful Ideas*. 1993: Basic Books.
73. Salomon, G., D.N. Perkins, and N.I.o. Education, *Transfer of Cognitive Skills from Programming: When and How?* 1985: National Inst. of Education.
74. Perkins, D.N. and G. Salomon, *Teaching for Transfer.* Educational Leadership, 1988. **46**(1): p. 22.
75. Mayer, R.E., J.L. Dyck, and W. Vilberg, *Learning to program and learning to think: what's the connection?* Commun. ACM, 1986. **29**(7): p. 605-610.
76. Shute, V.J., *Who is likely to acquire programming skills?* Journal of Educational Computing Research, 1991. **7**(1): p. 1-24.
77. Kyllonen, P.C., Stephens, D.L., *Cognitive abilities as the determinants of success in acquiring logic skills.* Learning and Individual Differences, 1990. **2**(2): p. 129-160.
78. Gellenbeck, E.M., Cook, C.R., *An Investigation of Procedure and Varialble Names as Beacons during Program Comprehension*, in *Empirical Studies of Programmers: Fourth Workshop*, J. Koenemann-Belliveau, T.G. Moher, and S.P. Robertson, Editors. 1991, Ablex Publishing Corporation. p. 65-81.
79. Tulving, E., *Episodic Memory: From Mind to Brain.* Annual Review of Psychology, 2002. **53**: p. 1-25.
80. Altmann, E.M., *Near-term memory in programminga simulation-based analysis.* International Journal of Human-Computer Studies, 2001. **54** p. 189-210.
81. Baddeley, A., *The episodic buffer: a new component of working memory?* Trends in Cognitive Sciences, 2000. **4**(11): p. 417-423.
82. Douce, C. *Long Term Comprehension of Software Systems: A Methodology for Study*. in *13th Workshop of the Psychology of Programming Interest GroupProc.* . 2001. Bournemouth, UK: Psychology of Programming Interest Group.
83. Collins, A.M., Loftus, E., *A spreading activation theory of semantic memory.* Psychological Review, 1975(82): p. 407-428.
84. Anderson, J.R., *ACT: A simple theory of complex cognition.* American Psychologist, 1996. **51**: p. 355-365.
85. Ritter, F.E. and J.W. Kim. *ACT-R FAQ*. 2012 [cited 2012 12/10/2012]; Available from: http://acs.ist.psu.edu/act-r-faq/act-r-faq.html.
86. Anderson, J.R., R. Farrell, and R. Sauers, *Learning to Program in LISP.* Cognitive Science, 1984. **8**(2): p. 87-129.
87. Anderson, J.R., *Acquisition of cognitive skill.* Psychological Review, 1982. **89**(4): p. 369-406.

88. Anderson, J.R., Fincham, J. M., Douglass, S., *The role of examples and rules in the acquisition of a cognitive skill.* Journal of experimental psychology. Learning, memory, and cognition, 1997. **23**(4): p. 932-945.
89. Cheng, P.W. and K.J. Holyoak, *Pragmatic reasoning schemas.* Cognitive Psychology, 1985. **17**(4): p. 391-416.
90. Cheng, P.W., et al., *Pragmatic versus syntactic approaches to training deductive reasoning.* Cognitive Psychology, 1986. **18**(3): p. 293-328.
91. Kolodner, J.L., *Towards an understanding of the role of experience in the evolution from novice to expert.* International Journal of Man-Machine Studies, 1983. **19**(5): p. 497-518.
92. Gilmore, D., *Expert programming knowledge: a strategic approach*, in *Psychology of Programming*, J. Hoc, Green, T., Samurcay, R., Gilmore, D., Editor. 1990, Academic Press: London. p. 223-234.
93. Martin A. Conway, M.A., Cohen, G., Stanhope, N., *Very long-term memory for knowledge acquired at school and university.* Applied Cognitive Psychology, 1992. **6**(6): p. 467-482.
94. Douce, C. *The Stores Model of Code Cognition*. in *In: Psychology of Programming Interest Group*. 2008. Lancaster University, UK.
95. Tulving, E., Thomson, D.M., *Encoding specificity and retrieval processes in episodic memory.* Psychological Review, 1973. **80**: p. 352-373.
96. Schneider, W., Shiffrin, R.M., *Controlled and automatic human information processing: I. Detection, search, and attention.* Psychological Review, 1977. **84**(1): p. 1-66.
97. Traxler, M.J., M.D. Bybee, and M.J. Pickering, *Influence of Connectives on Language Comprehension: Eye tracking Evidence for Incremental Interpretation.* The Quarterly Journal of Experimental Psychology Section A, 1997. **50**(3): p. 481-497.
98. McKeithen, K.B., et al., *Knowledge organization and skill differences in computer programmers.* Cognitive Psychology, 1981. **13**(3): p. 307-325.
99. Guerin, B. and A. Matthews, *The Effects of Semantic Complexity on Expert and Novice Computer Program Recall and Comprehension.* The Journal of General Psychology, 1990. **117**(4): p. 379-389.
100. Widowski, D. *Reading, comprehending and recalling computer programs as a function of expertise*. in *Proceedings of CERCLE Workshop on Complex Learning*. 1987.
101. Pennington, N., *Stimulus structures and mental representations in expert comprehension of computer programs.* Cognitive Psychology, 1987. **19**(3): p. 295-341.
102. Simon, H.A., *Problem Solving and Education: Issues in Teaching and Research*, D.T. Tuma, Reif, F., Editor. 1980, John Wiley & Sons Inc.
103. Wiedenbeck, S. and N.J. Evans, *Beacons in Program Comprehension.* SIGCHI Bull., 1986. **18**(2): p. 56-57.
104. Von Mayrhauser, A. and A.M. Vans, *Program comprehension during software maintenance and evolution.* Computer, 1995. **28**(8): p. 44-55.
105. Landauer, T.K., Bjork, R.A., *Optimum rehearsal patterns and name learning*, in *Practical Aspects of Memory*, M.M. Gruneberg, Morris, P.E., Sykes, R.N., Editor. 1978, Academic Press: London. p. 625-632.
106. Catrambone, R., *The subgoal learning model: Creating better examples so that students can solve novel problems.* Journal of Experimental Psychology: General, 1998. **127**(4): p. 355-376.
107. Kellman, P.J. and P. Garrigan, *Perceptual learning and human expertise.* Physics of Life Reviews, 2009. **6**(2): p. 53-84.
108. Kellman, P., C. Massey, and J. Son, *Perceptual Learning Modules in Mathematics: Enhancing Students' Pattern Recognition, Structure Extraction, and Fluency.* Topics in Cognitive Science, 2010. **2**(2): p. 285-305.
109. O'Brien, M.P., *Software Comprehension - A review and research direction*. 2003: University of Limerick, Ireland.
110. Brooks, R., *Towards a theory of the comprehension of computer programs.* International Journal of Man-Machine Studies, 1983. **18**(6): p. 543-554.

111. Shneiderman, B. and R. Mayer, *Syntactic/semantic interactions in programmer behavior: A model and experimental results.* International Journal of Parallel Programming, 1979. **8**(3): p. 219-238.
112. Soloway, E., K. Ehrlich, and J. Bonar, *Tapping into tacit programming knowledge*, in *Proceedings of the 1982 conference on Human factors in computing systems*. 1982, ACM: Gaithersburg, Maryland, United States. p. 52-57.
113. Dahl, O.J., E.W. Dijkstra, and C.A.R. Hoare, *Structured programming*. 1972: Academic Press.
114. Linger, R.C., H.D. Mills, and B.I. Witt, *Structured programming, theory and practice*. 1979: Addison-Wesley.
115. Von Mayrhauser, A. and A.M. Vans, *Program Understanding: Models and Experiments*, in *Advances in Computers*, C.Y. Marshall and Z. Marvin, Editors. 1995, Elsevier. p. 1-38.
116. Mayer, R.E., *A psychology of learning BASIC.* Commun. ACM, 1979. **22**(11): p. 589-593.
117. Spohrer, J.C. and E. Soloway, *Novice mistakes: are the folk wisdoms correct?* Commun. ACM, 1986. **29**(7): p. 624-632.
118. Dehnadi, S., Bornat, R., *The camel has two humps.* 2006.
119. Dehnadi, S., Bornat, R., Adams, R., *Meta-analysis of the effect of consistency on success in early learning of programming*, in *Psychology Programming Interested Group (PPIG) Annual workshop*. 2009.
120. Schulte, C., Clear, T., Taherkhani, A., Busjahn, T., Paterson, J.H., *An introduction to program comprehension for computer science educators*, in *In Proceedings of the 2010 ITiCSE working group reports on Innovation and technology in computer science education - ITiCSE-WGR '10*, N. ACM New York, USA, Editor. 2010: Bilkent, Ankara, Turkey. p. 65–86.
121. Rist, R.S., *Schema creation in programming.* Cognitive Science 1989. **13**(3): p. 389-414.
122. Rist, R.S., *Learning to Program: Schema Creation, Application, and Evaluation*, in *Computer Science Education Research*, S. Fincher and M. Petre, Editors. 2004, Taylor & Francis: Lisse, The Netherlands. p. 175-195.
123. Rist, R.S., *Modeling object-oriented design*, in *Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*. 2005, ACM: San Diego, CA, USA. p. 344-349.
124. Widowski, D., Eyferth, K. *Representation of computer programs in memory*. in *Third European Conference on Cognitive Ergonomics*. 1986. Paris, France.
125. Ko, A.J., et al., *An Exploratory Study of How Developers Seek, Relate, and Collect Relevant Information during Software Maintenance Tasks.* IEEE Trans. Softw. Eng., 2006. **32**(12): p. 971-987.
126. Pirolli, P., Card, S.K., *Information Foraging.* Psychological Review, 1999. **106**(4): p. 643-675.
127. Soloway, E. and K. Ehrlich, *Empirical studies of programming knowledge*, in *Software reusability*. 1989, ACM. p. 235-267.
128. Jeffries, R., *A comparison of the debugging behavior of expert and novice programmers*, in *Annual Meeting of the American Educational Research Association*. 1982: New York.
129. Nanja, M. and C.R. Cook, *An analysis of the on-line debugging process*, in *Empirical studies of programmers: second workshop*, M.O. Gary, S. Sylvia, and S. Elliot, Editors. 1987, Ablex Publishing Corp. p. 172-184.
130. Mosemann, R. and S. Wiedenbeck. *Navigation and comprehension of programs by novice programmers*. in *Proceedings 9th International Workshop on Program Comprehension. IWPC 2001*. 2001.
131. Pennington, N., *Comprehension strategies in programming*, in *Empirical studies of programmers: second workshop*, M.O. Gary, S. Sylvia, and S. Elliot, Editors. 1987, Ablex Publishing Corp. p. 100-113.
132. Détienne, F., *Software Design–Cognitive Aspect*. 2001: Springer Science & Business Media.

133. Green, T.R.G., Petre, M., *Usability Analysis of Visual Programming Environments: a `cognitive dimensions' framework.* Journal of Visual Languages and Computing, 1996. **7**: p. 131--174.
134. Sheard, J., et al., *Going SOLO to assess novice programmers.* SIGCSE Bull., 2008. **40**(3): p. 209-213.
135. Lopez, M., et al., *Relationships between reading, tracing and writing skills in introductory programming*, in *Proceedings of the Fourth international Workshop on Computing Education Research*. 2008, ACM: Sydney, Australia. p. 101-112.
136. Winslow, L.E., *Programming pedagogy-a psychological overview.* SIGCSE Bull., 1996. **28**(3): p. 17-22.
137. Lister, R., et al., *Not seeing the forest for the trees: novice programmers and the SOLO taxonomy.* SIGCSE Bull., 2006. **38**(3): p. 118-122.
138. Mannila, L., *Novices' Progress in Introductory Programming Courses.* Informatics in education, 2007. **6**(1): p. 139-152.
139. Soloway, E., *Learning to program = learning to construct mechanisms and explanations.* Commun. ACM, 1986. **29**(9): p. 850-858.
140. Mayrhauser, A.v. and A.M. Vans, *Comprehension processes during large scale maintenance*, in *Proceedings of the 16th international conference on Software engineering*. 1994, IEEE Computer Society Press: Sorrento, Italy. p. 39-48.
141. Sweller, J. and G.A. Cooper, *The Use of Worked Examples as a Substitute for Problem Solving in Learning Algebra.* Cognition and Instruction, 1985. **2**(1): p. 59-89.
142. Cooper, G. and J. Sweller, *Effects of schema acquisition and rule automation on mathematical problem-solving transfer.* Journal of Educational Psychology, 1987. **79**(4): p. 347-362.
143. Chase, W.G., Simon, H. A., *Perception in chess.* Cognitive Psychology, 1973. **4**: p. 55-81.
144. Adelson, B., *When novices surpass experts: The difficulty of a task may increase with expertise.* Journal of Experimental Psychology: Learning, Memory, and Cognition, 1984. **10**(3): p. 483-495.
145. Fix, V., S. Wiedenbeck, and J. Scholtz, *Mental representations of programs by novices and experts*, in *Proceedings of the INTERACT '93 and CHI '93 conference on Human factors in computing systems*. 1993, ACM: Amsterdam, The Netherlands. p. 74-79.
146. Schwonke, R., et al., *The worked-example effect: Not an artefact of lousy control conditions.* Computers in Human Behavior, 2009. **25**(2): p. 258-266.
147. van Gog, T., L. Kester, and F. Paas, *Effects of worked examples, example-problem, and problem-example pairs on novices' learning.* Contemporary Educational Psychology, 2011. **36**(3): p. 212-218.
148. van Gog, T., F. Paas, and J.J.G. van Merriënboer, *Effects of process-oriented worked examples on troubleshooting transfer performance.* Learning and Instruction, 2006. **16**(2): p. 154-164.
149. Schworm, S. and A. Renkl, *Computer-supported example-based learning: When instructional explanations reduce self-explanations.* Computers & Education, 2006. **46**(4): p. 426-445.
150. VanLehn, K., *Cognitive skill acquisition.* Annual Review of Psychology, 1996. **47**(1): p. 513.
151. Cooper, G., Sweller, J., *Effects of schema acquisition and rule automation on mathematical problem-solving transfer.* Journal of Educational Psychology, 1987. **79**(4): p. 347-362.
152. Chi, M.T., R. Glaser, and E. Rees, *Expertise in problem solving*, in *Advances in the psychology of human intelligence*, R. Sternberg, Editor. 1981, Lawrence Erlbaum Associates, Inc: Hillsdale, NJ. p. 7-75.
153. Paas, F., A. Renkl, and J. Sweller, *Cognitive Load Theory and Instructional Design: Recent Developments.* Educational Psychologist, 2003. **38**(1): p. 1-4.
154. Moreno, R., *When worked examples don't work: Is cognitive load theory at an Impasse?* Learning and Instruction, 2006. **16**(2): p. 170-181.

155. Gerjets, P., K. Scheiter, and R. Catrambone, *Designing Instructional Examples to Reduce Intrinsic Cognitive Load: Molar versus Modular Presentation of Solution Procedures.* Instructional Science, 2004. **32**(1-2): p. 33-58.
156. Chi, M.T., et al., *Self-explanations: How students study and use examples in learning to solve problems.* Cognitive Science, 1989. **13**(2): p. 145-182.
157. Kalyuga, S., et al., *The expertise reversal effect.* Educational Psychologist, 2003. **38**(1): p. 23-31.
158. Rey, G.D. and F. Buchwald, *The expertise reversal effect: cognitive load and motivational explanations.* J Exp Psychol Appl, 2011. **17**(1): p. 33-48.
159. Wood, D., Bruner, J.S., Ross, G., *The Role of Tutoring in Problem Solving.* Journal of Child Psychology and Psychiatry, 1976. **17**(2): p. 89-100.
160. Newell, A. and H.A. Simon, *Human problem solving*. 1972: Prentice-Hall.
161. Wertheimer, M., *Productive Thinking (Rev Ed)*. 1959, Chicago IL.: University of Chicago Press.
162. Schwill, A. *Cognitive aspects of object-oriented programming*. in *IFIP WG 3.1 Working Conference "Integrating Information Technology into Education*. 1994.
163. Dusink, L. and L. Latour, *Controlling functional fixedness: the essence of successful reuse.* Knowledge-Based Systems, 1996. **9**(2): p. 137-143.
164. McCaffrey, T. *Why We Can't See What's Right in Front of Us*. 2012 May 10, 2012 [cited 2012 August 9, 2012]; Available from: http://blogs.hbr.org/cs/2012/05/overcoming_functional_fixednes.html.
165. Forisek, M., Steinova, M., *Metaphors and analogies for teaching algorithms*, in *Proceedings of the 43rd ACM technical symposium on Computer Science Education*. 2012, ACM: Raleigh, North Carolina, USA. p. 15-20.
166. Brownell, W.A., Moser, H. E., *Meaningful vs. mechanical learning: A study in grade III subtraction* 1949, Durham, N.C: Duke Univ. Press.
167. Resnick, L.B., Ford, W., , *The Psychology of Mathematics for Instruction*. 1981, Hillsdale, NJ: Lawrence Erlbaum Associates
168. Du Boulay, B., T.I.M. O'Shea, and J. Monk, *The black box inside the glass box: presenting computing concepts to novices.* International Journal of Human-Computer Studies, 1999. **51**(2): p. 265-277.
169. Mayer, R.E., *Different Problem-Solving Competencies Established in Learning Computer Programming With and Without Meaningful Models.* Journal of Educational Psychology, 1975. **67**(6): p. 725-34.
170. Neeman, H., et al., *Analogies for teaching parallel computing to inexperienced programmers.* SIGCSE Bull., 2006. **38**(4): p. 64-67.
171. Doukakis D., G.M., Tsaganou G, *Understanding the Programming Variable Concept with Animated interactive Analogies*, in *Conference Proceedings of HERCMA 2007*. 2007: Athens. p. 74-75.
172. Doukakis D., T.G., Grigoriadou M., *Using animated interactive analogies in teaching basic programming concepts and structures*, in *An ACM Conference on the State of: Informatics Education Europe II*. 2007: Thessaloniki. p. 257-265.
173. Gick, M.L., Holyoak, K. J., *Schema induction and analogical transfer.* Cognitive Psychology, 1983. **15**: p. 1-38.
174. Holyoak, K.J. and K. Koh, *Surface and structural similarity in analogical transfer.* Memory & Cognition, 1987. **15**(4): p. 332-340.
175. Gentner, D., *Structure-Mapping: A Theoretical Framework for Analog.* Cognitive Science, 1983. **7**: p. 155-170.
176. Gentner, D. and C. Toupin, *Systematicity and surface similarity in the development of analogy*. 1985: University of Illinois, Center for the Study of Reading.
177. Spencer, R.M., & Weisberg. R. W., *Context-dependent effects on analogical transfer.* Memory & Cognition, 1986. **14**: p. 442-449.

178. Heydenbluth, C. and F.W. Hesse, *Impact of Superficial Similarity in the Application Phase of Analogical Problem Solving.* The American Journal of Psychology, 1996. **109**(1): p. 37-57.
179. Dunbar, K., *The Analogical Paradox: Why Analogy Is So Easy in the Naturalistic Settings, Yet So Difficult in the Psychological Laboratory*, D. Gentner, K.J. Holyoak, and B.N. Kokinov, Editors. 2001, The MIT Press. p. 199-253.
180. Chi, M.T.H., et al., *Self-explanations: How students study and use examples in learning to solve problems.* Cognitive Science, 1989. **13**(2): p. 145-182.
181. Reed, S.K., *A structure-mapping model for word problems.* Journal of Experimental Psychology: Learning, Memory, and Cognition, 1987. **13**(1): p. 124-139.
182. Bassok, M. and K. Olseth, *Judging a book by its cover: Interpretative effects of content on problem-solving transfer.* Memory & Cognition, 1995. **23**(3): p. 354-367.
183. Catrambone, R. and K.J. Holyoak, *Overcoming contextual limitations on problem-solving transfer.* Journal of Experimental Psychology: Learning, Memory, and Cognition, 1989. **15**(6): p. 1147.
184. Alexander, R., *Learning from Worked-Out Examples via Self-Explanations: How it Can(not) be Fostered*. 2007.
185. Gick, M.L., Holyoak, K.J., *The cognitive basis of knowledge transfer*, in *Transfer of training: Contemporary research and applications*, S.M. Cornier, Hagman, J.D., Editor. 1987, Academic Press: New York. p. 9-46.
186. Detterman, D.K., *The case for prosecution: Transfer as an epiphenomenon*, in *Transfer on Trial: Intelligence, Cognition, and Instruction*, D.K. Detterman, Sternberg, R.J., Editor. 1993, Ablex: Stamford, CT. p. 39–67.
187. Johnson, A.M.F., P.U. Curriculum, and Instruction, *The Beliefs and Practices of General Chemistry Students and Faculty Members Regarding Knowledge Transfer: A Phenomenographic Study*. 2007: Purdue University.
188. Grotzer, T.A. *Teaching Thinking Skills: Does it Add Up for Math and Science Learning?* 1996 [cited 2012 02/11/12]; Available from: http://www.pz.harvard.edu/Research/MathSciMatters/BK2THKSKRv03.pdf.
189. Bassok M., C., V.M., Martin, S.A., *Adding Apples and Oranges: Alignment of Semantic and Formal Knowledge.* Cognitive Psychology, 1998. **35**: p. 99–134.
190. Chrysikou, E.G., *When Shoes Become Hammers: Goal-Derived Categorization Training Enhances Problem-Solving Performance.* Journal of Experimental Psychology-learning Memory and Cognition, 2006. **32**(4): p. 935-942.
191. Chrysikou, E.G., *Your creative brain at work*, in *Mind*. 2012, Scientific American NY. p. 24-31.
192. Pane, J.F., C.A. Ratanamahatana, and B.A. Myers, *Studying the language and structure in non-programmers' solutions to programming problems.* International Journal of Human-Computer Studies, 2001. **54**(2): p. 237-264.
193. Hoc, J.-M., Nguyen-Xuan, A., *Language Semantics, Mental Models and Analogy*, in *Psychology of Programming.* , J.-M. Hoc, Green, T. R. G., Samurçay, R., Gilmore, D. J., Editor. 1990, Academic Press: London. p. 139-156.
194. Pea, R.D., *Language-Independent Conceptual "Bugs" in Novice Programming.* Journal of Educational Computing Research, 1986. **2**(1): p. 25 - 36.
195. Putnam, R., Sleeman, D., Baxter, J., Kuspa, L., *A Summary of Misconceptions of High School Basic Programmers.* Educational Computing Research, 1986. **2**(4): p. 459-472.
196. Putnam, R., et al., *A Summary of Misconceptions of High School Basic Programmers.* Educational Computing Research, 1986. **2**(4): p. 459-472.
197. Chen, T.-Y., et al., *Commonsense computing: using student sorting abilities to improve instruction.* SIGCSE Bull., 2007. **39**(1): p. 276-280.
198. Ma, L., et al., *Using cognitive conflict and visualisation to improve mental models held by novice programmers.* SIGCSE Bull., 2008. **40**(1): p. 342-346.

199. Johnson-Laird, P., *Models of deduction*, in *Reasoning: Representation and Process*, R. Falmagne, Editor. 1975, Erlbaum: Springdale, NJ.
200. Johnson-Laird, P., Steedman, M., *The psychology of syllogisms.* Cognitive Psychology, 1978. **10**: p. 64--99.
201. Bornat, R., S. Dehnadi, and Simon, *Mental models, consistency and programming aptitude*, in *Proceedings of the tenth conference on Australasian computing education - Volume 78*. 2008, Australian Computer Society, Inc.: Wollongong, NSW, Australia. p. 53-61.
202. Du Boulay, B., *Some Difficulties of Learning to Program.* Journal of Educational Computing Research 1986. **2**(1): p. 57 - 73.
203. Bonar, J.G., Cunningham, R., *Bridge: tutoring the programming process*, in *Intelligent Tutoring Systems: Lessons Learned*, J. Psotka, Massey, L.D., Mutter, S.A., Editor. 1988, Lawrence Erlbaum Associates: Hillsdale p. 409–434.
204. Bonar, J. and E. Soloway, *Preprogramming knowledge: a major source of misconceptions in novice programmers.* Hum.-Comput. Interact., 1985. **1**(2): p. 133-161.
205. McQuire, A.R. and C.M. Eastman, *The ambiguity of negation in natural language queries to information retrieval systems.* J. Am. Soc. Inf. Sci., 1998. **49**(8): p. 686-692.
206. VanDeGrift, T., et al., *Commonsense computing (episode 6): logic is harder than pie*, in *Proceedings of the 10th Koli Calling International Conference on Computing Education Research*. 2010, ACM: Koli, Finland. p. 76-85.
207. Epp, S.S., *The role of logic in teaching proof.* American Mathematical Monthly, 2003: p. 886-899.
208. Herman, G.L., et al., *Proof by incomplete enumeration and other logical misconceptions*, in *Proceedings of the Fourth international Workshop on Computing Education Research*. 2008, ACM: Sydney, Australia. p. 59-70.
209. Perkins, D., Schwartz, S., Simmons, R., *Instructional strategies for the problems of novice programmers*, in *Teaching and Learning Computer Programming*, R.E. Mayer, Editor. 1988, Lawrence Erlbaum Associates. p. 153-178.
210. Perkins, D.N., Farady, M.l; Bushey, B., *Everyday reasoning and the roots of intelligence.*, in *Informal reasoning and education.*, J.F. Voss, Perkins, D.N., Segal, J.W., Editor. 1991, Lawrence Erlbaum Associates: Hillsdale, NJ. p. 83-105.
211. Berendsen, Y.A., Krammer, H.P.M. , *Problem decomposition using programming plans.* Tijdschrift voor Didactiek der B-wetenschappen, 1992. **10**(3): p. 178-191.
212. Bloom, B.S., Englehart M.D, Hill, H.H., Furst, E.J., Krathwhol, D.R., *Taxonomy of educational objectives : the classification of educational goals. Handbook I, Cognitive domain*, B.S. Bloom, Editor. 1956, David McKay Company Inc: New York. p. 62-197.
213. Krathwohl, D.R., *A Revision of Bloom's Taxonomy: An Overview.* Theory Into Practice, 2002. **41**(4): p. 212-218.
214. Scott, T., *Bloom's taxonomy applied to testing in computer science classes.* J. Comput. Small Coll., 2003. **19**(1): p. 267-274.
215. Kanu E.O. Nkanginieme, M., FmcPaed, *Clinical Diagnosis as a Dynamic Cognitive Process: Application of Bloom's Taxonomy for Educational Objectives in the Cognitive Domain.* Med Educ Online, 1997.
216. Lahtinen, E. *A categorization of novice programmers: A cluster analysis study.* in *Proceedings of the 19th Annual Workshop of the Psychology of Programming Interest Group*. 2007. Joensuu, Finland.
217. Biggs, J.B., Collis, K. F., *Evaluating the Quality of Learning: The SOLO Taxonomy*. 1982, New York Academic Press.
218. Shuhidan, S., M. Hamilton, and D. D'Souza, *A taxonomic study of novice programming summative assessment*, in *Proceedings of the Eleventh Australasian Conference on Computing Education - Volume 95*. 2009, Australian Computer Society, Inc.: Wellington, New Zealand. p. 147-156.

219. Fuller, U., et al., *Developing a computer science-specific learning taxonomy.* SIGCSE Bull., 2007. **39**(4): p. 152-170.
220. Clear, T., Whalley, J.L., Lister, R., Carbone, A., Hu, M., Sheard, J., Simon, B., Thompson, E., *Reliably classifying novice programmer exam response using the SOLO taxonomy*, in *21s t Annual Conference of the National Advisory Committee on Computing Qualifications (NACCQ2008)*, S. Mann, Lopez, M., Editor. 2008: Auckland, New Zealand.
221. Burkhardt, J.-M., Detienne, F., Wiedenbeck, S., *Object-Oriented Program Comprehension: Effect of Expertise, Task and Phase.* Empirical Softw. Eng., 2002. **7**(2): p. 115-156.
222. Schulte, C., *Block Model: an educational model of program comprehension as a tool for a scholarly approach to teaching*, in *Proceedings of the Fourth international Workshop on Computing Education Research*. 2008, ACM: Sydney, Australia. p. 149-160.
223. Kintsch, W., *Comprehension: A Paradigm for Cognition*. 1998: Cambridge University Press.
224. Karagiorgi, Y., Symeou, L., *Translating constructivism into instructional design: Potential and limitations.* Educational Technology & Society, 2005. **8**(1): p. 17-27.
225. Greening, T., *Building the Constructivist Toolbox: An Exploration of Cognitive Technologies.* Educational Technology, 1998. **38**(2): p. 23-35.
226. Squires, D., *Educational Software for Constructivist Learning Environments: Subversive Use and Volatile Design.* Educational Technology, 1999. **39**(3): p. 48-54.
227. Brown, J.S., Collins, A., Duguid, P., *Situated Cognition and the Culture of Learning.* Educational researcher, 1989. **18**(1): p. 32-42.
228. Mordechai, B., *Constructivism in Computer Science Education 1.* Journal of Computers in Mathematics and Science Teaching, 2001. **20**(1): p. 45-73.
229. von Glasersfeld, E., *Questions and answers about radical constructivism*, in *Scope, Sequence, and Coordination of school science. Volume II. Relevant research*, M.K. Pearsall, Editor. 1992, National Science Teachers Association: Washington, DC. p. 169-182.
230. Richardson, S., *Cognitve Development to Adolescence*. 1987: Taylor & Francis.
231. Vrasidas, C., *Constructivism versus objectivism: Implications for interaction, course design, and evaluation in distance education.* International Journal of Educational Telecommunications, 2000. **6**(4): p. 339-362.
232. Cunningham, D., Duffy, T., *Constructivism: Implications for the design and delivery of instruction*, in *Handbook of research for educational communications and technology*, D.H. Jonassen, Editor. 1996, Simon and Schuster: New York. p. 170-198.
233. Jonassen, D.H., *Thinking Technology.* Educational Technology, 1994. **34**(4): p. 34-37.
234. Savery, J.R. and T.M. Duffy, *Problem Based Learning: An Instructional Model and Its Constructivist Framework.* Educational Technology, 1995. **35**(5): p. 31-38.
235. Perkins, D.N., *The Many Faces of Constructivism.* Educational Leadership, 1999. **57**(3): p. 6-11.
236. O'Donnell, A.M., *Constructivism by design and in practice: a review.* Issues in Education, 2000. **3**(2): p. 285-294.
237. Mayer, R.E., *Should there be a three-strikes rule against pure discovery learning? The case for guided methods of instruction.* Am Psychol, 2004. **59**(1): p. 14-9.
238. Kurland, D.M. and R.D. Pea, *Children's mental models of recursive Logo programs.* Journal of Educational Computing Research, 1985. **1**(2): p. 235-243.
239. Pea, R.D. and D.M. Kurland, *On the cognitive effects of learning computer programming*, in *Mirrors of minds: patterns of experience in educational computing*, D.P. Roy and S. Karen, Editors. 1987, Ablex Publishing Corp. p. 147-177.
240. Lee, M.O.C. and A. Thompson, *Guided Instruction in Logo Programming and the Development of Cognitive Monitoring Strategies Among College Students.* Journal of Educational Computing Research, 1997. **16**(2): p. 125-44.
241. Sweller, J., R.F. Mawer, and W. Howe, *Consequences of History-Cued and Means-End Strategies in Problem Solving.* The American Journal of Psychology, 1982. **95**(3): p. 455-483.

242. Sweller, J., *Instructional Design in Technical Areas. Australian Education Review, No. 43*. 1999: PCS Data Processing, Inc., 360 W. 31st, New York, NY 10001; Tel: 212-564-3730; Fax: 212-967-0928.
243. Revans, R.W., *Sketches in Action Learning.* Performance Improvement Quarterly, 1998. **11**(1): p. 23-27.
244. Peterson, R.L., *An action learning approach for the development of technology skills*, in *Proceedings of the 2000 information resources management association international conference on Challenges of information technology management in the 21st century*. 2000, IGI Publishing: Anchorage, Alaska, United States. p. 542-544.
245. Vat, K.H., *Training e-commerce support personnel for enterprises through action learning*, in *Proceedings of the 2000 ACM SIGCPR conference on Computer personnel research*. 2000, ACM: Chicago, Illinois, United States. p. 39-44.
246. Sherry, L., *A model computer simulation as an epistemic game.* SIGCSE Bull., 1995. **27**(2): p. 59-64.
247. Belland, B., Walker, A., Olsen, M. W., Leary, H., *Impact of Scaffolding Characteristics and Study Quality on Learner Outcomes in STEM Education: A meta-analysis*, in *Annual Meeting of th American Educational Research Association*. 2012: Vancouver, Canada.
248. Bruner, J., *The role of dialogue in language acquisition*, in *The child's conception of language*, A. A. Sinclair, Jarvella, R., Levelt, W. J. M., Editor. 1978, Springer-Verlag: Berlin, Germany. p. 241-256.
249. Vygotskiĭ, L.S., *Mind in society: the development of higher psychological processes*. 1978, Cambridge: Harvard University Press (Cole, M., Jon-Steiner, V., Scribner, S., Souberman, E. Original works published 1930-35).
250. Foley, J., *Key concepts in ELT: Scaffolding.* ELT Journal, 1994. **48**(1): p. 101-102.
251. Harel, I., Papert, S., *Software Design as a Learning Environment.* Interactive Learning Environments, 1990. **1**(1): p. 1-32.
252. Collins, A., *Cognitive apprenticeship and instructional technology.* Educational values and cognitive instruction: Implications for reform, 1991: p. 121-138.
253. Guzdial, M., *Software-Realized Scaffolding to Facilitate Programming for Science Learning.* Interactive Learning Environments, 1994. **4**(1): p. 1-44.
254. Yelland, N., Masters, J., *Rethinking scaffolding in the information age.* Comput. Educ., 2007. **48**(3): p. 362-382.
255. Applebee, A.N. and J.A. Langer, *Instructional Scaffolding: Reading and Writing as Natural Language Activities.* Language Arts, 1983. **60**(2): p. 168-75.
256. Applebee, A.N., *Problems in process approaches: Toward a reconceptualization of process instruction*, in *The teaching of writing: Eighty-fifth yearbook of the National Society for the Study of Education, Part II* A.R. Petrosky, Bartholomae, D., Editor. 1986, National Society for the Study of Education: Chicago. p. 95-113.
257. Glogowski, K. *Instructional Scaffolding*. 2007 [cited 2012 19/11/12]; Available from: http://www.teachandlearn.ca/blog/2007/07/30/instructional-scaffolding/.
258. Tharp, R.G. and R. Gallimore, *The Instructional Conversation: Teaching and Learning in Social Activity*. 1991.
259. Puntambekar, S., Hübscher, R., *Tools for scaffolding students in a complex environment: What have we gained and what have we missed?* Educational Psychologist, 2005. **40**(1): p. 1-12.
260. Jackson, S., Krajcik, J., Soloway, E., *Model-It™ : A Design Retrospective. In (Eds.), . ,* in *Advanced Designs For The Technologies Of Learning: Innovations in Science and Mathematics Education*, M. Jacobson, Kozma, R., Editor. 2000, Erlbaum: Hillsdale, NJ.
261. Hannafin, M., Land, S., Oliver, K., *Open learning environments: Foundations, methods, and models.* Instructional-design theories and models: A new paradigm of instructional theory, 1999. **2**: p. 115-140.
262. Davis, E.A., *Prompting Middle School Science Students for Productive Reflection: Generic and Directed Prompts.* Journal of the Learning Sciences, 2003. **12**(1): p. 91-142.

263. Linder, S.P., D. Abbott, and M.J. Fromberger, *An instructional scaffolding approach to teaching software design.* J. Comput. Small Coll., 2006. **21**(6): p. 238-250.
264. Thomas, L., M. Ratcliffe, and B. Thomasson, *Scaffolding with object diagrams in first year programming classes: some unexpected results.* SIGCSE Bull., 2004. **36**(1): p. 250-254.
265. Narayanan, N.H. and M. Hegarty, *Communicating Dynamic Behaviors: Are Interactive Multimedia Presentations Better than Static Mixed-Mode Presentations?*, in *Proceedings of the First International Conference on Theory and Application of Diagrams*. 2000, Springer-Verlag. p. 178-193.
266. Whalley, J.L. and R. Lister, *The BRACElet 2009.1 (Wellington) specification*, in *Proceedings of the Eleventh Australasian Conference on Computing Education - Volume 95*. 2009, Australian Computer Society, Inc.: Wellington, New Zealand. p. 9-18.
267. Belland, B.R., French, B.F., Ertmer, P.A., *Validity and problem-based learning research: A review of instruments used to assess intended learning outcomes.* Interdisciplinary Journal of Problem-based Learning, 2009. **3**(1): p. 5.
268. Hmelo-Silver, C.E., R.G. Duncan, and C.A. Chinn, *Scaffolding and achievement in problem-based and inquiry learning: A response to Kirschner, Sweller, and Clark (2006).* Educational Psychologist, 2007. **42**(2): p. 99-107.
269. Rane-Sharma, A., et al. *A methodology for enhancing programming competence of students using Parikshak*. in *Technology for Education (T4E), 2010 International Conference on*. 2010.
270. Gamma, E., et al., *Design patterns: elements of reusable object-oriented software*. 1995: Addison-Wesley Longman Publishing Co., Inc. 395.
271. Kölling, M., *The problem of teaching object-oriented programming, Part 1: Languages.* Journal of Object-oriented programming, 1999. **11**(8): p. 8-15.
272. Koulouri, T., S. Lauria, and R.D. Macredie, *Teaching Introductory Programming: A Quantitative Evaluation of Different Approaches.* Trans. Comput. Educ., 2014. **14**(4): p. 1-28.
273. McNiff, J. and J. Whitehead, *Doing and writing action research*. 2009: SAGE.
274. Remenyi, D., et al., *Doing Research in Business and Management: An Introduction to Process and Method*. 1998: SAGE.
275. Gogoi, l., Goowalla, H., *A study on the impact of research methodology in Ph. d course: An overview:.* International Journal of Development Research, 2015. **5**(11): p. 6065-6067.
276. Miller-Cochran, S.K. and R.L. Rodrigo, *The Wadsworth Guide to Research*. 2008: Cengage Learning.
277. McQueen, R.A. and C. Knussen, *Research methods for social science: a practical introduction*. 2002: Prentice Hall.
278. Goulding, C., *Grounded theory: The missing methodology on the interpretivist agenda.* Qualitative Market Research, 1988: p. 50-57.
279. Scott, D., Usher, R., *Understanding educational research*. 1996: Routledge.
280. Matavire, R. and I. Brown, *Investigating the use of "Grounded Theory" in information systems research*, in *Proceedings of the 2008 annual research conference of the South African Institute of Computer Scientists and Information Technologists on IT research in developing countries: riding the wave of technology*. 2008, ACM: Wilderness, South Africa. p. 139-147.
281. Strauss, A., Corbin, J., *The Basics of Qualitative Research: Techniques and Procedures for Developing Grounded Theory*. 1998: Sage.
282. Glaser, B.G., *Basics of Grounded Theory Analysis*. 1992: Sociology Press.
283. Pandit, N.R., *The Creation of Theory: A Recent Application of the Grounded Theory Method* The Qualitative Report, 1996. **2**(4).
284. Lau, F., *A review on the use of action research in information systems studies*, in *Proceedings of the IFIP TC8 WG 8.2 international conference on Information systems and qualitative research*. 1997, Chapman \& Hall, Ltd.: Philadelphia, Pennsylvania, United States. p. 31-68.

285. Baskerville, R. and J. Pries-Heje, *Grounded action research: a method for understanding IT in practice.* Accounting, Management and Information Technologies, 1999. **9**(1): p. 1-23.
286. Craig, D.V., *Action Research Essentials*. 2009: John Wiley and Sons.
287. Haberman, B., E. Lev, and D. Langley, *Action research as a tool for promoting teacher awareness of students' conceptual understanding.* SIGCSE Bull., 2003. **35**(3): p. 144-148.
288. Liu, Q., L. Chen, and Z. Zhou, *Action Research on Construction of Basic Courses Chief Teachers Pedagogical Model Based on School Network*, in *Proceeding of the 2005 conference on Towards Sustainable and Scalable Educational Innovations Informed by the Learning Sciences: Sharing Good Practices of Research, Experimentation and Innovation*. 2005, IOS Press. p. 775-778.
289. Baskerville, R.L., *Investigating information systems with action research.* Commun. AIS, 1999. **2**(3es): p. 4.
290. Santos, P.S.M.d. and G.H. Travassos, *Action research use in software engineering: An initial survey*, in *Proceedings of the 2009 3rd International Symposium on Empirical Software Engineering and Measurement*. 2009, IEEE Computer Society. p. 414-417.
291. Fredrik Karlsson, P.J.Å. *Multi-Grounded Action Research in Method Engineering: The MMC Case*. 2007.
292. Baskerville, R. and A.T. Wood-Harper, *Diversity in information systems action research methods.* Eur. J. Inf. Syst., 1998. **7**(2): p. 90-107.
293. Hallberg, L., R.-M. , *Some thoughts about the literature review in grounded theory studies.* International Journal of Qualitative Studies on Health and Well-being 2010. **5**(3).
294. Weiss, S.H.a.I., N *Predictive Data Mining: A Practical Guide*. 1998, San Francisco, CA: Morgan Kaufmann Publishers.
295. EVERITT, B.S., *Multivariate Analysis: the Need for Data, and other Problems.* The British Journal of Psychiatry, 1975. **126**(3): p. 237-240.
296. Bellman, R., *Adaptive Control Processes: A Guided Tour*. 1961, Princeton, New Jersey: Princeton University Press.
297. Wilkinson, L. *Tree Structured Data Analysis: {AID}, {CHAID} and {CART}*. in *Sawtooth/{SYSTAT} Joint Software Conference*. 1992.
298. Hall, M., et al., *The WEKA data mining software: an update.* SIGKDD Explor. Newsl., 2009. **11**(1): p. 10-18.
299. Su, X., et al., *An introduction to tree-structured modeling with application to quality of life (QOL) data.* Nursing research, 2011. **60**(4): p. 247.
300. Quinlan, J.R., *C4.5: programs for machine learning*. 1993: Morgan Kaufmann Publishers Inc. 302.
301. Quirin, A., et al., *Graph-based data mining: A new tool for the analysis and comparison of scientific domains represented as scientograms.* Journal of Informetrics, 2010. **4**(3): p. 291-312.
302. Gonzalez, J., L. Holder, and D.J. Cook. *Application of graph-based concept learning to the predictive toxicology domain*. in *Proceedings of the Predictive Toxicology Challenge Workshop*. 2001.
303. Han, J., J. Pei, and Y. Yin, *Mining frequent patterns without candidate generation.* SIGMOD Rec., 2000. **29**(2): p. 1-12.
304. Simmons, R.F., *Synthetic language behavior.* Data Process. Management, 1963. **5**(12): p. 11-18.
305. Zhang, S., et al., *A Multi-Semantic Classification Model of Reviews Based on Directed Weighted Graph*, in *Web Information Systems Engineering – WISE 2016: 17th International Conference, Shanghai, China, November 8-10, 2016, Proceedings, Part II*, W. Cellary, et al., Editors. 2016, Springer International Publishing: Cham. p. 424-435.
306. Cormen, T.H., et al., *Introduction to Algorithms, Third Edition*. 2009: The MIT Press. 1312.
307. Alistair Miles, S.B. *SKOS simple knowledge organization system reference, W3C Recommendation*. 2009 [cited 2017 04/01]; Available from: http://www.w3.org/TR/skos-reference/.

308. Baker, T., et al., *Key choices in the design of Simple Knowledge Organization System (SKOS).* Web Semant., 2013. **20**(C): p. 35-49.
309. Hung, S.-L., L.-F. Kwok, and R. Chan, *Automatic programming assessment.* Comput. Educ., 1993. **20**(2): p. 183-190.
310. Mengel, S.A. and V. Yerramilli, *A case study of the static analysis of the quality of novice student programs.* SIGCSE Bull., 1999. **31**(1): p. 78-82.
311. Edwards, S.H., *Improving student performance by evaluating how well students test their own programs.* J. Educ. Resour. Comput., 2003. **3**(3): p. 1.
312. Jackson, D., *A software system for grading student computer programs.* Computers & Education, 1996. **27**(3–4): p. 171-180.
313. McCabe, T.J., *A Complexity Measure.* Software Engineering, IEEE Transactions on, 1976. **SE-2**(4): p. 308-320.
314. Rohaida Romli , M.R., *Penyukatan Automatik Kekompleksan TugasanAturcara Java.*, in *National ICT Conference at Universiti TeknologiMARA*. 2006: Arau, Perlis, Malaysia.
315. Striewe, M. and M. Goedicke, *A Review of Static Analysis Approaches for Programming Exercises*, in *Computer Assisted Assessment. Research into E-Assessment*, M. Kalz and E. Ras, Editors. 2014, Springer International Publishing. p. 100-113.
316. Tegarden, D.P., S.D. Sheetz, and D.E. Monarchi, *A software complexity model of object-oriented systems.* Decision Support Systems, 1995. **13**(3–4): p. 241-262.
317. Li, W. and S. Henry, *Object-oriented metrics that predict maintainability.* Journal of Systems and Software, 1993. **23**(2): p. 111-122.
318. Chen, J.Y. and J.F. Lu, *A new metric for object-oriented design.* Information and Software Technology, 1993. **35**(4): p. 232-240.
319. Al-Radaideh, Q.A., E.M. Al-Shawakfa, and M.I. Al-Najjar. *Mining student data using decision trees*. in *International Arab Conference on Information Technology (ACIT'2006), Yarmouk University, Jordan*. 2006.
320. Bhardwaj, B.K. and S. Pal, *Data Mining: A prediction for performance improvement using classification.* arXiv preprint arXiv:1201.3418, 2012.
321. Minaei-Bidgoli, B., et al. *Predicting student performance: an application of data mining methods with an educational Web-based system*. in *Frontiers in Education, 2003. FIE 2003 33rd Annual*. 2003.
322. Powell, R.M., *Improving the persistence of first-year undergraduate women in computer science.* SIGCSE Bull., 2008. **40**(1): p. 518-522.
323. *Computing curricula 2001.* J. Educ. Resour. Comput., 2001. **1**(3es): p. 1.
324. Langrich, M. and J. Schulze. *A Systematic Approach to Immediate Verifiable Exercises in Undergraduate Programming Courses*. in *Frontiers in Education Conference, 36th Annual*. 2006.
325. Schulze, J., M. Langrich, and A. Meyer. *The success of the demidovich-principle in undergraduate C# programming education*. in *Frontiers In Education Conference - Global Engineering: Knowledge Without Borders, Opportunities Without Passports, 2007. FIE '07. 37th Annual*. 2007.
326. Sherriff, M., et al., *Early estimation of defect density using an in-process Haskell metrics model.* SIGSOFT Softw. Eng. Notes, 2005. **30**(4): p. 1-6.
327. Han, J. and M. Kamber, *Data mining: concepts and techniques*. 2000: Morgan Kaufmann Publishers Inc. 550.
328. Agrawal, R. and R. Srikant, *Fast Algorithms for Mining Association Rules in Large Databases*, in *Proceedings of the 20th International Conference on Very Large Data Bases*. 1994, Morgan Kaufmann Publishers Inc. p. 487-499.
329. J. Raven, J.C.R., *Section 4: Raven Manual: Advanced Progressive Matrices*. 1998, Harcourt. p. 37.
330. J. Raven, J.C.R., *Raven Manual: Section 4. Advanced Progressive Matrices*. 1998: Harcourt.

331. Asghar Ghasemi, S.Z., *Normality Tests for Statistical Analysis: A Guide for Non-Statisticians.* International Journal of Endocrinology and Metabolism, 2012. **10**(2): p. 486–489.
332. Shapiro, S.S.W., M.B., *An Analysis of Variance Test for Normality (Complete Samples).* Biometrika, 1965. **52**(3/4): p. 591-611.
333. Elliott, A.C. and W.A. Woodward, *Statistical Analysis Quick Reference Guidebook: With SPSS Examples*. 2006: Sage Publications Pvt. Ltd.
334. Kim, H.-Y., *Statistical notes for clinical researchers: assessing normal distribution (2) using skewness and kurtosis.* Restorative Dentistry & Endodontics, 2013. **38**(1): p. 52-54.
335. Sheskin, D.J., *Handbook of Parametric and Nonparametric Statistical Procedures*. Third ed. 2003: CRC Press.
336. Dunst, C.J., Hamby, D.W., Trivette, C.M., *Guidelines for Calculating Effect Sizes for Practice-Based Research Syntheses* Centrescope, 2004. **3**(1).
337. Cohen, J., *Statistical Power Analysis for the Behavioral Sciences*. 1988: L. Erlbaum Associates.
338. Glass, G.V., McGaw, B., Smith, M.L., *Meta-analysis in social research*. 1981: Sage Publications.
339. Zaiontz, C. *Real Statistics Using Excel*. 2015 [cited 2015; Available from: http://www.real-statistics.com/chi-square-and-f-distributions/effect-size-chi-square/.
340. Nandy, K. *Online Slides: Understanding and Quantifying Effect Sizes*. [cited 2015; Available from: http://nursing.ucla.edu/workfiles/research/Effect%20Size%204-9-2012.pdf.
341. Freeman, E., et al., *Head First Design Patterns*. 2004: O' Reilly \& Associates, Inc.
342. Larman, C., *Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development (3rd Edition)*. 2004: Prentice Hall PTR.
343. Warren, I., *Teaching patterns and software design*, in *Proceedings of the 7th Australasian conference on Computing education - Volume 42*. 2005, Australian Computer Society, Inc.: Newcastle, New South Wales, Australia. p. 39-49.
344. Astrachan, O., et al., *Design patterns: an essential component of CS curricula.* SIGCSE Bull., 1998. **30**(1): p. 153-160.
345. Astrachan, O. and D. Reed, *AAA and CS 1: the applied apprenticeship approach to CS 1.* SIGCSE Bull., 1995. **27**(1): p. 1-5.
346. Wallingford, E., *Toward a first course based on object-oriented patterns.* SIGCSE Bull., 1996. **28**(1): p. 27-31.
347. East, J.P., et al. *Pattern-based programming instruction*.
348. Bassok, M., *Analogical transfer in problem solving.* The psychology of problem solving, 2003: p. 343-369.
349. Spraul, V.A., *Think Like a Programmer: An Introduction to Creative Problem Solving*. 2012: No Starch Press. 256.
350. Kirkley, J., *Principles for teaching problem solving.* USA: PLATO Learning Inc, 2003.
351. Gugerty, L. and G. Olson, *Debugging by skilled and novice programmers.* SIGCHI Bull., 1986. **17**(4): p. 171-174.
352. Kernighan, B., *code testing and its role in teaching.* ; login:: the magazine of USENIX & SAGE, 2006. **31**(2): p. 9-18.
353. Runquist, W.N., *Interference among memory traces.* Memory & Cognition, 1975. **3**(2): p. 143-159.
354. Soloway, E., J. Bonar, and K. Ehrlich, *Cognitive strategies and looping constructs: an empirical study.* Commun. ACM, 1983. **26**(11): p. 853-860.
355. Spohrer, J.C., *Marcel: Simulating the novice programmer*. 1992: Intellect Books.
356. Simon, *Soloway's Rainfall Problem Has Become Harder*, in *Proceedings of the 2013 Learning and Teaching in Computing and Engineering*. 2013, IEEE Computer Society. p. 130-135.

357. Srinivasan, S., *Design Patterns in Object-Oriented Frameworks.* Computer, 1999. **32**(2): p. 24-32.
358. Neville, A.J., *Problem-based learning and medical education forty years on. A review of its effects on knowledge and clinical performance.* Med Princ Pract, 2009. **18**(1): p. 1-9.
359. Koh, G.C., et al., *The effects of problem-based learning during medical school on physician competency: a systematic review.* Cmaj, 2008. **178**(1): p. 34-41.
360. Michalewicz, Z. and M. Michalewicz, *Puzzle-based Learning: Introduction to Critical Thinking, Mathematics, and Problem Solving*. 2008: Hybrid Publishers.
361. Saurabh R Shrivastava, P.S.S., Jegadeesh Ramasamy, *Problem-based learning in undergraduate medical curriculum: An Indian perspective.* Arch Med Health Sci 2013. **1**(2): p. 200-201.
362. Schmidt, H.G., J.I. Rotgans, and E.H. Yew, *The process of problem-based learning: what works and why.* Med Educ, 2011. **45**(8): p. 792-806.
363. Bartlett, F.C. and C. Burt, *Remembering: A Study in Experimenta and Social Psychology.* British Journal of Educational Psychology, 1933. **3**(2): p. 187-192.
364. DeMarco, T., *Structured Analysis and System Specification*. 1979: Prentice Hall PTR. 352.
365. Applin, A.G., *Second language acquisition and CS1.* SIGCSE Bull., 2001. **33**(1): p. 174-178.
366. Martin, R.C., *Agile Software Development: Principles, Patterns, and Practices*. 2003: Prentice Hall PTR. 710.
367. Kransner, G.E., Pope, S. T., *Cookbook for using the Model-View-Controller User Interface paradigm.* Journal of Object Oriented Programming, 1988: p. 26-49
368. Fowler, M., *Patterns of Enterprise Application Architecture*. 2003: Addison-Wesley.
369. Holdener, A.T., *Ajax: The Definitive Guide*. 2008: O'Reilly Media.
370. Fielding, R.T., *Architectural styles and the design of network-based software architectures*. 2000, University of California, Irvine. p. 162.
371. International, E., *ECMA-262: ECMAScript@2016 Language Specification*. 2016.
372. T. Ruutmann, H.K. *Teaching strategies for direct and indirect instruction in teaching engineering*. in *Interactive Collaborative Learning (ICL), 2011 14th International Conference on*. 2011.
373. Scott, M.J. and G. Ghinea, *Educating programmers: A reflection on barriers to deliberate practice.* arXiv preprint arXiv:1311.0390, 2013.
374. Dweck, C.S., *Messages that motivate: How praise molds students' beliefs, motivation, and performance (in surprising ways)*, in *Improving academic achievement: Impact of psychological factors on education*. 2002, Academic Press: San Diego, CA, US. p. 37-60.
375. Gibbs, G., et al., *Conditions under which assessment supports students' learning.* 2005.
376. Green, T.R.G., R.K.E. Bellamy, and M. Parker, *Parsing and Gnisrap: a model of device use*, in *Empirical studies of programmers: second workshop*, M.O. Gary, S. Sylvia, and S. Elliot, Editors. 1987, Ablex Publishing Corp. p. 132-146.
377. Dunlosky, J., et al., *Improving Students' Learning With Effective Learning Techniques.* Psychological Science in the Public Interest, 2013. **14**(1): p. 4-58.
378. Rohrer, D. and K. Taylor, *The shuffling of mathematics problems improves learning.* Instructional Science, 2007. **35**(6): p. 481-498.
379. Taylor, K. and D. Rohrer, *The effects of interleaved practice.* Applied Cognitive Psychology, 2010. **24**(6): p. 837-848.
380. Rau, M., V. Aleven, and N. Rummel. *Blocked versus interleaved practice with multiple representations in an intelligent tutoring system for fractions*. in *Intelligent tutoring systems*. 2010. Springer.
381. Schneider, V.I., A.F. Healy, and L.E. Bourne, *What is learned under difficult conditions is hard to forget: Contextual interference effects in foreign vocabulary acquisition, retention, and transfer.* Journal of Memory and Language, 2002. **46**(2): p. 419-440.

Appendices

Appendix 1 Computational Thinking Test

Answer only the questions you can – if you cannot answer a question, move on to the next question. This test is not negatively marked, so you may wish to guess if unsure. Questions 1 and 2 to be answered on this paper, questions 3 and 4 to be answered using IDLE and saved.

1) You have been selected to program a new robot intended to create hot drinks. The robot is capable of following simple, tea and coffee-oriented commands precisely, but has no understanding either of the process, or the fundamental principles which underpin it (e.g. that a kettle requires power). The robot has access to the following items:
   - Kettle (initially unplugged)
   - Tea bags
   - Jar of ground instant coffee
   - 1L carton of milk
   - Unopened bag of sugar
   - 1 metal tea spoon
   - 1 large mug
   - Access to a sink for water and an electrical socket for power

   a) You are required to give the robot the set of instructions necessary to successfully make a cup of milky coffee with 1 teaspoon of sugar. Each instruction should be on a new line. Ensure that your instructions are in a logical order and no steps are missed; while highly capable of following instructions, the robot cannot solve problems independently.
   b) The robot is, of course, capable of making many variations of hot drink. The user must be permitted to give information about their drink preferences to the robot before it is created. What are the pieces of information the robot must collect before starting?
   c) If the instructions were converted to code, explain briefly how this data may be stored. What is the name given to a piece of data stored by a computer?

2) Write a single flowchart which:
    a) Says “Hello” to the user at the start.
    b) Asks the user how many addition operations they would like to perform.
    c) Loops the number of times requested by the user.
    d) For each loop, takes two new numbers from the user, adds them together and outputs the result.

3) You are required to create a computer system, in Python, which mimics the functionality of the national lottery. Each lottery draw results in six balls, numbered between 1 and 49 being selected, plus one “bonus ball” making a total of seven. The balls are not replaced after each selection, so each number may only be selected once. Remember, in Python, a random number may be selected through the statement:

```
random.randint(0, 10
```

This code will pick a random number between 0 and 10 inclusive, and duplicates are possible.

The following requirements specification was created for the application:

- Seven random numbers should be selected – six regular numbers plus the bonus ball
- The numbers should be displayed to the user

The output format should mirror the following (where the numbers following colons are generated randomly):

```
Starting lottery selection.
Ball 1: 17
Ball 2: 31
Ball 3: 5
Ball 4: 44
Ball 5: 28
Ball 6: 33
Bonus ball: 22
Lottery selection complete.
```

- No randomly selected number should be repeated

You should aim to implement as many of these requirements as possible within your Python solution. Focus on the logic of the program, and do not worry unduly about syntax. If you feel a requirement will be too difficult to implement, ignore it and focus on the others.

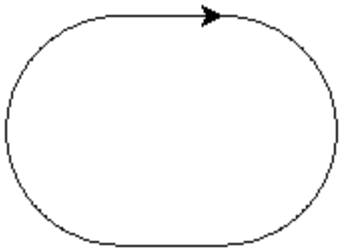
4) Using Python with turtle, draw the following shapes:

   a) An oval. Remember that the code to draw a circle is:

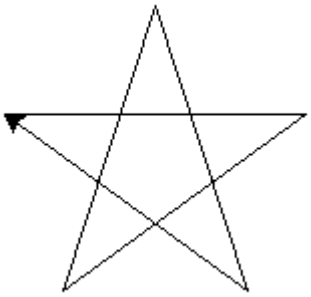
```
import turtle

count = 0
while (count < 360):
   turtle.forward(1)
   turtle.right(1)
   count = count + 1
```

You should start with this as a base, and modify it to form an oval similar to:

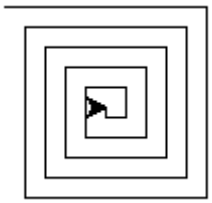


   b) A 5 point star. This can be achieved without lifting the pen. It should look similar to:



Hint: the angle of rotation at the end of each point is 144 degrees. (medium)

   c) A spiral. This may be a square spiral similar to:



You should not equate each edge to a line of code. Instead consider what you can use to reduce the amount of code you are required to produce.

Appendix 2 ACI and Problem Solving Tests

Test 1: Test for Assessing Student's Knowledge of Variables

1. Identify the types need and suitable values for the following:

(a)

```
type postCode;
postCode = value;
```

(b)

```
type numOfAnimals;
numOfAnimals = value;
```

[2 Marks]

2. Write a program that allows the user to calculate the cost of purchasing a number of cars of the same type. They must be able to enter the car model name, the price and the quantity that they wish to buy

Print out the model name, price and the total cost including a tax of 12.5%.

[12 Marks]

3. A shop owner requires a program to calculate the running costs and profits of their business. The business employs a number of people but each earn the same wages and the operational costs of the shop include supplies, manufacturing and utility costs. These values will be entered into the program. It has been agreed that the owner will calculate the total sales and will also enter this value. However, the program should calculate the overall profits made, allowing for VAT which will be alterable but have a default value of 17.5%. For security, the program will also require the owner to login with a pre-set username and password.

Identify the variables, selecting appropriate names and types.

[11 Marks]

Test 2: Test for Assessing Student's Knowledge of Branch Statements

1. (a) Enter two values:

```
type cupCount = ???;
type maxCups = ???;
```

Display “You have purchased up to or over the max cups allowed”

[3 Marks]

(b) Enter the original price of a product and a sales discount as a percentage (e.g. 12.5 for 12.5%).

Calculate the discount and the new product price.

[4 Marks]

2. Given a temperature under 321 check:

- pressure is below 48 and display “pressure too low”
- pressure is 12 or under and display “warning pressure is falling too low”

Given a temperate at 459 or more check:

- Pressure is above 35 and display “warning pressure is rising to high”
- Pressure is above 126 and display “pressure too high”

[11 Marks]

3. A program is required to monitor the water level in a pumping station. The water level is measured and entered into the program 4 times during the day (you may assume it is rerun everyday), and must display the highest level the water has reached during the day.

[7 Marks]

Test 3: Test for Assessing Student’s Knowledge of Array and Loop Statements

1. (a) Enter two values:

```
type isReady = ???;      ← assume not ready initially
type postCode = ???;
```

[2 Marks]

(b) Create the variables for the following:

```
type[] prices = new type[??];  // Store 22 prices
```

Given a dog owner has 8 dogs, and requires a program that can remember all their names.

Given there are 78 streets, provide a variable that can store the number of houses in each street.

[4 Marks]

(c) Create an array of 12 dog names, and set the following three names.

Set first name:	dogName[??] = “Fido”;
Set second name:	dogName[??] = “Biff”; … ←Don’t care about remaining names
Set last name:	dogName[??] = “Bones”;

[3 Marks]

(d) Display a count that increments from 0 to 99.

```
type count;
for(count = ??; count < ??; ???)
{
   Console.WriteLine(“Count is {0}”, count);
}
```

[3 Marks]

2. Allow the user to enter 30 numbers.

- After they have all been entered, print all the numbers in the order.
- Print numbers entered in the reverse order.

[6 Marks]

3. Allow the user to enter the names of 20 books. Once all the book names are entered, allow the user enter the name of one of the books, and then check that it was one of the previous names entered. Print a message if it is found.

[8 Marks]

4. A program is required to monitor the water level in a pumping station. The water level is measured and entered by the user on a continuous basis until they decide to quit the program e.g. they type “quit”.

- When it exceeds 50m, a warning message should be displayed
- When it exceeds 100m, an “overflow” alarm message should be displayed
- When it falls below 20m, a warning should be displayed
- When it falls to 0m, an “empty” alarm should be displayed
- After quitting and before exiting the program, the average water level should be displayed.

NOTE: You do not need to store all the water levels.
[8 Marks]

5. Display the following menu:

1. Choose Max Numbers
2. Enter Number
3. Add All Numbers
4. Calculate 12.3% of All Numbers
5. Quit

You may assume that the default max numbers that can be entered is 5, and if no numbers are entered the results displayed should all be 0.

When executing the program would look something like this:

```
Max numbers you can currently enter is 5  ← max starts at 5

1. Choose Max Numbers
2. Enter Numbers
3. Add All Numbers
4. Calculate 12.3% of All Numbers
5. Quit
Select Option > 1
Max > 3                                   ← User enters max

1. Choose Max Numbers
2. Enter Numbers
3. Add All Numbers
4. Calculate 12.3% of All Numbers
```

5. Quit

Select Option > 2

Please Enter 3 Numbers ← User is asked to enter max numbers

Enter Number > 10

Enter Number > 20

Enter Number > 30

1. Choose Max Numbers

2. Enter Numbers

3. Add All Numbers

4. Calculate 12.3% of All Numbers

5. Quit

Select Option > 3

Sum is 60

1. Choose Max Numbers

2. Enter Numbers

3. Add All Numbers

4. Calculate 12.3% of All Numbers

5. Quit

Select Option > 4

12.3 of 60 is 7.38

1. Choose Max Numbers

2. Enter Numbers

3. Add All Numbers

4. Calculate 12.3% of All Numbers

5. Quit

Select Option > 5

Goodbye

Start by getting the menu to work and only allowing the user to quit by entering number 5 for the menu option.

[10 Marks]

Test 4: Comparison Test for ACI and Non-ACI Focus Group Prior to Problem Solving Instruction

1. (a) Enter two values:

type numberOfPeople = ???;
type maxLength = 3/2;

[2 Marks]

(b) Ask the user to enter price of a book. When the price is £20 or more the book is delivered for free, otherwise the cost of delivery is £1.50. Display the told purchase and delivery cost of the book.

[3 Marks]

(c) Enter the original price of a product and a sales discount as a percentage (e.g. 22.5 for 22.5%). Calculate the discount and the new product price.

[5 Marks]

2. Write the code for a stock checking application.

Check the number of outstanding deliveries exceeds 1010 then check:

- Boxes in stock is below 897 and display “order more stock”
- Boxes in stock 467 or under and display “warning stock level is low”

Check the number of outstanding deliveries at 459 or lower then check:

- Boxes in stock are above 2033 and display “warning stock level is getting high”
- Boxes in stock is above 5456 and display “stop ordering stock”

[25 Marks]

3.

A program is required to monitor the pressure level in a pumping station during the day. The pressure level is measured and the user must continuously enter it into the program. At the end of the day, the user exits the program and the program displays the highest and lowest levels the pressure reached during that day.

[20 Marks]

4. Allow the user to enter exactly 25 numbers. After they have all been entered, print all the numbers in the order they were entered and in the reverse order.

[20 Marks]

5. Allow the user to enter exactly 12 numbers. After they have all been entered, allow the user to search for a number and print a message telling them if the number was previously entered.

[15 Marks]

6. Display a multiplication table. The user enters two for the max rows and the max columns e.g. 2 and 3 and the table should be displayed like this:

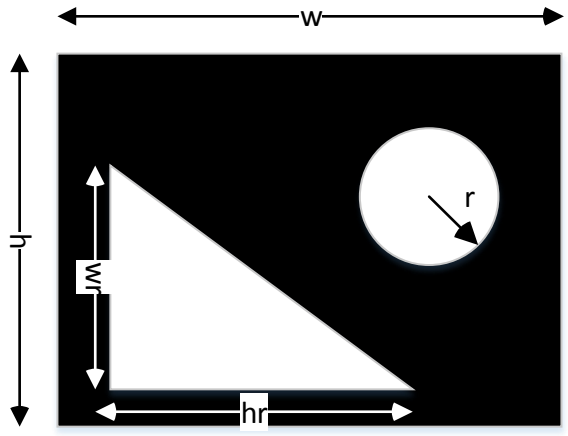
1 x 1 = 1
1 x 2 = 2
1 x 3 = 3
2 x 1 = 2
2 x 2 = 4
2 x 3 = 6

[10 Marks]

Test 5: Comparison Test for ACI and Non-ACI Focus Group Post Problem Solving Instruction

1. Write a function to calculate the area of right angled triangle and provide appropriate test code. Hint: Think half a rectangle.

2. Calculate the black area of the following shape:



The User should enter all the dimension values required.

FOR THE FOLLOWING QUESTIONS, MAP WHAT YOU KNOW AGAINST THE PROBLEMS YOU NEED TO SOLVE USING THE TABLE PROVIDED

3. Write the program that allows the user to enter 5 numbers, and then print the numbers in reverse order e.g. they enter 1, 3, 4, 5, 8 and then you display 8, 5, 4, 3, 1.

4. Write the code to randomise the selection of 5 lottery balls. Code has been provided below to help you:

```
static void Main(string[] args)
{
       Random rnd = new Random();
       int[] balls = new int[] { 1, 2, 3, 4, 5 };

       RandomiseBalls(balls, rnd);  ← Randomise the numbers 1 to 5
       DisplayBalls(balls);             ← Display the numbers

       Console.ReadKey();
}

// Select a random value between the provided min and max values (both the min
// and max values can also be chosen).
static int RandomNumber(Random rnd, int minValue, int maxValue)
{
       return rnd.Next(minValue, maxValue + 1);
}
```

MAPPING TABLES PROVIDED FOR STUDENT USE

Appendix 3 Structured Problem based Programming Online Survey

Question	Type
I recognise the importance of solving problems in programming	Likert 1 to 10
I find solving problems challenging	Likert 1 to 10
I find solving coding and solving problems interesting	Likert 1 to 10
I have learnt more by attempting to solve problems myself in class	Likert 1 to 10
In working on the exercises provided: I spent very little time attempting them I would like to have spent more time attempting them I was too busy or unable to attempt them for other reasons I felt I dedicated enough time I spent too much time	Single Choice
Engaging in problem solving learning leads to more class interaction between students and lecturer	Likert 1 to 10
I felt I was solving problems WITH the lecturer	Likert 1 to 10
I found the class more interesting when trying to solve the challenges presented by the lecturer	Likert 1 to 10
I prefer to follow code or solutions, step-by-step, developed by the lecturer	Likert 1 to 10
Problem solving activities provide gave me a better understanding of the technologies or principles being taught	Likert 1 to 10
The context of the problem is important (I like to know why it is important to solve a problem)	Likert 1 to 10
It is more interesting to discover next problem(s) myself, as a consequence of completing a previous exercise.	Likert 1 to 10
I prefer partially solved problems to new problems with no initial code provided	Likert 1 to 10
I prefer to learn new technologies or concepts by attempting to build my own solutions	Likert 1 to 10
I reviewed the completed solutions offered by the lecturer after attempting the problems myself	Likert 1 to 10
Sufficient documentation was provided to attempt the exercises	Likert 1 to 10
Providing hyperlinks between the code in the documentation enabled me to follow the code more easily	Likert 1 to 10
The exercises provided a gradual increase in difficulty (allowing for the complexity of the concepts being taught)	Likert 1 to 10
I found this approach gave me confidence in my ability to develop my own learning skills	Likert 1 to 10
I will be more confident in studying new technologies in the future	Likert 1 to 10
Please describe any benefits you felt you gained from the problem based learning approach	Open text
Please provided details of any drawbacks or anything you disliked in problem based learning	Open text