CONCEPT DEVELOPMENT IN NOVICE PROGRAMMERS LEARNING JAVA

CONCEPT DEVELOPMENT IN NOVICE PROGRAMMERS LEARNING JAVA

by

Walter William Milner

A thesis submitted to

The University of Birmingham

For the degree of

Doctor of Philosophy

School of Education

The University of Birmingham

October 21st, 2010

ABSTRACT

It is hypothesised that the development of concepts in formal education can be understood through the ideas of non-literal language and conceptual integration networks. The notions of concept, understanding and meaning are examined in some depth from philosophical, psychological and linguistic standpoints. The view that most concepts are grasped through non-literal means such as metaphor and conceptual blend is adopted. The central contention is that this applies both to everyday ideas and to those presented to students in formal educational contexts, and that consequently such learning is best seen in those terms. Such learning is not founded upon literal language, but a construction by the student of a complex network of metaphor and conceptual blends.

This is examined in the context of students learning programming, in particular in the language Java. The hypothesis is tested by analysing transcribed interviews with a wide range of students, triangulated with an examination of teaching materials, and the data is shown to be consistent with the hypothesis. However the approach is fundamental and is not concerned with specific features of programming or Java, so that conclusions are relevant across a wide range of disciplines, especially mathematics, science and engineering.

The thesis provides a new way of examining course design and learning materials including lectures and textbooks. Discourse which might seem to be literal is in fact metaphorical and blended, since it is in that way that the expert community understands the ideas. The students' construction of corresponding blends is on the basis of their learning experience, and course design features such as examples can be explained and evaluated in such terms.

DEDICATION

This thesis is dedicated to David Tall, Sue, and the cat.

ACKNOWLEDGEMENTS

I must thank staff at the relevant teaching institutions for their permission to carry out the student interviews, and their co-operation in arranging them. Thanks to the students interviewed for their time, and their courage in allowing their understanding to be exposed to examination, when for some this was a difficult issue. Thanks to my wife Carol for her tolerance of my bizarre ramblings. Thanks to Dave Hewitt, Helen Joinson for their support, and to the University of Birmingham for allowing me to complete this work. Most of all, my thanks go to Professor David Tall, a most remarkable man, whose support, encouragement and insight have been central to this thesis.

TABLE OF CONTENTS

LIST OF ABBREVIATIONS

CIN - conceptual integration network

CS - computer science

GCSE - General Certificate of Secondary Education

CSE - computer science education

GUI - graphical user interface

JLS – Java Language Specification

ICT - information and communications technology

IDE - integrated development environment

OO - object-oriented

OOD - object-oriented design

OOP - object-oriented programming

STM - short-term memory

CHAPTER ONE- Introduction

In my beginning is my end.

from East Coker by T.S.Eliot

1.1 The subject of this thesis

This work has its origins in the author's many years of classroom practice, teaching physics, mathematics and computing at school, college and university level. Part of that experience was the vivid sensation of *understanding*, both in its presence and absence. At one time educational theory was in thrall to behavioural psychology and setting the aim of a lesson to be that students should understand something was not allowed, since understanding was not an observable. Since then cognitive science has developed and there is a general acceptance that *understanding* is a valid notion, even though it is difficult to grasp.

This thesis is an attempt to investigate how students develop understandings of concepts in 'scientific' disciplines. In order to have a reasonable focus, its scope is limited to learning computer programming, and centrally, coming to understand the concepts of object-oriented programming (OOP) in Java. It is restricted to novices – students with little or no previous experience of programming.

It is not about the cognitive processes of professional programmers. A considerable amount of work has been done on this, from an essentially ergonomic point of view, with the motivation of making software development as effective as possible. This thesis is about *learning* programming rather than professional practice.

Neither is it about problem-solving. It is obvious that programming, both by novices and experts, contains some element of problem-solving, whether it be radically innovative or just the recognition of a context and the application of routine techniques. But the focus here is on students coming to understand the foundational concepts of OOP, rather than original uses of them to solve problems.

The approach tries to be as fundamental as possible. This derives from the assumption that people understand Java the same way that they understand anything else. Human cognition is assumed to be a consequence of evolutionary pressure which has resulted in neural and social structures which enable us to 'make sense' of a remarkable range of ideas. It is unreasonable to suppose that students have special cognitive apparatus devoted to Java and OOP. Consequently it is necessary to look at how humans understand concepts in general. Java is simply the context.

1.2 Why Java?

While some investigation in this thesis looks at general notions of computers and the execution of short pieces of pseudo-code, the focus is on the Java language and associated OOP concepts. There are several reasons for this. Java is currently a very widely used language, for desktop applications, client-server systems (using 'Java Enterprise Edition') and mobile phone applications on Android. It is also very often the first programming language taught at high school and undergraduate level. However numerous studies have shown that many students encounter problems when learning it - such as the McCracken Group Report (2001), The ACM Java Task Force (Roberts et al 2006), Hu (2006), Fleury (2000), Holland, Griffiths and Woodman (1997), Garner, Haden and Robins (2005), and Thomasson, Ratcliffe and Thomas (2006).

One aspect of this is that Java was designed so that professional programmers accustomed to languages such as C would be able to develop proficiency in it quickly. The syntax of variable declaration, assignment statements, loops and conditionals in Java is similar or identical to that of C, and there is an assumption that the person learning Java is already familiar with these concepts, which of course is not true for novices.

A second aspect of Java is that it involves a set of inter-dependent ideas, which cannot be isolated. As a fairly 'pure' OOP language, even the simplest executable program makes reference to a host of unfamiliar concepts, represented by the keywords *public, class, static, void* and *[]*. Beyond the first program, OOP entails a number of ideas which are distinct but closely linked, such as class and object. This presents a difficulty in designing a learning sequence, which one would ideally like to lead the student to encounter new ideas one at a time – impossible in Java because the concepts are dependent upon each other.

A third aspect is that there is a convention that classes and methods are named so as to make them self-descriptive (contrasted with languages like Perl where brevity is more important). There are clear advantages in this, but for a student there is the problem that they easily acquire a 'bleached' idea of what something is, and this can mask a precise understanding. For example a student may well think they 'understand' what Math.PI is, but not be aware that it is a final static member of the class Math. This also makes it difficult to investigate students' understanding, since they can often make shallow but correct guesses as to what a program fragment means.

Fourthly, the notion of metaphor has peculiar significance in Java. It is argued here that most of the content of all programming languages, and discourse about computers, is derived from metaphors and other conceptual blends. These metaphors are mostly unobserved – such as computer 'memory' or program 'instruction'. But in the case of OOP there is a further layer of more or less conscious metaphor, such as class, object, method and member. These correspond to literary metaphors, like 'All the world's a stage', where the author is aware that they are inventing a metaphorical way of looking at something. Consequently the student of Java has to deal with two layers of metaphor, even though typically neither layer is pointed out in an instructional course.

These features mean Java is an effective vehicle for exploring this thesis, and provide affordances and constraints in the research. Pragmatically Java is used because it is what most novice students are taught.

1.3 Concepts

What are the concepts used in OOP? Armstrong (2006) examined 239 sources published between 1966 and 2005 to identify the most commonly used set of OOP ideas. The 'top ten' were in order (these ideas are outlined in 1.5):

1. Inheritance
2. Object
3. Class
4. Encapsulation
5. Method
6. Message Passing
7. Polymorphism
8. Abstraction
9. Instantiation
10. Attribute

However this treats the ‘concept concept’ as if it were non-problematic and well understood. This is not the case. Chapters Two and Three examine the ideas of concept and meaning from philosophical, psychological and linguistic viewpoints.

1.4 The idea of the wrong answer

Most of the data collection involves asking students questions about short programs, and students are anxious to provide correct answers. However Java and OOP is an invented system – who decides what the correct answers are?

The approach adopted here is to consider ‘the normative response’, as being what the community of practice would treat as the ‘correct answer’. This approximates to the response which would be marked correct in an examination paper.

In fact this is rather simplistic – the ideas of authors of student text-books, professional programmers, and professors (see section 9.4) are not identical. Nevertheless it shows how students’ ideas are placed in a spectrum of other people’s ideas.

If a student demonstrates an understanding which does not correspond to the normative approach, this is very useful, in that it shows how students’ thinking can diverge from the expected. Ethical aspects of this are considered in 1.6.

1.5 Data collection methodology and theoretical stance

The research question here is to identify how students develop an understanding of concepts when learning Java OOP.

Research in the sciences proceeds with confidence in hypothesis testing based on repeatable experiments yielding objective quantitative data. Social science research using qualitative data always needs to justify itself. It would be comforting to be able to identify, for example, three possible answers to the research question, and then to devise some experimental procedure which would yield quantitative data which would demonstrate which of the possibilities were true. However examination of the question shows that things are not so simple as that. Most significantly there is the question of what a 'concept' is, and what it means to 'understand'.

Consequently the route taken was to first establish some theoretical understanding of those terms, through psychological, philosophical and linguistic approaches, and the work of previous computer science education research. This is described in Chapters Two and Three.

Data collection proceeded through several phases, as described in Chapters Five to Nine. These differ in detail, but they broadly consist of talking to students about computer programs and examining the transcripts.

Students were drawn from two institutions. One was a College of Further Education, with students with modest previous educational achievements, enrolled on vocational courses. The second was a research-led Russell Group university, with first year undergraduates with good A Level grades, and some mature students. This ensured a wide range of student profiles was encountered.

The data collection process took place in parallel with the theoretical work, and was informed by it as the theoretical position emerged. Early interviews were rather general and atheoretical, and became more focussed in the later stages.

There were several broad groups of interviews. The first stage was very exploratory, involving a small class of mature computing students, together with some work on ‘class’ ‘object’ and ‘set’ with some psychology students. The second stage was with some undergraduates at the start of their course, exploring their ideas of computers and programming. The third involved college students, and a detailed analysis of five pseudo-code program fragments, in order to form a picture of how programs varied in ‘difficulty’. The final stage involved case studies of a small number of university and college students as they worked through a first course in Java, and this focussed on the way OOP concepts were developing.

The issue was whether student responses could be interpreted in terms of the emerging theoretical position, which was that understanding meant having a mental space structured by a frame built on an appropriate conceptual integration network and revealed in figurative language (ideas elaborated in Chapter Three). Even if the answer to this was positive, obviously it only leaves us with the absence of a denial. In other words the data collection and analysis would do no more than show that the theoretical position was possible. Nevertheless it was felt that this in itself would be a significant advance.

1.5.1 Tutorial and research roles

As the interviews progressed, it became clear that tutorial and research roles stood in a relationship which was not straight-forward. Simplistically, the researcher asks a question, such as ‘what do you think a Java object is?’. The student answers, and the researcher later treats that answer as data which can be analysed to reveal the student’s understanding. But in a real interview, the answer may be unclear, or even the answer reveals that the question is not clear, at least from the student’s point of view. In that situation there is usually some elucidation of the situation, such as re-phrasing it, giving an example and so on. This means the researcher shifts into the role of the teacher. This does not invalidate the process, but it does make it more complicated, since we are not simply finding out what the student’s concept is. Rather we are examining a ‘micro-teaching’ process, and watching what happens as a student learns. This is also useful when the topic is re-visited in a later interview.

A second aspect of this is ethical, since we want to avoid leaving students with the 'wrong idea'. This is considered in section 1.6.

1.5.2 Subjective interpretations

The useful outcome of an interview is not simply what the student says, but what he has been shown to be thinking. For example, every student thought (at least at one point) that the keyword 'static' meant constant. This was simple – they said what they thought. But sometimes the student's concept is only implicit and there is sometimes a subjective element in identifying it. Several tactics were used to try to mitigate this problem.

One was to seek confirming evidence. An example of this was a student who's answer appeared to imply that a statement such as 'x=y+2;' defined x to be a function of y, rather than a single calculation and assignment. This was confirmed by constructing an appropriate scenario and asking what would happen.

The difficulty with this approach is that the interviewer has to interpret the student's answer in 'real time'. Sometimes this is not possible, and the interpretation only emerges later when the transcript is examined. In these cases similar questions are given to different students; if matching 'wrong' answers are obtained, and if an interpretation explains the response, this is good evidence to support it.

Interpretations which are most general, and so deepest, are both the most important and the most difficult to justify. An example is that novices must use three fictive mental spaces when considering what a program fragement will do. The justification here is three-fold – people in general use fictive mental spaces, they are logically required in this case, and they explain why some program fragments are 'harder' that others.

1.6 Ethical issues

Approval was sought, and obtained, from the Research Committee of the Centre for Lifelong Learning at the University of Birmingham before any work was done with students.

Issues included the anonymity of students, teachers and institutions, awareness and understanding of the situation on the part of students, and possible impact on their study time. Before students were interviewed, the nature and purpose of the work was outlined to them, and if they were willing, they were invited to sign a ‘consent’ form, a copy of which is included as Appendix Four. The case study students, who were interviewed at approximately fortnightly intervals, were asked whether there was any impact upon their progress, and none indicated there was a problem.

In some interviews students expressed ideas which were clearly non-normative – put plainly, they mis-understood OOP in Java. In these situations we moved to a tutorial stance and an attempt was made to improve the students’ understanding.

1.7 An outline of OOP Concepts

It is inevitable that reference will be made to the ideas of OOP, with which the reader may not be familiar. Consequently a brief summary of those ideas is given here.

1.7.1 Before OOP

One aspect of OOP is a way of looking at the structure of computer programs, and it exists as a stage in the historical development of this topic. Early programs before around 1970, written in FORTRAN or COBOL, typically consisted of single blocks of instruction code, with the flow of execution controlled by 'GOTO' statements which switched the flow to instructions other than the next one in the program. As program size grew it became apparent that this approach (commonly called 'spaghetti code' because of the lack of clarity of flow) had serious problems -

- The single block of code implied the solution to the given problem was a single item - which was therefore likely to be a very complex solution. It offered no support for a 'divide-and-conquer' approach of decomposing the problem into smaller and simpler sub-problems.

- Extremely large single blocks are difficult to approach using teams of programmers. As program size grew, the programming time required increased significantly. This implied that it was essential to use a team to write the code - but it was hard to apportion sections of the spaghetti to team members.

- Testing of software ideally requires the analysis of all possible routes through the code, for all possible inputs. As code size grows, the number of routes increases exponentially, so if there are a thousand times as many lines of code, there are a million times more routes to test - an impossible task.

- A problem might well require the solution of several sub-problems. These sub-problems (such as standard mathematical tasks) would also be required in other problems, but there was no way to 're-use' code. In fact this was twisted into the spaghetti as a whole, and could not be extracted and re-used.

The seminal short paper by Dijkstra (1968) signalled the start of what became known as the structured programming movement. He wrote very straight-forwardly

> *The* ***go to*** *statement as it stands is just too primitive; it is too much an invitation to make a mess of one's program.*

The use of GOTOs was avoided by splitting code into parts. The terminology varied between languages, but these parts were known as routines, sub-routines, procedures or functions. A sub-routine could 'call' another sub-routine, which meant that large problems could be successively decomposed into simpler sub-problems. Sub-routines could be apportioned to team members, they could be tested separately, and they could be placed into libraries of code which could be re-used to solve other problems. The idea of sub-routines was one version of the idea of modularity. This is also expressed in different ways in different languages and paradigms, but at heart it means software which is structured into parts which are more or less independent.

Structured programming became the dominant approach from the 1970's, typically in colleges and universities in the language Pascal, and in real-world implementations often in the language C (Kernighan and Ritchie 1988) usually in the context of the operating system UNIX.

However the structured programming paradigm still had problems, one of which was the use of global data. This was data which was accessible throughout all parts of a program. Sub-routines would be called to process this global data - but bugs in a sub-routine might have unintended consequences on this global data, which in turn would mean other sub-routines would fail. Modularity required that interaction between sub-routines be precisely controlled, but global data was a way in which sub-routines would affect each other. Programmers could be given guidelines of good practice to try to avoid this, but what was required was a language which would make the modularity unavoidable. This would happen if somehow the data could be split up, as well as the code, and there was a need to co-ordinate the relationship beween code and data.

1.7.2 The object concept

OOP had its first versions in the work of Ole-Johan Dahl and Kristen Nygaard with Simula, and Alan Kay with Smalltalk. The central notion of OOP was an object, which the programmer could ‘tell’ to do things. More precisely, the programmer could write a program which when executed would result in metaphorical 'objects' being created in memory, and then those objects would be instructed to carry out tasks. Three broad kinds of objects are usually programmed:

- Objects that correspond to real-world things. In a payroll program they might be employees. In a stock control program they might be supermarket products. In a simulation for a shipping company (as written in Simula-67) they would be ships at sea.

- Objects that correspond to computer system components on the screen. These would include windows, buttons, textboxes and so on.

- Objects that correspond to computer system things which are internal and not visible. These include users, files, folders and threads of execution.

Objects possess data. The employee has a payroll number, the button has a position on the screen, the user has an ID and password. However objects also ‘know’ how to do things. The supermarket product knows what happens when it receives delivery of new stock. The window knows how to change its size (in response to the user's request, shown by dragging the mouse). The button knows what to do when clicked. The file knows what to do when it is deleted.

These are all metaphors. The object 'knows' what to do by virtue of having a 'method' coded in the program. The coded method is intimately linked with the object - not simply some code drawn from a library, but written for that type of object alone. The object can be sent a 'message' (another metaphor) telling it to carry out a method.

Beyond the central concept of object, a range of other ideas are the basis for the advantages of OOP, as described next.

1.7.3 Other OOP concepts

Class

Objects come in classes. A class is a type of object. An example is the Button class. Individual buttons on the screen are *instances* of the Button class. When a class is instantiated, this means that an object belonging to that class is made. OOP consists of designing, implementing and using classes.

Encapsulation

This is the idea of enclosing the data within objects in order to ensure that the data is always valid. This is another version of the idea of modularity. In effect we want a class to be armour-plated so it cannot be damaged from outside. Data inside a class should be encapsulated so that it cannot be accidentally corrupted by other parts of the program.

On the other hand, we do sometimes need to find out the values of the data in an object, or possibly to change them. This can be done while still maintaining encapsulation if there is a good mechanism (in the language design) to control this. The process by which this encapsulation is brought about is called *access control.*

Inheritance and polymorphism

This relates to the idea of code re-use. A common situation is where we want something very similar to, but not identical to, some piece of software which we already have, but we want to avoid starting from scratch. This leads to the ideas of inheritance and polymorphism.

Classes exist within hierarchies. For example, there is a standard Java class called Calendar, and a *subclass* of this is GregorianCalendar (a subclass does not correspond to the mathematical idea of subset - a subclass extends a base class). The subclass *inherits* the data fields and methods of the superclass. This means that a GregorianCalendar object can do everything that a Calendar object can do. Another Java example is the Number superclass, which has subclasses of Integer and Double (numbers with decimal parts), among others.

As well as inheriting methods unchanged, subclasses can also over-ride them, which means that they will do the same kind of thing, but in a modified way appropriate to a sub-class. As a real-world example, the super-class of employee might have two subclasses, one for monthly salaried workers and another for the hourly-paid. Both subclasses would have a pay method, but it would operate differently. This idea, whereby we can change the shape of a class, is called *polymorphism*.

Abstract classes

Typically the classes towards the top of a hierarchy are more general in nature than subclasses. Sometimes that generality is such that it does not make sense to make an instance of the general class. An example is the Calendar class. This does not specify how the calendar is organised, and as such it would be meaningless to have a Calendar object. There could be a 'rule' that a programmer should not make a Calendar object, but a better approach is to design the language such that it is *impossible* to do so, in that such a program would fail to compile. To ensure this, Calendar is an *abstract* class, which means it cannot be instantiated.

By contrast GregorianCalendar, which is a sub-class of Calendar, is *concrete* and can be instantiated.

Constructors

A constructor is some program code which 'makes' a new object. The constructor is called when a new object is required. Constructors usually initialise data values to appropriate values. Some languages (C++) also have destructors which do what is needed when an object is no longer required, which most often involves releasing the memory occupied by the object. Java does not have destructors, and it automates the treatment of objects which can no longer be used, in a process called garbage collection.

Overloading

In OOP overloading means having different versions of the same thing with the same name. As an example, the division operator is overloaded. We divide integers by saying

```
x = 10 / 3;
```

and divide real numbers by

```
y = 10.0 / 3.0;
```

These have the same general idea, but they are not the same - x gets the value 3, while y gets the value 3.333.

In some languages (C++) operators can be overloaded programmatically, so that for example we could define a class called Matrix, then overload * as applied to matrices so it would carry out matrix multiplication. In Java operators cannot be overloaded, but constructors and methods can.

For example there are 3 (actually there are more) constructors for the GregorianCalendar class:

- one that makes a calendar with the current system time, and local time zone
- one that makes a calendar with a given time (supplied as a parameter to the constructor), and local time zone
- one that makes one with the current time and a given time zone, such as Pacific Standard Time (supplied as parameter).

1.7.4 OOP languages

The OOP paradigm is expressed more or less clearly in a range of programming languages. Initial OOP languages were rather slow. Between 1983 and 1985 Bjarne Stroustrup (1997) developed C++ as an extension to C with a full range of OOP features added. C++ is a large and fast language which is widely used, most commonly for systems programming. However it is not a 'pure' OOP language, in that global variables and functions (sub-routines) can still be used - and many people write C programs in C++.

By contrast Java (Arnold Gosling and Holmes 2001) was developed by Sun Microsystems (first released in 1996) and is a 'pure' OOP. Java is a general-purpose high level language which is cross-platform, meaning Java programs will run on a wide range of different types of computers - as required for operation over the Internet. During the 1990s many languages incorporated OOP ideas to a greater or lesser extent - examples are Visual Basic, JavaScript and PHP. However this work uses Java since it is one of the cleanest and best-defined versions of OOP principles, and is very frequently taught in undergraduate programs.

Chapter Two - Literature Review - Concepts and Concept Development

2.1 Introduction

This thesis is concerned with concept development, in the context of students learning computer programming. This means the thesis is about concepts, and associated ideas such as meaning and understanding, how concepts relate to each other in structures, and how conceptual structures develop in individuals - thus this literature review considers work relevant to this area.

The idea of concept spans three areas of human intellectual endeavour, namely philosophy, psychology and linguistics. Some of this material is extremely fundamental and slightly esoteric, and might be considered to be irrelevant. In order to maintain some focus, I am establishing what I will call the 'touchstone question', which is a typical question which would be put to a student during the data collection process. The touchstone question is "What does 'static' mean?" All ideas in the literature described here are placed in a relation to the touchstone question. A normative answer to the question is given in the next section.

This chapter starts with consideration of the work of H. P. Grice, since this makes some significant points about the way hearers understand speakers. It then moves on to an idea of Putnam concerning the extent to which meaning is mental, and touches on the formal approach to meaning with Montague grammar. It then moves from philosophy into psychology through an idea of Fodor, and then considers how psychologists have treated the 'concept' idea. It looks at some aspects of the work of Piaget and some approaches which are related to that. It concludes by considering how human beings think of the notion of concept.

Ideas such as frame semantics and conceptual blends are mentioned here, but described more fully in the next chapter.

2.1.1 What does 'static' mean?

For readers with limited familiarity with Java, it might help to provide the normative answer to the touchstone question. The OOP paradigm models *objects* in the real world with machine representations. For example a table (piece of furniture) might be modelled in a Java program by a table *object*. The object would have items of data, such as the dimensions of the table, its colour, material it is constructed from, and so on. The program might need to have several table objects.

A *class* in OOP is a *type of object.* So in our example program we would have a Table class (classes conventionally start with a capital letter). The class definition, which would be in the program code in a file called Table.java, sets out the data fields which an *instance* of the class (that is, a table object) would have. So the class definition would say that a Table object would have a width, depth, height, colour and so on. The program might well also define Chair, Cushion, and Sofa classes.

The values of the data fields of different tables are in general different. The width of one table would not usually be the same as another table. In other words these fields 'belong' to the object, not the class. They are 'per-object' not 'per-class'. Even though every table has a width, the value of the width depends on which table it is. This means the width field is *non-static*. If the program had two table objects, called table1 and table2, we could give them different widths by saying

```
table1.width = 60;

table2.width = 70;
```

The Table class might also have a data field called numberOfLegs. This would have the value of four (for our example we will pretend that all tables have four legs). We could have a non-static field called numberOflegs, and simply assign the value of four to that for all Table objects. But, as well as being wasteful of memory, this also does not capture the idea that the number of legs is 'about' tables in general, not about individual tables. Thus, numberOfLegs is about the class (is per-class), not about objects in the class (which would be per-object). We can achieve this if the numberOfLegs field is declared to be *static*. Then there is just a single value, which would be referred to as Table.numberOfLegs, and we would say

Table.numberOflegs = 4;

Further, classes have *methods* as well as data fields. A method is a piece of code which executes some process. For example the Table class might have a draw method which draws a picture of the table on the screen. The idea of static and non-static applies in the same way to methods as to data fields. The draw method, for example, would be non-static. It makes sense to draw a particular table *object*, in its own colour, but it does not make sense to draw the Table *class*. (Note a class is not a set - the Table class is not the set of all tables. So a class in OOP is not like a class in Mathematics).

Consider this from the student's point of view. It is impossible to understand *static* without understanding the concepts of *class*, *object* and the relationship between them. It is also necessary to accept this and reject the expected meaning of static, namely constant, fixed or not changing - which many students fail to do. This is what makes the touchstone question interesting.

2.1.2 Research on the psychology of computer programming

This began as soon as mainframe computers started to be widely used (Weinberg 1971), and so by now there is a very large body of work. At the end of the first decade of this, Sheil (1981) wrote a review of the psychological studies of programming, and concluded that they had had little impact on professional practice, due to poor experimental design and a shallow view of the nature of programming. Around ten years further on, Soloway and Spohrer (1989) edited a major set of pieces which were focused on the novice programmer, followed by Hoc et al (1990) again on the psychology of programming.

As described above, this chapter is structured along the lines of different theoretical approaches to concepts. Work about the psychology of programming which has an explicit theoretical base will be discussed in the relevant sections.

2.2 Conversational Implicature

This idea is considered first, because it is in the light of this that the empirical data has to be analysed.

Paul Grice was concerned with the philosophy of language, and in particular with the meaning of discourse from a *pragmatic* context - which is to say, discourse with meaning interpreted by the people involved, rather than by some kind of objective logical analysis. Pragmatics is therefore at the other end of the spectrum from *formal semantics*, briefly outlined in section 2.4.

Grice (1981) identified some 'rules of conversational implicature'. The term implicature is used as reminiscent of 'implication', indicating what follows from what is said, but also as being not the same as logical implication. These are sometimes called Grice's Maxims, but are not intended to be guidelines on how to speak well - rather they are descriptions of how listeners usually interpret the meanings of speakers. They are

1. Say what you believe to be true, and only if you have supporting evidence
2. Say as much as you know
3. Be relevant
4. Be brief, orderly and unambiguous

For example, suppose two people are driving through the countryside, the driver announces "We are running out of fuel", and the passenger replies "There is a petrol station in the next village". Then the driver will interpret the meaning of the passenger's reply as follows:

1. The passenger thinks the proposition 'There is a petrol station in the next village' is true, and that the passenger has evidence to support this, rather than just being a guess.

2. The passenger does not know, for example, that the petrol station closed down last year, since such knowledge would have contravened maxims 2 and 3.

3. The passenger knows there is only one petrol station in the next village. Otherwise his statement would have been ambiguous, and he would have said less than he knew.

These maxims lead to highly efficient conversation. For example a mathematician who has three children might say 'I have two children', and be happy that this is a true proposition in logical terms. But in conversation, this is taken to mean what mathematicians usually express as 'I have precisely two children.' Implicature means the conversational form conveys more meaning in a shorter form.

Why is this relevant? Because in formal terms, the question "What does 'static' mean?" is very difficult, because it entails the question of what 'mean' means, and a large part of the Western philosophical tradition has been devoted to answering that. But no student replies by asking what meaning means, and that is because the speaker's meaning is interpreted in pragmatic terms.

For example, if I ask a Romanian "What does 'scoala di soferi' mean?", the hearer will imply that I do not know. However if I ask a student "What does 'static' mean?", an aspect of the context (that I know and understand Java) indicates that I am not trying to find out what 'static' means in Java. Rather it is understood that I want to know what *the student believes* 'static' means, and I am using 'mean' in its common-sense meaning in Gricean terms.

2.3 Putnam - 'meaning just ain't in the head'

The touchstone question is, in practice, interpreted in actual interviews as described above, so that this is in the sense of what the student thinks, and is therefore 'in the students' head' and is a psychological issue. Consequently Putnam's assertion that 'meaning just ain't in the head' seems disturbing.

Putnam's argument (Putnam 1979) is based on (amongst other arguments) a thought-experiment in which there exists a 'Twin Earth', which is in all respects identical with Earth, except that on Twin Earth the liquid which the locals refer to as 'water' is made of XYZ, and not H_2O. Then on Earth there is a person called Oscar, and a molecule-by-molecule equivalent Twin Oscar on Twin Earth. So when the two Oscars use the term 'water', they are in identical mental states, but they are referring to different things. This implies that it is not possible to decide what someone is referring to on the basis of their mental state. Instead we must check whether the actual stuff is H_2O or not. In this way the meaning of water is not a mental state, but an objective truth, namely that it is H_2O.

This argument has been criticised, for example by Searle (1983). The question here is whether this is relevant. The idea is applied to concepts of 'natural kinds'. Kripke (1980) gives water, gold, cats and tigers as examples of this notion. So are things in Java like 'static' natural kinds? There is no universal definition, but Quine (1969), who first introduced the term, used it to refer to a set which was 'projectable', in that a judgement about one element could be reasonably projected to apply by induction to other members. In that sense 'static' is a natural kind, in that we can judge that for any class member which is static it is per class and not per object. And this is indeed meaning which is not in the head - the member is static (or non-static) irrespective of what the student thinks..

But when the touchstone question is asked, the student understands it to mean, in a Gricean manner, "What do *you think* 'static' means?" In other words the student is aware that they might 'get it wrong', and that their belief about the universal and objective meaning of 'static' may be incorrect. So Putnam's meaning 'not being in the head' is not a problem. He is referring to the 'real' meaning of static, which is not mental, and we are interested in what the student *thinks* the meaning of static is, which obviously is.

2.4 Formal semantics - Montague grammar

An alternative approach to meaning comes from formal semantics (Cann 1993, Portner and Partee 2002). This involves

- Establishing a grammar (a set of production rules) which will generate a subset of English sentences

- A set of translation rules such that sentences generated by the grammar can be mechanically translated into expressions in a formal language

- A way of interpreting the meaning of expressions in the formal language.

The interpretation of expressions in the formal language is based on truth-conditional semantics, which takes the meaning of a sentence to be those states-of-affairs under which the sentence is true. For example, the meaning of 'The cat sat on the mat' is those states of affairs in which the proposition *The cat sat on the mat* is true. Adopting a formal language approach to the analysis of natural languages was first proposed by Montague (1974), and such creations are often called Montague grammars.

The advantage with this approach is the clarity and precision with which the analysis of sentences can be carried out by formal means. However that comes with several restrictions. It separates meaning from, for example, the speaker's mental state, and gives a single meaning for every sentence. It is also only applicable to utterances which are assertions, and not interrogatives or imperatives. For example it would not give:

(1) "Do you think it's hot in here?"

any precise meaning, since it is not a proposition, while in fact (1) would typically be understood to mean a request to reduce the temperature in the room. However it has been argued that formal semantics provides a core, propositional sense of meaning, which people elaborate upon in actual conversation.

How does this relate to the touchstone question - "What does 'static' mean?" In one sense none, since the interviewee understands the question in pragmatic terms, as described above. But 'static' is a term used in Java, which is a formal language with an explicit formal grammar. In effect the context of programming with a Java compiler corresponds to a *model* (Cann 1993 page 39), which represents some state-of-affairs in the context of which formal meaning is established. In other words a Java program which contains the word 'static' will behave in a certain way, and this effectively defines a 'real' meaning of static. This then goes beyond 'normative' views of the meaning of 'static', since in practice there may exist a range of such views (for example, see Reddy 2002 for alternative concepts of class and object). Intuitively this corresponds to how the student will interpret the question, as meaning something like "what difference does it make if you use the word 'static' in a Java program", and consequently "when is it appropriate to use 'static'?". Consequently we have a two-level situation:

1. The meaning of 'static' with truth-functional meaning, as determined by the Java compiler, and corresponding to Putnam's meaning not in the head. This relates to the definition-based view of concepts.

2. The students' mental representation of that meaning - which may or may not be a good representation, in that a student may incorrectly predict what a Java program will do when compiled or executed. This relates to the concept image view (Tall and Vinner 1981).

2.5 The philosophy/psychology border - Fodor and the Language of Thought

Fodor’s work is located somewhere on the border of psychology and philosophy - he calls *The Language of Thought* (Fodor 1976) an essay in 'speculative psychology'. He argues that:

1. Folk psychology is correct (discussed below)

2. Cognition is a computational process

3. The representational theory of mind is true

4. Those representations exist within a 'language of thought', which is not a natural language

Folk psychology, or what Fodor calls 'common-sense psychology', holds that thinking beings have desires, and seek to achieve those desires according to a set of beliefs. (Fodor 1987, Fodor 1976 page 27). Cognition is claimed to be concerned with deciding which action among the various possibilities will give the best chance of achieving the desired outcome, and as such is a computational process.

A representational theory of mind treats cognition as operating not directly upon aspects of the real world, which many would think impossible, but on some kind of representation of those aspects. If we equate 'representation' with 'idea', this goes back to Hume deriving ideas from sensory impressions: (Hume 1999 page 98) :

> "We may prosecute this enquiry to what length we please; where we shall always find, that every idea which we examine is copied from a similar impression"

He treats cognition as computation (Fodor 1994 page 7)

> *"I assume that psychological laws are typically implemented by computational processes"*

and that (page 8)

> *"Computational processes are ones defined over syntactically structured objects; viewed in extension, computations are mappings from symbols to symbols; viewed in intension, there are mappings from symbols under syntactic description to symbols under syntactic description."*

(Intension and extension are discussed in 2.6.2). Fodor argues that as soon as we have 'syntactically structured objects', we must have a 'language of thought', since we cannot have syntax without a language. That this is not simply a natural language is shown by the fact that animals and pre-verbal infants can think.

How does this relate to the touchstone question? The folk psychology viewpoint seems tolerable - students have desires to understand, and beliefs in the form of what they have learnt, and sometimes doubts as to the 'accuracy' of what they have learnt. Cognition as computation seems questionable, although his view of concept learning (Fodor 1976 page 34) is interesting:

> *"I think that what concept learning situations have in common is fundamentally this: The experiences which occasion the learning in such situations (under the theoretically relevant descriptions) stand in a confirmation relation to what is learned (under its theoretically relevant description). A short way of saying this is that concept learning is essentially a process of hypothesis formation and confirmation."*

This is said in the context of lab-based experiments of artificial categories (Bruner, Goodnow and Austin 1956), but it can also be seen as fundamentally *constructivist*. Hypothesis formation and confirmation correspond to Skemp's (1983 page 107) 'building and testing'. Constructivism is examined in more depth later on.

Further, a representational theory of mind seems essential when dealing with non-physical items such as Java keywords, and corresponds to the idea of a mental model. The touchstone question might be re-phrased as "Describe to me your internal representation of 'static' in Java".

2.6 Notions of concept from a cognitive psychology viewpoint

Murphy (2002) provides an extensive review of the literature relating to concepts. In this section I will consider ways of looking at the relationships between concepts in terms of the structure of semantic memory, then how concepts themselves might be represented, and finally relate these ideas to the touchstone question.

2.6.1 Models of semantic memory structure

Tulving (1972) distinguished between three types of memory - episodic (memory of events, like a birthday party), procedural (how to do something, like driving a car), and semantic (memory of meaning and understanding). Two basic models for the structure of semantic memory have been proposed.

2.6.1.1 Set-based theories

An early characterisation of concept was as *category*, which can be traced back to Aristotle (Apostle 1980). The key feature of this was a set of defining criteria for membership of the category - for example a bird was something covered in feathers - and so category membership was an all-or-nothing affair. Bruner, Goodnow and Austin (1956) used this as the basis for a series of experiments in which subjects were presented with cards bearing figures (such as two red striped, on one black square) which were members or non-members of a category, and subjects were required to identify what the 'concept' was. They looked at how many examples were required, and what kind of strategy subjects used. This approach lends itself to straight-forward experimental procedures, and so a lot of it has been done - for example Bourne (1970), and Feldman (2000). Unsurprisingly it is found that more complex rules were harder to use, and that conjunctive was harder than disjunctive - but the ecological validity of the 'artificial concept' approach seems doubtful.

By contrast, Meyer (1970) sought to investigate the structure of semantic memory by following a similar set-based approach, but using natural language terms. He measured subject's reaction times to questions asking for the truth-value of statements like "All S are P" and "Some S are P" where S is for example 'Alps' and P is for example 'mountains'. The relevant point here is that meaning was characterised as set membership. Interestingly, the paper does not make clear whether a clarification of what 'some' means was made to the subjects. The logical interpretation is that it means that the intersection of S and P is not null - but the Gricean interpretation would be that S is a proper subset of P. In other words a Gricean interpretation would mean that most people would assert that 'Some Alps are mountains' is false, since in fact all Alps are mountains. Meyer suggested that the data supported the idea that semantic information was recalled in two stages - deciding if S and P are related, then deciding how they are related.

2.6.1.2 Network models

An alternative structure for semantic memory is based on a network, with each node corresponding to a word or concept, and with connections between nodes with meanings like 'is a' or 'has a'. An early example of this is Collins and Quillian (1969). There they first sketch out how semantic information might be stored in a computer as a network, and then suggest the same might be true in human brains. They provide reaction time data to support this - primarily based on the possibility that the time to answer a question like 'Does X have property Y' would depend on the minimum number of links through the network from node X to node Y. Rumelhart, Lindsay, and Norman (1972) amongst others also use a network model.

2.6.2 Concept representation

Distinct from how semantic memory is structured is the question of how individual concepts are represented - in other words, what is an appropriate mental model of a concept. Some alternative approaches proposed in the literature are discussed next. Much of this uses the terms concept and category interchangeably, but Murphy and Medin (1985) offer a helpful distinction:

> *We use concepts to refer to mental representations of a certain kind, and categories to refer to classes of objects in the world.*

This corresponds to the philosophers' distinction between the *intension* of a concept (the meaning or idea or definition) and the *extension* (the set of things to which the concept refers). For example, the intension of bird is 'animal with wings and feathers' and the extension is the set of all birds. It also corresponds to two ways of defining a set in mathematics - by giving a predicate (like X = { x | x is an even number less than 12 }) or by enumeration (X = { 2,4,6,8,10})

2.6.2.1 The feature list approach

Going beyond a simple definition approach, one possibility is the feature list idea. For example, the 'robin' concept might have features such as bipedal, has wings, has feathers, flies, and has a red breast. Smith Shoben and Rips (1974) review how the feature list idea has been used to account for many aspects of language. In particular, Lakoff (1973) suggests that some features are *characteristic*, whilst some are *defining*. His evidence is based on *hedges*, which are qualifying phrases such as 'technically speaking' and 'loosely speaking'. For example, 'Technically speaking, a chicken is a bird', because it has a defining feature (feathers) but lacks a characteristic feature (flies). Again, 'Loosely speaking, a bat is a bird', because bats have the characteristic feature of wings, but lack the defining feature of feathers. Lakoff argues that 'Technically speaking, a robin is a bird' sounds odd, since robins have both defining and characteristic features, and so the hedge is not required.

2.6.2.2 Typicality

Wittgenstein (1958 page 66) showed that for some concepts (his famous example was 'game'), there were no defining features which applied to every instance. Instead there was the idea of 'family resemblance' - for each instance there was at least one other instance which had a lot in common with it (soccer and rugby), but there were many pairs of instances which had very little in common (water polo and chess).

In the absence of defining features, only characteristic features are left. Rosch and Mervis (1975) developed this idea to consider if it might be possible to rank features in terms of *how* characteristic they were. They demonstrated this experimentally by showing close agreement between subjects in judging how characteristic the features of instances of furniture were. Ripps Shoben and Smith (1973) also supported this notion. A development of this was the idea of *prototype* (Smith and Medin 1981). The idea of this was that if people were making judgements of typicality, there had to be a perfect 'ideal' with which they were making the comparison, and this was termed the prototype. Another variation is the notion of *exemplar* (Medin and Schaffer 1978). They proposed that concepts consisted of recalled instances of observed examples of the concept. Their evidence was derived from experiments with artificial categories of geometric forms of different size and colour.

Does the exemplar approach relate to an educational context? It is a common practice when introducing a new idea to provide examples, but there is a distinction between the *idea* of the concept (see section 2.6.2.4) and the examples provided. But this does raise the point that if students are constructing their own idea of the concept on the basis of examples (along the lines of Fodor's hypothesis testing and Skemp's 'build and test') then the examples must be very carefully chosen, for otherwise students might use inappropriate aspects of the examples.

2.6.2.3 Category construction

Many experiments in this area are designed on the basis of subjects trying to choose between (usually artificial) categories constructed by the experimenter. However an interesting area of work involves looking at how subjects will construct categories of their own. For example, Medin, Wattenmaker and Hampson (1987) carried out a sequence of experiments where subjects were given a number of items and asked to partition them into 2 sets. The idea was to see if these would be chosen according to Rosch's family resemblance notion - but in fact the sorting was what they called uni-dimensional. Figure 2-1 illustrates this:

Example	D_1	D_2	D_3	D_4
1	1	1	1	1
2	1	1	1	0
3	1	0	0	0
4	0	1	0	0
5	0	1	1	1
6	0	0	0	0
7	0	0	1	0
8	1	0	1	1
9	0	0	0	1
10	1	1	0	1

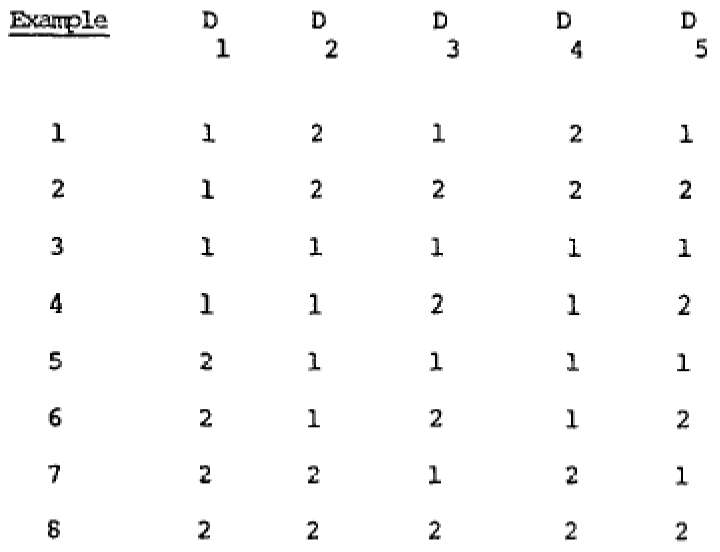
<u>Family Resemblance Sort</u>								<u>One-Dimensional Sort</u>							
Category A				Category B				Category A				Category B			
<u>Dimension</u>				<u>Dimension</u>				<u>Dimension</u>				<u>Dimension</u>			
D_1	D_2	D_3	D_4	D_1	D_2	D_3	D_4	D_1	D_2	D_3	D_4	D_1	D_2	D_3	D_4
1	1	1	1	0	0	0	0	1	0	0	0	0	1	0	0
1	1	1	0	0	0	0	1	1	1	1	1	0	1	1	1
1	1	0	1	0	0	1	0	1	1	1	0	0	0	0	0
1	0	1	1	0	1	0	0	1	1	0	1	0	0	1	0
0	1	1	1	1	0	0	0	1	0	1	1	0	0	0	1

Figure 2-1 Family and one dimensional sorts

D_1 to D_4 are 4 attributes, and a 1 means the example had that attribute, and 0 means it did not. A possible family resemblance sort is shown, into one set containing the examples which had all or all but one attribute, and the other having none or just one attribute. A uni-dimensional sort is also shown, with the partition depending solely on whether the example possessed attribute D_1 or not.

This was carried out in several ways, using cartoon-like drawings of animals and insects, and descriptions of people with differing personality traits. Almost all the subjects used a uni-dimensional sort, not family resemblance. This was still the case when the examples were set up to discourage this. For example with insects with short medium and long tails, they were split into long and short, and then the medium were split into longer than average and shorter than average.

However when there was some correlation between the attributes, subjects were more likely to use a sort based on that. For example, with the examples in figure 2.2,



<u>Example</u>	D_1	D_2	D_3	D_4	D_5
1	1	2	1	2	1
2	1	2	2	2	2
3	1	1	1	1	1
4	1	1	2	1	2
5	2	1	1	1	1
6	2	1	2	1	2
7	2	2	1	2	1
8	2	2	2	2	2

Figure 2.2 Correlated dimensions

attributes 2 and 4, and 3 and 5, are correlated. Subjects would sort on the basis of these, and especially if there was some 'theory' to explain the correlation. For example in one experiment the examples were patients and the attributes were disease symptoms.

Attributes like 'dizzy' and 'ear ache' were correlated, and might both be explained by an ear infection - in which case subjects used these as the basis of sorting. Similar family resemblance sorts were obtained if subjects were given a hint as to the basis of the categorization, such as 'things that can fly' or 'introverted people'.

The uni-dimensional sort is like Aristotle's definitional idea of category. It has the advantages of simplicity and precision, and it is the approach people will take, unless they are given correlated attributes or suggested concepts to sort on.

But consider this in relation to the touch-stone question. If students observed instances of static and non-static in this way, they would arrive at a normative meaning of the term. But learning Java takes place in the real world, not in the psychology lab, and students bring with them the ‘met-before’ meaning of static, that is, constant.

However the idea of a concept as something which is coherent and 'makes sense' leads on to the next section.

2.6.2.4 The theory theory

The approaches of simple definition, family resemblance, typicality, prototype and exemplar all assume that category members have, in some sense, something in common. Murphy and Medin (1985) challenged this view. They gave several arguments against the similarity view, and instead :

> *We propose that concepts are coherent to the extent that they fit people's background knowledge or naive theories about the world.*

Medin and Wattenmaker (1987) also took this view., as does Neisser (1989). Lakoff (1982, 1987a, 1987b) extended this approach, using the idea of an 'idealized cognitive model' (ICM) as how many concepts are represented. He used the example of 'bachelor', and suggested the ICM for this was as an unmarried male in an 'idealized' setting. Then a real-world example would be more or less typical to the extent that the particular circumstances corresponded to the idealized ones. The Pope, for example, would not be regarded as a typical bachelor since his circumstances meant he was expected to be celibate. Lakoff lists four characteristics of ICMs:

- They are structured wholes, in a gestalt manner. Thus the concept 'weekend' has to be related to the idea of 'work week'.

- The categories in ICMs are natural categories (like 'bird' as having typicality, so a robin is a typical bird and a penguin is not) not classical categories (like 'bird' defined as a creature with feathers)

- ICMs may contain image content in the wide sense, that is, not just visual image (concept image again)

and very significantly, so I quote verbatim (page 49 in the 1982 paper):

-ICMs provide holistic ways of 'framing' situations, where a 'situation' is taken to be an otherwise fragmentary understanding of either the real world or some imaginary or fictional world.

and he then says 'The word "understanding" here is crucial'. Trying to understand 'static' within the situation of programming, framed by the ICM of OOP, seems to correspond to this precisely. This is elaborated in the next chapter.

2.6.3 Concepts and the touchstone question

How do these ideas relate to '"What does 'static' mean?" We can construct a definition of 'static' - *Class members with the attribute 'static' apply per-class and not per-object.* This means the category of *static* has sharp boundaries and is an all-or-nothing affair (see 2.1.1 above for a more extended exposition). However the definition depends entirely on the concepts of *class* and *object*, and in turn this places it into a network of OOP concepts.

Can we apply typicality to *static*? Possibly. It is possible to identify a prototype use of static, namely to make class members per-class not per-object - as in the 'definition'. However there is a slightly less typical use - in the implementation of the *singleton* *pattern*. This is an OOP design pattern which seeks to model a situation where a class logically has precisely one instance. An example might be a class which modelled the computer system the application was running on (there is such a class, called System). One way to implement this in Java is to make all members of the class static, and to make its constructor *private*. That means the constructor cannot be accessed from outside the class, so that no instance can be made. In effect the class can be used as if it were a single object.

An even more non-typical use of static is to re-use the OOP class idea to create a module, in the sense of having a way of dividing up program code into separate units. This idea of modularity became important when applications became larger - software which comprises of a million lines of code cannot be handled as a single unit, and must be somehow structured into smaller modules to make it more manageable.

The actual way this is done varies between languages. In Java we can treat a class as a module, by having everything static and no constructor, but the idea is different from the singleton pattern where we want to have precisely one object, since a module has nothing to do with objects. An example is the Math class, which contains constants like Math.PI, and methods like Math.sin(). Having a Math object makes no sense, but having a module into which all code related to mathematics is placed does make sense. But this is as far from the typical use of static as a penguin is from a typical bird.

2.7 The Piagetian heritage

Jean Piaget wrote copiously, but it has been said that (Boden 1982 page 165)

> Those who require precision above all else will have learnt long since not to seek it *chez* Piaget.

Despite, or perhaps because of this, an entire library of work has derived from his ideas, some intended for academics, some for practising teachers. His recurring themes are well-known - a set of developmental stages and sub-stages, and a cluster of ideas treating a person as a biological system in homoeostasis, using the notions of assimilation, accommodation and schema. The following sections will look at these ideas and their heritage, in the light of the touchstone question.

2.7.1 Formal operational thought

Given that we are concerned with adults, the validity of the Piagetian stages is irrelevant. However the assertion that adults have and use the approaches to cognition known as 'formal operational thought' ought to be relevant. A Piagetian operation is a transformation of objects into a new state, with the operation being reversible and internalised. Concrete operations concern physical objects, while formal operations are about abstract non-physical objects.

Significant parts of Piaget's formal operational stage are the 16 binary operations resulting from all possible combinations of the p.q, p.not q, not p.q and not p.not q where p and q are propositions, and the INRC group. As usual his description (1953 p. 382) of this is not totally clear, but he seems to mean as follows:

I N R and C operate on a compound proposition involving two 'basic' propositions p and q (call it f(p,q)), transforming it into another compound proposition

1. I is the identity: I f(p,q) = f(p,q)

2. N is logical negation : N f(p,q) = not f(p,q)

3. R replaces p by q and q by p : R f(p,q) = f(q,p)

4. C is negation and exchange : C f(p,q) = not f(q,p)

The group table is (column is first operation, but group is commutative)

	I	**N**	**R**	**C**
I	I	N	R	C
N	N	I	C	R
R	R	C	I	N

I N R C

C C R N I

In *The growth of logical thinking* (Inhelder and Piaget 1958) an argument is presented that adolescents achieve greater reasoning ability by patterns of thought which reflect the INRC group. This is based on interviews with children of a range of ages concerning the various pieces of apparatus. However this formal mathematical approach to cognitive reasoning has attracted significant criticism, such as:

> "What fails to make logical sense can hardly be said to make psychological sense in a study of intellectual development" (Parson 1960 page 78)

> "Numerous investigators have found Piaget's propositional logic based on sixteen binary operations wanting both theoretically and empirically" (Lawson 1979 page 71)

> "His logical principles look unsatisfactory; his generalizations seem defective; and his basic categories and theorizing in this area appear to be devoid of empirical interpretation" (Ennis 1978)

On this basis, Piaget's algebraic conceptualization of formal thought will not be used here.

2.7.2 Piagetian schema

Piaget frequently uses the term schema, but appears to use it to mean different things at different times. Montangero and Maurice-Naville (1997) identify early motor schemas as developments of reflexes, representational schemas as knowledge structures, and operatory schemas as methods or ways of proceeding. A loose commonality is the notion of mental structure, and there is some correspondence between schemas, Lakoff's ICMs (2.6.2.4) and Fillmore's frames (3.5).

2.7.2.1 Schemas and programming research

In this loose sense the touchstone question sits very well with the notion of schemas. The novice programmer must acquire a logical succession of novel schemata which include the following:

1. Second-order action - the programmer tells the computer what to do, rather than doing it themselves
2. The use of a formal language
3. Consideration of the interaction between program text, machine state and input
4. The OOP web of class/object concepts

Various attempts have been made to mitigate these daunting tasks. One is the 'objects-first' approach, which was first described in Kölling and Rosenberg (2001), and Kölling, Quig, Patterson and Rosenberg (2003) describes BlueJ, a programming environment to support this. The basis for this is that students who first learn procedural programming find difficulty in later acquiring the OOP schema, and so the more effective approach is that OOP is learnt first. Another initiative using objects-first is Griffiths et al (2007), who devised a teaching approach which used a micro-world to develop OOP concepts before requiring students to use full Java syntax. The basis of this was the assertion that OOP concepts are meaningful and significant, so that storage of these in semantic memory would be stable, while syntax is largely arbitrary and contingent and so unstable in memory.

There is a thread in the programming research literature which connects problem-solving and 'schema'. This starts with the approach to human problem-solving of Newell and Simon (1972), which conceives of a problem space, a target state, and the identification of a sequence of state transitions to move from the initial state to the target. In the context of programming, Spohrer, Soloway and Pope (1989) suggest a 'Goal/Plan analysis' of programs, without actually referencing Newell and Simon but using a similar approach. Soloway and Spohrer further suggest (1989) that many bugs are not due to students' mis-understanding of program constructs, but are caused by faults in the 'plan' underlying the program. Soloway, Ehrlich and Bonar (1982) suggest that experienced programmers have implicit knowledge of successful plans (also known as schemas), an example being the 'running total' plan. They relate such things to the 'scripts' of Schank and Abelson (1977), and suggest that experienced programmers identify appropriate plans and use them, something which novices cannot do. Rist (2004) uses the term 'schema':

> *"A schema defines a set of slots or variables that are linked together in a coherent structure"*

which is therefore very reminiscent of Minsky's frames (1974), and says

> *"A plan schema is an abstract solution to a common programming problem; it stores a plan to achieve a common goal."*

How does this relate to the touchstone question? Much programming research has concerned procedural plans (of action), and static is an aspect of class design, which is more concerned with structure than process. However we can think of static as corresponding to 'an abstract solution to a common programming problem', namely the one of modelling a situation in which an aspect of a type (class) is required rather than an instance of a type (object).

Götschi, Sanders and Galpin (2003) reported on a study of students' mental models of recursion, which were in effect a schema. In the Götschi paper the term 'mental model' is taken to mean 'how recursive procedures execute'. The study was based on an analysis of answers ($n \approx 600$) to examination questions. An initial list of models was used (Kahney 1985) , and this was added to as students' answers were coded. Thus a phenomenographic approach was used, though it was not described as such. On the basis of this it was possible to see that those with a 'non-viable' model of recursion (around 50%) had problems dealing with flow of control, parameter-passing and return values.

Another recent thread in the programming literature is the idea of 'roles of variables', which is a type of schema. Ehrlich and Soloway (1984) suggested that programmers have a repertoire of 'variable plans', stereotypical uses of variables in programs, and that part of a plan was a 'variable role', such as a counter or a running total. They argued that plans and roles were part of the tacit knowledge which programmers had, and which students usually had to acquire themselves by studying examples.

Sajaniemi (2002a) took up this idea and analysed 109 novice level programs written by experts. He identified nine roles which accounted for 99% of all variables used. The roles were: constant, stepper, follower, most-recent holder, most-wanted holder, gatherer, one-way flag, temporary and organiser. He also described (Sajaniemi 2002b) some software which produced an animation of executing code including graphics displaying variables in roles, and Kuittinen and Sajaniemi (2003) reported an improvement in program comprehension and writing skills when roles of variables were taught explicitly. Ben-Ari and Sajaniemi (2004) confirmed that CS educators identified the roles of variables with at least 85% agreement. Byckling, Gerdt and Sajaniemi (2005) present evidence to suggest that these roles are also appropriate for OOP in Java.

2.7.3 Constructivism

I take constructivism in an educational sense to mean that a student learns something by mentally constructing it for themselves, in contrast with a traditional view of education in which teaching consists of the attempt to transfer what is in the teacher's head into the students' head.

Constructivism seems to have two roots, one philosophical and one psychological. The philosophical one concerns the problem of 'being sure' about our knowledge of the world, which is a thread which runs from the scepticism of the pre-Socratics through Descartes, Kant, Husserl and so on (for example Johnson 1978, Keeling 1968, Dennett 1991). This is the basis of von Glasersfeld's Radical Constructivism (von Glasersfeld 1984, 1991), but it is concerned with the nature of knowledge *as such*, rather than how human beings get to understand things.

However the psychological root lies with the work of Piaget, in texts such as *The Child's Construction of Reality* (Piaget 1955). In fact Piaget rarely explicitly discusses constructivism. However he provides overwhelming evidence that the child seems to think of the external world in a radically different way from the (modern Western) accepted version, such as Piaget 1973 page 100:

> *Does the wind know its name?*
> *Yes.*
> *Why?*
> *Because it makes it windy.*
> *Does the Rhône knows its name?*
> *Yes, because it is it that is the Rhône.*
> *Is it alive?*
> *Yes, because it flows into the Arve.*

This difference between child and adult perceptions of the world implies a representational theory of mind, and inevitably it must be the case that these representations are mental constructions of individuals. That is, the construction is due to the person as opposed to external reality, and not that it is independent of culture, since there exist animistic cultures in which the adult view corresponds to the child's view quoted above. Steffe has written extensively on this constructivism, (such as 1991) in the context of mathematics education.

Constructivism has at least three other versions. One is the *social constructivism* of Lev Vygotsky. This shares with Piaget the idea of the student's construction of the world, but emphasises the importance of *activity*, and the *social setting* - in other words, the teacher and fellow students. Vygotsky also introduced the term 'zone of proximal development' (ZPD) - the range from what students can do without help, to what they can do with help..

A second is *constructionist learning*. This is due to Papert and Harel, based on a 1987 NSF grant "Constructionism: A New Opportunity for Elementary Science Education", and joins the idea of the construction of mental models with students making things (typically physical objects, but also digital items) in the real world, leading on to Mindstorms (Papert 1980), promoting children learning mathematics by writing programs in Logo. And thirdly, there is *social constructionism*, which is a movement within the field of social psychology (Burr 2003) which sees the personality and identity of individuals being the result of societal forces.

In relation to the touchstone question, there is a clear possibility that students construct their own meanings for 'static'. This is supported by the fact that students offer a variety of interpretations of 'static'. It also explains this, which is otherwise paradoxical, in that students will have been told in textbooks and lectures that 'static means per-class and not per-object'. But for students who do not have a clear understanding of the difference between class and object, this makes little sense, and they are obliged to construct their own meaning for 'static'.

2.7.3.1 Constructivism and the psychology of programming

The most explicit reference to constructivism in computer science education is Mordechai Ben-Ari's paper with that title (1998), which argues that this theory of learning would be a useful basis for assessing teaching and research.

Lui et al (2004) describe an approach to helping weak students in a first programming course, using constructivist theory. They offered a model of the situation, shown in Figure 2.3:

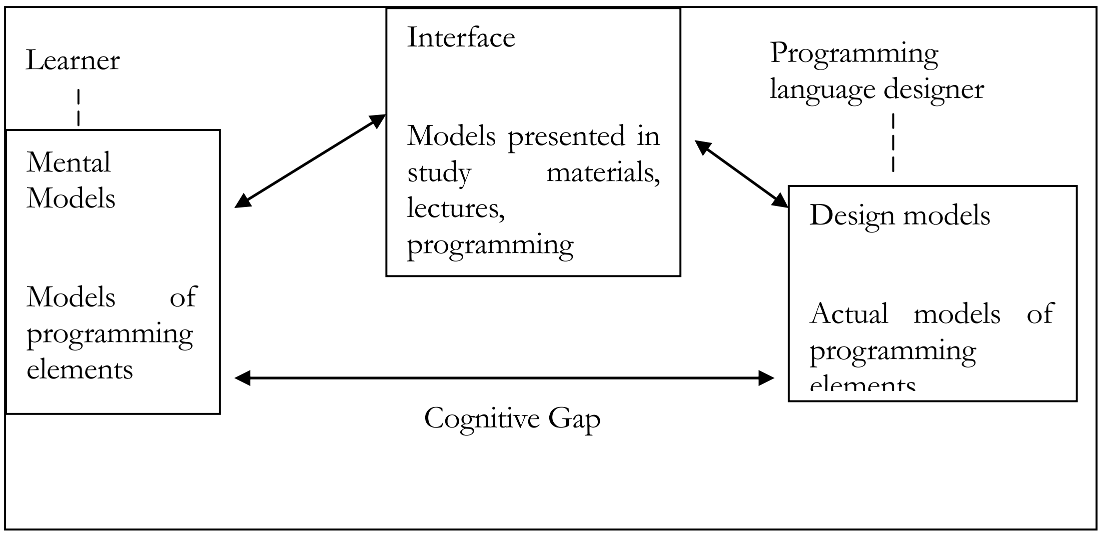

Learner
Interface
Programming language designer
Mental Models
Models of programming elements
Models presented in study materials, lectures, programming
Design models
Actual models of programming elements
Cognitive Gap

Figure 2.3 Cognitive Gap

They identify six guidelines:

1. Show the behaviour of new ideas, and avoid analogy. They give the example (due to du Boulay (1989)) of a variable as a number in a box, and point out that the analogy gives some students the ideas that a variable can have two values at the same time, or that when a variable value is used, the box is emptied. Instead they suggest providing many examples, so that students can create, test and improve their mental models of the new idea. See point 3 below.

2. Build concepts first, then introduce jargon. Griffiths et al (2007) echo this.

3. Repetition of examples, in order to provide the student with opportunities to test the viability of their mental models.

4. Checking readiness for concept construction, in the sense of ensuring students have viable models before trying to construct further ones which require knowledge of them.

5. Memorising a repertoire of key programs or program fragments. This is very unusual, with rote learning usually being poorly thought of. The basis of this guideline comes from the Suzuki method of learning a musical instrument, which was one of the bases of the approach.

6. Starting by writing programs on paper. They felt that weaker students struggled with the tasks of writing code, compiling and executing it, and that this additional load distracted from understanding program code. This generalises to the question of what environment is optimal in the early stages of writing programs. Issues here include exposure to the fundamental ideas of compilation, and balancing simple environments (such as a basic text editor) with a complex IDE (like Eclipse or NetBeans), and the additional tasks of learning a basic environment and then progressing to learning a sophisticated one, even though the sophisticated one might provide for largely automating routine tasks (such as import statements) enabling the student to focus on new concepts.

2.7.4 APOS

Another example of Piaget's legacy is Dubinsky's APOS (action-process-object schema) theory. This provides a range of ways of conceiving mathematical items (Dubinsky 1991, Brown, DeVries, Dubinsky and Thomas 1997, Asiala, Cottrill and Dubinsky 1997) . For example, seeing a function as a recipe for evaluating it treats it as an action. Seeing it as a mapping from one set to another sees it as a process. And seeing a function as an element of a set of functions sees it as an object.

Can we see OOP concepts such as 'static' in these three ways? Bogart (2009) observes that

> Students tended to blur the distinction between a class and an object, and they'd try to call a method, when they had not instantiated an object yet, and of course get an error message. They'd try making the method static, which in turn caused other errors, requiring them to make other methods and member variables static as well. They'd basically end up with a totally static class that they used as a singleton.

So the novice without clear concepts of class and object tries to invoke a method without an object. The compiler's error message tells them this should be a static method, so the student declares it as a static method, and this may solve the problem. In this situation, 'static' is an action - something you do to make the program work, without knowing why.

We can see 'static' as an APOS process from the viewpoint of a student who is designing a particular class, and wants to establish a member as pertaining to the class as a whole and not to an instance of the class. To see 'static' as an APOS object means separating it from any particular class design, and thinking of it as a possible attribute of any class member.

2.8 Tall's ideas

David Tall has been concerned with the development of concepts, primarily in a mathematics context, from the nineteen seventies through the first decade of the 21st. century. Two key elements of his current point of view are 'set-befores' and 'met-befores' (Tall 2011). The set-befores idea recognises that students think using brains which have been shaped through biological evolution. He identifies three features of this genetic heritage - recognition of patterns, repetition of physical, mental and joint processes, and the use of language. In turn these offer ways in which concepts can be constructed - by categorisation, encapsulation, and definition. Met-befores are previously-experienced perceptions, of sensation, pattern or concept, which are features of the setting in which new ideas emerge, but which may form 'epistemological obstacles' which block the assimilation and accommodation of new ideas.

Tall also uses the idea of the three worlds of mathematics. One is the 'conceptual embodied' world. This is different from the idea of embodiment as proposed by Johnson (1987) and others - they focus on personal embodiment, the experience of the infant in a body - but here the embodiment is initially the perception of things in the world, moving towards imagined, mental representations based on those perceptions. This is similar to Piaget's sensori-motor schemata and their development.

A second world is 'procedural/proceptual symbolic'. Here symbol is not used in the Peircean sense of an arbitrary sign. He illustrates this with an example of a word-problem like 'Amy is three times older than her brother, and in four years..', which is re-written as a linear equation, which is then solved, and the result again re-written in the form of 'So Amy's age is..'. During the solution, the terms in the equation cease to have significant referents in the real world - and consequently are not Peircean signs.

The third world is the 'axiomatic formal' world, an approach in which explicit axioms are used, and deductive logic is used to generate theorems.

At first sight these three worlds might be seen as constituting a simple progression as the student learns, and that the worlds are disjoint. However figure 2.4, from Tall 2009, shows the breadth, complexity and subtlety of the idea. Firstly the worlds are not disjoint. For example in the formal embodied intersection we have the Euclidean conception of geometry, based on what resemble axioms, and relating to actual geometric figures, but without symbols. And in the embodied symbolic intersection we have algebraic geometry, where in the Cartesian manner algebra is used to manipulate information from geometric

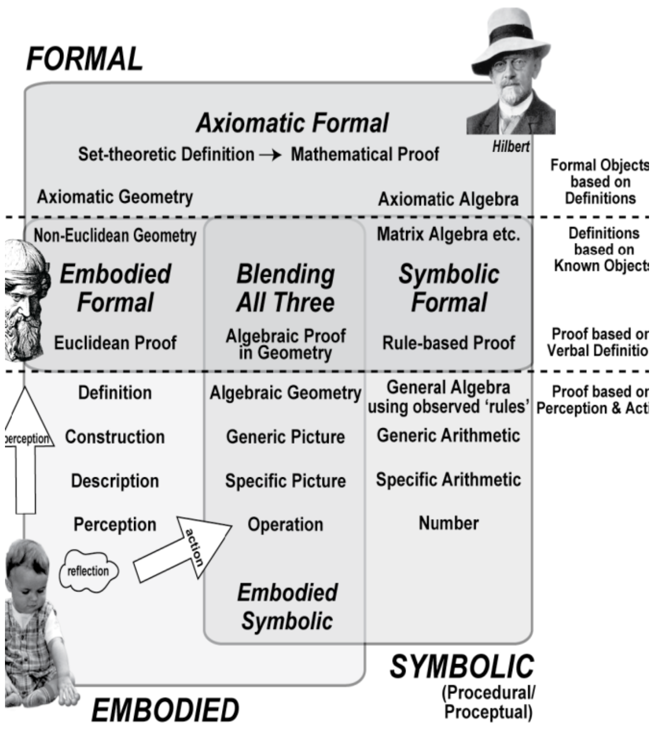


Figure 2.4 The Three Worlds

figures, but without an axiomatic basis.

The second striking aspect of this is the child in one corner and Hilbert in the opposite corner. This relates the development of an individual through their lifetime to the cultural development of mathematics over the centuries.

This framework is specifically mathematical. How does the touchstone question relate to it? The normative view of OOP would place it in the embodied/symbolic overlap. OOP is about models of real-world objects, structured into types (classes), and this is clearly a mental representation of reality, including physical entities (such as employees) and 'virtual' ones (such as bank account transactions). Corresponding to that mental representation is a symbolic one, in the form of Java program code. The concept of 'static' is one aspect of that symbolic embodiment. There is an 'axiomatic formal' aspect to this, in the definitions of Java as a formal grammar, but novices typically do not become involved with this.

However as already noted, many students do not acquire this normative view of OOP in a straight-forward manner. A possible reason for this is the epistemological obstacles caused by 'met-befores'. This in effect is the basis for the 'objects-first' pedagogical approach, which argues that an initial grounding of procedural programming establishes ways of thinking (in particular the separation of data and procedures) which constitute met-befores which hinder the establishment of the ideas of class and object.

2.9 Applied approaches - Ausubel, Biggs and Marton

Biggs (1999 page 11) writes:

> 'Learning has been the subject of research for the whole of the twentieth century, but remarkably little has directly resulted in improved teaching. The reason is that until recently psychologists were more concerned with developing the One Grand Theory of Learning than in studying the contexts in which people learned, such as schools and universities.'

Studying students in the psychology lab is a lot cleaner and simpler than studying them in the complex ecosystem of a learning and teaching institution - but it is likely to be less valid. This section considers three important and related dimensions to this situated approach.

Ausubel (1960) makes the assumption that 'cognitive structure is hierarchically organized in terms of highly inclusive concepts under which are subsumed less inclusive sub-concepts and informational data'

and hypothesises from this basis that students would learn more about a subsumed topic if they are first presented with an 'advance organiser' which places the topic in the context of the more inclusive concept, and he supports this with some empirical data. Since this first introduction of the idea, advance organisers have become widely used, criticised and misused. For example the DFES (2009) offers the following advice to teachers:

'To support lower-achieving pupils you might: .. use an 'Advance organiser' to reduce the amount of material a pupil has to consider.'

which clearly treats the advance organiser as an overview in the form of a simplification, rather than placing the topic in a more abstract context. Nevertheless Ausubel's original version is interesting when compared with SOLO, considered shortly.

Ference Marton made two significant and related contributions to this area. One was phenomenography, which Marton (1994 page 42) defines as:

'the empirical study of the differing ways in which people experience, perceive, apprehend, understand, conceptualise various phenomena in and aspects of the world around us'

The paper by Eckerdal and Thuné (2005) used this phenomenological approach. Rather than studying students' *mis*conceptions, they sought to identify the different ways that students conceived of class and object, by means of semi-structured interviews with 14 students. For objects, they found students had 3 ways of thinking of them:

1. object is a piece of code
2. as above, and in addition as something active in the program
3. as above, and in addition an object is seen as a model of some real world phenomenon.

The second contribution of Marton was an early outcome of the use of this approach (Marton and Säljö 1976), which was the idea of surface and deep approaches to learning. In turn this was taken up by Biggs (1999) starting with the idea of a 'learning outcome' as a way of characterising a taught module. This was developed into SOLO, which classified learning outcomes into five levels - pre-structural, uni-structural, multi-structural, relational and extended abstract. We can see that a deep approach to learning relates to attempting to move up these levels, and this hierarchy connects with Ausubel's hierarchies of concepts. Biggs also developed the idea of constructive alignment, which accepted the importance of constructivism in learning, and the alignment of teaching and learning experiences to target learning outcomes.

Another approach in this applied manner is that of threshold concepts. The idea of threshold concepts was first suggested by Meyer and Land (2003). They indicated that threshold concepts were the key building blocks of a subject, and that they had distinct characteristics:

- Transformative - changing how things are seen
- Integrative - relating different topics
- Irreversible - hard to unlearn
- Counter-intuitive - and so possibly troublesome to learn
- Boundary marking - relating to the boundary of the subject.

This idea was applied to geography and economics, and was first introduced into CSE by a poster (McCartney and Sanders 2005), which was followed by an identification (Eckerdal et al 2006) of two threshold concepts - *abstraction* and *object-orientation*. In the same year Vagianou (2006) suggested another, which she called 'Program Working Storage', which might be generalized to the idea of *state*. Eckerdal et al (2007) also used this idea as a backdrop to a study of students transitioning through partial understanding of computing concepts.

2.10 Reddy and the Conduit metaphor

Michael Reddy (1979) suggested that language can be seen as a conduit, by means of which ideas are transferred from person to person. He supported this with example such as

You still haven't *given* me any idea of what you mean.

Try to *pack* more thoughts into fewer words

Never *load* a sentence with more thoughts than it can hold.

There are two parts to this. One is that language can be seen (metaphorically) as a conduit. The second is that ideas, concepts or meanings are metaphorically thought of as objects. Lakoff and Johnson(1983) support this with examples like

I *gave* you that idea.

The point is that the English language community thinks of concepts *metaphorically*. While philosophers and psychologists have proposed explicit descriptions of the concept idea, most people do not use such ways of thinking. The implication is that the concept idea is difficult and abstract, and so is thought of in metaphorical terms.

If this is accepted, it would then be unreasonable to suppose that it would be possible to describe and explain the development of concepts in a given domain in simple, explicit, precise and definite terms, given that we can only grasp the idea of concept metaphorically. Rather we should only expect to be able to give a description which is impressionistic, organic and essentially human.

CHAPTER THREE - A cognitive basis for meaning

3.1 Introduction

This chapter sets out a sequence of related ideas which form a sketch of how people think. It broadens the notion of concept, discussed in the previous chapter, to encompass the ideas of meaning and words. It is on the basis of this that the data will be analysed.

The chapter starts with a problem, namely how any symbol-system can obtain actual meaning. It offers a solution to this, in the form of embodied cognition, and develops this on to metaphor as a way of grasping concepts above the purely concrete. The notion of a frame as a structure to hold meaning is described, followed by the idea of a mental space, a zone of awareness typically structured by a frame. Structures built out of mental spaces, namely conceptual blends, are outlined, and one class of blend, a compression, is exemplified.

3.2 The Symbol Grounding Problem

Harnad (1990 page 1) describes the symbol grounding problem as follows:

> *How can the semantic interpretation of a formal symbol system be made intrinsic to the system, rather than just parasitic on the meanings in our heads? How can the meanings of the meaningless symbol tokens, manipulated solely on the basis of their (arbitrary) shapes, be grounded in anything but other meaningless symbols? The problem is analogous to trying to learn Chinese from a Chinese/Chinese dictionary alone.*

If we have an English/Chinese dictionary we might be able to learn Chinese, but how did we get to learn English?

This is often related to Searle's Chinese Room argument. This thought experiment assumes that a computer program has been written which can pass the Turing test, in Chinese. In other words it receives input in Chinese, and responds in Chinese, and observers cannot tell whether this is done by a computer or a human. Then Searle uses this program, carrying out its instructions manually on the basis of a print-out of it, to also receive input and generate output. But Searle does not understand Chinese. He argues from this that there is more to cognition than just computation.

3.3 Embodied Cognition

Embodied cognition provides some answer to the symbol grounding problem. Anderson (2003) gives a 'field guide' to the idea. He outlines the distinction between the Cartesian dualism of the mind and body, and the Heideggerian idea of 'being in the world'. He contrasts embodied cognition with cognitivism:

> *Simply put, cognitivism is the hypothesis that the central functions of mind - of thinking - can be accounted for in terms of the manipulation of symbols according to explicit rules.*

Cognitivism is therefore the computational theory of mind. Anderson describes the embodied cognition project as follows

> *This project calls for detailing the myriad ways in which cognition depends on - is grounded in - the physical characteristics, inherited abilities, practical activity, and environment of thinking agents.*

The sub-title of Mark Johnson's 'The Body in the Mind' (Johnson 1987) is 'The bodily basis for meaning, imagination and reason' and this gives a good summary of the idea. Johnson develops the idea of an 'image schema' as a cognitive structure which is grounded in physical experience but which can be extended to abstract domains. For example, the In schema is a type of Container schema. We have a landmark, a locating feature, which in the case of In is the interior of the container, and a trajector, something which might move. So in

The milk is in the glass

the interior of the glass is the landmark, and the milk is the trajector. But in

Stalin is in power

Stalin is the trajector, and the power is thought of metaphorically as a container which you can be in (or out of).

It is possible to see the embodied aspects of these image schemata in Piaget's sensori-motor schemas.

3.4 Metaphor

Lakoff and Johnson (1980) argue that all conceptualisations, including those of abstract ideas, are thought of as metaphorical versions of other concepts, and that is demonstrated in language, both lexically and structurally. For example the metaphor of war for argument is exemplified by:

Your claims are <u>indefensible</u>.

He <u>attacked</u> every weak point in my argument.

His criticisms were right <u>on target</u>.

I <u>demolished</u> his argument.

and so on. Thus concepts are metaphorical extensions of other concepts, and at the base of the hierarchy we have a set of conceptual metaphors founded in bodily experience in early childhood - so-called 'primitive metaphors'. Lakoff (2008) argues that this has a neuronal reality. He gives the example of the 'up is more' metaphor resulting from countless situations where a child sees liquid poured into a container and the level rising. He claims this results in the simultaneous excitation of brain areas registering verticality and increasing quantity, and that the consequent circuit produced as being the physical form of the metaphor. Current work (such as Rapp 2004) does show some correlation between localised brain activity and metaphor processing.

3.5 Frame

Many concepts only have meaning inside a related set of ideas - a frame. The term 'frame' has been used by many people, sometimes with significant variations in meaning. The ideas of Minsky, Fillmore and Langacker will be outlined, and the idea will also arise in discussions of CIN.

Minsky (1974) described his version of a frame as follows:

> *When one encounters a new situation (or makes a substantial change in one's view of the present problem) one selects from memory a structure called a Frame. This is a remembered framework to be adapted to fit reality by changing details as necessary.*
>
> *A frame is a data-structure for representing a stereotyped situation, like being in a certain kind of living room, or going to a child's birthday party. Attached to each frame are several kinds of information. Some of this information is about how to use the frame. Some is about what one can expect to happen next. Some is what to do if these expectations are not confirmed.*

This arose in an AI context and might be described as the AI version of frame, quite different from the way others have used the term. However he does look at the idea in a range of situations, including human and computer vision, language and understanding, and memory. He views Piaget's concrete operations as direct frames, and formal operations as frames operating on frames. The title of the 1974 paper is 'A Framework for *Representing* Knowledge' (italics added) and so clearly lies in the Cognitivist camp.

The idea of *frame semantics* is primarily associated with Fillmore (1982, though this is hard to obtain, and is reprinted in Evans, Bergen and Zinken 2007). In this paper Fillmore gives a 'personal history of the concept'. He started by considering sentences forming a 'frame' like

John is Mary's husband ___ he doesn't live with her

and looking at what happened when words like BUT, YET, MOREOVER, HOWEVER, AND and OR were slotted into the gap in the frame. He then moved through considering syntactic frames, as was the Chomskian fashion at the time, into 'case frames' which combined syntactic and semantic structure, and 'case grammar', into frame semantics. The most commonly quoted example is the COMMERCIAL EVENT frame, which has slots of BUYER, SELLER, GOODS and MONEY, plus probably many more. Knowledge of this frame means that

John bought the car for a good price
delivers the meaning that the price was low, while

John sold the car for a good price
means the price was high. The meaning of 'good' depends on knowledge of the BUYER and SELLER roles in the COMMERCIAL EVENT frame.

Here is what Fillmore writes in the 1977 paper:

> *By the term 'frame' I have in mind any system of concepts related in such a way that to understand any one of them you have to understand the whole structure in which it fits.*

This seems very relevant to the situation of a student trying to understand OOP.

Similar to Fillmore's frames are Langacker's *domains*. He writes (1987 page 147):

> *All linguistic units are context-dependent to some degree. A context for the characterisation of a semantic unit is referred to as a domain. Domains are necessarily cognitive entities : mental experiences, representational spaces, concepts or conceptual complexes.*

He gives as an example of a domain the concept KNUCKLE, and suggests it has at least 3 components in its domain matrix:

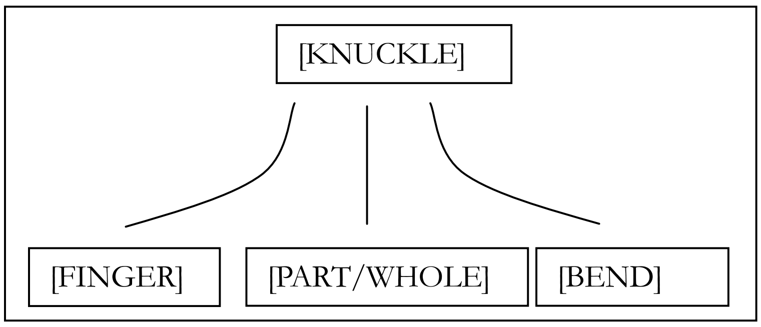


Figure 3.1 Concept domain

We therefore have a hierarchy of domains. Langacker argues that at the foundation of the hierarchy are basic domains, defined to be one which cannot be reduced to further components. He suggests that many of these are directly sensory, and suggests that [COLOUR] and [TEMPERATURE] are examples. Any domain which is not basic he calls an abstract domain. The basic domains are perhaps the foundational embodied concepts of Lakoff and Johnson.

Croft & Cruse (2004 page 15) describes domains as being identical with Fillmore's frames, although Evans & Green (2006 page 230) identifies 4 differences - which are largely matters of emphasis. Clearly frames and domains are very similar.

3.5.1 The profile-base relation

Langacker (1987 page 183) also introduced the terms *profile* and *base*. He gives as an example the CIRCLE domain as a base, with ARC 'profiling' a component of it, namely a section of the perimeter. Similarly RADIUS profiles another aspect of CIRCLE, namely the distance from the centre to the perimeter. Another example is the KINSHIP domain as a base, with UNCLE profiling one aspect of that. Profile-base relations are discussed in relation to OOP in the next chapter.

3.6 Mental Spaces

In the preface to his work of that title, (1994) Fauconnier writes that:

> *..in order for thinking and communicating to take place, elaborate constructions must occur, that draw on conceptual capacities, highly structured background and contextual knowledge, schema-induction, and mapping capabilities. Expressions of language do not in themselves represent or code such constructions - the complexity of the constructions is such that the coding, even if it were at all possible, would take very large amounts of time and be extremely inefficient. Instead, languages are designed, very elegantly it would seem, to prompt us into making the constructions appropriate for a given context, with a minimum of grammatical structure.*

and again:

> *This fundamental property of language is counterintuitive: in our folk theory, it is the words that carry the meaning - we 'say what we mean',..*

If this is true, this is of key significance for learning and teaching. The folk theory is that the words of the teacher carry the meaning 'at first hand', and that on that basis the student has to do no more than to hear in order to learn. But of course the folk theory is wrong.

Fauconnier's idea of mental spaces arose as a consequence of dealing with problems of referential opacity. For example, suppose Jack is the son of Philip, and also, secretly, the leader of the Black Brigade. Then consider the two sentences

Philip believes *his son* is a genius, and

Philip believes *the leader of the Black Brigade* is a genius

These do not mean the same thing. In 'reality' the two terms in italics refer to the same person - but not in terms of Philip's beliefs, and so the meaning is different. This demonstrates the fact that words do not simply carry meaning, but only once refracted through the mental space elicited by the words.

In The Way We Think (Fauconnier and Turner 2002 page 102) they write that

> *..mental spaces are small conceptual packets constructed as we think and talk, for purposes of local understanding and action.*

However the key point is that what we think and talk *about* is not simply our physical environment, nor even our 'mental' environment which is in some sense objective, but rather that they are personal constructions which have varying degrees of 'fictive' nature, ranging from pure hypothesis, through possibilities, to counter-factuals, and to things such as paintings or novels, where the fact that they are 'not true' is not important. Fauconnier claims that frames structure mental spaces - in other words that some mental spaces are thought of as being instances of a frame, so that the elements and relationships in the space acquire roles according to the 'slots' in the frame.

Relationships between mental spaces are the concern of conceptual blending.

3.7 Conceptual Blending

Fauconnier and Turner (2002) use the term conceptual integration network (CIN) to describe related spaces, and describe several topologies of CINs. A common feature of CINs comprises of some 'input' spaces yielding an 'output' space which contains some selected elements of the inputs, and producing emergent structure in the output - in other words, something more than a simple summation of the projected elements. Their claim is that such blends enable individuals and communities to be creative and imaginative and to conceptualize beyond basic concrete ideas.

For example, consider

If I were you, I'd hate myself

This was first discussed by Lakoff (1996). To start with, an objectivist view of this makes no sense at all - if I am actually you I would have all of your characteristics, and so would only hate myself if you hated yourself - in which case I would not have said it, and nor is this what most people would take the sentence to mean. The difficulty is what 'I' and 'myself' refer to - again a problem of referential opacity.

However we can satisfactorily model this if we invoke a blend in a fictive mental space:

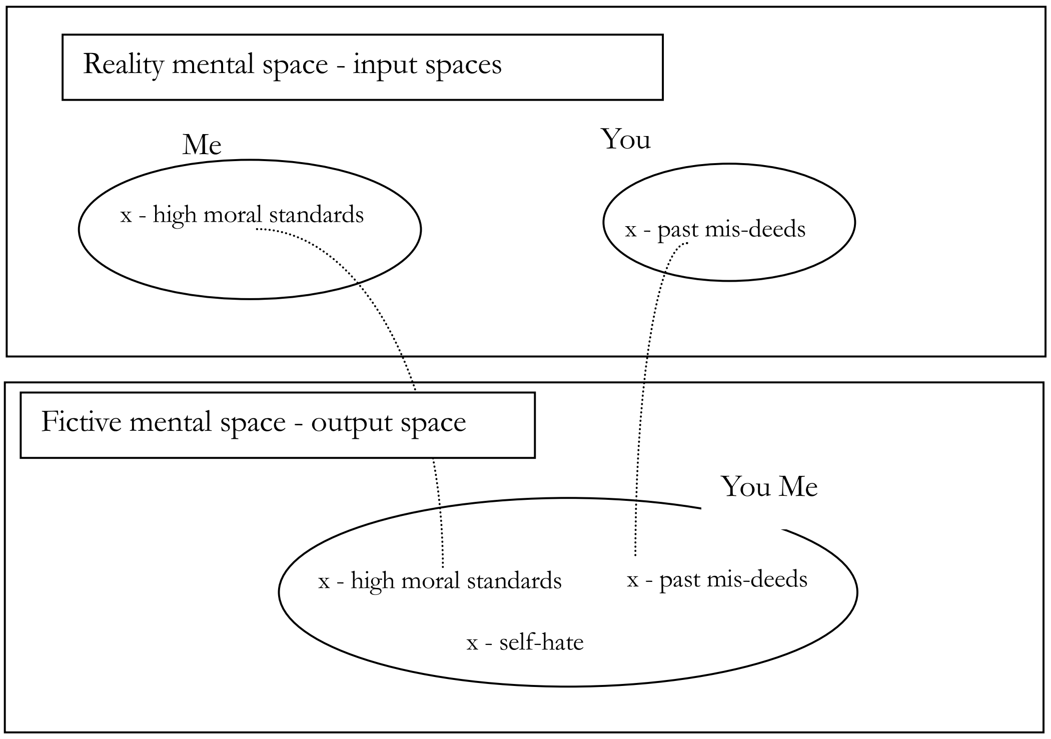


Figure 3.2 If I were you

Selective projection of elements in the input spaces produces an emergent structure in the output space, namely the self-hate consequent on the high moral standards and past mis-deeds.

However the sentence

If I were you, I'd hate me
requires a slightly more complex model:

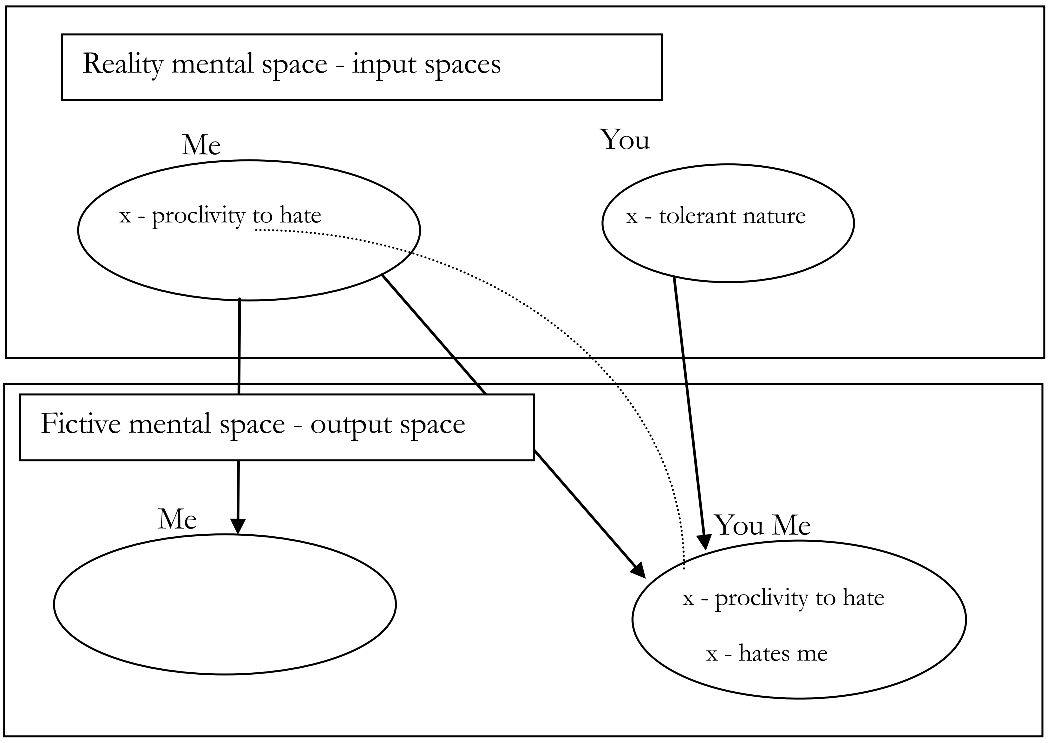


Figure 3.3 If I were you I'd hate me

Here the fictive space has both the original 'me', who is the one who is hated, and the you-me blend, who does the hating. This is isomorphic with McCawley's (1993) example

I dreamt that I was Brigitte Bardot and that I kissed me.

A pair of examples from Quine (1960) discussed by Johnson-Laird (1983 page 60) in the context of conditionals is effectively seen in terms of selective projection:

If Caesar had been in command in Korea, he would have used the atom bomb.

and

If Caesar had been in command in Korea, he would have used catapults.

Both of these have the same pair of input spaces:

1. Caesar and Roman military action:
1a Caesar
1b. His aggression
1c. Ancient Roman weapons

and

2. Korea
2a. Modern weapons
2b. Korean war

In the first case, we project into the blend 1a, 1c and 2b. In the second case, we project 1a,1b, 2a and 2b.

This also illustrates that a conceptual blend is not like a food blender. It does not merge elements together. It preserves the selected elements, and when they are put together, new emergent content results.

3.8 Compressions

Thurston (1990) wrote that

> *Mathematics is amazingly compressible: you may struggle a long time, step by step, to work through some process or idea from several approaches. But once you really understand it and have the mental perspective to see it as a whole, there is often a tremendous mental compression.*

This version of the compression idea seems to involve the encapsulation or packaging of a lot of thought and process into a single notion. However in terms of CIM, compression has a different meaning. A compression is one kind of CIM, in which a set of related items are replaced by a single one. The items are related by what Fauconnier calls a 'vital relation', such as analogy or dis-analogy. Some examples given by Fauconnier (2008) follow:

My tax bill gets bigger every year

we have a set of tax bills, one each year. These are analogous to each other, in that they share certain properties. This analogy is compressed into an identity - just *one* tax bill. However there are also dis-analogies - the amount increases between them. This dis-analogy through time is compressed into a change - getting bigger.

The second example is an illustration of the theory that dinosaurs evolved into birds. the illustration shows five animals - the first a dinosaur chasing a dragonfly, three intermediate forms, and a final modern bird. This contains several compressions. Firstly millions of individual dinosaurs are compressed (on the basis of analogy) into a single animal - and similarly for the other forms. But between these forms there are differences, and these dis-analogies are compressed into change. Further, what are in reality a very large number of intermediates over a very large length of time are compressed into just five animals. And finally those five are compressed into a *single animal*, which changes as we look at it.

Thirdly

Heidegger is very hard to read. Only Heidegger understands him.

Here the first Heidegger is a compression (by analogy) of the works of Heidegger, identified metonymically (the part of the author being used for the whole of the work). The second Heidegger refers to the individual. Similarly for

Chomsky found himself on the top shelf.

In

He saw himself in the mirror.

the man and his image are compressed by analogy into a single identity. But it is not compressed into uniqueness - the man and his image are not the same.

CHAPTER FOUR - Concepts, Programming and Teaching

A running program is a kind of mechanism and it takes quite a long time to learn the relation between a program on the page and the mechanism it describes.

du Boulay (1989 page 285)

4.1 Introduction

The previous two chapters outlined some ideas about understanding meaning in general. The purpose of this chapter is to consider those ideas in the particular context of computer programming, the teaching of programming, and the normative understanding of programming concepts. The following ideas are examined:

- definition-based concepts
- family resemblance concepts
- prototype and feature list concepts
- exemplar-based concepts
- category construction
- embodied cognition
- metaphor
- frame
- profile-base relationships
- mental space
- conceptual blend

The chapter has two parts. The first looks at some lecture slides from a first course in programming. This material has the advantage that it is physical evidence which literally shows how OOP concepts are normatively held and publicly displayed. The second part examines some key programming concepts, such as variable and function, from a cognitive point of view.

4.2 The teaching of programming

4.2.1 A case study approach

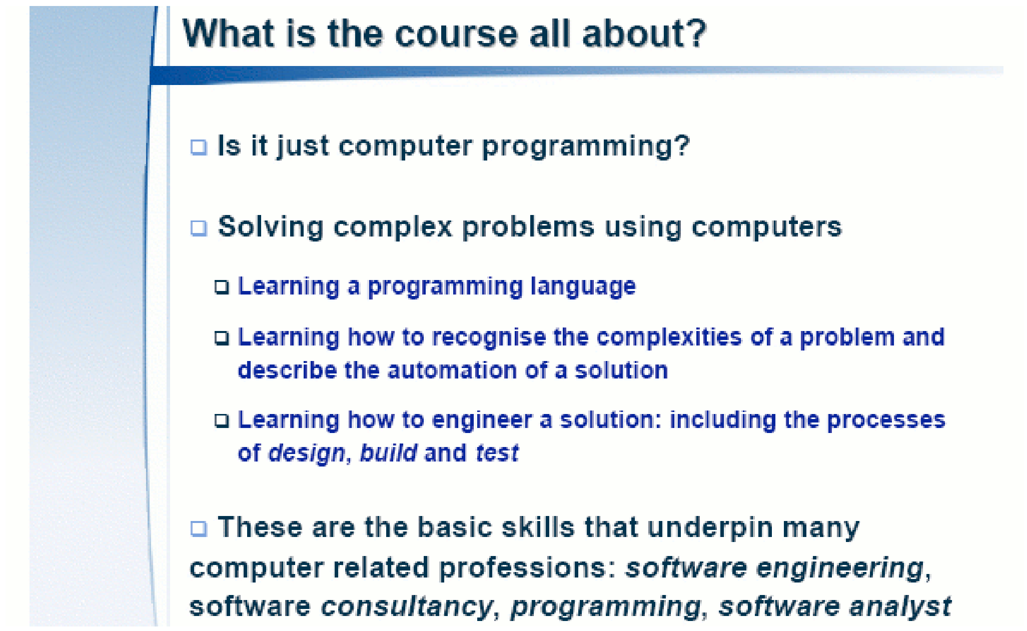
Teaching is a practical, not a theoretical, activity, and so this needs to be related to an actual course. Java is the most common first programming language in courses across the world from high school level and beyond, but a wide variety of approaches are used, dimensions of variation including the following

- an 'objects first' approach or progression through procedural programming first
- the use of different programming environments, such as a simple text editor and command line, BlueJay, or a professional IDE such as NetBeans
- the presence or absence of a consideration of 'problem-solving' in addition to language syntax and idiom.

Consequently it is not possible to identify a 'typical' course. In fact a case study approach is used, and the examples are drawn from a set of lecture slides from a module called 'Programming for Computer Scientists' delivered in 2006. This module took place at a university with an excellent research reputation, to students with strong academic backgrounds.

4.2.2 Lecture discourse

Typically a lecture involves something like a traditional blackboard, a more modern whiteboard, a PowerPoint or similar computer display. This provides a mediation for the discourse - in other words, a set of symbols to which teacher and students will refer to as the context of what the teacher is saying. For example, figure 4.1 is the first slide from the first lecture.



What is the course all about?

- Is it just computer programming?
- Solving complex problems using computers
  - Learning a programming language
  - Learning how to recognise the complexities of a problem and describe the automation of a solution
  - Learning how to engineer a solution: including the processes of *design*, *build* and *test*
- These are the basic skills that underpin many computer related professions: *software engineering*, software *consultancy*, *programming*, *software analyst*

Figure 4.1 First slide

4.2.3 Definitions

Programming languages are formal and have explicitly-defined syntax, but this is not usually presented in first courses. Neither is denotational semantics, which is an attempt to provide formal statements of what language features 'mean' in terms of what they 'do'. However we can find many items which are close to a definition. For example, the following are extracts from early slides:

Comments : Notes to the programmer

Reserved words : Part of the Java high-level language

Statements : Perform some action, terminated with a semi-colon

Classes : Programs representing some particular activity

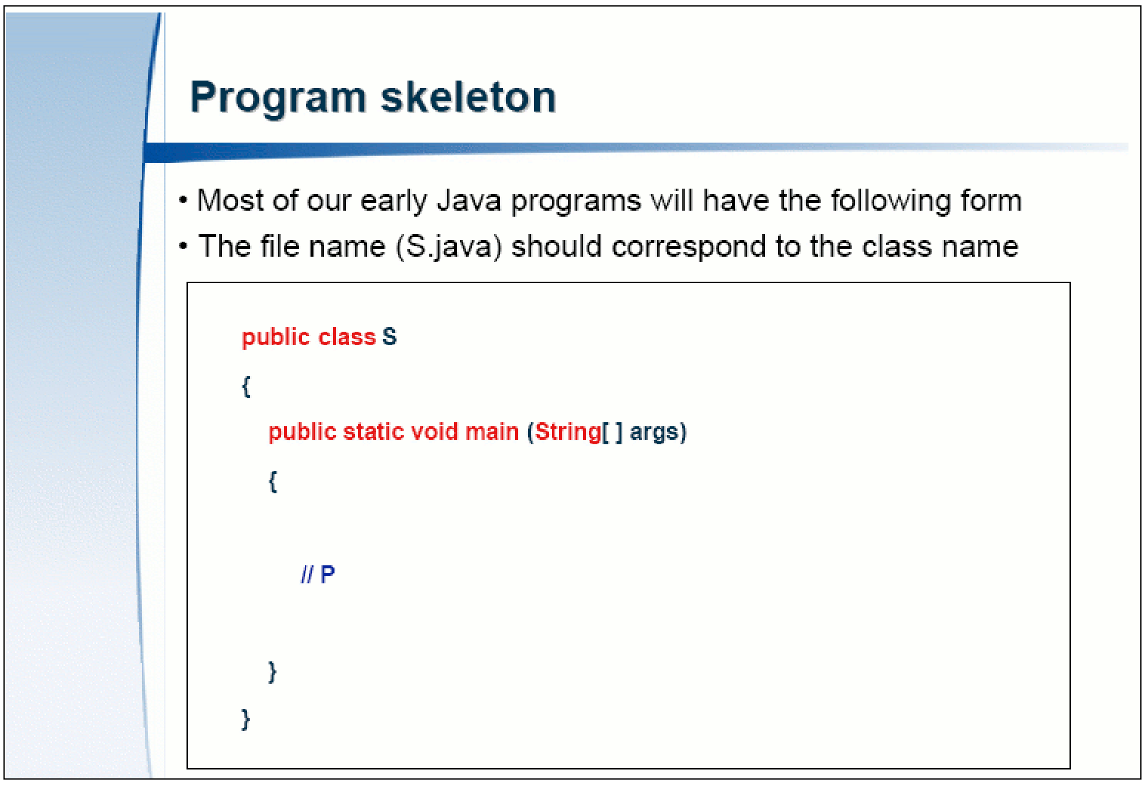
These are all supported by at least one example. These are not logical definitions, in the sense of setting out necessary and sufficient conditions. For example, being a 'Note to the programmer' is not a sufficient condition for being a comment in a program - they may be written on PostIt notes, and in the Extreme Programming paradigm they often are. But these pseudo-definitions are concise phrases which try to encapsulate the idea in a few words.

The last one above is interesting. It is on slide 42 of 48 in the first lecture, so is very early on in the course. It is quite vague, and an OOP expert would question its correctness. The normative idea is that a class is a type of real-world object modelled by the software, which is clearly not the same as a program, nor some particular activity. Because of the way that Java operates, (which is somewhat arbitrary), each public class must be defined in its own separate file. Consequently a typical Java 'program' or application will make use of several classes, each in its own file. But the very first Java programs will be often be chosen to be as simple as possible, so they will have just one class, and so be in just one file. Consequently 'a class is a program' is true in this limited context - but this is only 'accidentally' true. The problem is that some students may learn from this that what 'class' actually means is 'program' - a mis-conception which has been observed (for example Eckerdal and Thuné (2005) found some students thought a class was a 'piece of code').

But this is an example of one of the classic dilemmas of teaching Java - a full exposition of the meaning of *class* is impossible at this early stage, yet the word must occur in every Java program, however simple, and so there is some obligation to 'explain' it. We can see this as an example of cyclic base-profile dependency. The term class only makes sense in relation to ideas of object and the rest of the OOP paradigm, and so it cannot make sense in lecture one. The solution adopted here is a common one - class is established with a vague 'holding' meaning, which is elaborated and made precise later on.

These items can be seen as establishing a vocabulary in order to be able to talk about things. For example the term 'statement' is introduced, so that it is then possible to refer to 'that statement' rather than the clumsy 'this section of code ending with the semi-colon', or 'this line of code', the latter being not always one statement. However the idea is also being introduced as well as the vocabulary. So we have the triad of the sign, namely the word, its intension or the idea or meaning of the sign, and examples are usually given as samples of the extension of the sign, as the set of things it refers to.

4.2.4 Prototypes of concepts



Program skeleton

- Most of our early Java programs will have the following form
- The file name (S.java) should correspond to the class name

```
public class S
{
   public static void main (String[ ] args)
   {

      // P

   }
}
```

Figure 4.2 Prototype

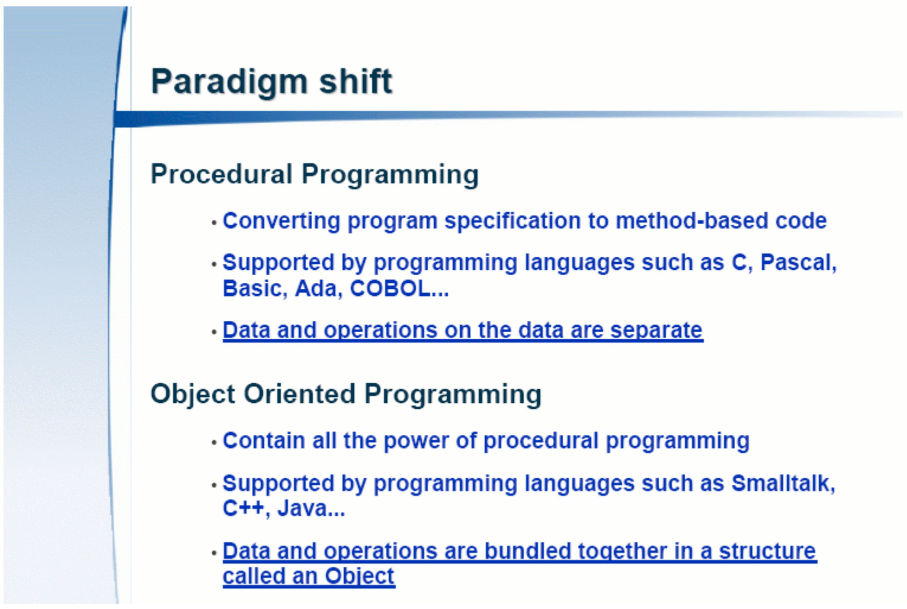
A possible example is shown in figure 4.2

This is possibly a prototype, in that it implies a feature list expected of a class definition, which at this stage (slide 5 of lecture 2) is termed a 'Java program'. Those features are -

- starting 'public class'
- followed by a {, with a matching } at the end
- then a 'public static void main'
- then a matching pair of { and }, which enclose the code

It also carries the characteristics of a prototype, in that the features are typical and expected, but not mandatory. This is indicated by the '*Most* of our *early* Java programs..', indicating that some may not be like this, and that later ones will be different.

But it could be questioned as to whether this is intended to provide a category-constructing prototype at all. In other words the idea is not to enable the student to be able to make a judgement of a piece of code which they encounter as to whether it is a Java program or not, so it is not about category construction. Rather it is presented to help the student write a Java program. The idea is that if you want to write a Java program, you first write this skeleton, then put in the code to do what is required. This is therefore to



Paradigm shift

Procedural Programming

- Converting program specification to method-based code
- Supported by programming languages such as C, Pascal, Basic, Ada, COBOL...
- Data and operations on the data are separate

Object Oriented Programming

- Contain all the power of procedural programming
- Supported by programming languages such as Smalltalk, C++, Java...
- Data and operations are bundled together in a structure called an Object

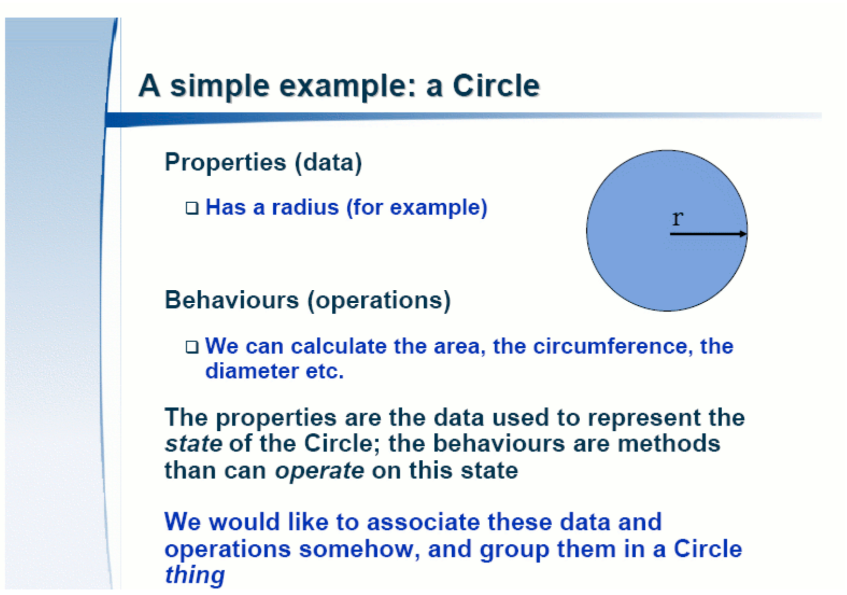
Figure 4.3 Definitions

support the skill of writing Java, rather than 'understanding' something. For example the question of why curly brackets are used, not square ones, is not addressed, since it is not significant - the answer is that this was inherited from C and C++, but this is immaterial to these students.

4.2.5 Introduction of the object concept

This happens in lecture 9, at a key point in the course, after ideas of procedural programming have been presented. The introduction shows the combination of definition, prototype and exemplars.The 'definition' is shown in figure 4.3

The device of underlining serves both to focus on the important sections, and also to emphasise the contrast between them. The central point is that an object binds data and operations. The next few slides seek to clarify the distinction between data and process, and then there is an example, shown in figure 4.4:



A simple example: a Circle

Properties (data)

- Has a radius (for example)

Behaviours (operations)

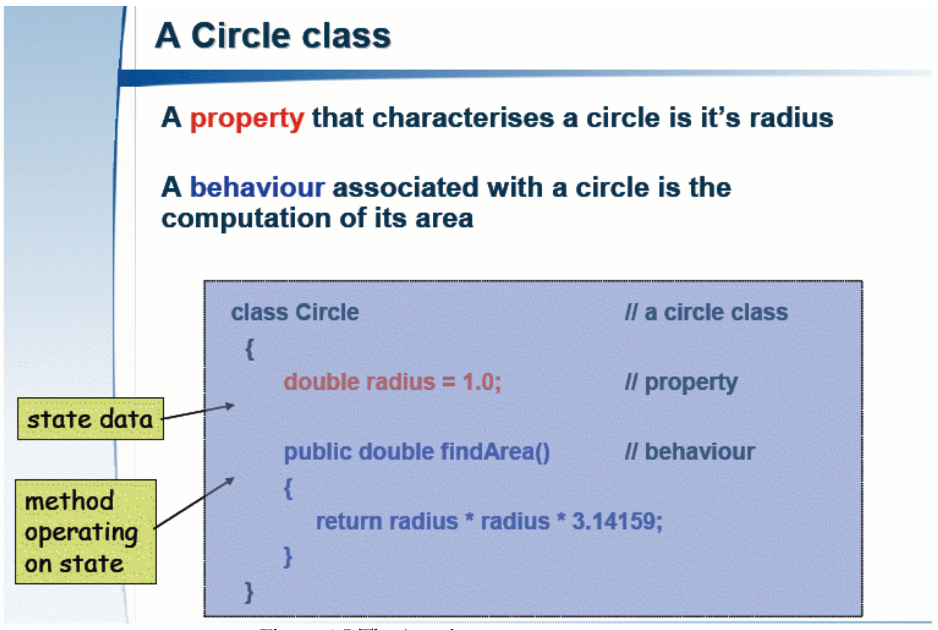
- We can calculate the area, the circumference, the diameter etc.

The properties are the data used to represent the *state* of the Circle; the behaviours are methods than can *operate* on this state

We would like to associate these data and operations somehow, and group them in a Circle *thing*

Figure 4.4 Example

Is this an example as it claims, or, since it is the first, is it a prototype with a feature list of data and operations? It is followed by a repetition of the ideas of data and behaviour, and



A Circle class

A property that characterises a circle is it's radius

A behaviour associated with a circle is the computation of its area

```
class Circle                              // a circle class
 {
     double radius = 1.0;                 // property

     public double findArea()             // behaviour
     {
         return radius * radius * 3.14159;
     }
 }
```

state data
method operating on state

Figure 4.5 The ‘state’ concept

also a first look at how these are coded in Java (figure 4.5)

This slide raises two major issues. First is the notion of *state*. A simple idea of state could be simply the values of the various data items in a program, together with which point through a program execution has progressed to. This corresponds, in the mental space structure described below, to the *machine state space*. However this is not a machine-wide state, but just the state of one instance of the class - that is, the radius of one circle. This is not a simple idea.

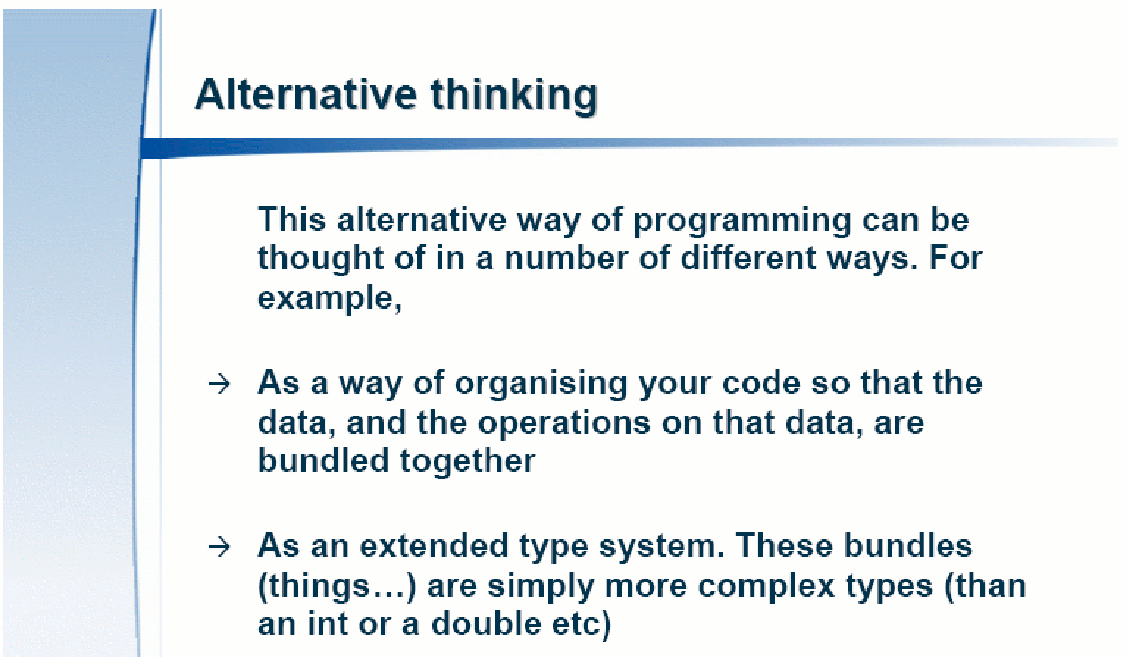
The second issue is the limited validity of the OOP class-object-method metaphor. The object in the machine is a model of the object in the real world, and a method is something the object does. Then the program is like the script for a play, where the author instructs the actors (the objects) to do various things. The metaphor struggles because most of the objects modelled by Java are inanimate, and cannot do anything. For example, there is no sense in which real circles find their area. This is something *people* do, manually, and the program code enables the computer to do it automatically. So the typical code sequence:

```
Circle myCircle = new Circle(5); // make a circle radius 5

double area = myCircle.findArea();
```

looks like we are telling the *circle* to find its area. This is often called message-passing, sending a message to an object to do something (which in itself is part of the metaphor that an OOP object is a person, and can understand messages). In fact we are telling the *system* to work out the area of the circle.

There is one type of object where this is not the case, namely what might be called an automation object. One example might be a thread of execution (the standard Java class Thread does this). Another would be something like a robot or an 'autonomous software agent'. In these situations, Java is not modelling an external object - it is creating an original internal object. For these, a method is not a model of what the object can do - it is the actual content of the method. For example the Thread class has methods to start execution, sleep (suspend execution for a given time), yield (stop momentarily and let other threads run) and so on. These do not *model* the behaviour of a Thread, they *are* the behaviour. However this breaks the OOP metaphor that objects in programs are models of external objects.

The next slide is also interesting (figure 4.6) :



Alternative thinking

This alternative way of programming can be thought of in a number of different ways. For example,

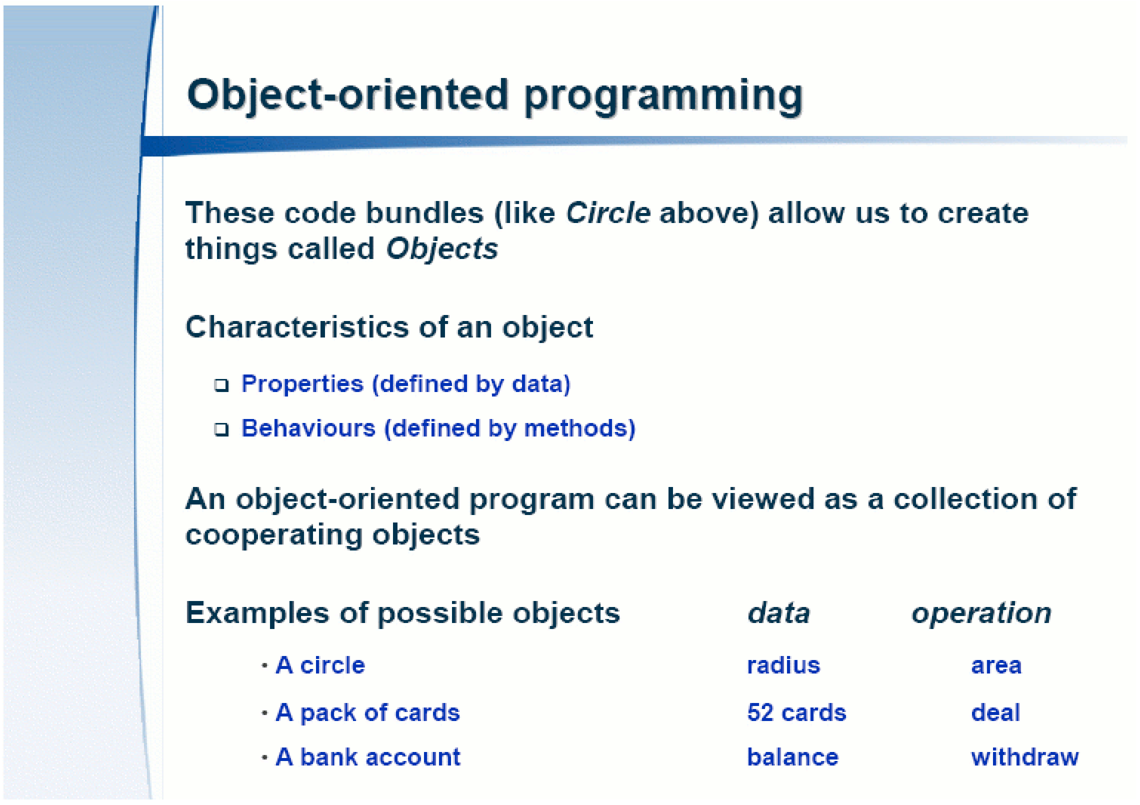
→ As a way of organising your code so that the data, and the operations on that data, are bundled together

→ As an extended type system. These bundles (things…) are simply more complex types (than an int or a double etc)

Figure 4.6 Dual concepts

This is probably intended as a way of helping students who have not constructed a clear concept of class (or object - at this point the difference between class and object has not been made clear), by offering another way of thinking of the idea. For an OOP expert, this this dual notion of code organisation and type definition is not a problem. They would probably see the type notion as being more significant, since the code organisation aspect can be partially achieved in other languages, such as C, by devices such as variable scope and separate source code files. But for the novice, it may be that the immediate introduction of this sophisticated way of thinking, after just one example, may not be helpful.

We then have two more examples:



Object-oriented programming

These code bundles (like *Circle* above) allow us to create things called *Objects*

Characteristics of an object

- Properties (defined by data)
- Behaviours (defined by methods)

An object-oriented program can be viewed as a collection of cooperating objects

Examples of possible objects	*data*	*operation*
• A circle	radius	area
• A pack of cards	52 cards	deal
• A bank account	balance	withdraw

Figure 4.7 Examples

and then an exposition of the difference between class and object, the idea of a constructor and the use of 'new'. It is interesting that the motivation for introducing the OOP paradigm is treated very briefly, as

> *We would like to associate these data and operations somehow, and group them in a Circle thing*

4.2.6 Use of metaphor

Even from the start, metaphorical expressions are used extensively, in accordance with the established vocabulary of computing. Slide 12 of lecture 1 says (metaphors are in bold):

> *Programming was done using a Machine **Language***
>
> ***Instructions** that directly controlled the processor*
>
> *Typically expressed as 8-part sequences of numbers*

expressed in base (2, 8 or) 16

 ***Fetch** data, calculate, **store** data, fetch data …etc.*

on slide 14:

***Translating** from mnemonic to machine language was done by an assembler*

On slide 27:

*Java is more than just a language, it is a **platform***

On slide 45:

***Type** the program in and **save** it as GCD.java*

Compile the program by typing

javac GCD.java

this converts the high-level description to bytecode

(assembly code) stored in GCD.class

***Run** the program by typing*

java GCD

this converts (at run-time) the bytecode to machine code

This simply illustrates how many of the ideas in computing are grasped metaphorically.

4.3 Mental spaces, Frames and Conceptual blends

This section considers some programming ideas from a cognitive viewpoint. These are variables, functions, instructions and scripts, and abstraction. These ideas are considered in the light of frames, mental spaces and blends, as described in Chapter Three.

4.3.1 Variables

4.3.1.1 The variable blend

We can analyse the ubiquitous concept of program variable to illustrate the complexity of this apparently simple idea. The idea of a variable is analysed at some length in these sections to demonstrate how much of this topic is submerged below the water-line and rarely explicitly taught, but still influences how the idea is used.

The concept of a variable is a result of a blend from 2 'input spaces', namely hardware and mathematics:

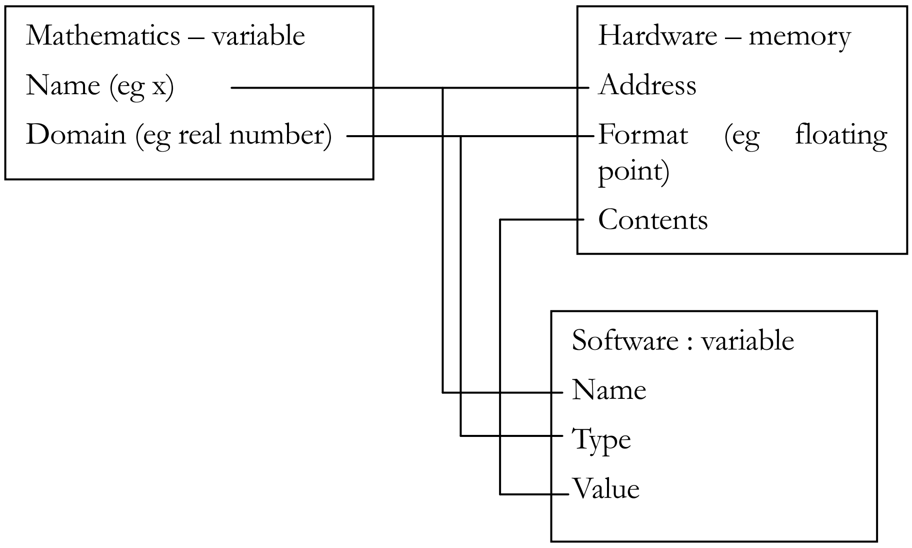


Figure 4.8 The variable blend

The name of a mathematical variable is a unique identifier, and consequently it is analogous to the address of the memory location. In the conceptual blend, that unique identifier becomes the name of the program variable. The mathematical domain relates to the method of data representation in the memory location, and in the blend this is the *type* of the program variable.

4.3.1.2 Blending creates a new idea - a program variable

Fauconnier argues that human cognition is based on integration networks in which 'vital relations' such as identity, analogy, and cause-effect, both within and between mental spaces, construct new spaces which have emergent structure - in that the output space is not just a merging of the inputs, but has properties different or additional to the inputs spaces. As such, these integration networks are the source of the imaginative powers which drive forward new ideas.

This can be seen in the variable blend. The earliest days of programming electronic computers was in terms of machine code or early forms of assembly language, where memory locations referred to by addresses were used to store data values interpreted by some format such as a version of floating point. The use of symbol tables to map addresses to labels might be seen as simply a convenience, but moving from label to name corresponded to thinking of memory contents not as *representing* some quantity, but as *being* that quantity.

4.3.1.3 The variable blend again

Close examination of the model of the variable blend above shows that it is in fact a great simplification of the real state of affairs. For example memory does not 'really' contain data. One byte of memory contains 8 two-state electronic devices, and we 'think of' those two states as mapping to 0 and 1 - so that in fact a bit is a blend between the space of electronics and the space of binary numbers. However this just gives us a pattern of bits in a location, and seeing that as a data value requires another blend, that of incorporating a method of data representation such as floating point in a given format. So the relationship between domain, format and value is more involved than that shown above. The format determines the set of possible values which can be represented, and this relates (but not equates) to the domain of the mathematical variable. For example all conventional floating point representations have limited range and precision, but real numbers do not.

Another factor is that 'mathematical variable' itself has several different meanings. This question is discussed by Beynon and Russ (1991). Some flavours of variables include

1. A quantity fixed by some constraint to an unknown value, such as $x+1 = 5$
2. A metonymic role enabling the statement of a proposition for all members of a set. For example for all real x and y, $x^2-y^2=(x+y)(x-y)$
3. A 'dummy' variable, or what in lambda calculus terms (and elsewhere) would be called a bound variable. For example in $\int_0^1 \sin xdx$ the x is 'used up' in the integral evaluation, so this represents the same as $\int_0^1 \sin ydy$
4. A parameter, such as C in $xy=C$ defining a family of hyperbolae.

Each of these can be crossed with the question of what kind of domain the variable might have - for example real, complex or other number, vector, matrix, set, function and so on.

It is interesting to see the result of each of these variations when blended with the mental space of memory location. (1) is straight-forward and gives nothing new - this corresponds to the numerical solution of equations, as in the first uses of electronic computers in ballistic calculations. (2) usually does not work in conventional imperative programming, since memory locations always contain one value, not 'any value' from a set - but it does correspond to the use in languages like Prolog.

(3) is very common - for example suppose we want to sum the array *a* which has 100 elements-

```
total=0;

for (int x=0; x<100; x++)

 total+=a[x]; // add a[x] into the total
```

Here x is a dummy variable, and could have been called anything else. The only problem might have been if the variable name 'x' was already being used for some other purpose, and the use in the loop might corrupt that other value. However declaring x in the loop header (int x = ..) means that the scope of this x is limited to the loop body, so even this concern can be discarded.

(4) corresponds to some extent to the idea of a constant, which cannot be implemented at machine level except by using read-only storage. Some high level languages can express the idea, which Java does with the keyword 'final'. Alternatively we can see a parameter as a function argument, which again cannot be implemented at the machine level, only in a high level language.

4.3.1.4 Primitive and reference type variables

Java distinguishes between 'primitive' types and 'reference' types. A primitive type corresponds to a simple value, such as an integer or a character. A reference type corresponds to an object - which probably has several data fields and several methods. Another difference is that the value of a variable of reference type is just a reference to an object, not the object itself. References are in fact pointers - the address of the object in memory. For example

```
SomeClass object1; // declares the type of object1;

object1 = new SomeClass(); // make an object - object1 points to it

SomeClass object2; // object2 is same type

object2 = object1;
```

The last line means that object2 now points to the same place as object1, so we now have two references to the same object. If we changed the data in object1, object2 would seem to change - since they are the same thing.

But how does this differ from primitive types? One difference is that for example

```
int x;
```

not only declares the type of x, it also reserves enough memory to hold an int - whereas

```
SomeClass object1;
```

only declares the type of object1, and reserves no memory. Assignment also seems to be different, since

```
int x=3;

int y;

y=x;
```

copies the value of x to y, whereas

```
object2 = object1;
```

does not copy any objects.

However, the 'value of a reference type' is usually taken to mean 'the object stored where the reference type references. But the reference is actually an address, and

```
object2 = object1;
```

does copy the address that object1 points to what object2 points to - so in fact assignments are the same for primitive and reference types..

The names of both primitives and references are stored in a symbol table, together with the address in memory to which the symbol refers. The real difference is that for primitives that address cannot be changed, whereas for a reference type it can.

For a novice this relationship between primitives and reference types is usually limited to the viewpoint that reference types are objects, and primitive types are simple pieces of data. But this is only part of the story, and a perceptive novice will realize that.

4.3.1.5 Variables and metonymy

We would often say that a statement like

```
x = x + 1;
```

increases x by one. To be precise, what it does is to increase the *value* of variable x by one. In other words it is common to use the name of the whole (the variable) in place of the part (the value). This is a metonymic use. As well as a value, a program variable also has a name (x or whatever), a type (an int and so on), possibly a range and precision, and at the implementation level, a method of representation (which in the case of Java int is a 32 bit two's complement) and a location in memory - which may change during execution.

The metonymic form is a convenient short-hand, but a novice may benefit from the more precise long-hand form.

4.3.2 The Function blend

Many programming languages have units which can be used to structure and modularise code. An early form was the sub-routine in FORTRAN. Pascal had two versions - a procedure which simply carried out some process, and a function, which 'returned' a value. Pascal procedures and functions also had parameters or arguments, by means of which data could be 'passed' into the routine. The influential language C only used the term function, but included 'void' functions which did not return a value, and so were like

Pascal procedures. C++ accepted functions like this, but also offered methods for classes. These were like C-type functions except that they were solely associated with the data values in an object of a particular class. Java has no 'stand-alone' functions or procedures, and only has methods.

We can see program functions and their variations in different languages as a blend between the mathematical notion of a function, together with the concept of a modular unit of program code :

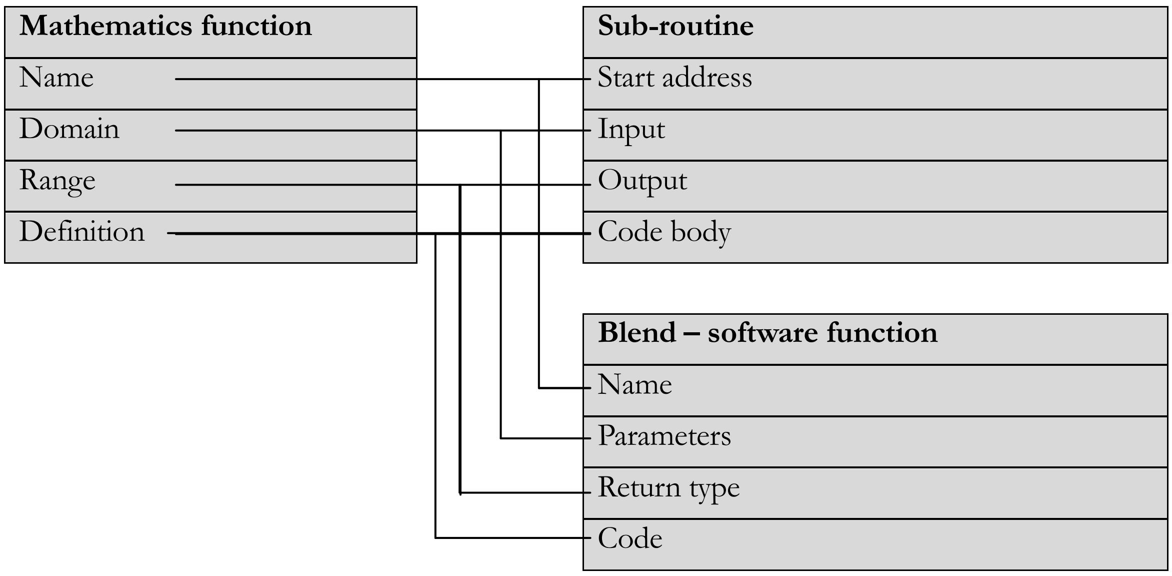


Figure 4.9 The function blend

As in the case of variable, this is a simplification. For example within mathematics we can distinguish between 'standard' functions such as the trigonometric functions which have widespread significance, and an arbitrary polynomial such as x^3+2x^2+1. In software we can see these as corresponding to system-provided library functions on the one hand, and programmer-defined functions on the other. Further, some programming functions directly correspond to 'standard' mathematics functions such as cosine and logarithm, whilst others do not - such as a function to open a file or input stream.

These examples are another illustration of the way that a blend helps us think of a new idea, and to extend the idea beyond its range in the input spaces. But pedagogically it might raise an obstacle, to the extent that a novice programmer may have limited previous mathematical experience, and not have the function concept to extend.

4.3.3 Instructions and scripts - metaphor as blend

The two blends of instructions and scripts are taken together because they are related. At machine code level the elements of a 'program' are usually called instructions, with examples like ADD and MOVE and CMP (for compare). But they are not literally instructions, which are commands issued by people requiring other people to do things. They are actually binary patterns which cause a CPU to transform though a sequence of subsequent electronic states. However we find it effective to blend the space of human interactions with the space of the electronic specification of CPUs, yielding a blend in which we 'tell the CPU what to do'.

In a blend there are options for which input space components are carried over. In this case, a human instruction has the possibility that it might be disobeyed - this must not be brought over into the blend. Further it is required that the person who receives the instruction must understand what it means. This is discussed below.

This metaphor of seeing program statements as instructions opens out to seeing a program as a script - which is another blend. The core meaning of script (now slightly archaic) is that of something written - but this is specialised here to words and directions to actors who will perform a piece of drama. Thinking of a program as a script supports the idea that programming means 'telling a computer what we want it to do.'

We can now see this as the reason behind what Pea called 'the super-bug' (1986) - the idea that the computer contains some intelligent agent which knows how to do things, and that programming means letting this agent know what we want it to do. It is not surprising that novice programmers often make this error, since the vocabulary of programming has developed as if we thought it was true. For example, Horstmann (2005 page 67) writes in a student textbook that programmers using structured programming became confused by large numbers of global primitive type variables and that

> *As a result, programmers gave wrong instructions to their computers, and the computers faithfully executed them, yielding wrong answers.*

This might be seen as nothing more than loose writing, but students might well interpret it as implying that computers are faithful servants which do what they are told. Some aspects of this metaphor are useful - for example that a program (plus input and initial state) determine what a computer 'does'. Other aspects are not so helpful - for example human servants have intelligence, and a certain amount of autonomy.

4.3.4 The thinking of expert programmers - living in the blend

Fauconnier and Turner (2002) point out that 'a cup of coffee' is a mysterious blend. They argue that what we actually sense is the visual appearance of the cup, the smell and taste of the coffee, the feel of the weight of the cup and so on. Only by a process of integration are these neural processes brought together to the perception of the construct 'a cup of coffee'. In particular, we do not perceive cups of coffee simply because they are there.

On the other hand, when having breakfast we are not aware of going through the process of gaining sense impressions and then integrating them into a cup of coffee. The process is very fast and we are not conscious of it - we 'live in the blend' in the sense that we think of the cup of coffee as if it were a single simple unity.

Expert programmers (that is, not novices) live in multiple blends, but for many of these they can reverse out and deal with the input spaces if the need arises. For example, consider the blend between bit patterns and real numbers to yield the concept of floating point number. Most of the time we live in the blend and treat floats as being real numbers - for example the Pascal language actually used the word 'real' to denote floating point type. But there are situations where we have to make the blend explicit and conscious. For example Knuth (1997 page 214-264) is doing this when he shows that the associative law does not apply to floating point addition.

4.3.5 Abstraction

Programmers are aware of the benefits which accrue from concentrating attention on the core of the task at hand - in other words, abstraction. For example, in a simple conditional statement:

```
if (x>0) .....
```

..

we know that we can answer the question 'how can a computer do that?', in that somewhere in the compiled code for this there will be a machine code instruction which will be a conditional branch, jumping to a new location if the plus flag is set in a program status word. However we usually abstract away the mechanism of implementation, and focus entirely on the if statement.

In terms of blending, an abstraction involves erasing the input spaces so that we can only see the output space - the grin of the Cheshire cat. Again this is living in the blend. However we are aware of (at least a possible mechanism for) an implementation, and we can reverse the blend and consider the input spaces if the need arises.

This raises the pedagogical question as to whether it is helpful, or distracting, to include an implementation mechanism when teaching the abstracted process.

4.4 Chapter summary

This chapter has examined some computer programming concepts from a cognitive perspective. It has shown that despite the fact that programming languages are formal with explicit syntax, they do not simply 'say what they mean'. In fact the normative versions of these ideas are based on metaphors and other types of blends, and this is reflected in the way that natural language is used when the subject is taught. The students' task in learning is then to develop their own versions of this blend-based understanding, on the basis of their learning experiences.

C H A P T E R F I V E - P i l o t D a t a C o l l e c t i o n

DATA COLLECTION 1

5.1 Introduction

Pilot data collection took place in two stages:

1. In May 2006, 5 students on a Computing programme were interviewed, once per student, about *main*, classes and objects in OOP

2. In June 2007, some students on Psychotherapy and Counselling courses were interviewed in connection with the use of the words ‘class’, ‘object’ and ‘set’

As pilot studies, these were conducted in the absence of a firm theoretical framework, but nevertheless they yielded some useful results. The first group is described below in section 5.2, and the second in 5.3. Section 5.4 summarises the findings.

5.2 Java Student Interviews

5.2.1 The context

The students interviewed here were a class of 5 members. They were part-time mature students studying a programme equivalent to the first year of a full time degree. They had previously completed modules in HTML, JavaScript, Visual Basic and C, and so had experienced ideas such as variable, data type, loop, conditional and so on, in a structured programming situation.

The students varied very widely in their previous academic background, from a post-doctoral research engineer to a welder with no formal qualifications.

At the time of the interviews they were completing the third of 3 modules in Java, taught by the author. They had started writing Java using Notepad and command-line compilation, and then started to use the Netbeans IDE. These modules consisted of 10 classes each 2 hours long, consisting of a workshop approach of brief presentations followed by practical exercises and tasks involving writing short Java applications. The first module had been about the principles of Java and OOP, in a character-based situation. The second was about writing GUIs in Java using Swing, and the third was about data structures and algorithms in Java. They had completed assignments at the end of each module. Consequently they had significant experience of learning programming.

5.2.2 The interviews

The research method selected was an interview which was then transcribed and analysed in the ethnographic tradition. Each interview lasted between 20 to 40 minutes, was held on a one-to-one basis (no-one else present) and was videoed. The interviewer was the author. The interview was loosely structured on the following lines -

1. An introductory question about the *main* method, asking what was special about it, and following up with questions about *public, static* and *void*. The plan had been to provide a gentle friendly start to put students at their ease, and as a preliminary for what came next. In fact for most students their grasp of main proved to be more problematic than had been expected, and these questions elicited some interesting responses.

2. The second stage was to present the student with a short piece of Java code, to ask what the program would do, and follow this up by asking how many objects were created as the program ran.

Within this framework questions were improvised based on what the students said, trying to follow up interesting statements and responding to the different levels of understanding of the individuals.

5.2.3 The Java code used

This is listed below. It was selected as a straight-forward class definition and use, illustrating the construction of several instances of the same class, and having a static method. Students had already seen the algorithm used in the findBiggest method in the context of other languages, so that it was thought they would be able to focus on the OO aspects of the code rather than how this algorithm worked.

Two versions of the code were used, one with meaningful identifiers and one without. The idea behind this was that using meaningful identifiers would make code interpretation 'too easy', so that identifiers such as ClassOne and methodOne were used. After the first two interviews it was found that students found code interpretation not at all easy, and consequently a second version (listed here) was used. The two versions differed solely in the identifier names as follows:

First version	Later version
ClassOne	City
methodOne	findBiggest
methodTwo	display
fieldOne	name
fieldTwo	population

Further thought revealed that the difference between the two versions was not simply that the later version was 'easier'. Rather the two versions emphasised different aspects of the code, with the first version focussed on its syntactic structure, while the second pointed to the semantic data modelling aspect.

Here is the first version (the second version is annotated for non-Java readers):

```
public class ClassOne
{
  public ClassOne(String f1, int f2)
  {
  field1=f1;
  field2=f2;
  }
public static ClassOne method1(ClassOne[] someObjects)
{
  int which=0;
  int c=0;
  for (int i = 0; i<someObjects.length; i++)
    {
    if (someObjects[i].field2>c)
      {
      which=i;
      c=someObjects[i].field2;
      }
    }
    return someObjects[which];
}

public void method2()
{
     System.out.println(field1+" "+field2);
}

private String field1;
private int field2;
}
```

and using the class in main:

```
ClassOne[] data = new ClassOne[3];
data[0]=new ClassOne("London", 10000000);
data[1]=new ClassOne("Birmingham", 1000000);
data[2]=new ClassOne("Coventry", 300000);
```

```
ClassOne anObject = ClassOne.method1(data);

anObject.method2();
```

The second version follows here. The comments (starting //) are for the benefit of non-Java readers, and were not given to students

```
public class City // class to model a city
{
 // construct a city, with given name and population
 public City(String argumentOne, int argumentTwo)
 {
 name=argumentOne;
 population=argumentTwo;
 }

// this static method is given an array of City
// objects, and it returns a reference to the one with
// largest population
public static City findBiggest(City[] someCities)
{
 int biggestOne=0; // local variables
 int biggestPopSoFar=0;
 // loop through the array
 for (int index = 0; index<someCities.length; index++)
 {
 // if we find a bigger one..
 if (someCities[index].population>biggestPopSoFar)
  {
  biggestOne=index; // remember which it is
  // and its population
  biggestPopSoFar=someCities[index].population;
  }
 }
 // return the biggest one
 return someCities[biggestOne];
}

// output info about the City
public void display()
{
 System.out.println(name+" "+population);
}

 // two data fields
 private String name;
 private int population;
}
```

And using the class in main:

```

// make an array of references to three cities
City[] cities = new City[3];
// make three cities and place in array
cities[0]=new City("London", 10000000);
cities[1]=new City("Birmingham", 1000000);
cities[2]=new City("Coventry", 300000);
// find the biggest
City anotherCity = City.findBiggest(cities);
// display it
anotherCity.display();
```

5.2.4 Notes on transcriptions

The videos of the interviews were transcribed carefully and a few excerpts are given here. These are given as fully as possible, including filled pauses (such as um) and discourse markers (such as like or well). A short pause is marked as .. and a pause of over 1 second is reported as such. Text written in italics is highlighted because it is regarded by the author as being significant, rather than emphasis given by the speaker.

5.2.5 Ideas about main

It had been expected that the question 'What can you tell me about main' would evoke the normative response, which is that execution started there. A Java application consists of definitions of classes, together with precisely one method called ‘main’. When the application starts, class definitions are loaded, and then the system looks for the ‘main’ method, and executes it. This convention is historical – it dates back to how programs written in C start.

In fact there were 4 aspects to the way in which students thought about main, some students holding more than one aspect:

1. The normative response - that main was the point where execution began
2. The idea that main was the textual section of code where one would see expressed the algorithm the programmer was using to achieve the required outcome
3. The idea that main meant important or key or central.
4. A confusion between a main method and a Main class.

So we see a range of understandings of main, some normatively 'correct' whilst others are obscured by a previously encountered concept, or ‘met-before’. Ragonis and Ben-Ari (2005) recommend that the main method is taught at an early stage, despite the fact that it is 'too procedural', since otherwise it 'interferes with understanding dynamic aspects of the program'. This evidence seems to support this.

5.2.6 Ideas about objects

The notion of object is central to OOP, and it might have been thought that these students after 3 modules would have a clear picture of what they were, especially having done a module on Swing and writing applications which instantiated many different classes.

However, several do not, which is very surprising. The following extract is the most significant section of this data collection phase, since it shows a student, who was working as an IT professional and had completed three modules in Java, had no clear idea of the OOP object concept:

(Interviewer) .. here's another question.. how many objects are there.. in this program?

(Subject A) How many objects..(8 seconds pause) one, and that's the array data ..

(Interviewer) OK ok .. what what's the name of that array?

(Subject A) Errr.. anObject..

(Interviewer) OK.. and.. um um OK

(Subject A) oh err name.. data

(Interviewer) OK so the array is called data

(Subject A) yeah

(Interviewer) huh? Um.. is that the only object which is in this program?

(Subject A) (10 second pause) Probably not (laughs) but um I suppose that is.. *what the definition of an object is*..

(Interviewer) OK OK

(Subject A) um I mean I suppose in my mind I *when I think of an object I'm thinking of some thing* and the only things that is *a thing* that I can think of is the array obviously you've got things like the fields and do they count as objects or you know the constructor itself is that an object as well er methods are they objects in which case there's lots..

This student was given the initial version of the program which did not have meaningful names - the array called data was later called cities.

There are several points here:

- Two long pauses indicates uncertainty and a conflict in the student's ideas
- He uses the terms ‘constructor’ and ‘method’ without hesitation, showing that in some ways he is familiar with OOP.
- The student has encountered the notion of object before
- A prominent characteristic of that notion is that it vaguely means a thing - 'in my mind I when I think of an object I'm thinking of some *thing*' (my emphasis)
- He is also aware that the Java notion of object must be more precise than that, but he does not know in what way - such as 'you've got things like the fields and do *they* count as objects?' - again my emphasis

This lack of a clear idea of 'object' was typical, although one student could identify objects quite readily, including the String object inside each City object

So as in the case of main, we see a range of ideas about the object concept, some of which are normative but others being dominated by the vague general sense of the object notion.

5.2.7 static

The significance of the keyword static was not well understood:

> (Interviewer) OK.. now the method.. called.. the method called method1 has a different keyword associated with it, than with method2, and that keyword is static
>
> (Subject A) Yes.. (subject smiles – probably because he is anticipating what is coming next – I will ask him what static means, and he does not know)
>
> (Interviewer) So method1 is being declared as static..um.

(Subject pulls funny face, then laughs, looks at interviewer then looks away)

(Interviewer) What difference do you think that makes?

(Subject A) What difference does that make? (18 second pause) If it's static it can't be err *it can't be changed by anything else* it's just what it is, it's just.. what.. takes the information in and spits it out and it doesn't - it can't be affected by anything..anything any outside influences changing it

Again there are several aspects to this:

- The subject's facial reaction, and the 18 second pause, again indicates uncertainty and conflict.
- He recognises the use of static with its everyday meaning of constant, and claims he understands in those terms - 'it can't be err it can't be changed..'
- There is also the trace of another idea, the idea of access control and the fact that private members cannot be changed from outside the class - 'it can't be affected by anything..anything any outside influences changing it'.

The third aspect is quite significant, in that it has a characteristic seen in later student interviews. There are two ideas which have an aspect in common. The two ideas are 'static' in its general sense (which is here inapplicable), namely not changing. The other idea is 'private', a Java access control modifier which means that the class member cannot be accessed from outside the class - and can therefore not be changed. In trying to construct a meaning for 'static', the student has (incorrectly) established a link between the met-before static as constant, and the OOP concept of access control. This strengthens for the student the coherence of the web of concepts - but since it is wrong, it is likely to form an obstacle to further understanding.

By contrast, the student who was clear about objects had no difficulties with static, giving a textbook response, and could also explain a different meaning of static in C.

This student is very free of the ‘met-before trap’ – she can entertain two completely unrelated meanings of the term, which are themselves not connected with the met-before of static as constant.

As was seen for the object concept, a range of ideas for static is shown, from the good grasp of subject E to the ideas of subject A which are dominated by the idea of static as unchanging.

We can also see here consequences of the inter-dependence of concepts. The idea of static (in Java) is concerned with class and object, and so students who do not have a clear idea of object and class could not logically be expected to handle the notion of static.

5.2.8 Summary of this interview set

1. Some of these students have concepts of key OOP notions such as object, class and the role of man which are very non-normative

2. They are non-normative because in some cases the student finds it hard to articulate, for example, what an object is, or because they suggest aspects of the idea which are inappropriate - for example that the main method is is the 'main' method.

3. Even with such non-normative ideas, these students are able to at least 'survive' assessed modules in Java/OOP

4. The two versions of the code suggest a reason for point 3, in that the Java convention of meaningful class and method names may be the reason why students can survive, and think they understand, when their understanding is actually very different from the normative view.

Overall, even after three ten-credit modules, many of these students had a very weak grasp of OOP concepts.

5.3 The psychology student interviews

In June 2007 some volunteers were interviewed in relation to the concepts of class, object and set. The (rather naïve) idea was that class, object and set were words in common usage, and that a determination of how non-programming individuals used those terms would provide the foundation upon which programming students would be building.

The volunteer subjects (with one exception) were students of psychology, counselling and psychotherapy, and all were mature students. One was a professor, some were graduates working at Masters level, others were undergraduates, but all had at least reasonable experience of academic work, and all had had significant life experience.

Each volunteer was given a semi-structured interview lasting about 35 minutes. They first received some brief written instructions :

> *In the attached text, some words and phrases are underlined. I would like you to classify each word or phrase into one of the following three kinds - classes, objects or sets.*
>
> *If you think it is a class, write C next to it*
>
> *If you think it is an object, write O*
>
> *If you think it is a set, write S*
>
> *Feel free to ask questions and talk about what you are thinking.*

They were then given 3 short texts, taken from newspapers, with selected noun phrases underlined. An example follows:

Text 1 - from the Times Online April 3 2007

Expecting mothers will have the choice to give birth at home or in hospital, according to a reform of maternity services announced today by the **Government.**

The Health Secretary, Patricia Hewitt is expected to confirm the option of **home births** as part of the guarantee to give **pregnant women** the “full range of birthing choices” by 2009. Under the plans women will also be attended by the same two or three midwives throughout their pregnancy, with one of them delivering **the baby**.

The **Royal College of Midwives** (RCM) described today’s plans as ambitious and said that more widwives would be needed to satisfy the resultant demands. Just 2 per cent of the **601,000 births** that took place in England in 2005 were delivered at home.

The Department of Health has been accused of watering down its planned reform of maternity care, after suggesting that women would be looked after by a single, named widwife during labour. But the DoH insisted today that it was not practicable to promise **women** that **one midwife** would look after them throughout their pregnancy.

The aim was to take a phenomenographic approach and to identify the crucially different ways of experiencing object, class and set. This was done and the following ways were found:

Object	Class	Set
As singular thing	Group or collection	Group or collection
As point, purpose or subject of discussion	Definition, description or classification	

The word 'set' was thought of as significantly different from the mathematical notion of set. The usual conception was what a mathematician might call a tuple - for example a set of golf clubs or a set of cutlery, a collection with each element having a distinct role. A collection with just one element was not thought of as being a set, nor a group with no elements.

However, the most significant outcome was that no subject used class in the way they usually would.

When presented with the instructions, some subjects asked for definitions:

> J: I'm wondering what class object and set are in all fairness, but, could you sort of say what does class mean, what does object mean, what does set mean?

One subject explicity constructed her own definitions:

> B: Umm.. I guess a class would be a type of something, an object would be the something, and set would be a collection of somethings. (Laughs) I don't really know, I'm just going to do it anyway.

but most expected to be able to construct definitions from the context:

> GR: OK, and the definition of object, class and set will be apparent in the text?

Towards the end of the interview subjects were asked how they usually used the word class:

> Interviewer: When you talk about class, what do you usually mean by it?
>
> S: Class, I suppose, well.. the way how its written there, if you think it is a class, that looks like its weird to me, because when I think of class I think of like status, initially, that would be the first thing that comes to mind, like different classes, or like a classification type thing
>
> Interviewer: OK so when you talk about status, can you say a bit more about that?
>
> S: Sort of like upper class, middle class, working class.
>
> Interviewer: What other situations would you talk about a class?
>
> S: A class as in like a school, as in a class group.

Most subjects chose these as the way the would normally use the word class, and this is confirmed by concordance analysis of corpus data. For example a sample from the British National Corpus (2007) shows the following relative frequencies

Meaning	% Frequency
Social class	53
Kind or type	20
Educational class	14
Merit (eg first class)	9
Unclear	4

So why did these subjects not use the social class sense, when they said that was how they usually used it, and corpus analysis shows most people do? It is proposed that it was because they thought that we were working in a formal frame (in the sense described in section 3.5 above) which had components CLASS OBJECT and SET, and they knew that SOCIAL CLASS could not be part of that. Their actual responses - asking for definitions, trying to ascertain definitions from the texts, or constructing new definitions - fits with this possibility.

This means the perspective of the orginal conception for this group of interviews was completely wrong. The idea had been that programming students would bring to a course a set of concepts for CLASS, SET and OBJECT, which would form the basis of developing OOP ideas. But these students showed that in a formal educational setting they would put aside 'everyday' word meanings, and would expect that alternative definitions and meanings would be assigned through the subject of study.

The second outcome from this work was equally serendipitous. It concerned one sentence in one of the newspaper articles:

> Under the plans women will also be attended by the same two or three midwives throughout their pregnancy, with one of them delivering *the baby.*

Was 'the baby' an object, a class or a set? It seemed to be none. Was it even singular or plural? Grammatically it is singular, but it seemed to refer to all of the babies, so it should have been plural. Reading of Fauconnier and Turner after these interviews were conducted provided the insight - 'the baby' was a perfect example of a *compression*, as described in section 3.7 above.

5.4 Chapter summary

At first sight these two data sets seem to present opposite characteristics. The psychotherapy students suppressed their pre-existing ideas of class, object and set (their met-befores), and expected new versions of these would be presented, or that they would be able to deduce them from the context. By contrast the computing students seemed to have concepts of *main* and *static* which were very heavily influenced by the everyday meanings of these words.

This can be resolved if we take into account the fact that the two groups were at very different positions through the learning programs (if we view the psychotherapy student interviews as being introduced to a new set of ideas). The psychotherapy students were at the beginning, and 'wiping the slate clean' as regards the concepts of object, class and set seems reasonable, for adults who were experienced in learning new domains of study.

But the computing students had completed three ten-credit modules on Java. Consequently they were expected to have 'got the idea' of class, object and *main* by this point, and they answered in a way which implied that they thought they did have at least some understanding. But they show that by this time, they had constructed rather fluid and organic ideas of these terms, that these were largely non-normative, and were heavily influenced by 'met-befores'.

We can sketch this out for the web of concepts comprising *class*, *object*, *static* and *access control*. Several students showed that they had no clear idea what an object was. Consequently that could not grasp the relationship between class and object, and consequently the normative meaning of static was inaccessible to them. They fell back on the met-before meaning of static, that is, not changing, and one student had constructed an (incorrect) link with part of the idea of access control, namely private, in that a private member could not be changed by something outside the class.

So we have a picture of students starting out on a course suppressing their met-befores of previously encountered concepts, but as the course progresses they construct their own versions of these ideas, which are complex combinations of met-befores and partially grasped normative ideas.

CHAPTER SIX - Phase One Data - University

How do computers work?
How?
Yes.
Err how do computers work? I have no idea!

Extract from student interview

6.1 Introduction

This chapter and the next describe two sequences of data gathering. The first took place at a research-based university with Computer Science undergraduates, and the second was at a vocational post-16 College of Further Education, with students on a range of vocational courses

Both groups were interviewed towards the beginning of their courses, before the OOP paradigm had been presented, and so concepts of class and object were not examined. Instead, ideas about basic notions like computer, program and variable were examined, and some short sequences of procedural code involving assignments, if statements and loops were discussed.

This was not a comparative study - there was no attempt to identify differences in response between academic and vocational students. Consequently the interview plans were not identical. The intention was to identify how students thought of basic computing ideas and dealt with short program fragments, in order to establish a context within which more specific OOP concepts would be held. The two groups were used so that it would be possible to make statements about this which would encompass students with a wide range of backgrounds and learning experiences.

In this chapter, the plan and purpose of the interview is described. Responses are then presented. The responses for each question are used to construct a set of critical variations, in a phenomenographic analysis. Selected transcript excerpts are presented, to exemplify these. The chapter concludes with a short summary.

6.2 The University interviews

6.2.1 Introduction - Context and Content

These interviews were carried out in October and November 2007. The objective was to interview first year Computer Science undergraduates in order to provide an insight into how they perceived computers and programming. The students were following a two semester module which is primarily concerned with programming, Java and OOP, and at the same time they are following a parallel module about data structures and algorithms, which provides a broader perspective on the fundamentals of Computer Science.

Fourteen students were interviewed. In the interview transcriptions, interviewer statements are in plain text, and *subject responses are in italics*. Students are referred to by a coding scheme as follows:

First character - gender - F = female, M = male

Second character - previous programming experience - 0 = none, 1 = a little for example a weekend or a short course several years before, and 2 = reasonable experience, usually through A Level Computing

Third character - distinguishing letter

For example M0D is a male student with no previous experience of programming. These codes are given at the start of each transcription extract.

6.2.2 The interview plan

The interviews are semi-structured - a broad sequence of questions is pre-planned, but variations occur as answers are given. The students at this stage have a variety of backgrounds. Some have no prior experience of programming while others have some, mostly from A Level Computing. The theoretical framework adopted posits that the OOP frame is a 'high pinnacle', built on concepts which themselves represent frames built on others. Those components are likely to include the concept of a computer and the idea of programming, and these were the topics the first interview sought to explore.

The interview plan was as follows - relevant points are expanded below:

1. Ask about progress in the programme in general

2. Have you done programming before? If so, how much, what languages, any OOP.

3. What is a computer?

4. How do computers work?

5. What is programming?

6. What is a variable?

7. Ask about the short code sequence about assignments

8. Give the first pseudo-code fragment and ask 'what does this do?'

9. As 8. with the second fragment

10. Discuss System.out.println and public static void main

11. What does OOP mean?

The plan was not rigidly used for all subjects - for example those with no experience of programming were not asked what they thought OOP meant.

6.2.3 The assignment code sequence

This was not present in the first few interviews. However in the course of an early interview the following occurred. This followed a discussion of 'what is a variable' which had established the predominant idea that variables changed value:

(F1A) When you give it a value, what kinds of values can you give to a variable?

Integers, or for example, n plus something, um, different..

n plus something?

I think for example, if i, we have i, it might have a value of 8, 9, or for example another integer plus something, or for example another something, and we can change that

OK so that if you have say i equals, if we say this is an expression, involving other variables?

Yes

This suggested there were 2 qualitatively different options, namely assignment to a constant and assignment to an expression. This was pursued in an improvised manner:

(F1A) OK suppose we have a little piece of code, suppose we say j = 2, and then we say i = j + 1,

Yeah

Now does the computer remember i = j + 1, or does it remember i = 3?

I think it remembers 3

Now suppose these are the instructions in order, in the program, so we have first instruction j = 2, second instruction i = j + 1,

3

now suppose I change the value of j, (writes down j = 3)

and i will be 3 plus 1

so at this point i will change its value?

Yes

Because this was unexpected and interesting, the following was added to subsequent interviews:

```
a = 1
b = a + 2
a = 3
How big is b now?
```

Most students responded 3 as expected, but a few (mostly but not all with no programming experience) gave the answer 5. Lest these students might labour under a misconception, it was explained to them that in most languages including Java the answer was 3, and that this was because expressions were evaluated and the value assigned to the left hand side, rather than expression being retained as a definition.

6.2.4 Pseudo-code sequence 1

The purpose of this was to explore how students reacted to pieces of new code, what they thought the question 'what does this do?' actually meant, and to try to clarify the process of 'understanding code'. The sequence, which outputs the maximum of an inputted sequence of numbers terminated by -99, was as follows:

```
x = 0
input n
while n is not equal to -99
     {
     if n > x then x = n
     input n
     }
output x
```

6.2.5 Pseudo-code sequence 2

This sequence, which outputs the average of an inputted sequence of numbers terminated by -99, was as follows

```
total = 0
count = 0
input value
while value is not equal to -99
  {
  total = total + value
  count = count + 1
  input value
  }
output total / count
```

Initially only this sequence was used, but in fact there are 5 processes at work here:

1. accumulating a total
2. counting
3. repeated input
4. termination at -99
5. recognising total/count as average

together with the steps involved in the student's grasping these processes. Consequently it was decided to precede this with sequence 1, to increase the chances that a student might be able to give at least a limited response, and to offer a 'stepping stone' for those who might be able to respond to sequence 2 after having dealt with sequence 1.

As the interviews proceeded, sequence 2 was seen to be fairly insignificant. Subjects who found sequence 1 difficult to follow were spared sequence 2, while those who found sequence 1 easy found 2 trivial. However responses to sequence 1 were very illuminating, as described below.

6.2.6 System.out.println

At this point in the module subjects were just starting to code in Java, and were being presented with 2 code fragments. The first was

```
System.out.println( .. )
```

which would display textual output, and

```
public static void main(String[] args)
{
...
```

as the start of execution of code. This raised the question of how subjects reacted to items which could not be completely understood but which had to be used in order to execute simple programs. This was the reason why subjects were asked about these items.

6.3 Responses

What is a computer?

Three variations in this response were found, as follows:

A computer is a servant which can be told to do useful things by human beings	A computer is a machine which can be used to do useful things	A computer is a concrete implementation of an abstract 'device which computes'

This is a phenomenographic style of analysis. This means that individual students sometimes answered in terms which were a combination of two of the above, while other students explicitly offered more that one answer. However at this stage the assertion is made that all answers could be placed somewhere along these three dimensions. The significance of this is considered later.

Examples - what is a computer?

(F0A) What would you say a computer is?

A computer? Err - my English not so good,

OK

A computer - its a machine, its designed by human beings, so we can do whatever we want the computer do things

OK

They listen to us and we know the Java and we know how to work it

Yes

And they just obey our decisions

OK

We can control the computer

OK

So sometimes the computer is more smart than the human being, but we can control it if we know the method

> OK so it's a device that you can err get it to do what you want
>
> *If we know the way to do it*

This sees the computer as a machine, but firmly placed in a human context of doing useful things, provided we can control it.

Another example:

> (F1A) What do you think a computer is?
>
> *Computer for the users or for the programmers?*
>
> If its different, both
>
> *I think for the users its something to use Internet, to play games for the boys for example, to write maybe their homeworks, just to use written programs*
>
> Yes
>
> *And for programmers its like to express, to express herself, the programmer*
>
> Yes
>
> *To write the programs, its to tell the computer what to do, its not something vaguely*
>
> Yes
>
> *You can tell them what to do, what not to do*

Again this is in terms of fulfilling a human purpose, even for a programmer, for whom the computer is a means of self-expression.

However this example:

> (M2F) What do you think a computer is?
>
> *A computer, um, I think a computer in today's terms is a device that computes signals and forms an output based on inputs*
>
> OK
>
> *That's what I'd say, but it doesn't necessarily have to be electronic because you may in the future have biological computers. I think the whole idea of having a computer is that it does computations, hence the name computing.*

shows the abstract notion of something which 'does computations.'

How do computers work?

Some students who viewed computers as servants reacted to this question with a certain amount of blankness, perhaps because 'working' is a characteristic which is applied to machines, and these people were not seeing computers as machines. These students tended to offer response 1:

Response 1	**Response 2**	**Response 3**
They listen for what you tell them to do, and do it	Coded instructions are input and the appropriate output generated	As a consequence of the electronic operation of their components

Clearly responses 1 and 2 share the idea of something representing 'requests' going into the computer, and some kind of output being produced. However they are at the two ends of a spectrum regarding whether or not the computer really 'understands' the input, or whether this is only metaphorical in the sense that it is 'really' just an automatic response to coded input - which was a common position.

There is also the slightly more complex issue of language. Some thought the computer understood English, while others thought they cannot understand English because it is too complex in practice, leaving the possibility open that in the future more powerful computers could understand English. No students could offer unprompted the idea of a distinction between formal and natural language.

It was tempting to explore what 'understand' meant to these students, but this was not done. What was clear however was that the subjects did not hesitate to use the term ‘understand’ in connection with computers. In other words some said computers could understand English and others said computers did not really understand. No student responded by saying 'what do you mean by understand?'. To repeat this important point - students readily used the term 'understand' in connection with computers.

What is a variable?

There were 2 variations in this:

Response 1	Response 2
Something that changes	Memory location and contents

All combinations of this were found. In other words some students said the key thing about a variable was that it changed, but did not relate variables to the notion of storage and hence to memory (while the idea of changing directed them to places in the computer which seemed to change a lot, such as graphics display). Others would say that it was changing data, which was held in memory. And the third group referred to memory and only indirectly related that to software.

The assignment sequence

This was :

```
a = 1
b = a + 2
a = 3
How big is b now?
```

Most students replied 3 - a typical example of those who did not is:

> (M0C) We've got 2 variables here, a and b, and we have a = 1, then we have b = a + 2, then we have a = 3. After that, how big is b?
>
> *5*
>
> OK. So after that sequence, b is 5?
>
> *Yes.*

These students hold an alternative conception of assignment to an expression, which they take to be a functional definition, so that

```
b = a + 2
```

is taken to mean what would be written in mathematics as

$$b = f(a) = a + 2$$

This is not unreasonable, and is correct for a few languages or environments, (such as a spreadsheet), but is incorrect for the majority of languages including Java. A student who regards assignment in this way is likely to be faced with what would appear to be anomalous behaviour of programs encountered in the future unless their idea is corrected.

Pseudo-code sequence 1

For convenience the sequence is repeated :

```
x = 0
input n
while n is not equal to -99
     {
     if n > x then x = n
     input n
     }
output x
```

Here the variation was not in the answer (the content of learning) but what students did in response to the question (the approach to learning). This was complicated by the fact that the question "What would this program do?" in itself did not make sense for some, since they knew the output would depend on the input, and the input could be anything, so the output could be anything, so how can we say what it will do? This provided some insight into what was happening:

Approach 1	Approach 2	Approach 3
Study the code and deduce what it would do	Imagine some input values and trace execution step by step	Recognise the sequence

No student succeeded with approach 1, and only one student could use approach 3, not on this sequence but on sequence 2.

Some students could think of no way to answer the question - not surprising if they had never written a program before. These were led towards approach 2, with interesting results.

Examples of responses

The interviewer usually started by reading through the code with the subject like this:

> There are 2 variables in this program, one called x and the other n. So we have x = 0, input n, that means read in a value for n from the keyboard, and that would be a number, while n is not equal to -99, we have a loop, so we do from there to there in this loop, so long as n is not equal to -99, and we have an if, if n is greater than x, n equals x, then we've got another input n, so here we are invited to type in another value, and the loop continues, until n is -99, and at the end of it we output x.
>
> *Yes*

The student throughout this section is M0C.

Then we have the key question:

> So what would that program do?
>
> *Err, it could, in terms of what it would do if you put a value in for n? How it would work through, or what it would do in terms of what it would do here in the middle between these curly brackets?*
>
> Either

So this is an example of *the question not having a clear meaning* to this student, who had done no programming before. He suggests 'what would it do?' might refer to the execution of the code steps, but at this point it is very vague.

Nevertheless he *spontaneously has the idea of a 'trace',* involving imagining actual input values being entered and following what would happen :

> *Well we could put a value for n in, provided n is not equal to -99, it would run this loop, and for example if n was 5, 5 is greater than x, so x becomes 5, and then you input a new value for n, say 6 to go with a corresponding, a consecutive value, so 6 would go in here, 6 is greater than x, so x becomes 6, this would be a never-ending loop, because it would keep going up and up, but if you were working with negative numbers it would stop, because you would get to, oh no you wouldn't, because if you put -1 in, -1 is not greater than 0, so it would output x,*
>
> OK
>
> *Well that's how I looked at it.*

Here he is trying to deal with the loop. Because he inappropriately fixes on a positive sequence starting at 5, he reasons the loop will not end. But then he entertains the idea of inputting a negative number to stop it, but he is side-tracked by the realization that then the conditional would behave differently, and x would not be changed, and seems to link this with the loop ending. However the question 'what does this program do?' still has no clear meaning. The interviewer provides some guidance which leads the student to the idea of *input being independently chosen* by thinking of an analogy with a game *:*

> OK now we can put in any numbers that we want to,
>
> *Yeah*
>
> Yeah, and they don't need to be in a sequence.
>
> *OK. I suppose the program there, could be to do with, it could randomly, the user could randomly select a number, or there could be something that generates numbers according to what the user of that program does, and then those values would run through*
>
> OK yeah

And then depending upon what the user was doing it could be in a game etc. and then the moves of the user would determine the values of n, and then it could be that the output was something for the computer to do.

This is very significant, since it shows that dealing with the fictive space of user input is not trivial.

The interviewer then offers some input data to consider:

Now suppose we ran this, and I put in, the numbers 7 8 4 2 -99. Does it stop?

Yeah..

I'm putting in 5 numbers, it stops does it?

I would say so

Why?

Because when the 7 goes in at the end, 7 is greater than 0, then input the next number, which is 8, x becomes, sorry x becomes 7, x takes the value of n, 8 is greater than x, so x takes the value of 8, then n takes the value of 4, 4 is not greater than 8, so x does not take the value of n, but you could input another value of n, because there is no else condition, so it would stop when n was -99, and then x would be outputted.

So the student has traced this through to completion, but needs to work through it again (silently) to see the actual outputted value:

OK OK what then is the value of x? The outputted value? If I typed in these, what would come out?

Err.. (9 second pause) x would be 8

8?

I think, 7, yeah 8.

We try another input set:

OK now suppose I ran the program again and I put in, err, 12 3 13 1 -99. What would be the output for those numbers?

(16 second pause) The output would be 13.

13?

That's if I'm carrying out the program correctly?

Yes, you are correct. So what does this program do?

The student has still not yet reduced this completely. The looping is clarified but the output is not related in general to the input data:

The program carries out the operation according to the variable n, and then if it becomes equal to -99 at any stage, then it will output the value of x,

So the student is now clear about this ends when -99 is input, but all he can say is that x is output (as the code states), and cannot say what x is. We do a third run. The grasp is still not firm:

OK suppose we do it another time, and we put in 3 22 -99, what will I get out?

22

22?

Looking at it, I'm not sure whether the loop will just keep running, because even when n becomes -99, that could just stop - no, no ignore that actually

When we put in the -99, it then says, while its not equal to -99 do this, if it is equal to -99 we stop

It goes onto the next line

OK so we've been through this program three times, can you see some kind of pattern, here we get 8, here we get 13, here we get 22.. so which number do we get?

-99 in all of them

As being the last value which is put in?

That's when the if condition effectively finishes

So still the attention is on ending at -99, not the output. Finally we have it:

OK suppose we put in 1 8 23 97 31 48 12 -99 what number gets to be output?

97

Why?

Because its the largest number. It would increase normally here, but then when n is 97, its greater than x which is the previous value, which is 23, then none of the others are greater than that.

So if we summarise this, what does this program do, in summary what does this program do?

It selects the largest value, it will always output the largest value.

6.4 Chapter Summary

- There was a range of ways of thinking about computers, programs and variables. All these ways could be seen in terms of blends (in the sense of section 3.6), such as machine and person, or value and memory location. No student showed an understanding in entirely literal terms, since the computer science community of practice does not.

- The short pseudo-code sequences were in general not trivial, even for students with a good academic background.

- The idea of data input is not trivial

CHAPTER SEVEN - Phase One Data - The College

7.1 Introduction

This data collection process followed that of the previous chapter. That had shown that even for students with a strong academic background, short program fragments could be difficult. This prompted the question as to what aspects of a program fragment produced a difficulty – in other words, are some ‘kinds’ of program harder to understand than others? This round of data collection was based on five different program 'types':

1. A simple calculation on constant values, outputted (Program 1)

2. A loop, with constant initial values (Program 2)

3. A simple calculation on values inputted at run-time (Program 3)

4. A loop using values inputted at run-time (Program 4)

5. A loop which would iterate a number of times determined by user input. (Program 5)

It was conjectured that these would present varying levels of difficulty, and that the level of difficulty would relate to the closeness with which the program code related to changes in machine state. For example, Program 1 had four steps (lines of code), and the resulting execution had four steps. Program 5 had seven steps, but the number of execution steps would depend on input values.

7.2 Overview

Sixteen students were interviewed in relation to their understandings of 5 short program code fragments. The students were following one of the following courses:

- City and Guilds Software Development level 2 - no academic entry requirements, one year full time
- BTEC National Diploma in IT full time 2 years level 3 - entry requirements 4 GCSE's at grade C
- Foundation Degree in Software Development, 2 years full-time, entry requirement BTEC National

Some were mature students who had returned to study, and the rest were aged 16 to 19 years old. The College delivers programmes with a very strong vocational flavour to the local community, in fairly small groups in an informal setting.

One program was altered slightly after the first interview, so the quantitative data presented later refers to n=15.

The idea of these interviews was that work with less able students would reveal some insights into aspects of this situation which might be concealed by direct normative responses from more academic students - and this proved to be true.

7.3 The interview

The interviews were one to one, lasting about 30 to 40 minutes, with an audio recording being made. Students were presented with the 5 programs on paper, and asked ‘what does this program do?’. No explanation for what that actually meant was offered, and no student questioned what it meant.

After several interviews had taken place, a codification scheme was drawn up which would encompass all the responses which were being seen. The scheme is ordered by the degree of 'facility' which students displayed. So code 1 is simply looking at the program and saying what it will do, while 7 is a failure to understand even after having been led through it, and 8 is a failure to understand the question. This ranking is *a priori*, in that we have no empirical basis for saying that 1 shows more insight than 2, say, but it gives a rough indication of how easily the students could understand ('mental trace' is described next):

Code	Description
1	Study program text and deduce answer correctly
2	Recognise algorithm
3	Carry out mental trace correctly
4	Do mental trace and fail, then prompted and succeed
5	Do mental trace and fail, then prompted and fail
6	Study text and answer incorrectly - then prompted trace and succeed in understanding
7	Study text and answer incorrectly - then prompted trace and fail in understanding
8	Not understand question

For 3, 4 and 5, 'mental' trace meant a verbal rehearsal of what the program would do with suitable data values, but *not written down*. No student carried out a trace on paper unless prompted, as in responses 6 and 7. The following illustration (figure 7.1) shows a scan of a typical set of traces, with the interviewer showing the idea and then the student writing down the traced values. In 4 and 5, 'prompted' means being shown how to do a trace on paper.

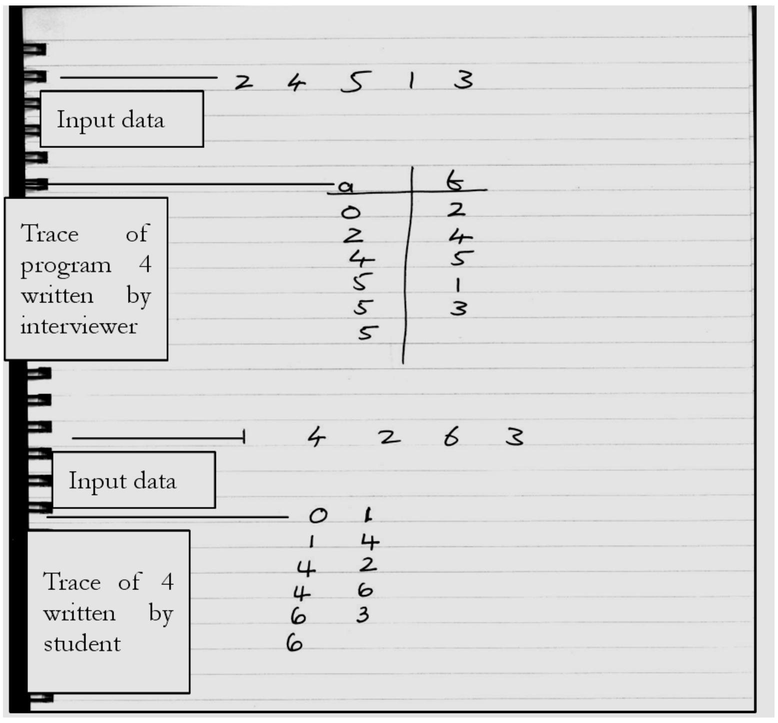


Figure 7.1 Typical trace

What ‘understanding’ means is described for each program below. Students are usually eager to be told whether they are 'correct' or not - but there is some tension with this and a phenomenological point of view, where we are using what the students say as data and notions of the 'right' answer are not appropriate. But for these programs there are clear *normative* versions of what these programs 'do', and a student is deemed to understand if they can describe that. Even within a constraint of normative correctness, there are different depths to which a student can grasp what the program does. This is discussed later.

Typically several traces were worked through for programs 2, 4 and 5, and this provided students with some empirical evidence upon which they could base some inductive reasoning - in other words they could try and ‘spot the pattern’. This was not coded as understanding unless they could see the cause of the pattern.

After the 5 programs had been worked through, the student was invited to put them into an order of difficulty. Finally they were asked if they had seen program tracing before, or if it seemed similar to anything they had encountered before.

7.4 Student Codes and Transcriptions

Students are identified by a faceted code with the format

<institution><gender><course><number>

All these students had the institution CC. The course codes were

C = City and Guilds 7266 Diploma for IT Practitioners Level 2

B = BTEC National

F = Foundation Degree

<gender> is M or F

Hence CCMC10 is a male student following City and Guilds and is the 10th student interviewed.

In the transcripts, interviewer's words are in normal text and student's words are in italics.

Many of these students found the programs difficult, and they find it hard to articulate with any precision what they think is happening. This makes the transcription extracts rather hard to follow – but this reflects the students' grasp of the situation.

7.5 The Programs

Pseudo-code was used so that language-specific issues could be ignored. The idea of pseudo-code was outlined at the start, and no student expressed a problem with this.

Program 1

```
a = 4
b = 5
c = a + b
output c
```

Normative understanding

The programs adds 4 and 5 and outputs the result.

This was chosen to be extremely simple, carrying only the basic ideas of variable, value, arithmetic and output. Given that all of these students had completed at least one course section in programming, it was thought these ideas would be familiar. It was chosen to be the first so that it would relax students and give them confidence. All students gave response 1 to this, and all but one ranked it as the easiest. Nevertheless one student found it non-trivial, ranking it third out of the five:

CCMC10

So number 1 was in the middle (in ranking of difficulty)

Yeah

What was difficult about that?

Er number one like, the difficult was this just about how to do it its the 4 when you go the a equals to 4

Yeah

b goes to 5, and you know, and c equals to a, plus the b, know what I mean like, it muddles you up even more,

Why does it muddle you up?

I don't know like, its just when I put a equals to 4 like, b equals to 5, I don't know which I should choose, from, numbers,

OK

and the output c when you're saying c equals a to b, a plus b, output c I didn't get like that part.

OK

This is unexpected. The student cannot articulate how he is 'muddled', and since the interviewer also had no idea at the time, it was not pursued in the interview. It was not simply the case that the student was very weak, since he responded with 6 on programs 3 and 4. This is returned to in Chapter Ten.

Program 2

```
a = 0
b = 0
while a not equal to 4
     b = b + a
     a = a + 1
output b
```

Normative understanding

The program calculates 0+1+2+3

There are three ways in which this is more complicated:

1. The loop, including its negative aspect

2. Both statements b=b+a and a=a+1 were not easy to make sense of for many students.

3. The fact that two things happen in the loop - the use of variable a as a counter, and changing b based on this counter which would itself change.

Here is an example of a student who could show some insight into this:

CCBM04

> So it outputs 6 - could you, I mean, how come it ended up at 6?
>
> *Its because it adds the a's to the b for from each time it goes round the loop, until it gets to 4.*
>
> So you said, it adds the a's?
>
> *The a, to the b that's already there, which is like, on the second time it will 1, plus 2 which equals 3, so it adds the a's to the b that is already there from the loop before each time it goes round*
>
> OK
>
> *Till a gets to 4*
>
> So could you do a little calculation that would end up as 6? How's it got the 6?
>
> *How's it got the 6? It would be a plus b which is zero, the first time,*
>
> Yeah
>
> *So it will be 1, plus another one which is 2,*
>
> Yes
>
> *Equals 2*
>
> Yes, its got 2,
>
> *Then its got 3 because you add the 2 to the 1, the a's gone up to 3 because you add another 1 to that, then you add those 3s together and it comes out with 6.*

However, twelve out of fifteen students responded with 7 for this. The student would study the program text for a while, possibly saying the output would be zero, because b=b+a and a=0, thereby ignoring the fact that a will change. If they gave an incorrect answer like this, or were unsure what the output would be, they were then led into the idea of a program trace, and this was worked through with varying levels of difficulty on the part of the student. Only 2 students could come to a full understanding (response 4).

At the end of the prompted trace, students agreed the output was 6. The usual response to 'why 6' was that 6 was the value of b when the loop finished. For example:

CCFM11

> OK so this outputs 6 - why does it output 6? How did it - why is it 6?
>
> *Because the last increment on the b the last variable stored at b is 6 so it is showing the output b.*
>
> OK er how come we arrived at the number 6? Its done some calculations and the result of the calculations is 6 - how - why is it 6?
>
> *Because there are conditions when as long as the loop is running it is adding numbers*
>
> OK
>
> *so when it comes up to the 4 the loop will stop then the last variable or the last number stored in b that will be 6, so it will store the 6*
>
> OK is there any reason why it turned out to be 6?
>
> *Why it turned out to be 6?*
>
> Yeah
>
> *Because of the loop*
>
> OK
>
> *Because we put the condition while a not equal to 4*
>
> OK

Program 3

```
input a
input b
c = a + b
output c
```

Normative understanding

The program inputs 2 numbers and outputs the total.

This is a modification of program 1, with the introduction of user input. A program can be seen as a function, mapping input data to output, and program 1 is a constant function – it always outputs 9. This program is the function f(a,b) = a+b. Students were not expected to see it in these terms, but the point was to see if they could cope with input values which were not specified.

Eight students ranked this as second easiest, and a further 2 ranked it as third. An example of a student who had some problems with this, ranking it 4th out of 5, was as follows. The student can only grasp it when actual input values are specified:

CCMC08

> This is program 3
>
> *Input a*
>
> So we've got input a, so the person would input a value for a through the keyboard, input b, c=a+b, output c – so what does that program do?
>
> *c= a + b.. c=a+b.. so the person puts input a, then input b,*
>
> Yeah
>
> *And the value of that should be.. b, c, um, I don't know, b, a, it puts the value a it puts the input a*
>
> Yes
>
> *Then input b,*
>
> Yes
>
> *And then the output, the output should be from a and b, c=a+b,*
>
> OK
>
> *Actually b, don't get it what the answer should be*
>
> OK suppose we ran it, and the first number we typed in was 2, and then when it said input b we said 4, what would it display?
>
> *It would display 2 and 4, and it will add it up, 6*

Program 4

```
a = 0
repeat 5 times
```

```
          input b
          if b>a then a = b
output a
```

Normative understanding

The program inputs 5 numbers, and outputs the largest (if at least one input value is greater than zero).

Two students produced response 1 - study the program text and give the correct answer. For example:

CCBF14

> So what will that do?
>
> *You put something in for b, yeah?*
>
> Yeah
>
> *Then if b is greater than a, then a becomes b, a is equal to b*
>
> Yeah
>
> *..Oh, OK yeah, so I guess it will um whatever you put in for the input to b,*
>
> Mm
>
> *then um if its greater, if its greater than a then the um number for a becomes b because its greater, yeah?*
>
> Yeah
>
> *Then after doing that 5 times if the numbers are higher than what a is yeah higher than what a is it keeps doing that but if its not then at the end of the while it displays whatever a is at the time so*
>
> OK

There is a pause before the '*..Oh, OK yeah'* and after that point she rapidly articulates what will happen, so it seems likely that at that point she had understood. This continues:

> *I guess like if you were to lets say you were to lets say you a starts at zero so it doesn't really have a number so whatever you put in for b becomes a*
>
> Yeah

So its equal to 3 then after that if you put 4 in it becomes 4 but if you put like 3 again then a still stays at 4 because its higher then you do that 5 times I guess whatever the highest number that's put in then that's what a outputs at the end of it

Here she is doing a 'mental trace' in the sense of choosing input values and thinking what would happen, but without writing it down. However she has already indicated what the program does, so this is likely to be for the sake of explanation, rather than trying to see what it would do.

The interviewer asks for an explicit statement of what the student thinks the program does:

Right
Is that right?
So this program - how many numbers does this program input?
It inputs 5
OK, and what does it output?
It outputs the highest number
Right, OK
Is that right?
Yes, yes.

However this response was unusual. Eight students gave the response 6 – initial reading yielded a wrong answer, but one or (usually) more traces resulting in a pretty secure idea that this would input 5 values and output the largest. A common initial error was to take a=0, so the if b>a would always be true (negative values were not considered), so this would just happen 5 times and the output would be the final b – so not seeing that a might change. Five students responded 7 - even after tracing it through several times they could not see the largest value would be outputted.

Most students, but not all, saw that 5 numbers would be input, and that these might be not all equal.

The typical problem with this was the if statement. There are 2 difficulties:

1. Some students would read this as something which was self-contradicting, in that it was read that if b>a, then it was asserted that a and b were equal. This relates to the confusion between the assignment a=b which changes a, and the boolean condition usually written a==b, which is true if a equals b

2. Some students intermittently in effect lost the ‘idea’ of the if, namely that a=b was only carried out if the condition was true. This may have been due to a cognitive load problem – attention was focussed on b>a, and effort devoted to seeing if that was true or false, but this drove out the idea of the if, and the next task was seen as a=b, irrespective of the condition.

This characteristic of understanding coming and going rapidly is illustrated here:

CCBM03

> Its got 2 variables, a is 0, then its got a loop, another kind of loop, its not a while loop, it just goes around 5 times. So those 2, are inside the loop, and it does that 5 times, and inside the loop it inputs a value for b, and it says if b is greater than a, the a equals b
>
> *So if zero is greater than, um, a is zero, yeah? If b is more than, greater than a, then the a should be b, so its opposite sides like, like to opposite sides, if b is greater than a, then a should be b, and the output will be a,*
>
> OK
>
> *But the output should be 5 then, 5 times, zero times 5 is zero,*

This implies the student does not understand the loop construct 'repeat 5 times', thinks multiplication is involved because of the use of the word 'times', and the 'zero' implies variable a is being multiplied. But this is immediately altered:

> OK where does it.. OK so, suppose we had, suppose we ran this program - how many numbers does it input?
>
> *5*

So now he correctly understands the 'repeat 5 times' construct:

It inputs 5 numbers. Suppose those numbers were 3, 7, 8, 2 and 1. Suppose we inputted those 5 numbers - what would this program output?

Zero. Because there's a.. because its 5, 5, so it should be.. it should count all these numbers, its going to output them, as well

He responds immediately, but with two different answers - 'zero', and 'them'.

OK lets track it through, OK, so to start with a is zero, so then we do this 5 times, we input b, and we're saying that would be 3, the first value we input is 3. Then it says if b is greater than a, then a equals b

So if it's greater than a

Yeah?

Yeah b is greater than a

Yeah it is, so what does it do?

Then the outcome should be b

So does a change its value?

Yeah because its b its going to becoming a.

So a will become?

3

Here, b is 3, a is 0, and the student is trying to work out the effect of 'if b>a then a=b' . His speech pattern is restricted - he says 'if it's greater than a' rather than 'if b is greater than a', and 'the outcome should be b' rather than 'a will be b'. Consequently establishing the fact that the effect is that a becomes 3 takes time.

3 so a becomes 3. So that's the loop once, do it again, input a value for b, we input 7. It says if b is greater than a, a equals b

So if b is equal a

He should have said 'if b is greater than a'.

If b is greater than a, then a equals b

So it should be 7 in here

a becomes 7?

Mm

OK that's doing it twice, do it again three times, b becomes 8. Then we have the if statement. How does that change, does that change a?

Because it's 1 number different, so a will be.. no it won't because

He has noticed the fact that b differs from a by 1, and seeks to determine some consequence of this, but there is none, since the fact is irrelevant. He needs to be refocussed on the task of deciding whether b is greater than a.

> OK so its saying if b is greater than a.. is b greater than a?
> *Yeah*
> Yeah? So it says a equals b - so how big does a become?
> *8*
> 8? OK so in the fourth time we input a 2,
> *Mm well this can be less than a*
> OK so will it change the value of a?
> *No*
> So a..
> *is lesser than b*
> So a is still?
> *Bigger than a b*

The a b is probably a correction - saying a but meaning b. Otherwise he says that a is bigger than a, which seems unlikely. So he means that a is bigger than b, correctly, since a is 8 and b is 2

> So after this how big is a*?*
> *How big is a? 8*
> Its still 8?
> *Yeah*
> OK. Then we do it the last time and we get 1, so b becomes 1,
> *Mm*
> So we do that if, if b is greater than a?
> *Then the a should be 1*

Even after 4 times, he is not clear about the if statement. It could be that he is speaking hypothetically, as in 'if b were to be greater than a, then a should be 1'. But he uses the actual current value of b, so he is not speaking generally. He needs to be focussed on deciding if the boolean condition is true:

Is it? Is b greater than a?

Err b is lesser than lesser than a

So it doesn't do this?

No no its lesser than - if it was bigger than a, a should be 1, but it wont like 9

Yes OK

It should be 9 there, its lesser than, it shouldn't be changing there

So it stays at the 8

Yeah 8

8. And then that's what's output, yeah?

Yeah that's the output

8 is what is output. OK

Mm

The next question is to determine if he can see what the program as a whole does:

So could you summarise what this program does?

It means like if something's bigger than the other one, if its like a bigger than b the output should be like working to a so if a was zero like and b was 3, the 3 comes to a

This relates to a single execution of the if, not the cumulative effect after 5 iterations. Another set of input data is used to see if he understands that:

OK suppose we ran it again with another set of 5 numbers, suppose those 5 numbers were 2, 9, 4, 12, 3. What output would you get?

Um.. 12

Why would you get 12?

Because its bigger number.

OK

The 3 is the less number.

The next question concerns whether this is just for this input data set, or always:

> So could we summarise what this program does? It inputs 5 numbers and it outputs..
> *Outputs the maximum number.*
> The maximum number? OK. Good.

Program 5

```
a = 0
b = 0
input c
while a not equal to c
     b = b + a
     a = a + 1
output b
```

Normative understanding

This program inputs an integer n and outputs 0+1+2+..n-1

This is a generalisation of program 2, generating the (n-1)th. triangular number, where n is the input.

The idea was to see if students were able to absorb, and recall, the structure of a program, since this was needed in order to recognise it. In fact 9 students did not recognise it, even after an explicit prompt ('Have you seen a program like this before?')

Five students ranked this as 4th hardest, and 8 ranked it as the hardest of all.

The following example illustrates the case of a student who has a partial grasp, seeing the result as a sequence of additions, rather than encapsulating the sequence into a single step:

CCBF14

> So this program outputs 10 - how come its 10?
> *Because um - that's what the numbers add up to?*

This is the common first response. To take it further, the interviewer suggests 'working backwards'

> Right the numbers add up to 10, it comes out as being 10, yeah, and how did it end up as 10?
>
> *I don't know a way*
>
> Could you sort of work it out backwards?
>
> *So that's..*
>
> So its10, how er what
>
> *How did we get the 10?*
>
> Yes how did we get the 10?

The student interprets this very literally, in that rather than following the steps backwards she actually tries to reverse the process, doing subtractions rather than additions:

> *4 and 6*
>
> OK so we added 4
>
> *So we take away 4 wouldn't it, go back to 6,*
>
> OK yeah
>
> *3, go back to 3, take away 2 go back to 1, take away 1 it goes back to 0.*

She can then go forward again, but still as a set of steps:

> Yeah - so the 10 is equal to..
>
> *10 is equal to 4 plus 6*
>
> Yeah and where does the 6 come from?
>
> *6 comes from the 3 plus 3,*
>
> OK where does the 3 come from?
>
> *2 + 1*
>
> So can you do it like, altogether, the 10 is
>
> *The 10 is..*
>
> Four, plus
>
> *Four plus 6*
>
> Yeah?

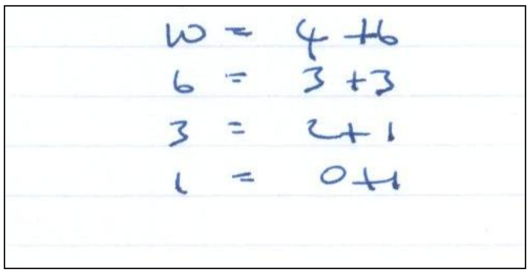
That's it isn't it?

But where does the 6 come from?

Oh the 10 is 4 + 6 and the 6 is the 3 + 3, and the 2+1 and the 1+0. I don't know how to explain it.

Yeah yeah can you write it down?

This is what she wrote down:



10 = 4 +6
6 = 3 +3
3 = 2+1
1 = 0+1

Figure 7.2 No compression

and not 10 = 4 + 3 + 2 + 1 + 0

She has a clear grasp of the sequence of statements but is not compressing them into a single process.

The following transcript sequence shows a student who's understanding develops considerably during the interview. He has problems reading the code, but he has a breakthrough in realizing how the program changes the variable values, and how those values need to be recalled to see what the program code does:

CCMC10

So what we can do is er to write down on paper these values as we go through it, so we don't have to keep it in our heads. So to start with a is zero

Right

and b is zero, then it says b = b + a, so, b+a is?

Going to be zero, both of them

Zero, then it says a = a + 1

a equals plus one is going to be one

He reads a=a+1 as a equals plus 1 - something he repeats several times

Yes, now it says while a is not equal to four, a is not equal to four so we do it again

Right

b=b+a

b equals plus a?

Similarly, he reads b=b+a as b = + a

Er b=b+a. b becomes?

Zero - no actually that's going to be wait there b plus that's going to be er one

OK so b becomes one

Its already got one there so that's one isn't it?

Yeah?

When you plus the a

Then we say a = a + 1, so a becomes equal to?

a goes to two no a equals plus one its going to be one as well

He corrects his initial mistake, but we clarify this:

a equals a plus one

oh a oh a plus one is going to be zero wait there er

So a starts off as one,

Yeah

Then we say a=a+1

So a equals so a (laughs) equals a equals plus one that's going to be er.. er plus one that's going to be two then?

He laughs because he is aware that he is having problems saying a = a + 1

Two

You're adding that as well aren't you? Yeah alright

So a becomes equal to two?

Yeah

Its still not equal to 4, so we do it again, b=b+a,

That's going to be one er another one but you can add a two instead

So b is one, a is two,

Oh that's going to be three

Three. So b becomes three. Then we say

At this point the student interrupts excitedly. He has had a sudden insight into what we are doing. He sees we must use the variable values, not just look at the program. After this point it's easy - he leads it forward quickly

So you're saying you don't add these (points to program text) you add up these (points to trace of variable values) instead on the sheet like what you've put in like you say put two there and like you're going to say a=a+1 so a equals a equals a yeah that's going to be these (trace of values) that's this one now

Yeah

plus one so that's going to be three as well.

Yes yeah

Yeah. Then you're going to write b = b + a

Yeah?

So you're going to add this time this one now so

Yeah?

add that to er wait there that's going to be four then

b=b+a

b equals plus a that's going to be b equals so you're going to add that one

Yeah?

So that's going to be 6 then?

So we're going to have 6 - then it says a=a+1

a - so a equals plus one, so that's going to be four

..

However the most common response (eleven students) was 7 – following a trace, they did not grasp what the program did. For the trace a value of c=5 was used, since 4 would have given the same as program 2, and more than 5 would have been very tedious.

However those eleven students did not perceive that the output was 10 because 10 = 0+1+2+3+4.

The following example illustrates the common features of intermittent understanding, mixing up the values of a and b, and mis-applying the two action patterns (incrementing the variable, as in a=a+1, and adding them, as in b=b+a)

CCBM03

> We've got 3 variables, a b and c, a=0 and b=0, input c, while a is not equal to c, b=b+1, a=a+1, and then we output b after the loop.
>
> *a is b a is zero b is zero OK zero b is equal to b plus a, so something coming a b the answer should be coming c.. the answer is based on c*
>
> Yeah?
>
> *Mm so a will be equal.. so if a is zero, the answer should be zero as well,*
>
> OK

This is a common first response, reasoning that a=0 and b=b+a, so b remains at 0. This therefore ignores the fact that a changes, and its significance is discussed later.

> *Equals zero b equals..*
>
> Have you seen a program a bit like that before?
>
> *Er no I don't think so.. a equals zero b equals zero input c while.. What it means? b? and b plus – b is zero*
>
> Yes
>
> *And b is zero as well, if you add one, should be coming 1,*
>
> Yes – was that a becoming 1?
>
> *Yeah because look it says b is zero,*
>
> Yes
>
> *b equals b zero plus a, so a is 1 er a is 0 if you add 1 its going to come 1 then output 1*

He has seen that a will become 1 at the end of the first iteration. But he ignores the loop, and the fact that b is output, not a.

We start the trace and complete the first iteration:

OK OK suppose we ran that

Mm

And the person typed in 5 for c, OK? So c is 5 (starts to write table of values)

Mm

And a is zero and b is zero, and we come in here, while a is not equal to c, and its not so we carry on, and it says b=b+a, b is zero and a is zero, so nought plus nought is nought

The answer should be zero

b is still zero yeah? Then it says a = a + 1,

It should be 1

a becomes 1 yeah?

On the second iteration, he needs help both with b=b+a and the a=a=1.

Go around, well, while a is not equal to c, 1 is not equal to 5, so we carry on, b=b+a, so b becomes how big?

Er b equals zero, no no sorry

b is zero, a is 1, so b+a equals?

5 no sorry 1

OK so b becomes 1, because it says b=b+a

Yeah

So its 1 OK? Then a=a+1,

It should be 1, it should be 2

a becomes 2, it is 1, then we add 1 to it so it becomes 2

Because its adding in numbers like

On the third iteration he needs help with a=a+1. His first answer for this, 4, may be because he is using the wrong variable, incrementing b (which is 3) not a (which is 2)

Mm go round again, b=b+a, so how big does

Three

b is 2 + 1 is 3, OK, then a = a + 1, a becomes?

Er 4 - no, a becomes 3

Becomes 3, um,

On the fourth iteration he needs help with both statements. His first answer for b=b+a is consistent with using the wrong pattern - he is incrementing b not adding a and b:

Go around again, b=b+a,

It should be 4

b is 3, and a is 3

6

So it becomes 6, and a changes to

Er

a=a+1

Er 7 oh no 6

He may have arrived at 7 by using the right pattern (increment) but again mixing up b=6 for a=3

> a er right a equals 3, how big is a plus 1?
>
> *a + 1 .. 4*
>
> a becomes 4, and the loop while a is not equal to c, they are not equal
>
> *No so it should*
>
> Carry on,

At the start of the final iteration a=4 and b=6. His first answer for b=b+a is again consistent with using the wrong pattern - he has incremented b not added b and a:

> b=b+a, so b becomes how big?
>
> *b becomes seven er six*
>
> b = b+a
>
> *4 + 6 – that should be 10*
>
> 10. a becomes 5
>
> *Yes 5*
>
> And its going to stop it says while a is not equal to c, now it does equal c,
>
> *Yeah it should be equal to*
>
> So it stops. It outputs b
>
> *Should be 10*

Not surprisingly, he cannot see why the result is 10

> Which is 10. OK so this program – if the person said c was 5, this would output 10. Yeah?
>
> *Yeah yeah because you know the thing the number is going up by these numbers and we are counting and these are the numbers its going by this way like there's 1, like its going 1 2 3 and the number is going up by b like*

OK so a goes up 1 at a time

Yeah

0 1 2 3 4 5

Mm

This 10, how did we get 10?

By adding it, like adding a, from a to b,

OK

And b can a like by adding like there and the number each time is a, so we are adding numbers in b, so they kind of go

All he can see is that it involves addition:

OK so if we covered that up (covers up the a column) 10 comes from?

a

Adding up what?

Adding up numbers

What numbers?

Er from a I think, adding up numbers

OK

Like 5 like

7.6 Seeing tracing before

Very few students could recall seeing program tracing being used in a programming class.

Two students were reminded of maths classes without being able to remember specific topics, but two mentioned working out a table of values prior to drawing a graph, two referred to drawing up tally charts in simple statistics, and one mentioned long division.

For example:

CCCM05

OK this way of looking at programs, of checking through the variables, the values of the variables step by step, have you seen that way of doing things before?

No, not in the class, no. In maths I suppose I have, in the past. Not so much in programming.

OK OK. In maths, can you remember like where in maths you saw it?

Algebra probably, no er kind of when you're using ranges and, do you know what I mean? What is it called now? A lot of ticks and things, range boxes. Any idea what I mean?

Ticks in range boxes?

Or even er x and y graphs, kind of

Graphs? Graphs? For drawing graphs? So how does this connect with drawing graphs?

I can't remember, its a long time since I've been doing maths - I think I definitely recognise it more from maths, I'm not sure where..

Yeah yeah

In there, he doesn't tend to write things out like this which is probably a bad thing

So its to do with drawing graphs, back when you were doing maths so its a long time ago?

Mm

OK so we are talking about using graph paper, not on a computer

Yeah yeah

On paper, plotting points

Yeah plotting points

OK

We'd have an algebraic sum

OK

and work it out and then plot its

So you work it out then you plot it?

So

So you'd have something like y = x + 2?

The x would be here and the y would be here, you'd draw your dots, then join up the dots kind of thing.

OK would that thing have been called a table of values?

Yeah that's the one, yeah

For one student, this process was reminiscent of a card game:

CCMC10

This way of doing things, where you, its called a trace, when you trace through a program you have the variables, the values of the variables you write them down after each step - have you ever seen that before?

No

No. Never seen anything like it?

Yes I've seen it once in cards in games when you play games

Yeah?

Like you say I'm a zero but I want to be higher than you, like, higher and lower

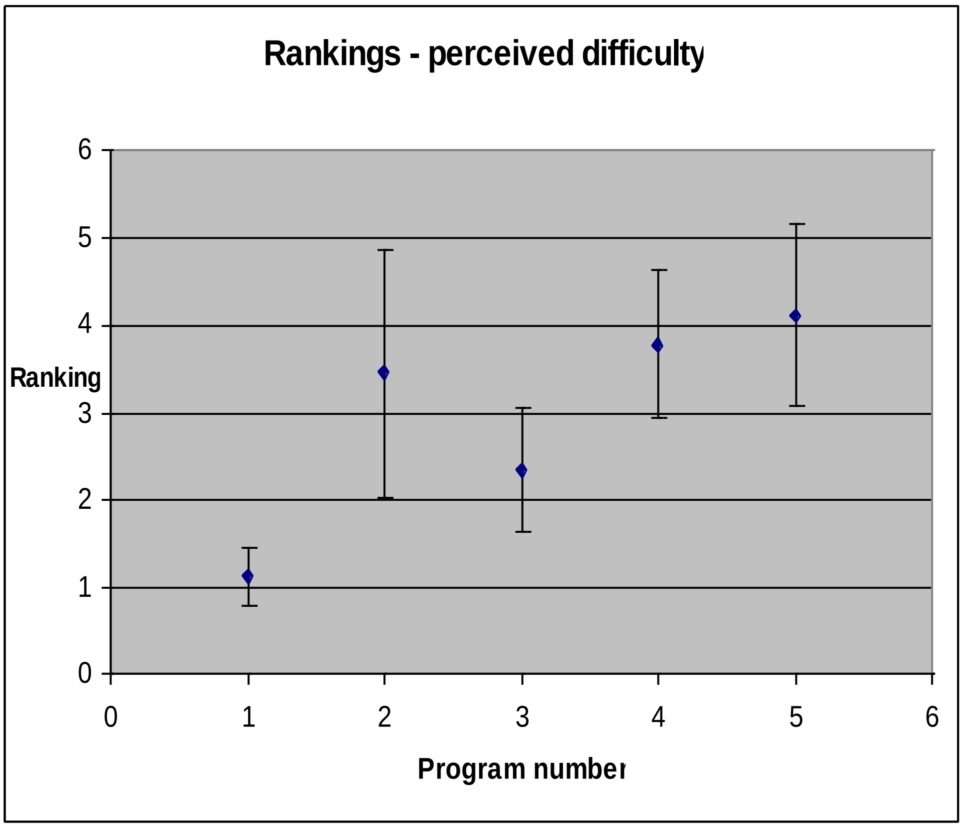
OK, so like these things written down, these are like the cards on the table

If you want higher and lower, you've got to add, like I'm a number one, and you're a number two, and OK I'll have these two.

If students develop an understanding of program tracing, and thereby a picture of program execution, by a process of the radial extension of meaning, the point is that they are coming from many different directions, in that there is a wide range of things which students will relate tracing to when it is first presented to them.

7.7 Perceived Difficulty

The perceived difficulties of the 5 programs based on their rankings is:



This shows the arithmetic mean of the rankings of each program, with the vertical bar showing one estimated standard deviation above and below the mean.

As expected, this shows 1 being seen as very easy, with program 3 not so easy, since the data values are run-time input and not constants. Programs 2, 4 and 5 are harder, but with little difference between them.

7.8 Responses by program

These are as follows:

		Counts of responses				
		Program				
Response		**1**	**2**	**3**	**4**	**5**
1	Study program text and deduce answer correctly	15	0	12	2	1
2	Recognise algorithm	0	0	0	0	0
3	Carry out mental trace correctly	0	0	0	0	0
4	Do mental trace and fail, then prompted and succeed	0	2	0	0	2
5	Do mental trace and fail, then prompted and fail	0	0	0	0	0
6	Study text and answer incorrectly - then prompted trace and succeed in understanding	0	1	3	8	1
7	Study text and answer incorrectly - then prompted trace and fail in understanding	0	12	0	5	11
8	Not understand question	0	0	0	0	0

All students can determine what program 1 does 'by inspection', and the same is almost true for program 3.

For program 2 (sum 0 to 3), all but 3 students needed to be prompted to follow through traces, but still could not understand it. Program 5 (sum 0 to n) was very similar.

For program 4 (maximum of five inputs) there is a mixed response. Two could understand from the program text alone, just over half could understand it after following through the trace, and the rest could not even after tracing it.

7.9 Chapter Summary

The chapter looked at the ways vocational students understood five fragments of program code. The interviews show the following:

- As expected there was a range of difficulty, and the ranking according to their actual responses was consistent with their perceived difficulty. Calculation on constants is easy, calculation with run-time input not so easy, and loops and ifs are harder still.

- Some students found very simple code sequences difficult, and some seemed to only become aware during the interviews that program code changed variable values, and that variable values affected what program code did.

- Students responded to the question 'what does this program do?' at a variety of levels. Some saw this only at the level of individual statements operating on given values. Others could say what the program as a whole did, for a given set of input data. Others could say what the program would output for any input data.

- Some of the transcriptions show how some students struggled to be aware of wholes at the same time - such as remembering an if condition whilst grasping the conditional body, or remembering the loop as well as the if contained in it. This corresponds to the previous point - these students could not grasp the program as a whole, and so could not 'understand' it.

C H A P T E R 8 - P h a s e T w o U n i v e r s i t y

CASE STUDY DATA

8.1 Introduction

Up to this point, all data gathering had involved a single interview with each subject. While this yielded interesting insights into possible theoretical frameworks, it could not show anything about the development of concepts in an individual pursuing a programme of study. Consequently it was decided to interview a small number of individuals over as much of an entire academic year as possible.

Two institutions were happy to co-operate in this work. One was a research-led Russell Group university which was enrolling students with high academic grades. For this institution the subjects were undergraduates on an Honours BA degree in Computer Science. The other institution was a College of Further Education, which was focussed on vocational courses of a very practical nature. These students were in the first year of a Foundation Degree programme. At both institutions students were following units in software development using Java. This chapter describes the university students, and the the FE College students are described in the next.

This chapter first discusses abstract classes and interfaces in Java, because some of the interviews concern these topics. The rest of the chapter consists of a series of sections, each about one student. The three students are called Fiona, Jennifer and Matthew, and there is some cross-referencing between them. Each section comprises an introduction, an account of each interview together with a summary of the longer ones, and a concluding overview.

8.2 Java abstract classes and interfaces

These interviews touch on the concepts of abstract classes and interfaces, and so for the benefit of non-Java readers, a little background is provided here.

Classes usually occur in hierarchies, with sub-classes extending base classes. The idea of an abstract class is that it would be the base class of a set of related sub-classes, at a level of generalisation such that defining methods would be inappropriate. For example a Shape class might be abstract, and have non-abstract sub-classes of Circle and Rectangle, The Shape class would have a method declared with the keyword *abstract*, but with an empty method body, like

```
public abstract void draw()

{

}
```

The consequence of this is that any concrete (that is, non-abstract) sub-class of Shape must have a draw method, which must be implemented - that is, how to actually draw it must be programmed. Otherwise the sub-class would itself have to be declared as abstract.

The idea of *abstract* is to express the idea of a general 'thing' which can do something, but is too general to be able to say how it does it. Java syntax prevents a programmer unthinkingly instantiating a 'general thing', and requiring an 'actual thing' to have the method coded. There cannot be an instance of an abstract class - this is important for the interviews.

An interface is a set of methods, in the form of method names, but not coded definitions. In other words an interface is a set of things that a class might be able to do. For example, the KeyListener interface has three methods:

```
void keyPressed(KeyEvent e)

void keyReleased(KeyEvent e)

void keyTyped(KeyEvent e)
```

Pressed and released deal with basic key actions, such as the SHIFT or ALT keys being pressed or released. KeyTyped deals with actual characters, such as 'A' (a combination of SHIFT and the A key).

Any class which 'implements' the KeyListener interface must have those three methods, with coded definitions of them.

Interfaces and abstract classes therefore have features in common - methods with names but no definitions, and the 'no object' idea. But the hearts of the concepts are different - an abstract class is a class, but an interface is not.

8.3 Fiona

8.3.1 Interview One

When these interviews took place, Fiona was an 18 year old undergraduate. In the previous year she had attended a sixth-form college, where she took A levels in Computing, Biology and Film Studies, obtaining grades of B, C and A respectively. In her second sixth form year she took Mathematics at AS level, obtaining a D grade.

For her Computing A Level she studied Pascal, and wrote her project using Visual Basic and a MS Access database. Consequently she had a good deal of experience of programming at an elementary level.

These facts were brought out at the time of the first interview, and then a pseudo-code fragment used during phase one was introduced, with the familiar phrase 'what would this program do?':

```
a = 0
b=0
repeat 5 times
    a = a + 1
    b=b+a
output b
```

Fiona immediately wrote (see Figure 8.1) a trace, to give the correct result that b=15 is output. She was not familiar with the term 'trace', but had previously seen examples of variable values being displayed as code executed step by step.

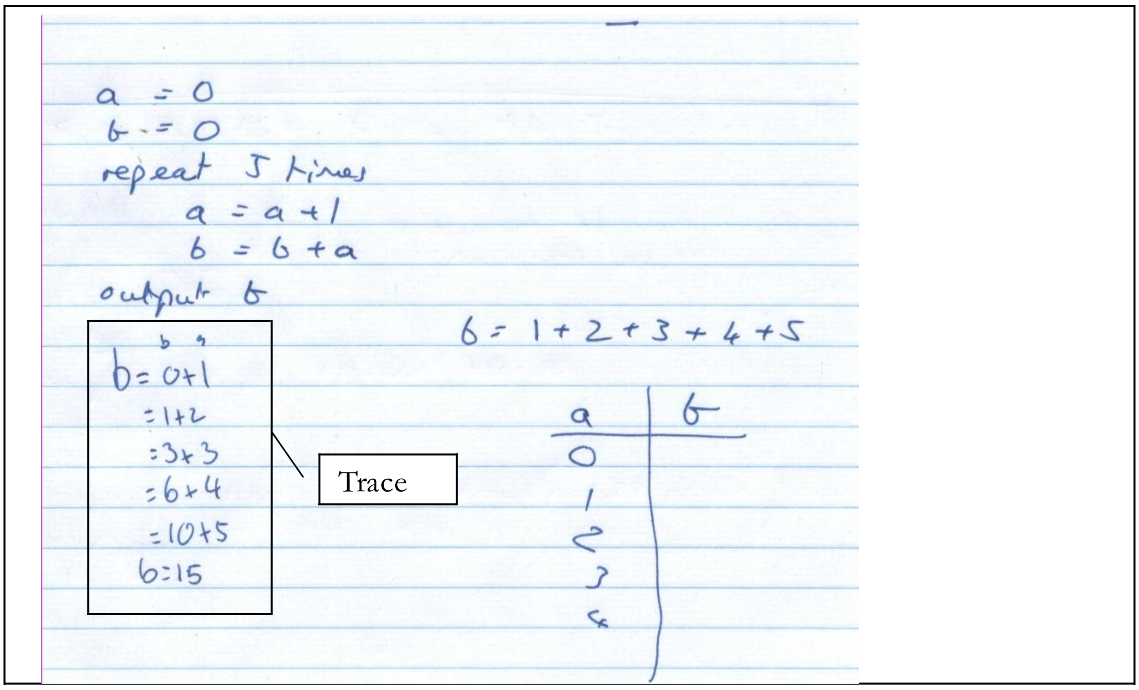


Figure 8.1 Program trace

The question 'why is it 15?' was answered by the common reason that successive values of 'a' were added, and that 'a' was also being incremented. The question was asked whether this could be done in one step, and when there was no response, it was suggested that this was could be done in one step by:

b = 1 + 2 + 3 + 4 + 5

Fiona denied this was the same, since in the code b was added in as well as the successive values of a. This implies that she does not see this as figure 8.2

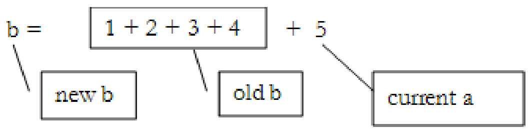


Figure 8.2 Compression

8.3.2 Interview Two

This began by checking which topics had been covered so far. The following terms were recognised:

class, object, constructor, method, field, parameter passing, return value, static, abstract, public, private, interface, and Swing.

Encapsulation and access modifiers were not recognised, and polymorphism and inheritance were doubtful. Swing (a set of standard Java packages used for GUI programming) was only used for the Timer class.

Several interesting points emerged:

> Encapulation?
> *Um.. I don't think we've done that*
> OK probably not. Access modifier?
> *Not done that yet.*
> Public?
> *Yep.*
> Private?
> *Yeah*
> OK so what's public and private all about?
> *Public, we've been using with the methods, so like it would be public static void main, or public boolean or whatever, and private is in defining fields in classes, so we haven't been making private methods, we've been making private fields*
> OK um so if a field is private, what difference does that make?
> *You can't change it unless it is within a public method, so its just - when you are making a class, it means that people can't tamper with that field unless they run a method to change the value*
> OK - so you could have a public field, is that right?

Yeah

But what - you are not supposed to?

Um XXXX has just been teaching us to make private methods because its good practice - people can't tamper with your program.

Private fields

Yeah

Because its good practice, because they can't tamper..

So if you make a program on the Internet or something, if its private it will be harder for people to be able to change the initial value you set for those fields

This is interesting, because encapsulation and access modifiers are unfamiliar, yet public are private are thought of as well-understood, and are seen in the light of security. In fact public and private *are* access modifiers, and are intended to enable the idea of encapsulation (providing access to the state of an object in a controlled manner, intended to prevent a *programmer* accidentally corrupting object state from code outside the class), rather than anything to do with *security*. Interviews with other students (see 8.4.2 and 8.5.2) indicate some students shared this view, whilst others had a more normative viewpoint.

Interfaces were recognised. This was pursued:

Have you used an interface in programming?

Ur yes last week's exercise was all about interfaces.

OK so what is an interface?

Um its like a class but you can't make instances of it so it has methods which can be implemented but only in a class that implements the interface.

OK

So you can make an interface that has 3 methods, you can only use those methods in a class that will extend that interface.

OK. So what would be the point of an interface? I mean what's the purpose of the idea?

Sort of like, XXXX is saying its like an adaptor so like if you've got a class of one type, you want to return something of another type, you could use the method to return the right type.

OK

(laughs)

On a scale of 1 to 10, how much sense does that make?

Not a lot.

Interfaces are used for (among many other purposes) dealing with event-handling in a GUI - such as programming what happens when the user clicks a button. This often done using 'adapter' classes, which is what Fiona referred to. However adapter classes entail the following concepts:

- abstract classes
- inner classes
- anonymous classes
- anonymous objects

in addition to the idea of an interface. We will see that some students do not have a clear concept of a (standard) class even by the end of the module, and it is unsurprising that at this stage Fiona is unclear.

The interview then proceeded to a discussion of The Game (see Appendix One). The student was given this on paper, a check was made if there were any questions about it, and then it was discussed.

Fiona suggested there would be a class called FirstNumberAssignment, which would have a method called numberGetterPlayer. This would return an int, which would be called from main and the returned value would be assigned to a variable local to main. Similarly that class would have another method called numberGetterComputer.

Another class would be called Game, and it would have a method called randomNumberGetter, which would be called twice. This class would also have a method called comparison, which would decide who had won.

This approach is in fact procedural - the methods relate to the *processes* which must occur in the game, and the classes are bundles of related processes. This is shown when the question of 'objects' is addressed:

> What objects would you have?
>
> *Um err you'd probably have a player and a computer which would have their names like player or computer and then the values that had been assigned to them.*
>
> Right two objects and then .. ok these two objects, computer and player, what class or classes do these objects belong to?
>
> *They probably go in FirstNumberAssignment, although.. where to k..um where to (inaudible).. because you could probably do it without classes if all you are going to do is return a string based on who won.*
>
> You could do it without classes?
>
> *You could do it without objects.*
>
> OK

In this, the interviewer asks which classes the objects *belong* to, whereas the student talks about where they *go*. In other words a class is seen as a textual *containe*r to put things in, rather than a *type*. Further, this is a container of source code, rather than a container of data like an array. Since the player and computer objects are only thought of as having name strings, the student questions whether they need to be objects at all.

Further discussion shows that Fiona would not have any instances of the FirstNumberAssignment and Game classes. Then this happened:

> If you have this situation, where you have a class, and there are methods which belong to the class, but you don't expect to have any objects in there, is there.. what we're saying then is that these methods belong to the class rather than to an object, is there a keyword for that?

> *Abstract - an abstract class, because you don't make any instances of it.*
>
> OK

The phrase 'these methods belong to the class' is a clear 'steer' towards the static keyword, but abstract is offered by the student, without hesitation.

The difference between abstract and static was discussed. She had just done an exercise using the idea of abstract, involving a messaging system with an abstract class called BillUnit, with descending classes SMSUnit, CallUnit and GPRSUnit:

> Suppose if we didn't call this abstract, suppose we said this was just an ordinary class, and these descended from it, what's the difference?
>
> *Um (5 second pause) I guess there wouldn't be except um.. um you'd have to define all the methods that are in BillUnit because well*

This might indicate the student thinks that abstract is essential to enable the inheritance of methods, but she is now very uncertain.

In fact abstract and static are very distinct in meaning, but they do share the *no objects* aspect. There cannot be an object belonging to an abstract class, and a static member belongs to the class, not to an object. Consequently the two concepts share an aspect of *no object*. But otherwise they are unrelated. Yet this has led her to believe that abstract is appropriate here, when in fact (in her class design) static is the correct option.

Uncertainty about this called into question the whole idea of OOP. This was examined:

> What would you say is the point of object-oriented programming?
>
> *Um.. so instead of writing loads and loads of methods and having loads of variables you could have an object which is able to do all the methods to find in the class so instead of having to re-define all the methods in the class you can just refer to an instance of that object which will already know how to do all the methods*
>
> Yes OK right right so the object will know how to do the methods. OK now have you been told explicitly in class, the idea the purpose of object-oriented programming or are you working it out for yourself?
>
> *We're sort of working it our for ourselves we haven't been told explicitly.*

So as previously shown in this interview, a class is seen as a *bundle of methods*, which an object can use because it belongs to the class, rather than a class being a *type of object*. This mis-conception in turn is then being seen as the motivation for OOP. Further, Fiona is in the process of constructing the purpose of OOP.

The phrase ' *object which will already know how to..'* shows the 'object is a person' metaphor.

Summary

The student has a solid conceptualisation of 'private', but her idea is non-normative and unrelated to the 'encapsulation frame'.

The program design approach is still procedural

Classes are seen as containers of methods and objects

Abstract, static and interface have been used but the concepts are unclear or mistaken.

In general the student uses much of the vocabulary of OOP, but without clear conceptualisations.

8.3.3 Interview Three

This started by Fiona's confirming that she felt the interviews were not affecting her approach to study, since this was primarily motivated by her concern for course assessment. Since the last interview she had covered recursion and exceptions. Her idea of recursion is confident and normative:

> What do you think recursion is?
>
> *When I was doing A Level we got told it has a method that calls itself when its running, so that's how I've learnt it, something that calls itself.*

The example used in the Java class was the Fibonacci sequence. An *exception* is seen as a programming error, and the example given is an array bounds error.

Then the following task was presented:

input 5 numbers

output the largest

> *Well, each of the numbers would be assigned to a variable, and then you can compare them to each other, so you'd compare the first two, if the first one's larger you'd compare that to the third one, if that one's larger you'd compare it to the fourth, until you get the largest, almost like a search,*
>
> OK almost like a search, but not a search, you'd compare the different numbers
>
> *I'm trying to think which of the searches it was,*
>
> Ah these are the searches that you've done in Foundations?
>
> *Yeah*
>
> Right OK
>
> *I think it was the bubble sort where the largest or the smallest depending on what you are looking for is compared to all the elements then would end up at the bottom of the search pile.*
>
> OK
>
> (laughs)
>
> Right OK um so we have searching and sorting, are they the same or are they different?
>
> *No they are different,*
>
> They're different, so you're
>
> *I was getting them confused, yeah*
>
> OK so we're talking about sorts, so your method for this would be inputting the 5 numbers, and then sorting them,
>
> *Yeah*
>
> and seeing what is at the end.
>
> *Yeah*

So the approach to this is to use ideas recently encountered in class which seem relevant, rather than considering the problem directly. Further discussion elicited the fact that an algorithm to do a linear search for an extreme value had not been encountered in class (although Jennifer in another interview could recall this).

The interview ended with a discussion of classes and objects:

> What is an object?
>
> *Something that you can make instances of, it has its own methods and fields, that can be run by instances of that object*

(in fact this more describes a class, and a class cannot be 'run', though it can be used, which is possibly what she means).

> OK and there is the idea of class, and the idea of object. OK?
>
> *So an object is an instance of a class.*
>
> OK an object is an instance of a class, alright, what's in an object?
>
> *All the methods and fields that are in the class, that are able to be executed by the object.*
>
> OK where are objects?
>
> *Um, I'm not sure*

Further discussion shows that Fiona (correctly) thinks that objects 'start' when the keyword 'new' is used to invoke a constructor. We return to

> Where is it?
>
> *I'm not sure*
>
> Suppose I ask you - which part of the computer is it in?
>
> *In memory.*
>
> In memory?
>
> *Yes.*

So at this point her understanding of object is fairly normative, with a little more emphasis in 'running' things rather than the static nature. There remains the idea that a class is a repository for methods that an object can execute.

8.3.4 Summary of the Fiona interviews

It was only possible to conduct three interviews with Fiona. Nevertheless some useful evidence is gathered:

1. At the start she is able to work through simple procedural code fragments without help. Her understanding of this is limited, in that she does not see the one in the first interview as summing the first n integers.

2. She quickly acquires some of the vocabulary of OOP, but with limited or clearly non-normative conceptions.

3. She acquires an idea of *private/public*, but it is non-normative, relating it to security and not the OOP concept of encapsulation.

4. She acquires *abstract* and *static*, but conflates them probably because of their shared 'no object' aspect.

5. Her first notion of *class* is as a library of related procedures, which an object can call. This develops slightly by the third interview.

6. She is constructing the purpose of OOP - *"We're sort of working it our for ourselves"*

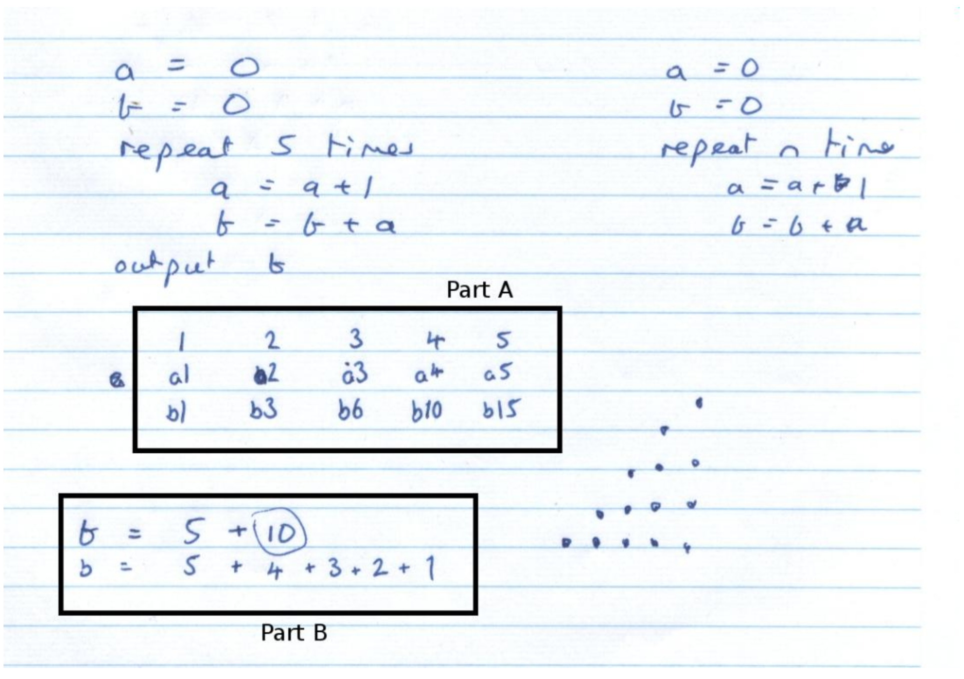
8.4 Jennifer

8.4.1 Interview One

Jennifer was a nineteen year old first year Computer Science undergraduate. She took A level Mathematics, Further Maths and Film Studies, obtaining A grades in all three. She initially intended to read for a degree in Mathematics, but decided that she really wanted to do computing. Her previous experience of programming was limited to some Visual Basic, used in a GCSE ICT project.

Some program code was introduced (Figure 8.3), with the question 'what would this program do?', and this produced a description of the loop, 'a' being incremented and added to 'b'. Then this happened:

> Can you tell me what the answer would be?
>
> *um (laughs) um*
>
> You could write things down if you wanted to - you don't have to write things down, you could work things out in your head, or you could write things down, as you want - whichever - its up to you
>
> *OK (25 seconds pause) is it 18?*
>
> OK do you want to work it out on paper as a confirmation of that?



```
a = 0
b = 0
repeat 5 times
    a = a + 1
    b = b + a
output b
```

```
a = 0
b = 0
repeat n times
    a = a + ~~b~~ 1
    b = b + a
```

Part A

	1	2	3	4	5
a	a1	a2	a3	a4	a5
	b1	b3	b6	b10	b15

$b = 5 + \textcircled{10}$

$b = 5 + 4 + 3 + 2 + 1$

Part B

Figure 8.3 Program trace

This yields section A of Figure 8.3, yielding an answer of 15 . The initial question was then re-visited:

My original question was, what does this program do? and you gave me an answer. Could you say it in a different way now?

OK.. err.. how?

Could as it were, well lets put it this way, it does some calculations, it does some calculations for a value for 'b', and then it displays 'b', it does these calculations which you've worked out. Would it be possible, could we replace this by a single, long calculation? Could we work it out in one go, such that we could say 'b = 'something, and work it out in one go?

(26 second pause) Well I could um (9 second pause) I can't think

Um suppose we did it backwards, what's the last step it takes before it finishes?

b = b + a

Yeah

Which

Yes but in a sense it always does that, it always does b=b+a, but in the loop, the numbers change, which

Yeah

So maybe you could say that the key to it is what way do the numbers change?

Well its 10 plus 5, so its

Can I say, its 5 plus 10,

Uhu, yeah

In fact the way its written down here its b=b+a, so it is 10 plus 5, but that's the same as 5 plus 10, so I could say to you, how did it get the 10, where's the 10 come from? (in section B of Figure 8.3, the b = 5 + 10 is drawn, and the 10 circled by the author)

Well um it's the um it's the 1 plus the that 6 plus the 4 which is

OK could you write that down?

OK

OK is that correct does it work out to be 15?

Yep

> OK so you've written down b = 5 plus 4 plus 3 plus 2 plus 1 (in figure 8.3 part B) so that is a way of saying what this program does. Um could you say it in English what the program does?
>
> *It um it outputs a sequence of numbers well it doesn't output a sequence of numbers but it works out the sum of all the values up to 5, and including 5.*

Interview summary

This very able student can trace the action of the code directly, but she cannot initially see it as a whole - but she can do so after a little leading.

8.4.2 Interview Two

The keyword 'private' was not seen by her as motivated by security, but maintaining object field values in valid states - the normative view. She said that her understanding of 'interface' was weak, and was reassured that several students had this problem and that a better understanding would come after seeing more examples.

Her understanding of the term, 'abstract' was normative, as was her understanding of static, and she quotes the Math class as an example of static, but something interesting arises. A class hierarchy, with an abstract base class and 3 non-abstract desendant classes is being discussed:

> Suppose we programmed it, and we missed out the word abstract, would it still work, or what?
>
> *Umm, possibly umm .. the um would the methods in it have to be defined if it wasn't abstract, I'm not sure.*
>
> Would they have to be defined?
>
> *Well in the*
>
> Ah well OK would they have to be defined? Er what do you think?
>
> *Um if err if the sub-classes can still be an extension of the base class which isn't abstract, I'm not sure if that is possible*
>
> Ah err right say that again

> *I'm not sure - is it possible that a super-class could not be abstract but still have sub-classes that extend it?*
>
> Oh OK the answer to that question is yes
>
> *OK*
>
> They can be

The interview then developed as a recap of the idea of inheritance, confirmation that base classes did not have to be abstract, and a discussion of what was gained by the use of the 'abstract' keyword.

Consideration of 'the Game' (see 8.2.2 and Appendix One) showed that for her, identifying classes and objects was difficult, and that the unassisted approach was procedural, but that given a suggestion of objects involved (human player and robot) she was able to outline appropriate classes, including a hierarchy with an abstract base class.

Interview Summary

At this point the student:

- had some awareness of the ideas of inheritance and 'abstract'
- knew they were related
- was unaware that inheritance does not depend on the base class being abstract - she thought base classes must be abstract
- was aware that she did not know
- could reason along the lines of ' if inheritance does not depend on abstract then .. but otherwise …'

Her belief that base classes had to be abstract is an over-generalization - she had seen an example where a base class was abstract, and had concluded from that that base classes had to be abstract.

She is able to apply logical reasoning to her understanding, in that she can say 'if .. then I think .. otherwise I think ..'

8.4.3 Interview Three

Jennifer had been ill before Christmas, and this interview was during the same week as Matthew's fourth interview, where the new topic area was starting GUI programming with Swing.

Previously her grasp of the interface idea had been weak, and so the line of questions pursued whether this was now clearer in the context of concrete use

> Can you remember anything about ActionListener?
> *Well it um it um like takes note of what the user, how the user interacts with the program, and it does certain things based on what how they interact with it, like through um mouse clicks and things*
> OK so if you've got a button
> *Yeah*
> on there - this thing called ActionListener, do you know what kind of thing it is?
> *Is it a um can't think is it a (11 second pause) a err I can't think of the word, like there's a something there's a sub-class of it, I've just forgotten the word*
> Is it interface?
> *Um..*
> Or no?
> (laughs)
> So interface is not the word you were thinking of?
> *No*
> Ah that's interesting. So what is the word you are thinking of?
> *I don't know.*
> Not class,
> *Its like something which some things can be a sub-class of, can't think of the word.*
> Well that would be a base class or a super-class or an ancestor class?
> *Because I presume that MouseListener and WindowListener*

> Yeah?
>
> *are sub-classes of that - I wasn't sure if it was a class or interface*
>
> OK

What is happening is that the hierarchy surrounding these interfaces (which has EventListener as a base, with sub-interfaces ActionListener, MouseListener, WindowListener and others) has been mistaken for the typical hierarchy of base and sub-classes, leading her to think of these as classes rather than interfaces.

The interface idea is pursued directly:

> OK so if I ask you a question - what is an interface? what would you say?
>
> *Its um.. it's a declaration of all the methods that the classes that implement it should um should write out*

which is essentially a text-book answer. So although there is some confusion with specific cases, the general notion is well-grasped - the opposite of Matthew's situation. But she does describe the idea of ActionListener in human terms - *it um like takes note of what the user, how the user interacts with the program,*

Also she shares Matthew's appreciation of the concrete visual nature of GUI programming:

> How do you feel about graphics programming and Swing is it er
>
> *I think its slightly more interesting than what we were doing last year because you can see something that whereas with the other ones you could tell that it worked because it produced the output that you were expecting but you can't really um I think its more interesting to actually like see something visually than just..*
>
> OK

Her concept of an object involves the term *instance*:

> What do you think an object is? What would you say?
>
> *It's an instance of a particular class. You create.. you can create a new object which has methods defined in the class of the object*
>
> OK so it's an instance of a class?

Yeah.

Yeah? So what's a class then?

It's a series of methods that can be called when a new instance of the class is created.

OK so you have methods, you might also have fields, what are they all about?

They are variables that the object would need to use and work with.

This notion of a class bears some resemblance to that of Matthew's, as a library of functions which an object can call - rather than the object *having* those methods.

8.4.4 Interview Four

Since the last interview, work with a JTable filled with data from a database had been completed, and the current topic was to construct a syntax tree from an input string in a programming language.

The task of exchanging two int variable values was introduced, and immediately solved, on the basis of the standard approach which had been seen in class. This was then developed to exchanging objects, as reference types. In other words with

String s1 = new String("One");

String s2 = new String("Two");

we want to exchange s1 and s2 values.

Jennifer's initial response was that since primitives and reference types are stored in a different way, this would not work in the same way.

Well could you create another reference to "One", the s1 value, and then change the s1 reference to "Two" and change the s2 reference to where the new reference was pointing to

OK do you want to code it then?

She writes

String s3 = s1;

s1=s2;

s2=s3;

(8 seconds later) That wouldn't work I just realized.. because I'm going to say s2 = s3, but if s3 is pointing to the same place as s1 and then s1 changes then wouldn't s3 change as well?

It is agreed that we could work out what happens by using a diagram (see below):

s1 points there, and s3 will point to there..

Yeah

and then..

Can I just ask, does s3 point at s1, or does s3 point here (at "One")

Well it points to there, but it points to wherever s1 is pointing,

which is there ("One")

but then if s1 changes it follows s1, I would assume,

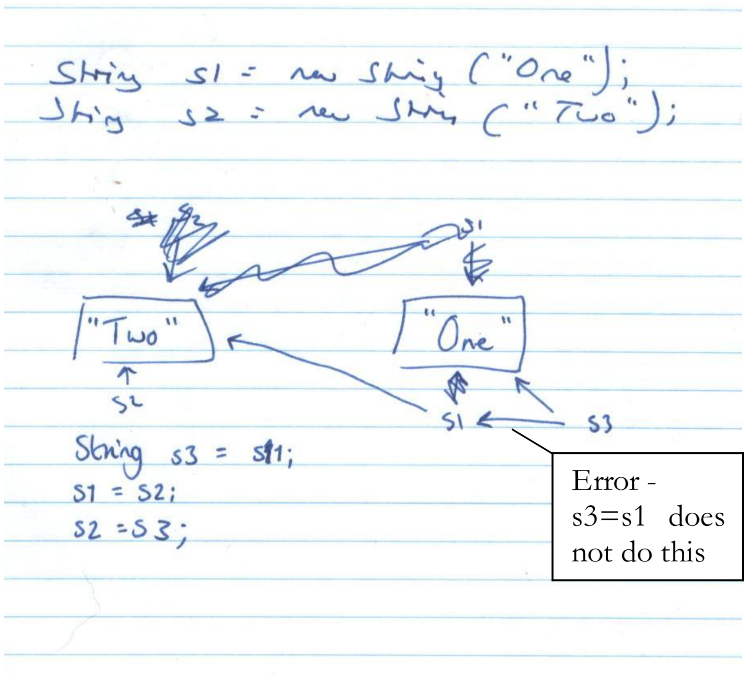

String s1 = new String ("One");
String s2 = new String ("Two");
"Two"
"One"
s2
s1
s3
String s3 = s1;
s1 = s2;
s2 = s3;
Error - s3=s1 does not do this

Figure 8.4 Reference error

Ah right I see

Jennifer's reasoning is that s3=s1 causes s3 to point to s1, and so if s1 then changes (in s1=s2 so s1 points to "Two") then s3 will also change, following the pointers and so also pointing at "Two". Hence the final s2=s3 would do nothing, since s2 already points at "Two". It is striking that this mis-conception is also reflected in the object diagram.

This mistake is discussed, and then Jennifer agrees this would in effect work the same as for primitive type exchange.

In fact she makes the same mistake as Matthew in 8.5.6, taking s3=s1 as meaning that s3 points to *s1*, rather than *at the same place that s1 points to*.

The task of reversing the data in an array is then presented. As for Matthew, she first uses a second array to copy the array in reverse, and then transfer them back, but quickly sees that this can be done with the single array and the value exchange in the start of the interview.

Interview summary

1. The exchange is easily recalled from the topic seen in class

2. The exchange of reference types shows a lack of facility in the use of references, and in particular that for example s1=s2 means that s1 refers to s2, rather than what s2 refers to.

3. The array reverse is solved first by the use of a second array, and then by the use of the exchange algorithm. The fact that this involves a loop is taken as obvious.

8.4.5 Interview Five

This was concerned with Listing 2 (Appendix 2), with the purpose of investigating ideas of static and non-static fields. The use of Vector was discussed, since this would not have been encountered in class. Jennifer then read through the code, and could immediately give a good description of what this was about.

> In the fields of the class Organism, the first two, vaccinated and infected, are different from the rest of them, to the extent that the rest of them are static, and vaccinated and infected are non-static, so what difference does that make?
>
> *Well the boolean fields infected and vaccinated could be changed later on, you could, using the methods for the Organism class, they could change to true, whereas the static, doubles, they don't change*
>
> OK so they don't change, because they're static?
>
> *Mm.*
>
> Now some of these, four of them in fact, vaccination rate, infection rate, death rate and reproduction rate, they're declared to be final, as well as being static. What does final mean?
>
> *Um that its never going to be changed? Like err even though err the static indicates that.. I'm not quite sure of the difference between static and final, because I'm assuming that static means that it doesn't change, and final means it doesn't change either, so I'm not entirely sure of the difference between the two.*

There follows an explanation that final means it cannot change. Then we have:

> These non-static values, they apply on a per-object basis, so if you have one object
>
> *Oh!*
>
> it has that, and if you have another one, it also has vaccinated and infected, but the data values of one object are independent of the other one. OK, so non-static means it is on a per-object basis, so what does static mean?
>
> *Its universal for everything, every single incidence of Organism, so every single Organism which is constructed will have the same vaccination rate, its like dependent over all*
>
> OK have you sort of like, when I said that for those two, the non-static, they are per object, did that give you a clue to recall? Now if I say the non-static values are per object, what are static values?
>
> *Per class?*

So initially static is understood as 'cannot be changed', as others have. Here Jennifer shows she actually has this idea, which is part of Java, and also associates it with the correct keyword, 'final'. When she sees there are two keywords with the same meaning, she sees there is something wrong. However, the extract above shows she grasps the correct meaning of static extremely quickly, suggesting that she already had this idea (in fact she showed it in the first interview), but that it had been obscured by the meaning of static as fixed. This shows that 'understanding a concept' is not a simple yes or no thing, and that in this case there were two understandings, one of which had come to over-shadow the other. It also shows a logic-based constructivism - she sees that it is unlikely that static and final mean the same thing, so she is obliged to construct an alternative meaning for static.

Interview Summary

1. Static was reported as meaning 'not changing'

2. The real, per class, meaning was latent, and quickly revealed with very little prompting.

3. The construction (or probably re-construction) of the 'static' concept is shown,

8.4.6 Interview Six

At this point Jennifer had completed the module and was in a revision period prior to a written examination paper. The sequence of class definitions given in Appendix 3 was looked at. Her understanding was largely normative - only the few non-normative conceptions are reported here.

In version two, this code:

```
Employee person1=new Employee("Joe",3);
Employee person2=new Employee("Jane",5);
Employee another = person1.getMoreSenior(person2);
```

How many objects have we got?

Three. Because its still err the err the person that you return from the getMoreSenior method is um another um.. You've only um created two instances but *you've assigned to another employee the err the result of the getMoreSenior um there are still two instances of it but there is another employee um err variable.*

OK you say variable err yeah

I know not variable another.. like its not its not another instance of the Employee class it's the um its basically creating another um.. I its because um that the third line it doesn't say it's a new employee its not another instance of the class its just its like a copy of the err err the person that's returned by the getMoreSenior method.

It might be suggested that Jennifer is just having problems recalling the word *reference*. But in fact what she says implies she does not have a clear distinction between objects and references to objects. She first says we have three objects, then changes that to two instances, and another variable. She also suggests that we have a copy of the person - when the normative view is that a copy of an object is another object, and that is quite distinct from a second reference to the same object. An interpretation is that the OOP frame is not that firmly structured for her at this point, and she is having problems viewing this code fragment in that frame.

In version five, introducing sub-classes, we have the following:

Looking at the ManagerialStaff, what - for a ManagerialStaff object, what fields have they got?

Um just well its has the field getsPaid, and it also has the fields from Employee because it extends the Employee class -it's a sub-clas of Employee, it has all the fields from that.

OK so the fields are inherited, what about the methods?

The methods, the constructor is inherited, and well all the other methods are inherited.

So she incorrectly says that constructors are inherited. She has over-generalized - since methods are inherited, she has assumed that constructors are also.

Each of these instances, one each of the two sub-classes, they have a pay method invoked upon them. Would it be sensible to modify the design of the classes in some kind of way?

> *Um how do you mean? Like having a um like an undefined method in Employee and then a defined method in the sub-classes?*
>
> Yeah? And if its undefined? What's the keyword for that?
>
> *I can't remember.*

This contrasts with not recalling the word *reference*. There her grasp of the idea was also poor, while here she clearly has the idea of *abstract*, but simply cannot remember the word. She immediately recognises it:

> OK well let's see what happens let's take it a bit further (shows version six). OK so we have um this so we have public abstract class Employee, and we have this abstract
>
> *abstract that was the word I was thinking of.*

Interview summary

1. Jennifer's understanding of many OOP ideas is now normative

2. The distinction between object and reference is still fragile

3. She has over-generalized inheritance to include constructors.

4. She had the idea of *abstract*, but had forgotten the word

8.4.7 Jennifer Summary

1. She starts with significant proficiency in handling procedural code fragments (interview one)

2. She has mostly normative ideas about *private*, *abstract* and *static* (interview two)

3. She had over-generalized that base classes had to be *abstract* - but she could easily change this idea (interview two)

4. She develops normative concepts of *class* and *object* (interview three)

5. Her awareness that *Listener forms a hierarchy leads her to see these as classes rather than interfaces. (interview three)

6. Her ability to use references is limited - s1=s2 means that s1 refers to s2, rather than what s2 refers to (interview four)

7. She does the array reverse using a second array (four)

8. She reverts to *static* meaning constant (five)

9. The clash with *final* means she easily recovers the correct meaning of *static*. (five)

10. She has problems with the reference concept to the end (six)

11. She over-generaizes to thinking constructors are inherited (six)

12. She forgets the word abstract but maintains the concept (six)

13. Over-all, she can generalize from examples and instances to abstract and general concepts were easily, which she uses to construct effective mental models of the domain. However this sometimes leads to an over-generalization.

8.5 Matthew

8.5.1 Interview One

Matthew is an 18 year old undergraduate on a Computer Science degree programme. He took 3 A levels, Maths Physics and Chemistry, obtaining grades of A C and C respectively. He has had no experience of programming before the start of the course.

The interview began with the same code code fragment as for Jennifer, summing 1 to 5. The question 'what would be the final value of b?' produced the trace as shown in part A of Figure 8.5. After some discussion this followed:

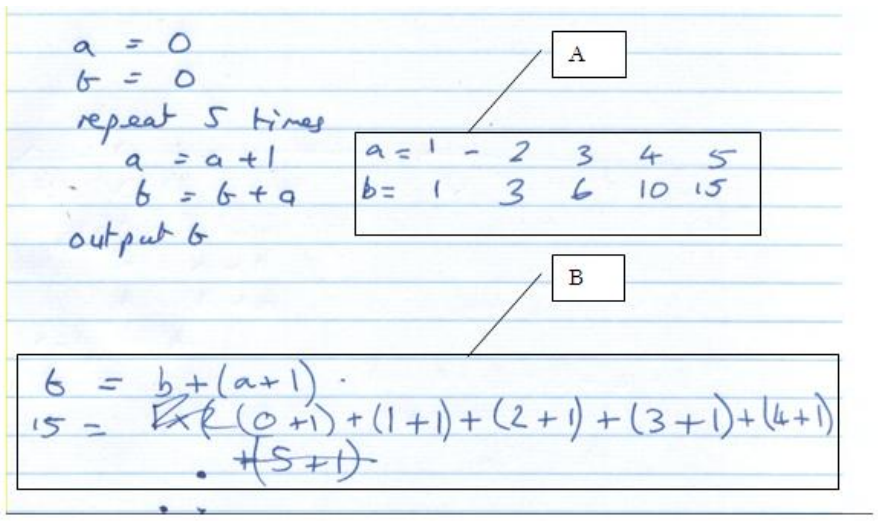


Figure 8.5 Program trace

Why 15? How did it arrive at 15? In summary, what does it do?

Counts up with, um, like um, its an exponential which will kind of how do you call it, exponential, its incrementing more each time

Its incrementing more each time?

Yeah, the value of 'b' is more each time.

yeah um if you can, I mean it does these steps 5 times, so it does the calculation there 5 times, would it be possible to do it in one big step as it were

Um, how do you mean?

If we replaced all of that with a single 'b =' some calculation, maybe a long calculation, would it be possible to write down an expression in mathematical terms, could you write down an expression which would work out to be this?

Um yes I should think so. Um (6 seconds) yeah.

Could you write it down what it would be?

(20 seconds)

You could have several goes at it, you could write down stuff, and maybe its wrong, and we could cross it out and do it another time.

(27 seconds)

OK so you've got b = b + a + 1 (section B of Figure 8.5) could you sort of like - I can guess how you got to that point, but could you tell me why you wrote that down, what you were thinking?

Well I was thinking here you've got b=b+a, and here a is equal to that

OK right

so that's just merging those 2 into 1

Yeah

It does that 5 times so err

Could you do it with the numbers rather than the letters, the numeric values?

So you could say 15 is equal to..

Yeah, because its b, but you know its 15 so where does the 15 come from in terms of numbers?

It comes from the addition of .. these..

So could you write that down?

So yeah um (43 seconds)

You might want be tempted to do some algebraic manipulation of this, but we could do that later, but to start with how does the computer what does the computer do - the computer's not doing any multiplying.. its just adding

I dunno nought plus one

Yeah, zero plus one yeah

Plus um its going to be one plus one

Yeah

And then I'm just going to be creating a massive string of

OK

OK so um 2 plus 1, 1 2 3, 3 plus 1, (inaudible - writes down b = (0+1).. in B in Figure 8.6)

OK so this is 1 plus 2 plus 3 plus 4 plus 5, what does that come to?

That's 15, so just get rid of this one (deletes (5+1)) then you have it.

OK

Yeah that makes sense

Interview summary

So as for Jennifer, Matthew can trace execution with ease, but when trying to encapsulate it into a single process, he is misled by an algebraic approach, and needs considerable support before he can see how the values are changing at a higher level.

8.5.2 Interview Two

Matthew confirmed the topic list covered at this point (see 8.3.2). He showed the same mis-conception that private was motivated by security concerns, and confirmed that he does not 'understand' interfaces.

The 'Game' task (8.3.2 and Appendix One) was discussed, but he found OOP data modelling very difficult.

Like Fiona he was unaware of the purpose of the keyword abstract.

Interview Summary

Even though at this stage many OOP concepts have been introduced, he finds the use of classes and objects to model a situation very difficult.

8.5.3 Interview Three

This is a key interview, in which Matthew shows that he has good recall of material encountered in class, can invent his own solutions, yet has little understanding of classes and objects.

We start by establishing that the interviews are not affecting his approach to study, which is primarily motivated by course assessment.

He has not encountered recursion before. He cannot 'define' recursion, and finds an example not easy.

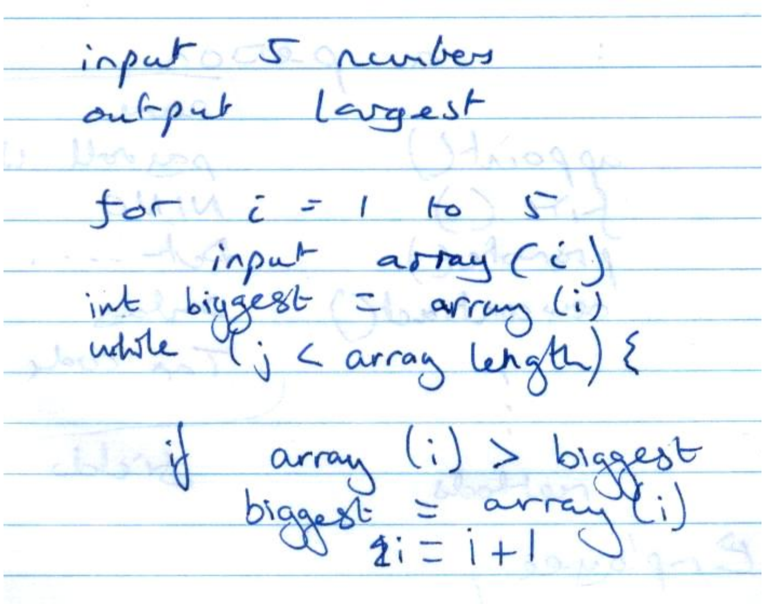
He is then given the task:

input 5 numbers

output the largest

His solution is to input the data into an array, then search the array for a maximum:

> Did you just invent that? Or have you seen anything like that before?



```
input 5 numbers
output largest

for i = 1 to 5
        input array(i)
int biggest = array(i)
while (j < array length) {

        if      array(i) > biggest
                biggest = array(i)
                i = i+1
```

Figure 8.6 Array method

> *I've seen err yes something similar to this before, in one of my homeworks, exercises of going through an array and finding the biggest.*

Discussion led Matthew to the idea that it might be possible to do the task without using an array. This prompted him to produce an original invention, something like a mergesort. Hence he can recall ideas, and invent his own.

But then we move on to classes and objects:

> What is an object? In a Java program?
>
> *A new method?*
>
> Well you could say - what is an object made of?
>
> (8 seconds pause)
>
> Or does an object have parts?
>
> (18 seconds pause)

No

No? I mean what would you say what is an object in a Java program, its object-oriented programming, what's an object?

Objects are (11 seconds) objects would they be lists of, well not lists, various different commands, or getting something

Well, yeah,

The objects themselves are referred to to do the work,

Yeah, right, then um so how do you how do you get them to do the work?

By (8 seconds) by including in the object what you want them to do

Right is there a kind of technical word for what you want them to do

I don't really know.

OK do objects store any um values, pieces of data, pieces of information?

Er, no.

OK. Can you give me an example of what would be an object, in any old Java program?

(27 seconds) Would it be something like, getName, something like that?

In fact getName is a typical name for a method, not an object.

The rest of this interview was spent as a tutorial about classes and objects, using examples of an Employee class in a personnel system, and a Shape class in a graphics program. The student seemed to follow these ideas with understanding.

Interview Summary

1. He has shown in this interview that he can recall with understanding ideas presented (linear array search for a maximum), and can invent his own procedures to solve a problem.

2. This student had almost no normative understanding of the concepts of class, object, field, method and so on. He can suggest a likely method as getName(), but this must be simple recall and cannot be meaningful to him, since he has denied that objects carry any data, so the ideas of encapsulation, fields with access modifier set to be private and accessor methods like getX() cannot make any sense. He has also encountered in the

course at this stage a large set of ideas about OOP, (see 8.4.2), including elaborated ideas such as 'abstract' and 'static'. Yet he has almost no idea what an object is.

8.5.4 Interview Four

This starts by establishing new topics encountered:

> *New - we've done frames, like creating windows on screen, and as well as that we've done paint methods, so using graphics to draw lines and things on screen,*
>
> Yeah
>
> *Um, I'm trying to think, using, what was it, window, window action listener, something like that, so you're assigning like when you hit the cross on the window it closes, things like that. Yeah, apart from that we've done buttons, briefly*
>
> So what's your reaction to that, is it kind of OK?
>
> *Yeah I mean its err I quite like it because its getting to be a bit more you know you're coding something that the sort of thing that you'd actually see on computers like everyday kind of thing like buttons, things like that, so its quite nice to see we're moving on to something a bit more advanced*
>
> Right OK there's two things there, one is that it corresponds to actual applications
>
> *Yeah*
>
> That are useful, and the other is that its visual, you can actually see it - which is the most interesting? or maybe they are both interesting?
>
> *Yeah dunno probably they are both interesting yeah but definitely the fact that you can see it, yeah it helps to create interest in it.*

The student is saying that the fact that UI objects can be seen makes it more interesting. There is a possibility that the interest derives from the fact that it is easier to understand if the objects are in a sense visible. An alternative is that the objects are recognised as being used in 'professional' applications.

The mechanism of Java event-handling is seen by him as entirely in terms of embodied cognition, with the application 'listening' for an event - and there is no recognition of ActionListener as an interface:

> One aspect of that is ActionListener um did that make sense?
>
> *Yes, sort of, cause it seems to be like in a crude way I guess the way I see it is if the method, its telling the program that you're running to listen for a particular event so for the window closing its listening yeah its kind of almost I dunno if its right but its like its always running and waiting to see if the action happens its kind of hard to explain its kind of listening.*
>
> Yeah yeah yeah so the idea of calling it an action listener idea makes sense.
>
> *Yeah*
>
> And you find it quite straight - you know, is it straight-forward to program?
>
> *What I've done so far, yeah.*
>
> OK now this relates to the idea of an interface
>
> *OK*
>
> Do you remember that?
>
> *I remember it from last term - I can't quite remember what it is.*

So Matthew is happy with the embodied notion of the program listening, but does not see this as a example of the idea of an interface, introduced last term in the rather abstract context of timing.

The interview then moved on to ideas of class and object:

> If you had to tell somebody what is an object, in Java, what would you say?
>
> *(29 second pause) I'm not entirely sure*

So just as at the previous interview, Matthew cannot formulate a description of what an object is.

> OK could you give me an example of an object, which, you know, you've used in programming?
>
> *Is it, like, when you create an object do you, say I have like, this is like what we did for the last exercise, say you had a class called Line and to create which had various methods and things of that*
>
> Yeah

> *to display all the information for a line, and is it you create the object by saying public Line, this would be in a different class but public Line and then just the name of the line, or the name of what you wanted to refer to*
> Yeah, do you want to write that down?
> (he writes down
> public Line line())

So even in the case of an example, he does not have a strong grasp of the object concept, and certainly not the syntax. A short tutorial about

```
public Line line1 = new Line();
```

then followed. Line() was not recognised as a constructor invocation.

However he can use the terms field and method appropriately:

> This Line class - what methods would you have?
> *Ones for getting the start point, like co-ordinates*
> Ah OK yeah
> *So you'd create some fields x and y*
> OK so these would be data fields
> *Yeah*
> They'd be ints
> *Yeah*
> So you might have an x1 y1 as start points, x2 y2 as the endpoint OK so you'd have that there, and you say you'd have a method to get those? Yeah err OK so those data fields would they be public or private?
> *Private.*
> Why would you make them private?
> *So you can't accidentally change them.*

Matthew has very recently done an exercise on this - but his answers are immediate, direct and correct. This is very significant:

- He can't say what objects are.

- He does not recognise constructor use
- He correctly talks about fields and methods
- He grasps the notion of encapsulation as implemented with access control and accessor methods

This might be seen as 'illogical'. How can he talk about fields and methods, when he cannot say what an object is? But he does.

Even after this discussion of an example, he still cannot say what an object in general is, or what a class is:

> What are Java objects in general?
> (21 seconds pause) *Just a way of calling classes and methods, or..*
> OK OK What would you say a class is?
> (12 seconds pause) *Probably just a list of methods including a constructor*
> OK so in a class you'd have a constructor and you'd have methods and data fields
> *Yeah*
> Yeah OK so when you actually program Java you define classes
> *Yeah*
> What's the connection between classes and objects?
> (5 seconds) *The object will contain.. the.. everything the class has in it?*
> The object will contain everything the class has in it. Like, methods do you mean?
> *Yeah. Well.. um.. is it just using all the methods and things in the class?*
> So an object uses, or can use, the methods in the class?
> *Yeah*

So Matthew suggests a class is a set of methods which an object can use, like a library, but also that an object is a set of methods. There is a possibility that he would like a tutorial on 'what is an object' at this point, but this is resisted, and the idea of 'type' pursued.

OK so I mean it can be said in different ways - there are different versions of it. Yeah OK. One way - if I say the word 'type' to you, what do you think of in terms of type?

Type..

OK not very much

No (laughs)

So if I say int and char and double and float, what are they?

They're types of fields.

This is very striking, showing the same pattern. He can say nothing about 'type' as such, but he immediately and confidently says that int and so on are types of fields.

Interview Summary

1. He can talk about some ideas (field, method, encapsulation, constructor) associated with the object/class concept, in the context of one example, sometimes confidently but not uniformly.

2. He cannot talk about class and object in general

3. He can talk about examples of types

4. He does not respond to 'type' itself.

5. We can characterise this then as at this point, he can handle some examples of the ideas of class/object, and of type. But these instances have not yet been compressed into structured frames for class/object or type.

8.5.5 Interview Five

This started by establishing that the module had continued with more work on Swing-related topics.

The interview sought to investigate Matthew's grasp of this, and so a simple GUI program was examined. This was looked at in the NetBeans IDE - the program consisted of just one class, and was a simple calculator, which when running would look like:

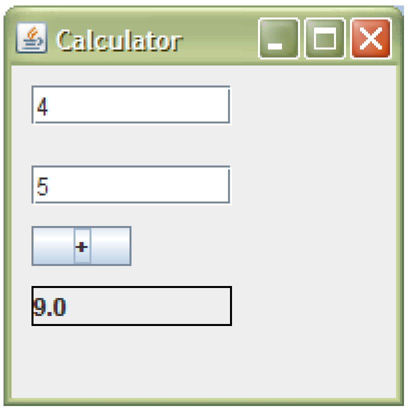


Figure 8.7 The calculator

The code is listed below. It was reviewed briefly, and then some points were examined:

> This is in the class called Main - that's public Main() - so what is that then?
>
> *Is that your constructor?*

So now he can correctly identify a constructor in code.

> Those five instructions (starting with setBounds in the constructor).. they don't refer to - they're method names, but there's no object name there - there's no object.setbounds - so what object is it talking about?
>
> *Is it talking about the object main, which is or yeah which I guess is your JFrame?*

So he can see that a method invocation in a constructor refers to the object being constructed.

> What do you make of that bit there? (Pointing at public static void main())
>
> *Um I guess its yeah you're testing if you like to see whether um to see what to see if everything is working um so you're just running your one and only class. Without that your program would not run. Because you need the main method.*
>
> OK so the special thing about public static void main - what's special about that?
>
> *Um oh its that's what runs the code*

OK

It just - I dunno it just - I don't think its ever really been explained like you know - you just what it will do is it'll just run through your code.. and produce the results from it, I guess

This is interesting because it shows that Matthew

1. Thinks that a class can be 'run'

2. Does not have a clear grasp that main is simply where execution *starts*.

Java event-handling is rather mis-understood:

Quite a significant part of this is the button.addActionListener(this) - what's that all about?

Um.. you're assigning an actionListener to the button, so the button is well the button is effectively listening to see if its being clicked or not

The button is listening?

Oh its its it's the setting er yes I guess its not the actual button its um I dunno its setting up a listener its listening to see if the button's being pressed or not.

OK that's quite interesting - *what* is listening?

Um

Because its not the button listening to itself

Is it - is the actionListener listening to it? Or.. is it .. or the constructor?

OK in the - here, we call the method addActionListener, button.addActionListener, and there's an argument *this*, addActionListener *this*. So what is *this* referring to?

Um.. would it be the program, or

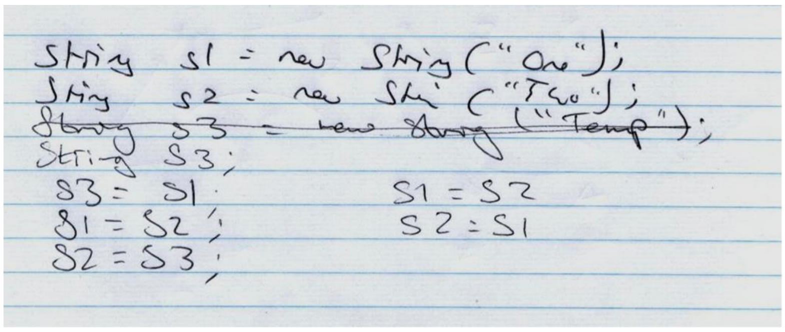
At this point an explanation of the event-handling mechanism was attempted by the interviewer. A reason for the confusion here is discussed in the next chapter.

```
package interview;

import java.awt.Color;
import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import javax.swing.BorderFactory;
import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JTextField;

public class Main extends JFrame implements ActionListener
{

    JTextField num1;
    JTextField num2;
    JButton button;
    JLabel label = new JLabel();

    public Main() {

        // set up frame..
        setBounds(200, 200, 200, 200);
        setVisible(true);
        setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        setTitle("Calculator");
        setLayout(null);
        // set up input textfields
        num1 = new JTextField();
        num2 = new JTextField();
        add(num1);
        add(num2);
        num1.setBounds(10, 10, 100, 20);
        num2.setBounds(10, 50, 100, 20);
        // set up label for output
        label = new JLabel();
        add(label);
        label.setBounds(10, 110, 100, 20);
label.setBorder(BorderFactory.createLineBorder(Color.black)
);
        // set up Add button
        button = new JButton("+");
        add(button);
        button.setBounds(10, 80, 50, 20);
        button.addActionListener(this);

    }

    public static void main(String[] args) {
        Main main = new Main();
    }

    public void actionPerformed(ActionEvent e) {
        if (e.getSource() == button) {
            // do the add process
            double n1 = Double.parseDouble(num1.getText());
            double n2 = Double.parseDouble(num2.getText());
            double result = n1 + n2;
            label.setText("" + result);
        }
    }
}
```

Interview summary

In a GUI programming context Matthew has a reasonably strong grasp of the practicalities of Swing programming.

8.5.6 Interview Six



```
String s1 = new String("One");
String s2 = new Stri ("Two");
~~String s3 = new String("Temp");~~
String S3;
 S3 = S1.               S1 = S2
 S1 = S2;               S2 = S1
 S2 = S3;
```

Figure 8.8 String value exchange

This started with establishing that work had continued on using a JTable with data sourced from a database using code supplied. Then the task of exchanging two ints was introduced. Matthew immediately solved this with the standard third location approach - he recalled seeing this in class. The extension of this to using objects rather than primitive

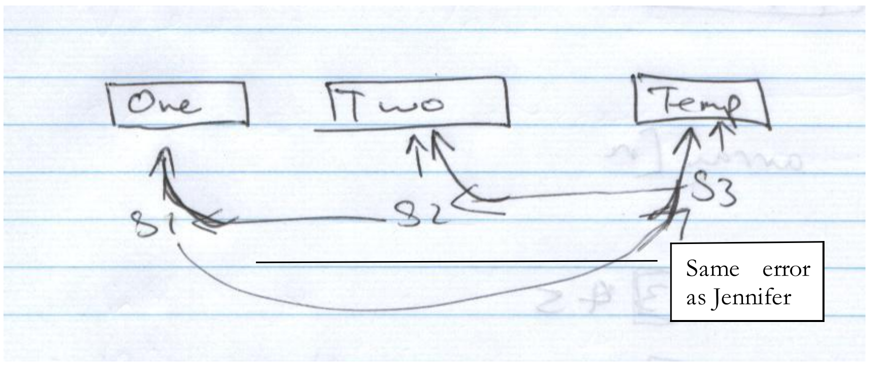


Figure 8.9 Reference error

types was presented, and he produced a corresponding solution (see Figure 8.8).

He was unsure whether this would work, and was prompted to draw a diagram showing how the references changed, following a reminder that references were pointers to objects. The result (Figure 8.9) shows two mistakes. Firstly he draws the arrows in the (normatively) wrong directions - for example, the first

s3=s1

yields an arrow *from* s1 *to* s3.

Secondly, the arrow goes *to the pointer*, not *what the pointer points to*. In other words the normative version is

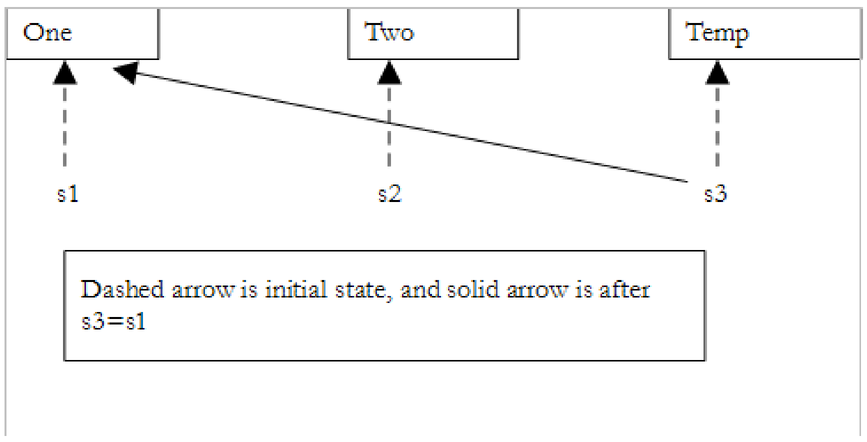

One
Two
Temp
s1
s2
s3
Dashed arrow is initial state, and solid arrow is after s3=s1

Figure 8.10 Reference correct

These mistakes lead Matthew to think his code will not work. The arrow diagrams are reviewed, and he is led to see that in fact his code would work. He then says

> *But I feel almost like we've got an unnecessary string here hanging around.*
> Ah OK so it's a matter of, what happened to this then (string 'Temp')?
> *It just I guess it will disappear by caught up by the garbage collector?*
> OK it will be caught up by the garbage collector. Why would it?
> *Because you've no longer got anything pointing to it anymore so there's no way that you can access it anymore.*
> OK that's right so could you improve that anyway?

This suggests we do not need s3, so he writes s1 = s2 and s2=s1 (Figure 8.8) but it is quickly seen that this will not work, and after a brief discussion he crosses out the s3=new String("Temp") and simply declares s3.

The task of reversing an array is then presented. This is first solved by creating a second array, then moving the element at the end to the start of the second, one from next to the end to the second place in the second array, and so on in a loop, then copying them back into the first array in another loop.

He is then asked if he could do it without using a second array, which leads to a realization that this uses the exchange process:

> *If you were, would you be using, if these were integers you would use something similar to this, so you would take this value, put it in a value outside the array, swap that to be the first one, that over there, then increment that by one, increment that by one, in terms of their position, then you get the array switched around.*

He could see this would stop in the middle of the array.

Interview summary

1. The standard exchange algorithm was recalled from class

2. References as pointers were not handled proficiently and misled him into thinking this would not work with objects

3. The array reversal task was initially solved using a second array

4. It was obvious this involved a loop

5. It was easily solved using the exchange without using a second array

8.5.7 Interview Seven

This started by establishing that the current class topic continued to be writing a Java project to construct syntax trees. We then proceeded to look at Listing 2 (See Appendix Two). The purpose of this was to investigate the way Matthew understood individual object states, in contrast to class fields. The use of a Vector was discussed, since this was not something taught in class. This was described as like an array except elements could be added or removed at run-time - Matthew had no problem with this. The definition of two classes in the same file was also discussed.

> These two fields, boolean vaccinated and infected, they're different from some of the other ones, because some of the other ones are static - those four, vaccination rate, infection rate, reproduction rate and death rate, they're static, and these two are not. What's the idea of that?
>
> *Um.. static are values that can't be changed, and they can be either true or false*

OK boolean is true or false, yeah, so boolean data type is true or false. These like vaccination rate is 0.2, so these are double, double precision floating point. OK so.. the keyword static um is a bit deceptive - it sounds like fixed and unchanging. ..

So Matthew's conception of static is non-normative - like other students including at times Jennifer, he thinks static means 'cannot change'. Static is explained to mean per-class rather than per-object. He responds:

So you can't have more than one vaccination rate for the class Organism

which is good, but in fact his grasp of non-static fields is wrong:

So vaccinated and infected they're not static, so, do we, so what does that mean?

That you can have them, you can have them outside the class, so it would be for, using it in the main() method? Or..

Ah OK yeah I think you're thinking of public and private, which is not quite the same as that.

At first sight it is difficult to see why he thinks non-static means, in effect, public visibility. But a possible explanation can be obtained. Static is taken as being a property of the class, and is therefore 'in' the class, and so it is reasonable to suppose that non-static signifies 'out' of the class. And there is an idea pertaining to this, that of access control, determining whether parts of a class can be accessed from other classes. So he connects the out-of-class access idea with the 'static is in the class' idea. Unfortunately he has been misled by the way he is conceptualising static.

An attempt is made to explain this by visualising some Organism objects as cows.

So there's one Organism - does it have any values, does it have any data associated with one of those?

err, not initially I don't think, no.

This again suggests that he does not understand non-static per-object field values, but:

So this is the constructor,

Yeah

So it says if Math.random is less than the vaccinated rate, vaccinated is true.

OK so would you have in that case, whether its vaccinated or infected? So those would be the two values that it would have.

So the idea of per-object field values is picked up. It appears that the concept is unstable, being absent then re-appearing with very little prompting.

It is established that field values are distinct for distinct objects, and that declaring population as being static is reasonable, since there is only one population. The way that the Organism methods are coded is then seen as being understandable, as is the syntax Organism.population. One problem is the use of the keyword 'this' in the die() method.

We have the keyword 'this' - what does that mean?

Um, well its talking about the object, the Organism object, is that.. its just the way of.. yeah the way of.. (quietly) Population? telling it to remove that organism from the population.

This method called die() - what does it belong to? Does it belong to the class, or what?

Yeah, it belongs to the class Organism, yeah.

In fact the interviewer's question is ambiguous - what exactly does 'belong' mean? It does belong to the class, in the sense that it is a method of the class Organism. But it is a non-static member, so it applies to objects, not the class. This is discussed further in the section 8.5.8.

OK is it static or non-static?

Non-static

So what does that mean?

It doesn't return any..

Err

Oh sorry if its non-static it doesn't belong to a class?

So what does it belong to?

The object?

Not returning a value would be *void*. Again, his grasp of static/non-static is weak.

The interview ended with a discussion of *this*.

Interview Summary

1. Matthew started by thinking 'static' meant constant.

2. His grasp of the meaning of non-static fields was unstable - it was absent, then re-appeared with a little prompting.

3. His grasp of the meaning of 'this' was rather diffuse.

8.5.8 Interview Eight

At this stage, Matthew had just completed the 20 credit module relating to Java. He would have an examination on it after a revision period.

This interview was devoted to consideration of some class definitions in code (see Appendix 3). This consisted of 6 versions, developments of an initially simple Employee class, which sequentially exposed a range of OOP concepts.

Version 1 was treated with confidence. The keyword 'private' was examined:

> The fields of this class, name and grade, are declared to be private - what difference does that make?
> *It will mean they can't be accessed or modified outside this class, um so ..yeah*
> OK so is there any advantage in that?
> *Its so that you can't accidentally alter that person's name or grade..*

We established that 'you' meant the programmer. Then

> In object-oriented theory, as it were, there's a term called 'encapsulation' - have you come across that?
> *Yeah*
> So what's encapsulation mean to you?
> *(very quietly) Encapsulation.. um, er..is it pretty much like it sounds, just, getting that field and um, just making sure nothing else can.. no other class can touch it?*

For a field, or for?

Methods, I guess, they could be private.

So he recognises the term encapsulation, but is uncertain about it. "Is it pretty much like it sounds" is an attempt to construct a meaning by relating it to the everyday meaning of encapsulation, together with the fact that we've just been talking about 'private'. So his conceptualisation of encapsulation is weak, and is related to the specifics of the use of 'private' in relation to a field. He does not relate it to a general OOP notion distinct from part of its implementation in Java.

We go on to Version Two, which is the same except for the addition of the method getMoreSenior. This allows us to explore the idea of a reference. Matthew describes what this code does:

OK. So in this case again, you are creating 2 objects this time, person1 and person2, and.. Employee another.. now here you're creating another Employee ob .. are you?.. mm.. now here I think you're calling the method is it yeah the object of getMoreSenior? Which is taking person1 and calling getMoreSenior with person2.. so here then it calls this method.. this.grade, so .. in this case it will take person1, and see if its greater than someoneElse dot grade which will be person2 in this case, return this meaning person1 if its person1 is greater, or if its not then return someoneElse which would mean that someoneElse's grade is greater. So in that case it would be person2. Then another.display yeah just print out the person that is more senior.

Now when you started looking at that, you started to say that we were making a third object but then you stopped- have you changed your mind?

Um.. I'm trying to.. I did at first because there is no = new, like new Employee or something, but looking here it looks like it is creating an object because its public Employee getMoreSenior, with Employee referring to the class and that class is which doesn't it mean that its creating an object in that case?

This is quite involved. It shows:

1. He has worked out what getMoreSenior does, at a superficial level. The name must be a help.

2. He is tentative. He has self-awareness, in that he knows his grasp of the ideas involved here is uncertain.

3. He first thinks there is no new object, because the declaration of the variable 'another' is not followed by '= new Employee()'

4. Because the definition of getMoreSenior starts 'public Employee getMoreSenior', he has thought that it returns a new Employee object.

So he is still not clear about the big difference between a reference type and a primitive type - although he does know that he is not clear. A declaration of a primitive type like

```
int x;
```

looks structurally the same as the declaration of a reference type

```
Employee person;
```

but the former involves the reservation of memory to hold the int value in, while the latter does not. Similarly for a return type, for example

```
public int someMethod()..
```

looks like

```
public Employee someMethod()..
```

The former returns a value of type int, whereas the latter returns something of 'reference to an Employee object' type. But the latter need not be a reference to a new object created by the method - it might (as in this case) be an existing object. The former, by contrast, will always return a value in newly used memory.

Version 4 involves the use of a collection of objects, in an array, and the use of a static method. He follows through the code for mostSeniorEmployee with little difficulty, and recognises the pattern of the code, but cannot name it (such as a linear search for a mximum value). The use of an array as a collection of objects raises no problems. However he still has problems with static:

> This method called mostSeniorEmployee is declared with the word static in front of it, so what difference does that make?
>
> *It makes it belong to the class Employee..*
>
> As opposed to belonging to?
>
> *Um, …*
>
> Something like display which is not static, that doesn't belong to the class in the same way that this belongs to the class, so what does it belong to?
>
> *.. um (very quietly) trying to think*
>
> I mean there's some clue when we try to use it, it says Employee boss = Employee.mostSeniorEmployee,
>
> *OK so .. with this one you can call you can call an assignment to for display but with Employee err with this static one you have to call it through the class?*
>
> Through the class?
>
> *Yeah*
>
> So we say Employee dot mostSenior, whereas for display you say boss dot display and boss is an object.
>
> *Uh huh*
>
> So we can say that a static method belongs to the class, whereas a non-static belongs to?
>
> *Anything, or.. yeah .. oh would it belong to the objects?*

This shows that Matthew has absorbed the idea that 'static belongs to the class', but does not clearly grasp what this signifies, in terms of the implication for what non-static is.

Once again there is a problem with the term 'belong'. The word has several shades of meaning, such as

- Being a member of a group, as in "3 belongs to the set of odd integers"
- Ownership, as in 'This car belongs to me'
- Association, as in 'This sock belongs to that one', with the association sometimes being due to ownership.

The use of 'belong' in OOP is unfortunate, in that the first and third meanings apply at different times. All methods must belong to a class, in the first sense, treating a class as a set of methods (and data fields and constructors). For example the 'display' method belongs to the class Employee, in the sense that it does not belong to another class. But so does the static method mostSeniorEmployee. In other words both static and non-static methods belong to a class in this sense.

The difference comes with the third sense. A non-static method belongs to an object, in the sense that it must be used in association with an object, as in boss.display. By contrast a static method in use must be associated with the class, as in Employee.mostSeniorEmployee.

The situation can provoke a common mis-conception, in that it is often said that an 'object belongs to a class' and if this is interpreted in the first sense, that implies that a class is a collection or set of objects. This is true for methods, fields and constructors if we view a class as a set of those. But for objects it is only true in the third sense, that of an object being associated with a class.

Version five introduces inheritance through two sub-classes of the base class Employee. His response as regards inheritance is described next. This passage is not easy to follow, but it shows the way that Matthew is thinking about these ideas:

> The fact that these two classes are sub-classes of Employee, what effect does that have?

Um.. (5 second pause)

Let's say in terms of data fields, so what data fields does a ManagerialStaff have?

getsPaid and that's it - oh and name and grade.

OK so it has a new one, getsPaid, and it also inherits the two fields from there. Um, and Employee (sic - should be HourlyPaid) , has getsPaid as well, and also hoursThisMonth. What about methods? What methods does ManagerialStaff have?

It has a constructor and a method of pay.

OK

The constructor just.. I guess here it just.. it inherits the name and the grade which is in this one's constructor, so the ManagerialStaff would have to have the same fields as Employee, plus any other one's you want.

Would have to?

Would it? I'm not really sure on that. It would or wouldn't?

If we first of all establish what fields does it have? It has a field called getsPaid, and does it have any other fields?

No. Not in.. well apart from.. not in this class, but it inherits those two

Right so it has 3 fields then, getsPaid and those two, name and grade. OK so it has 3 fields. And you asked the question does it have to? So really you are saying does it have to have name and grade? Yeah?

In the constructor.

Ah in the constructor

Yeah

They are mentioned in the constructor, to the extent that we say super name and grade

Yeah

Um so we are saying does it have to. What would be the consequences to missing out that call to super?

ManagerialStaff would only have that getsPaid..um

Value?

> *Yeah yeah so like so whereas Employee has values of name and grade, ManagerialStaff would only have one of getsPaid.. Is that right? Or would it still inherit without super?*
>
> Right - would it inherit the fields? Yes.
>
> *OK*
>
> Because that's the way that extends works, it would inherit the fields. If we missed out the call to the super constructor, would those fields have values?
>
> *No*
>
> No. OK. So we'd have the fields, but no values assigned to them.
>
> *Oh OK.*

It is clear that Matthew has only a partially normative view of inheritance, but his ideas are clearly fluid and insecure. There appears to be 2 main points

1. The sub-class must have the same fields as the base class - as distinct from the normative view that a sub-class inevitably inherits the base class fields. A possible basis for this is some confusion with interfaces - a class must both state what interfaces it implements, and to actually implement them. If inheritance worked the same way, a class would need to say 'extends..' and also to actually declare the same fields as the base class. In fact it does not.

2. Matthew does not have a clear distinction between variable and value. He knows that without the call to super, name and grade would not have values, which connotes the idea that the fields are not there.

However because his grasp of these ideas is tentative, he cannot clearly articulate them.

Version six addresses two ideas - abstract classes and polymorphism. A long extract is presented next, but it is needed by virtue of what he says at the end:

> If we first of all think about - this is about Employee becoming abstract - and it gets a pay method - is that sensible?
>
> *What do you mean having a pay method?*

Right really we are talking about class design here, OK, so is it sensible to give Employee a pay method?

Um, yeah I think so because it means that for each of the employees you can say how much they get paid or, yeah.

OK. In the previous version we had 2 sub-classes, each of them declared a pay method. We've now put it into the Employee class - is that an improvement?

Well I suppose it's a bit pointless in a way

Ah OK why is it pointless?

Because if you've got them already in Managerial and HourlyPaid, then there's no need to have it in Employee as well.

OK

Yeah I don't know because when you say when you're writing a new class like WorkForce and you've got this abstract method in the abstract class Employee, and you extend Employee in another class then you have to implement all the abstract methods so, yeah I don't know, it doesn't seem like it makes a difference, to me.

OK the pay method is abstract, its declared to be abstract, what's the significance of that?

Um, does it mean all classes that extend the class have to implement the pay method?

OK yeah I think you've already said it - they have to implement the pay method. OK - suppose you didn't - what would the compiler say?

It would complain that you would need to have a pay method.

OK - that is if the sub-class was not abstract - if they were still abstract it would be OK, if they were not abstract you'd have to implement the abstract methods. So what's the point of it? Given that if you were to define a third sub-class of Employee, you'd have to implement the pay method, in which case what's the point of that?

.. Is it kind of like a reminder?

Ah - a reminder to what extent?

That you need to have a pay method like if you knew that you wanted to in all classes you wanted a pay method for all these different employees, then it might be a good idea because it stops you forgetting to code it in.

OK exactly saying abstract pay means that any subclass of this has to have a pay method, which is reasonable because all employees have to get paid, and so saying public abstract pay means that if you define another one, and if you forget to implement tha pay method the compiler will tell you you have to write the pay method. OK um but in that case why not implement the pay method in the Employee class?

.. um .. I suppose you could..

That would save you time implementing it in all the sub-classes. But there must be a reason why not then.

Because is it because .. pay.. would it have to stay the same if it were in Employee class

OK yeah keep going

and you might want to have different rates of pay or whatever for the different employee classes, the extended classes.

We already have the possibility of different rates of pay, in that the HourlyPaid each person has a field called rateOfpay so the different objects will have different pay per hour

Well in that case is it because it wants to have in the pay method they aren't the same so you use so in this case you are using a field

In other words the method of how they get paid is different for different types of worker, so they all need a pay method but how its actually going to happen

Yeah

is going to be different for different sub-classes. Yeah. And that's a reason why we can't define it in the base class. What would happen if we did - what would happen if we gave a definition in the base class, and then a different definition of the same thing in a sub-class?

So if you gave it a definition of pay, and then a different one?

Yeah in a sub-class?

..I would think it would produce some sort of error.

This long extract is needed because that final statement is significant for everything that has gone before. It shows that Matthew is *unaware that an inherited method can be over-ridden in a sub-class* - which would alter his whole idea of inheritance. He would think that a base class method would be declared abstract, not because it would be impossible to formulate the method at that level of generality, but because any implementation could not be altered down the class hierarchy. It would also prevent him thinking that some functionality present in most hierarchy members could be factored out and implemented in a base class, for fear that it might need to be altered in a sub-class. This is a good illustration of a basic concept mis-conception forcing a perverted understanding of consequent concepts. For example, statements will be mis-interpreted, such as the interviewer's "And that's a reason why we can't define it in the base class." is taken to mean not that there is any pay method for a general Employee, but that if we defined one, all sub-classes would have to use it.

In the interview this is clarified, and then the class WorkForce is discussed, with its pay method, which invokes the pay method of all the employees. The idea of polymorphism is raised:

In OOP, there is a word called polymorphism

Yeah I've heard that before

You've heard that?

Yeah

What does it mean?

Um.. I think.. a demonstrator told me last week actually.. can't remember.. just.. does it kind of mean that you can have lots of different classes branching from one base class?

So he is guessing that polymorphism simply means a class hierarchy, and cannot recall what he has been told. Given that he did not grasp the idea of base class methods over-riding sub-class versions, no explanation would have made sense.

Interview Summary

He has some awareness of the idea of reference but not a firm grasp.

His conception of the static/non-static distinction is still weak.

His ideas about the inheritance of fields is uncertain, and he sometimes looses the distinction between variable and value.

Mis-conception of sub-class methods over-riding base class versions makes coherent ideas of inheritance and polymorphism impossible.

8.5.9 Interview Nine

At this point Matthew has completed the module and is in a revision period prior to a written examination paper. Consequently the interview starts with an invitation to him to discuss any topic. He raises the topic of loop invariance - this is discussed. The plan of the interview was to look at overloading (more than one method with the same name) contrasted with over-riding (a method defined in a sub-class over-riding a method in a base class), and then to re-visit the class design for the 'Game' considered in 8.5.2, where he could not suggest any appropriate classes, objects or methods..

> Now there's a term in object-oriented programming called 'overloading'.. um have you come across overloading?
> *Um.. I've come across overflowing - is it the same, or not?*
> So what would you say overflow is?
> *I've come across the error in NetBeans which is stack overflow*
> Oh right yes stack overflow
> *which I believe is if your program ends up going into an infinite loop? or like something in your code is wrong, not doing what you're intending it to do, then it ends up filling up all the memory breaking the program.*

So he does not recognise the word, but suggests it might be the same as overflow. The core meaning of 'overload' has an element of meaning of 'placing requirements on an system which cannot be met', such as overloading the weight on a girder. This has something in common with overflow, which means exceeding capacity. The two terms both start with over, so it is unsurprising that Matthew is seeking to construct a meaning for overload as a synonym for overflow.

His mention of an infinite loop, rather than infinite recursion, suggested his understanding of stack overflow in this context was non-normative. Since the examination was imminent this topic was explored, and indeed Matthew did not know why a stack was involved. The idea of needing to store return addresses on a stack was discussed, and how an infinite recursion would imply an infinite stack was covered. The other common use of overflow, that of arithmetic overflow, was also discussed.

The interview then returned to the idea of overloading in the OOP sense. This was described and exemplified in terms of methods and constructors, and also operators. Matthew seemed to not have met the term before.

> In OOP there's also something called over-riding - have you come across over-riding?
> *I think so. Is it if you have a set variable and if you assign a say a number then later on - dunno then you assign it to another number that ends up over-riding the variable's value?*

So in fact he is suggesting for over-riding a meaning which would usually be for *over-writing* - store a value in memory which would replace existing contents, which would be lost. He is therefore doing the same for over-riding as he did for over-loading - he does not have the normative OOP concept, but he constructs a meaning by borrowing a concept which seems to be related in some way - here over-riding and over-writing.

This was discussed, and then:

> In terms of object-oriented programming, in terms of class hierarchies, does over-riding mean anything there?

> *Um not that I can think of, no.*

So Matthew has no idea of OOP over-riding. This is presented to him:

> OK its used because in if you've got a class hierarchy, you have a base class you might have a method with a certain method in that base class, and you might also in the sub-class have a method with the same name
>
> *Right*
>
> and its about what happens in that situation. So if you've got your base class, lets call it Base, and its got a method in it which is called display, and it prints out something, now suppose there is a sub-class called Sub, and suppose we program that with a method called display as well, yeah?
>
> *Yeah*
>
> and it's a question of what happens when we have an instance of this class and call its display method, what happens.
>
> *OK*
>
> And what do you think happens?
>
> *Um.. would it be calling the same display method that's in the base class?.. Or*
>
> Or?
>
> *um (laughs).. the subclass would over-ride the display method in the base class?*
>
> Yes
>
> *Ah!*

This is an example of how the student constructs a concept, facilitated by a teacher. The 'Or?' implies to the student that the first possibility is wrong, so he must look for other options. In fact there are only two possibilities, and he sees that in the other option he has the possibility of using the term 'over-ride', so this would enable a radial extension of meaning from 'riding over something' to the metaphorical situation where the sub-class method is used in place of the base class method, and at the same time enables polymorphism to make sense.

In fact the idea of this had been discussed in the previous interview, where sub-classes had methods which over-rode base class methods. Matthew had developed some idea through the course of that interview, but had lost it again for this one. The idea was discussed further, largely in the context of the toString method.

The interview then proceeded to the 'Game' (Appendix One):

> What objects would you have, if you did that in Java?
>
> *Um.. (9 seconds) would you have objects for each of the human's and computer's number? With the values of the random numbers which are produced?*
>
> So you might have humanNumber and computerNumber?
>
> *Yeah Um.. and they'd be of the same object which yeah have values which would be*
>
> Is that like their 1 or 2 - when you say their number, they either choose 1 or 2
>
> *Yeah that would be yeah*
>
> So that's like their choice 1 or 2
>
> *Yeah*

So Matthew focusses on two values, and not objects, in the sense of agentive objects which do things

> Now would these things be objects?
>
> *I guess so because you'd have to.. you'd need to be able to say um like which number the human which number the human had chosen like if the human number had a value of 1 or 2*
>
> OK
>
> *And you could assign .. the number to that to just human number so I dunno*
>
> Yeah yeah
>
> *but in the second stage you need to have another number in the range 1 to 50*
>
> Yes
>
> *So you couldn't.. assign that as well without overwriting the 1 or 2 that's why I think there needs to be an object because it needs to have 2 different numbers*
>
> Yes there's the choice and the score
>
> *Yeah .. that's why I think you need objects.*

He might mean objects are needed because they would need to have two fields, for the choice and the score, but what he says later implies he does not see this clearly.

> OK you need objects the question is, which are the objects?.. So you need an object to rep.. what things have we got in here?
>
> *You've got the choice of 1 or 2, and you've got the random number generation of 1 to 50*
>
> Right OK now I'd say like the choice of 1 or 2, that's going to be 1 or 2, so that would be an integer, yeah? And then there's the bit about producing random numbers, and that would be, I'd say that was a process, um, so maybe it's a method, in Java speak, um OK, so what - what objects have we got?.. So an object is going to have associated with it, some values
>
> *Yeah*
>
> numbers whatever, and it'll have methods that will be capable of doing things
>
> *So in this case the object would have the value either 1 or 2 and a method for random number generation.*
>
> OK and what thing would that object model? What thing would it correspond to?
>
> *Um, the score?..(inaudible)*

So he is still thinking of an object as being a single value. He sees an object as a value, rather than an object having attributes which are values, and methods. He has to be basically told how to do it:

> OK if I were doing this, one of the objects I'd have in there would be as it were the human player, yeah?
>
> *Yeah*
>
> So I'd have a class called HumanPlayer. Right. What data fields would that have?
>
> *Um an integer which would be input by the user.*
>
> Which would be their choice?
>
> *Yeah*
>
> So there'd be a field which we might call choice, which is going to be 1 or 2, any other data members the class might have?
>
> *Yes int rand, or randNumber for the random number, then you'd have a method further down which would give that integer a number, a value*
>
> OK what name would you give that field?

> *Um randomNum*
>
> OK could do, but if you call it random number that suggest how we obtain it, it doesn't suggest its significance or what it means
>
> *OK so you’d call it score*
>
> Yes you'd call it score OK, so we've got 2 data fields, we've got this class HumanPlayer, how many instances of that class would we have?
>
> *You'd have one.*

and so on. Matthew is still finding it very difficult to utilise the ideas of class and object to structure the modelling of the Game. The phrase "further down" is significant, since it shows that he is thinking in terms of program text, rather than thinking about the Game in terms of classes and objects.

Interview Summary

1. He does not recognise the OOP term overloading, nor the idea, and instead he tries to construct the concept based on overflow

2. Neither does he recognise over-riding, even though it would have been covered in class, and in the previous interview. He constructs the idea in the course of the interview, but it may not be retained.

3. He finds it very difficult to model a situation (the Game) in class and object terms. He thinks of these as features of program code rather than modelling other situations.

8.5.10 Matthew summary

Matthew starts the interview sequence displaying some proficiency with procedural programming. However, even by the end, he has significant problems dealing with OOP concepts. He has limited understanding of references, static, encapsulation, inheritance, over-loading and over-riding. He cannot describe classes and objects in general, or use classes and objects to model a situation.

CHAPTER 9 - Phase 2 College

CASE STUDY DATA

9.1 Introduction

These interviews relate to those in the previous chapter, in that both take a case study approach. However these two students are on a Foundation Degree programme at a College of Further Education. Much of this content concerns structured programming rather than OOP, since their coverage of Java was slow and limited. This chapter concludes with an interview with a professor of Computer Science, to elicit an 'expert' view.

9.2 Aaron

9.2.1 Interview One

Aaron is a mature student, who left school with no qualifications and worked in the construction industry. He later passed some GCSEs in adult education classes, then in 2007-8 he completed a City and Guilds Diploma in Software Development, a one year full-time programme. At the time of these interviews (autumn 2008) he was in the first year of a Foundation Degree, a two-year full-time programme, which might be followed by a third year to obtain an Honours degree in Computer Science.

When presented with

```
a=0
b=0
repeat 5 times
    a=a+1
    b= a + b
output b
```

he tries to work it out in his head, but he misses the fact that 'a' will change, except for the first time when it is 1, and so suggests the result will be 5. However when invited to write it down, he does so and then sees that 'a' will change, and gets the correct result of 15.

We then move to discussing why it is 15.

> *The different values of a are..*
> Is there a pattern to them? It’s a variable, but is there a pattern?
> *The pattern, a goes up in 1 each time*
> OK yeah
> *And the b the b's got a I suppose the b's got a pattern if I look at it this has gone up 2 times this has gone up 3 times this has gone up 4 times and this has gone up 5 times*
> OK
> *Well not 5 times but by 5 so its gone up 2 there, 3 there, 4 there and 5 there.*
> Right OK so we've got in this program a loop, could we replace the loop by one assignment, to one expression, which might be quite a long expression, but which is just one expression, could you do that, and calculate it in one step?
> *(10 seconds) oh yeah yeah*
> So what would it be?
> *Well I can see what you're saying, because all the a's added together add up to 15 don't they?*
> So why don't you right that down then?
> *Um you could put a times 5 but that's not right because a times 5's not right*
> OK so you could kind of say its like a times 5, but you can't really
> *No*
> That's not right
> *Because that could be 5 or could be 10*
> OK so the a changes
> *Yeah*
> Yeah - so what else could you do?

(8 seconds) I know that when you tell me I'll go yeah yeah why didn't I think of that (8 seconds)

Suppose I start you off b =1 +

Oh yeah a?

b = you see the issue is that a changes, so that if I say b = 1 + 2 and you finish it off

Oh got you got you yeah I thought you were looking for letters

Ah well no well - I don't think you can do it with letters

Its like I said before, I thought there was some way of doing it with the letters

The phrase 'I thought you were looking for letters' is critical. He is thinking that 15 might be like his suggestion, 5a, an algebraic formula. It suggests Aaron is still focussing on the program text frame (see section 10.6.1), and not on the machine state frame or the blend.

9.2.2 Interview Two

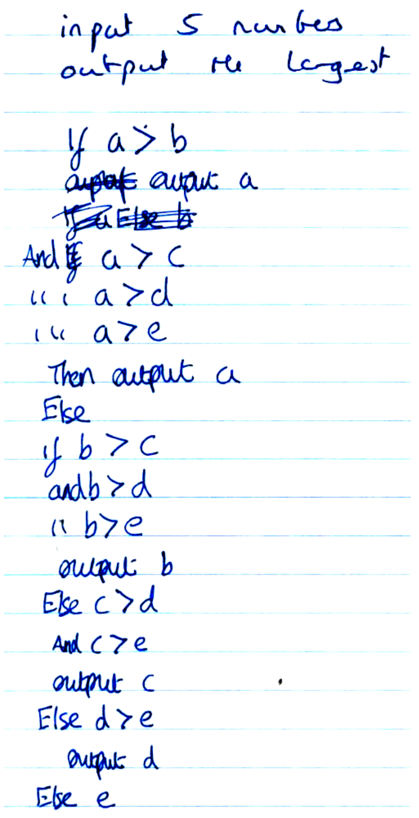
The task

input 5 numbers

output the largest

is given. Aaron thinks he has done this before, in VB, but is very uncertain. He establishes the need to use an 'if', and eventually produces a solution as shown in Figure 9.1.

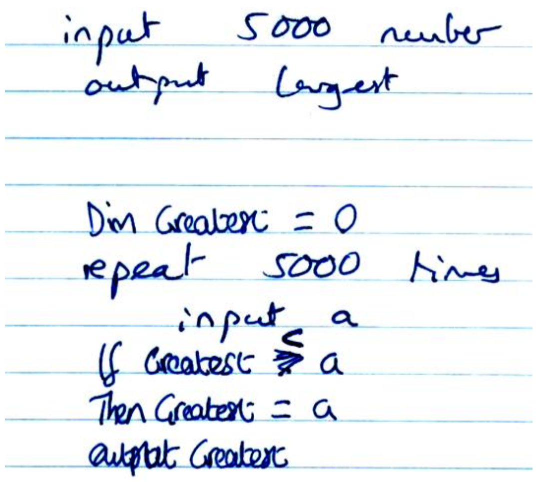
This shows a logical approach, first establishing if a is the greatest, then b, and so on. There is also an optimization - he sees that if a is not the largest, when checking b we do not need to see check again that b>a.



```
  input 5 numbes
  output the largest

  If a > b
  ~~output~~ output a
  ~~If a Else b~~
And ~~If~~ a > c
 " " a > d
 " " a > e
  Then output a
  Else
  if b > c
  and b > d
  " b > e
   output b
  Else c > d
  And c > e
  output c
 Else d > e
   output d
 Else e
```

Figure 9.1 Original method

The extension of this to a large number of numbers is



```
input 5000 number
output Largest

Dim Greatest = 0
repeat 5000 times
    input a
If Greatest $\overset{<}{\cancel{>}}$ a
Then Greatest = a
output Greatest
```

Figure 9.2 Method with loop

then discussed. At first Aaron cannot suggest a solution, and it is established that he has never before seen code to find extreme values. So the idea of maintaining a 'greatest value so far' is introduced, and he can produce a solution as shown in Figure 9.2.

Interview summary

1. Aaron shows he can devise original solutions to basic programming problems

2. With support, this can include loop constructs

9.2.3 Interview Three

Illness had meant that Aaron had missed some weeks of the course on Java, and at this point had done little more than seeing the use of primitive types.

The meanings of public, static and void were discussed. He saw public as meaning 'open to anyone', and static meant 'can't be moved', but said that these were largely guesses.

The problem of exchanging variable values was presented.

How would you do that?
In programming?

> Yeah, kind of. When I say kind of, I mean in pseudo-code, I don't care if its in Java or Visual Basic or whatever, how would you - x is 1 and y is 9, how would you swap the over?
>
> *..Times them by, no no, no idea. No idea.. In maths you'd just multiply them wouldn't you,*

The idea of multiplying is possibly that if x is multiplied by y, it becomes 9, the initial value of y. Of course y does not become 1. The interviewer tries to move away from the manipulation of numbers:

> OK let me stop having that being 1, suppose that was 3 x is 3, y is 9
>
> *That would be easier wouldn't it*

This has not worked - Aaron simply thinks we have 'easier' numbers to manipulate.

> Do you want to write it down
>
> *I think it would be anyway..you'd divide them both by 3 wouldn't you.*

So Aaron is trying to find an arithmetic solution. If you divide y by 3 you would get 3, which was the value of x. But of course again this does not work for the other variable.

> This method is going to have to work for all values, so whatever they are, we want to swap over their values… Ah because you're looking at these numbers and trying to see some kind of connection with them, but its going to have to work for all numbers, not just for these. So you can't try and figure out some kind of connection with the numbers.
>
> *..No.. unless.. I was thinking the divide as well, because you know when you're working it out with fractions, if you divide them you sw- you flip them over - you times them don't you? You flip them over then multiply them but - its obviously nothing to do with that.*

So this problem has been seen by Aaron as an aspect of the process of dividing fractions, in that 'flipping' the dividing fraction in one sense exchanges the values of the numerator and denominator, and so he is conjecturing that division might solve this problem.

This was discussed, then:

Suppose it said, x =, and make it really simple.

(He writes down x = 3 + 6, when x is initially 3 and y 9)

Ah no, start again. You've got x and you've got y, and you need to swap them over

Oh. x=y.. y=x

OK that's what most people say, but its not right

Its not right

Because if you imagine like these are memory boxes, yeah?

Right. Is it y - x? x = y - x? y = x - y?

.. x = .. what did you say? x =

y-x

Why would it be y - x?

Oh yeah I'm just swapping them round, it wouldn't, no. Right that's me. It would have been a different number anyway wouldn't it. It would have been 6 and -6 or something.

Alright OK if we take it as particular numbers just to see what happens, like 5 and 4 maybe, so to start with, we've got a 5 in that box (x) and a 4 in that box (y).

Yeah

Then we say x=y, so what happens?

So x equals 4 and y is 5

Aaron is saying that x=y accomplishes the swap. What follows suggests he is not clear about this. It may be that he thinks the interviewer thinks that x=y does it, so that he thinks he should agree..

So does that change what's in this box (x) ?

No, its just..

Does it change what's in this box(y)?

No its just a reference isn't it? To a location? So it stops in there doesn't it? It kind of swaps - no, it doesn't.

This makes sense if Aaron thinks of x and y as pointers, or references, to locations. If that were the case, the 5 and 4 are unchanged (*it stops in there*), and in fact x=y would change x to point to where y points. But he is not clear on this.

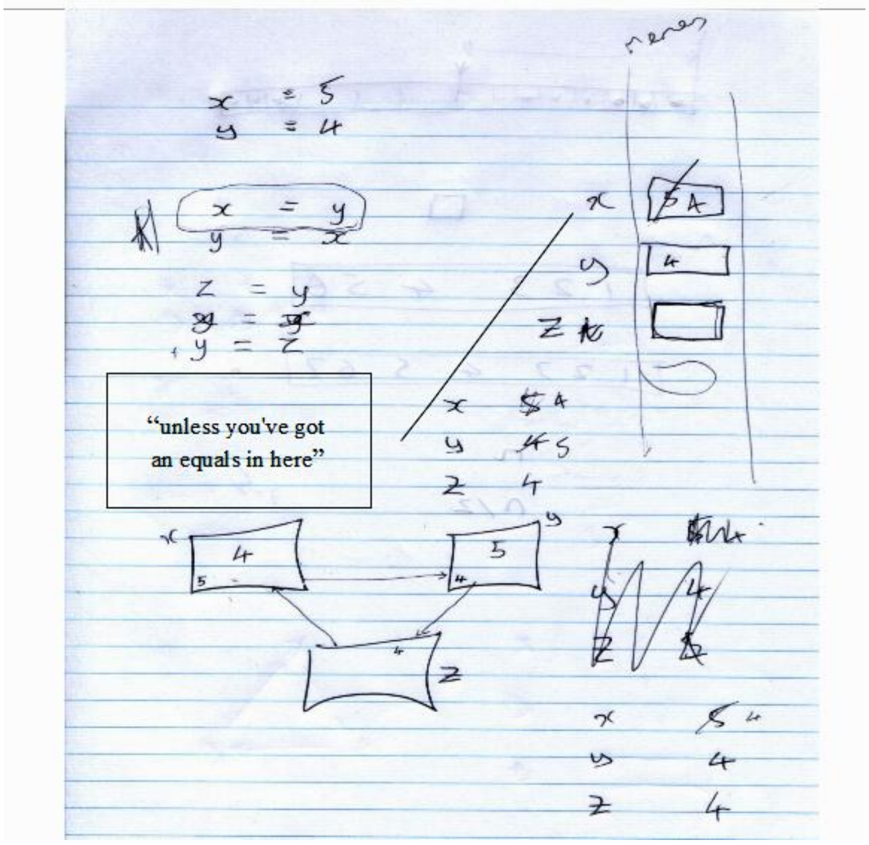


Figure 9.3 Trace

So that x=y - what does it actually make happen?

Stop getting the answer from there and get it from there. I don't know if it actually flips it around - unless you've got an equals in here (see Figure 9.3)

This also supports the possibility that he thinks of x and y as reference*s*

OK what we're thinking this picture as being, this is a picture of the computer's memory, yeah? a whole set of cells, storage locations in it, which are all used for different purposes, as the program is running

It would swap them around then if it's a program, it would swap those round.

OK if we take it one step at a time, that first step, x=y, does that change values in memory?

I think it does, yeah.

In what way?

It will take the 4 into the 5 box.

OK x= y, it will get the value of y, which is 4, and put it in that box.

It will substitute 4 for the 5. Yeah

So that's happened (cross out 5, write 4 in the x box)

So the next one, y=x, its going to have y=4 still.

He has now grasped why x=y y=x will not work, by seeing what those instructions would actually do. So what did he think they would do when he thought it would work? One possibility is that he did not realize that the first x=y would change x, so that the subsequent y=x would give a different value to y than if the first instruction had not happened. In other words he does not know that program execution changes state, and that state affects program execution. However another possibility is that he does not have a confident grasp of the nature of a variable as an address label (a named location, a symbolic address, or in Java terms, a primitive type) in contrast with a pointer (an actual address, or in Java terms, a reference type). This is discussed further in the next chapter.

In the interview the problem of finding a working solution was addressed. The interviewer had to suggest using a third storage location, and the first attempt to use this had to be corrected, as shown in Figure 9.3 above.

Once the idea of exchanging values through a third location seemed to be established, the task of reversing an array was introduced.

> How would you reverse the array - swap it?
> *Isn't there commands for this?*
> Ah, could be, but how would the command work?
> *Oh got you got you*
> How is it going to do it?
> *I would have thought it would have just swapped this one for this one, and this one for this one,*
> OK OK so you swap the first one with the last one,
> *Yeah second and one from last*
> How would you do the actual swapping?
> *Ah you'd probably need another empty array, no err, somewhere to put it*
> Well it could be an array, or it could be a single box
> *A single box, probably*

Right you've got just one box, and using that, how would you swap those two?

I would put that into there and that into there

And then that back into there

That back into there

So really it’s the same kind of thing as this

Yeah yeah

So you swap that and that, and then what?

The next one the same thing

That into there

There into there and there into there

So what kind of program control construct would you use?

Er.. replace, move,

OK what is a control construct? Um, could be an if? Would it be a loop?

It could be an if I think. Yeah? I don’t know.

So with a little support he can see the connection between this and the exchange process. But even though he has a good grasp of this algorithmically (*The next one the same thing*) he does not see that as a loop.

Interview summary

1. In the exchange problem, his initial concern is to exchange the numbers, not the variables. He looks for arithmetic transformations which would change over the numbers - as distinct from the variable values, to obtain a solution which would work for all values. This suggests he does not actually have the concept of *variable*, which would fit with his lack of awareness of the dual text/state frames.

2. His idea of a variable fluctuates between an address label and a reference.

3. He suggested doing the array reversal within the array, but could not see how it related to the exchange process without prompting.

4. For the array reversal problem, it was not obvious to him that this involved a loop, and he suggested it might need a conditional statement. This implied absence of

the loop concept is consistent with a lack of awareness of the dual text/state frames.

9.2.4 Interviews Four and Five

Aaron experienced financial problems causing poor attendance and leading him to the verge of dropping out. He made some recovery, but he failed the examination for the Java module, requiring a re-sit. Consequently the interviews changed in character to more direct tutorial sessions, in the hope that he would pass. Interview Four included looking at the 'syllabus', which covered a range of OOP concepts including inheritance and polymorphism - ideas which aaron had no idea about, since interview five shows he has only a limited grasp of the ideas of variable, loop and conditional statements. Interview four also introduced Aaron to the 'debug perspective' in Eclipse, the IDE he used in class. The reason for this was to support the visualisation of the 3 spaces (see section 10.6) - program text, state space and the I/O space.

Then I sent Aaron some small programming tasks, for the purpose of providing experience of thinking in those three spaces. Interview Five involved working through those tasks, and this is reported next.

The first task was a given example, and the second a very simple development of this, which Aaron could do by himself – theses tasks were as follows:

1. Here is a Java program which inputs 2 numbers, adds them up, and outputs the total:

```
import java.util.Scanner;

public class Main {


public static void main(String[] args) {
int x;
int y;
int z;
Scanner scanner = new Scanner(System.in);
x = scanner.nextInt();
y=scanner.nextInt();
z=x+y;
System.out.println(z);
}
}
```

Try this out.

2. Change it so it inputs *three* numbers, adds them up, and outputs the total.

However Aaron 'got stuck' on question 3:

3. Write a program which inputs a number, increments it (use something like x++;) and outputs the result.

The problem was that he had written

```
x = x++;
```

rather than

```
x++;
```

The reason why the former does not work, and the latter does, involves the slightly obscure difference between pre-increment (++x) and post-increment (x++). This was discussed.

Question four was:

4. Change question 3 so it increments it 4 times, and outputs the result (do not use a loop).

The idea of this was to illustrate an 'unrolled' loop in the form of repeated statements, which can be more effectively done in a loop. However Aaron has already experienced loop and other statement types in class, which simply leads to confusion - his response was:

> *Without using a loop - um*
>
> Yeah
>
> *So I can't do 'do until' or 'do while' that's all loops isn't it, which one's not a loop, um, for's a loop isn't it? for's a loop as well.. if.. is that a loop? if's not isn't it?*

The directive in the question to not use a loop has been focussed upon, which leads him to think 'what type of statement should I use then?', leading him to an if statement, since that is not a loop.

His previous classes involving programming statements are recalled, but they have not been fully assimilated, in that he is not sure that an *if* is not a loop, and it is not obvious to him that he can simply write x++; four times, although he eventually suggests it.

5. Write a program using a for loop which outputs 0,1,2,3.. up to 9.

He has the following problems writing a for loop:

1. Vagueness in recalling the role of the three parts in a *for* loop header

2. Using variable name 'i' for the loop variable, copied from notes, while declaring a variable named 'x'

3. Writing 'For' not 'for'

4. Putting a semi-colon after the loop header, as 'for (..); '

5. Struggling to see that an output statement is needed in the loop body.

Question 6:

6. Change 5. so it outputs 0,1,2,3 up to 100

is done by Aaron without help. However in question 7:

7. Change it so it outputs 0, 2, 4, 6, 8, 10

he has a problem in incrementing x by two. With support he eventually suggests

```
x=x+2;
```

and this is incorporated into the loop. Similarly he can do question 8 with support:

8. Change it so it outputs 10, 8, 6, 4, 2, 0

For question 9:

9. Write a program with a for loop which adds up 0 + 1 + 2 + 3 + ...10

he realises that he needs two variables, (named x and y), but has two for loops, one for x and one for y. This is corrected, and but he needs a lot of support to move to a statement like

```
y = x + y;
```

to sum successive values of x.

He can do question 10 with help:

10. Write a program which inputs a number, and outputs up to that number from zero. For example, if the user inputs 5, it should output 0 1 2 3 4 5

He sees this needs input so we need a Scanner object, but is then at a loss for how to go forward. He suggests we might need an *if*, possibly because the question says 'if the user inputs..', and he needs help to see that a loop is appropriate. He then finds constructing the loop difficult:

I'm getting stuck on this one because its inputting numbers and there's no actual, there's no actual number that it starts from, it doesn't start from anything, its just whatever the user inputs.

Aaron is having to use the concept of a variable in the sense of something having a value which is indeterminate. The condition for the loop to end cannot be expressed as an 'actual number', but is 'whatever the user inputs' - precisely the situation where this sense of the variable concept is needed. With support he is led through to a conventional solution.

For question 11,:

11. Write a program which inputs 5 numbers, adds them up, and outputs the total.

he starts:

Does that mean I'll need five inputs? a b c d e?

So he is thinking of five variables and no loop. This is discussed, and he says:

It seems quite long-winded it seems you could use a loop

showing that he is beginning to see when loops are appropriate. With help he sees that we now need three variables - the loop counter, the value inputted each time, and the total.

Interview summary

Significant aspects of this (not all transcribed here) are

1. Lack of facility in using Eclipse, the IDE

2. A range of miscellaneous errors which obscure the problem-solving - such as having an import before a package statement, or writing 'For' instead of 'for', or putting a semi-colon at the end of a 'for' loop header.

3. A partial grasp of the ideas of variable, loop and conditional.

The first two points in one sense are insignificant, in that they do not relate to a deep understanding of Java programming. However they stop Aaron from writing programs which work.

The third point shows Aaron being able to think in terms of the three spaces of program text, state and I/O space, but he needs support in all but the simplest problems. In fact he actually says

> *I wouldn't have done that on my own*

and this is clearly in Vygotsky's idea of the ZPD. It is suggested that this experience is useful in development of frames to structure the three spaces.

9.2.5 Interview Six

This continued in the tutorial manner, addressing nested loops and arrays.The first task is:

```
public class Main {
public static void main(String[] args) {
for ( int i =0; i<4; i++)
    for (int j=0; j<4; j++)
    {
        int x = i * j;
        System.out.println(x);
    }
}
}
```

Aaron first suggests this will output 1 2 4 8, then that it will 'go' 1X1, 2X2, 3X3 and 4X4, and so output 1 4 9 and 16. So he sees i and j increasing together in step. He then says the output statement will execute eight times - since the single loop executes 4 times, and we have two loops. The program is run, and he is surprised by the output. The idea of nested loops is then discussed with the analogy of a week cycle inside a month cycle, and eventually he can understand why the output is what it is.

We then go on to:

> In the last program, if you changed
> int x = i * j;
> to
> int x = i + j;
> what would the output be? Try it and see.

Aaron tries to work this out on paper, and does so correctly.

We then go on to:

> Modify this program to output
> a. 1 2 3 4 2 4 6 8
> b. 1 2 3 3 6 9 5 10 15

For (a), he first tries ending both loops at 2, then at 1. His attention is directed at looking at the pattern in the numbers, and with help can write the program. For (b), he starts by looking for the pattern. He suggests there are three loops - a possibility, but these would be three sequential loops, rather than two nested loops. He struggles to see how 3 6 9 relates to 1 2 3, since he looks at how much the numbers go up by rather than 1:3, 2:6 and 3:9. With help he sees multiplying 1 2 3 by 1 3 and 5 will give what is required. But he still needs help with

```
  j=j+2;
```

to make the outer loop iterate with the required sequence.

Moving on to arrays, he can say that an array is a group of values, and that elements are accessed through an index. The first task is:

1. Declare an array of 100 integers, and fill it with random integers in the range 0 to 50;

He has not 'done' arrays in Java, so he is shown the syntax for declaring an array. To fill the array, he knows there is a random number generator, and that a loop will be needed. With help he produces the correct code, like

```
for (int i=0; i<100; i++)
   numbers[i] = (int) (Math.random()*50);
```

The code for producing a random integer from 0 to 50 is given to him, since it is pretty idiosyncratic.

We then go on to

2. Print out the array

He sees we need a loop, and we copy and modify the first loop to

```
for (int i=0; i<100; i++)

System.out.println( Numbers[i] );
```

He says he understands this, but we go on to:

3. Add up the array

He sees we need another loop. He first says that we'll need

```
i = i + i;
```

Using a running total is suggested to him. He says this should be initialised to zero. For the assignment in the loop he first suggests

```
total = total +i;
```

When asked how to refer to each number in the array, he says to use the index, but for the loop body he suggests

```
total = total + numbers + i;
```

and

```
total = total + i + 1;
```

and

```
total = total+numbers[1];
```

and others. Eventually we get to:

```
total = total + numbers[i];
```

He has to put together here the ideas of

- adding successive values to a running total
- how to use a single addition inside the loop to stand for 100 actual additions
- how to move on to the next
- how to refer to a value in the array via an index variable which varies through the loop mechanism.

For the next task:

4. Find the largest in the array

he immediately suggests a nested if - so he has a significant grasp of the solution. He struggles to distinguish between i and numbers[i], but with help he can come up with a correct solution.

9.2.6 Aaron - Summary

Aaron clearly does not have a firm grasp of the basic concepts of variable, conditional and loop. His lack of attendance caused by financial problems meant he knew very little about OOP and Java, but he found the programme of study moving on when he had little understanding of foundational concepts.

9.3 Phillip

9.3.1 Interview one

Phillip spent his last year of school (age 15-16) in hospital and took no examinations, although he had been studying GCSEs and a GNVQ Intermediate in ICT, so he had shown an interest in Computing at an early stage. In 2006-7 and 2007-8 he followed a BTEC National Diploma in IT, completing this with 2 Merits and 1 Distinction, a good grade set. In 2008-9 he is in the first year of a Foundation Degree programme, which after 1 more year may lead to a third year to obtain an Honours Degree in Computer Science.

The first program:

```
a=0
b=0
repeat 5 times
    a=a+1
     b= a + b
output b
```

was presented, again with the question 'what would this program do?' He tries to do it 'in his head':

> So what would the result be?
> *What would the result be?*
> Yeah
> (10 seconds pause) Would it be .. I dunno.. (10 seconds) would it be somewhere between 15 and 20 I think - I don't think I'm right there?

He later describes how he worked this through but started to lose track at around a=3, and then estimated the result.

He was then invited to work this out on paper, which he did immediately. At the stage a=3 he makes a mistake, but he notices this himself and corrects it himself, and obtains the correct answer.

Pursuing the summarising of the program reveals an interesting mis-understanding:

> Could you do it in one step?
> *I suppose you could.. err..I don't think you could actually. I can't, no.*
> OK Suppose we had it like b = something something something something, in one step, is that possible?
> *I don't think so.*

Why not?

Because you have to do 2 calculations, each loop, so it would need more than 1 step.

(So 'doing it in one step' was interpreted to mean retaining the loop, but just having 1 statement in the loop. However doing without the loop was seen as impossible)

Right you do two things each loop

Yeah

Ah OK I see what you mean, right, I don't mean just having one thing in the loop, I mean not having the loop.

Ah so just write it out simpler like that instead of having the loop

Could you do that?

You could put a=1 and then you could put b = , no err no I don't think I could.

So again

b = 1 + 2 + 3 + 4 + 5

does not arise.

Interview summary

At this point Phillip is fairly proficient with program tracing, using the ideas of variables and loops.

9.3.2 Interview Two

This started with a discussion of Visual Basic programming, concerned with event-handling, which revealed an interesting mis-understanding - Phillip thought of an 'event' as meaning what happened as a *consequence* of the user clicking a button (for instance). So his notion of an event was what the 'on-click' method executed. This contrasts with the normative view, where the event is the user's button click (not its consequence), so that the on-click method could be seen as an event-handler.

We then moved to programming the task:

input 5 numbers

output the largest

He replies:

> *It would have to have some kind of calculation so it would know which was the highest number*
> OK
> *So then I suppose it would have to like do five steps of checking which one's checking each number , and then putting them in order, then it displays the last one in the order. So you could do it in five steps.*

This is then the common response of sorting the values and outputting the last one. Phillip has never written a program like this, but he can recall the term 'sorting'.

He is then asked how the computer would 'hold' the five values, in order to see if he would talk about an array. In fact he responds with a set of 5 text input fields on a form (Figure 9.4).

Here the left column are 5 input fields. On the right there are 5 buttons which 'process' the input values, and at the bottom of the right-hand column is where the output is displayed. This is interesting because it shows exposure to Visual Basic form design has coloured Phillip's approach to programming a task, with the idea of a data structure being conflated with interface design.

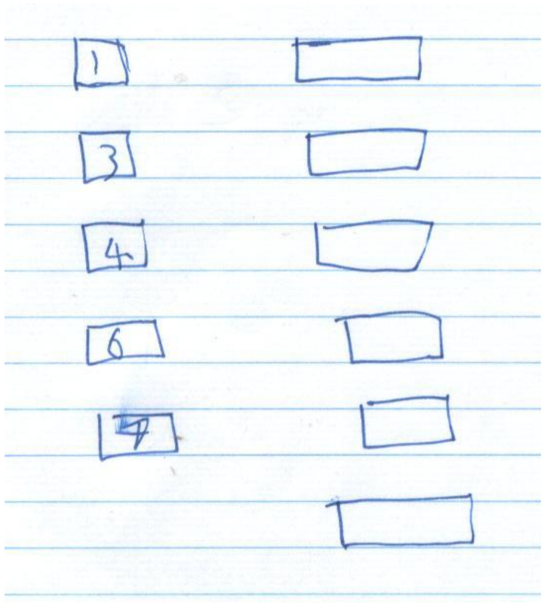
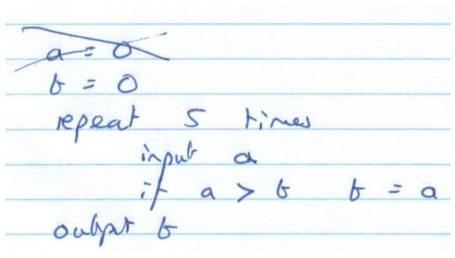


Figure 9.4 Visual approach

The pseudo-code in Figure 9.5 was then offered, with the question what would this do.



```
~~a = 0~~
b = 0
repeat 5 times
      input a
      if a > b   b = a
output b
```

Figure 9.5 Pseudo-code

Initially the first a=0 line is not crossed out.

> *That, it would repeat this five times so you'd have to input a, 5 times,*
>
> Yeah
>
> *But a's got a value of zero, so a wouldn't be greater than b they'd be the same and then*
>
> When we - that line, input a, what does it do?
>
> *It inputs zero, if a equals zero*
>
> OK OK let's change this program a bit, let's cross out that first assignment..

This shows the inappropriate perception of the text, state and I/O spaces. The normative view would be that the initial

a = 0

would be altered by the subsequent

input a

but Phillip sees the text space setting of 'a' to zero as requiring that the user must input the value of a, which is zero, so the value of a will always be zero. This therefore shows a mis-perception of the I/O space, and a dominance of the text space. Once the 'a=0' line is deleted, he says:

> *Would the user input the amount for a?*
>
> Yeah

Because there's nothing declared as a = , so if the user say was to input a and they thought a should be 3

Yeah

So Phillip does see that the user has a free choice, but this is over-written if the program text had previously assigned a value to it. But there is another issue - he thinks the user would decide a single value for a, and so input the same value five times:

It would repeat that 5 times, so first it would input a and then 3, if a is greater than b so it would so a is greater, because its 3, so then b would equal 3,

Yeah

Then it would do it again

Yes

and it would add another 3, so then a would be 6, then, if is that

Where is it doing the adding?

Oh it doesn't say to add it does it?

No

It just repeats the same thing 5 times

OK

So it would output b, which.. would it be 3, because you don't have to add any of them?

OK now suppose in the loop in the repeat 5 times, the user changes their mind..

In one sense this is logical, since in the fictive mental space the user is asked for a value of a, they have chosen that to be 3, and so if asked 5 times they will input 3 five times. However the normative view is that the user simply inputs 5 unrelated numbers - the 'user' cannot 'see' the program text and so does not know they are supplying a value for variable 'a' - but Phillip can see the program text and his interpretation of the user's hypothetical actions is understandable.

Once this is suggested, he can see the result immediately:

.. they put in different values for a, so it could be 3 to start with, but the next time they might choose a different number

It would output the largest one, out of all 5.

Interview Summary

1. His concept of internal data storage is influenced by ideas of GUI design in Visual Basic

2. His idea of the I/O frame is non-normative. He thinks user input is influenced by program text.

9.3.3 Interview Three

At this point Phillip had been working on a module concerned with OOP for about five weeks. The approach was based on Java in the Eclipse IDE. He was familiar with the use of primitive types, arithmetic, loops, and output with System.out.println, and that execution starts at main(). There was a discussion of 'public static void':

public?

Public is for the whole program - I think public is for the whole program - I think that's what we've been taught.

OK

Something like that anyway.

static?

Static, I don't know what that is, no. Does that mean it can't change, its unchangeable? Static, it has to stay that one thing all the time.

(brief discussion of static omitted)

void? Any idea about void?

You can't use it in other classes? I don't know.

So this shows

- he has a rather vague recall of the idea of access control by 'public'

- he is happy to guess meanings, in relation to static, and uses the ‘constant’ idea which many other students have done.

- he is not concerned that there are several ideas here with which he is not familiar

The interview then moved to the task of exchanging variable values. Phillip makes the classic error (x=y and y=x), but spots it immediately, and finds the correct solution with very little help:

> What we want to do here is to change their values, so you might have for example x=1 and y =9, and what we want to have after we've done it, is that x should be 9 and y should be 1, yeah? to exchange their values. So how would you do, what instructions would you use to exchange their values?
>
> *Well, couldn't you just write x=y, y=x, I think that might work. Would that work? I don't know if that would work.*
>
> OK let's trace through it, x holds a value in memory, y holds a value in memory, and we start off x is 1, y equals 9. Then
>
> *Oh if you done that, we'd lose the 1 wouldn't we? If we did x=y we'd lose the value of x so it wouldn't work.*
>
> OK so if you said x=y, we'd have a 9 in there, and we'd have lost the 1
>
> *Yeah*
>
> So they'd both be 9. OK. So how do we fix that problem?
>
> *Could you write a sub-routine for it? So you could store the numbers first, you could have the numbers stored, and then you could swap them round.*
>
> Yeah, but how would the subroutine work? How would it actually change their values?
>
> *It could - I don't quite know. It would have to store them first, and then it would have to recall them from the memory.*
>
> OK so do you need an extra storage location?
>
> *You would do I think yeah.*
>
> So if we call that t, t for temporary,

So you could do it, t=x, so it would be the 1 stored there, then you could do the x=y, so the 9 would move to the x, then it would be y=t, so the 1 moves from the t to the y.

OK. Have you ever seen that before?

Not really, no. I don't think so.

The task of reversing an array is then presented:

Suppose we wanted to reverse the array, to put that value into there and that value into there, to reflect them, backwards, how would you do that?

Well would you have to like make each box of the array have a variable, so this one would be called x, then each time you'd swap them, oh no cause you'd lose one.

Are you saying put the end one in the first one

But then you'd lose the first one

Yeah, OK

You'd have to write some sort of code to swap them at the same time, so you could move them across at the same time, so you could move that one and that one, both at once.

OK, right, is that possible in normal programming?

Yeah I think it is yeah

To do two things at the same time?

Well no its not is it?

No, in fancy kinds of programming it might be possible, but if you've only got one processor, it can only do one thing at a time. Is that similar to this problem (exchanging values) in some way?

It is, kind of, but with more boxes to fill out, more numbers

So we want to do them one pair at a time

So would it be the same sort of solution to that? You'd have a space, another temporary folder, to store one in

Is folder the correct word?

Well it wouldn't be would it it would be a temporary space, piece of memory

Piece of memory OK

> *And then if you wanted to move that one across you'd move the end one into the temporary, so it would be a bit like a circle, routine, cycle, you'd move that one across first, then the first to the last, then that there, then you'd move on to the middle, then you'd just keep running it in a sort of loop, you could have it in a loop.*

Interview summary

1. The exchange problem first elicits the classic error
2. The error is immediately spotted
3. The standard solution is found with very little help
4. The array reverse problem is treated as a reversal within the array
5. The array reverse problem is first seen as sharing the exchange problem difficulty (over-writing a value)
6. It is not seen as being solvable using the exchange solution
7. A solution using 2 simultaneous steps is suggested
8. With some prompting the exchange solution is used
9. The use of a loop is immediately suggested

9.3.4 Interview Four

At this point Phillip had completed the first term of a first module in Java. We started by establishing which topics he had encountered:

> Classes?
> *Yeah yeah yeah*
> Objects?
> *..Objects? We've done a bit of them, yeah.*
> OK yeah yeah yeah

> *Not much but we've done some.*
>
> OK classes objects, how about constructors?
>
> *Yeah yeah*
>
> Constructors, OK, how about methods?
>
> *Yeah, we've done a bit of that as well.*
>
> Yeah, methods, how about, you might call them fields, or data members, or instance variables,
>
> *No.*
>
> This is in an object, an object will have some methods, and it will have some values in there as well
>
> *Its got values and stuff, yes*
>
> OK so people use different names for those things, what would you call them?
>
> *Like a series of values?*
>
> Yeah
>
> *In an array?*
>
> Ah no not in an array as a.. no no an array is different.

So Phillip seems to have some familiarity with basic OOP ideas, but he does not recognise the idea of a data member. Later in the interview it becomes clear that this is optimistic, and in fact he has had little experience of classes and objects:

> OK here's a question - what is a class?
>
> *Well it’s a simple program, within a project, so you could link one class and information from that you could link it to another class, with all within one project.*
>
> OK yeah - have you come across the term 'package' in Java?
>
> *Yeah package you create the project then you add the package, and each package can have so many classes.*
>
> Right OK - what's the connection between objects and classes? What do you think?
>
> *Is the object the code that's actually inside the class? So for the class to work you need objects, the objects would need to be coded right for the class to work*
>
> OK so the object is..
>
> *What makes a class work*

and that's as a result of the code in it?

Yeah

The idea that a class is a program has been seen before, and derives from the fact that Java source code is organised into a separate file for each (public) class, so equating classes with 'programs' can be understood. But the idea that 'the' object is the code inside a class is very unusual. This (very non-normative) viewpoint sees a class as simply a unit of a program, and 'the object' is the code inside the class which makes it 'work'.

How can Phillip have this idea? Possibly because (as described later) the Java programming he has seen so far has been in a procedural style, with no objects instantiated. But he has heard the use of the terms class and object. He has therefore constructed for himself a meaning for the term 'object', and has decided (since he has no evidence for the contrary) that 'the object' is the code inside a class.

The interview proceeds by talking about String class objects. Phillip has just told us what he thinks 'class' and 'object' mean. Given that these are very non-normative, what is said next must make little sense to him, but through it his statements come to be more coherent. It is suggested that in this next section he is starting to develop better ideas of class and object:

If you declared a String variable, you'd usually say something like

String s = new String("Hello"); (this is written down)

for example. Yeah? You've seen code like that?

Yeah. I've seen that.

So what's happening in that statement?

Well its whenever the s is displayed in the code, it would display the word "Hello".. or if you if the user were to write Hello, the code would recognise it as the s, because that's what you've told the s to be.

This is a completely non-normative understanding, but it is reasonable given an unawareness of the meanings of class and object.

OK. Going through it a word at a time, we start off String, why do we need to say that?

Because that's how you would declare the variable, you could use a String or an int or a double

OK so that's a type

Variable type.

So Phillip sees

String s..

like the declaration

int x..

and so agrees that String is a type (informally understood). But this is in conflict with his understanding of a class as a program. This is pressure on him to re-construct his concept of class.

Now String starts with a capital letter - do you know why that is?

I don't know why that is. No.

And its the same here - new String and it has a capital letter. Not sure?

No.

OK that's because it's a class - String is a class, and conventionally classes start with a capital letter.

Mm.

So we know which are classes and which are not. Then it says s, that's the name of our variable, then it says new - what does new do?

Does that mean that it's a new class - its like a new String class, so like, it hasn't been used before? Is that right?

Now Phillip has indicated that he has 'done' constructors. But his answer to 'what does 'new' do?' consists of two questions. At this point he is trying to construct the concept of 'new', basing it on the everyday meaning of new, and its context here, associated with class. Because his concept of 'object' is completely wrong, he cannot answer 'new makes a new object' or 'new invokes the constructor'. He may have been told that, but since that conflicts with his current concept of 'object', it does not make sense.

> It's not a new class, it’s a new object.
>
> *A new object within the class. Is that right?*
>
> A new object within the class? .. nn yeah OK

Phillip is transitioning to a new concept of 'object'. The interviewer is struggling with this, and is unsure what 'within the class' might mean (and so ignores it):

> and new String("Hello").. how does it - what we're doing here is making a new object
>
> *Yes*
>
> What part is used to do that? What part of a class is used to make a new object?

The interviewer's use of the terms 'class' and 'object' conflict with Phillip's initial understanding. He thought 'objects' were the code inside a class - so how could 'part of a class' make a new object? To resolve the conflict he must alter his concept of object:

> *Its the String part isn't it?*
>
> OK but in general, OK we've gone through these ideas, Eclipse, classes, objects, constructors, methods, and the String class. Which one of those makes new objects?

There is a clear implication by the pattern of the discourse that the correct answer is one of the items listed. Given that we are looking for something to 'make new objects', together with their everyday meanings, the only possibility which makes sense is a constructor:

> *It would be the constructor would it?*
>
> The constructor? So what is happening here when we say new String("Hello") is that we are invoking the constructor of the String class, yes?
>
> *Yeah*
>
> So that constructs a new String object.

So this has presented Phillip with a way of re-constructing his concept of object to a larger coherent set of ideas - that an object is a thing like a variable, that objects are made by constructors, and that constructors are part of a class.

The distinction between primitive and reference types is then introduced - Phillip had not encountered reference types before. He is then given an example:

> Let's change this to String s1, then we said String s2 = s1 - how many objects have we got now?
>
> *Well, you've still got the one, which would be the Hello*
>
> OK
>
> *Because you've told s2 to be the same as that. You've got two objects, but they'd both be the same. Or have you just got the one? Ah you've just got the one, because you've told it not to make a new object.*
>
> Mm yes you've only got one object
>
> *and the s2 would be referencing to the first object, which would be Hello.*

This is remarkably fast progress. A few minutes before, Phillip has suggested objects are the code inside a class. He has now developed the idea of an object as a 'thing', picked up the idea of a reference, and has not been misled by the code:

```
String s1 = new String("Hello");
String s2 = s1;
```

into thinking we now have two objects.

After a short discussion of this code, diagrams are introduced. However the start of this reveals an interesting point:

> Have you ever seen diagrams drawn to help us see what's going on here?
>
> *No.*
>
> OK so one way you could think of it is to say OK we've got this Hello object - where is the Hello object?
>
> *The Hello object? It's within the String class. Its an object within the String*
>
> OK which part of the computer is it within?

Its in the memory

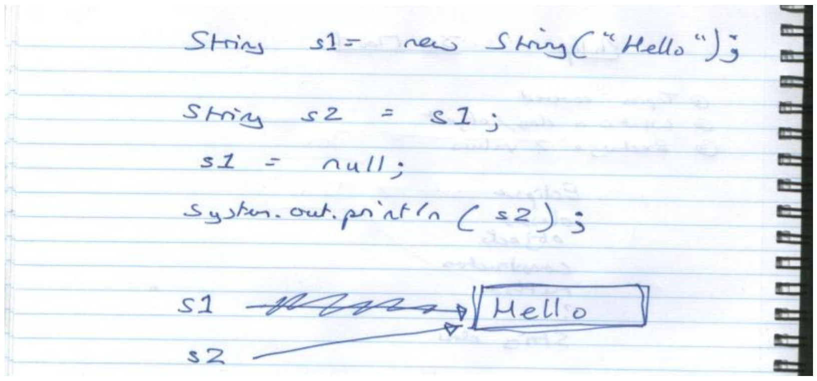
OK so this Hello object is in memory.

What does Phillip mean by 'within the String class'? The normative understanding would be to say the object *belongs* to the String class. This relates to the partonomy/taxonomy confusion. In other words two possibilites are

1. Phillip uses 'within' in the sense of belonging to a category. Then a class is seen as a type, and the object is 'in' that type
2. Phillip uses 'within' to mean belonging to a set. Then a class is seen as a container, a set of objects.

There may be other possibilities, and given that his concept of object is rapidly changing, the idea may have no precise nature. This is discussed further in the next chapter.

A diagram is then drawn and discussed - see Figure 9.6.



```
String s1 = new String("Hello");
String s2 = s1;
s1 = null;
System.out.println( s2 );
```

Figure 9.6 String exchange

The interview then moves on to the task of exchanging values

If you can remember the last time we met, we talked about changing over the values of two variables, exchanging two variables OK, now suppose we do that same kind of thing, but we do it with objects. What we were doing last time was using integers, which are primitive types, not references. Suppose we do it using references, so suppose we do it like this String s1 = new String("One"); and then we had another object, String s2 = new String("Two"); now suppose we wanted to exchange s1 and s2. How would we do it?

Well the easiest way would be to create another two strings, string 3 and 4, and then you'd get String 3, which would be s3
Do you want to write it down?
Yeah.. equals s1, so that would be stored into the s1, that would be stored into s3, string 3 would be 1, and you'd have.. then you'd have another string, s4, that equals s2, and then you'd just do String.. s1 equals s4, and string s2 equals s3 (see Figure 9.7)
So maybe I think I'm not sure whether this works or not
Yeah yeah - could draw one of those diagrams
Yeah? Draw one of these diagrams then:

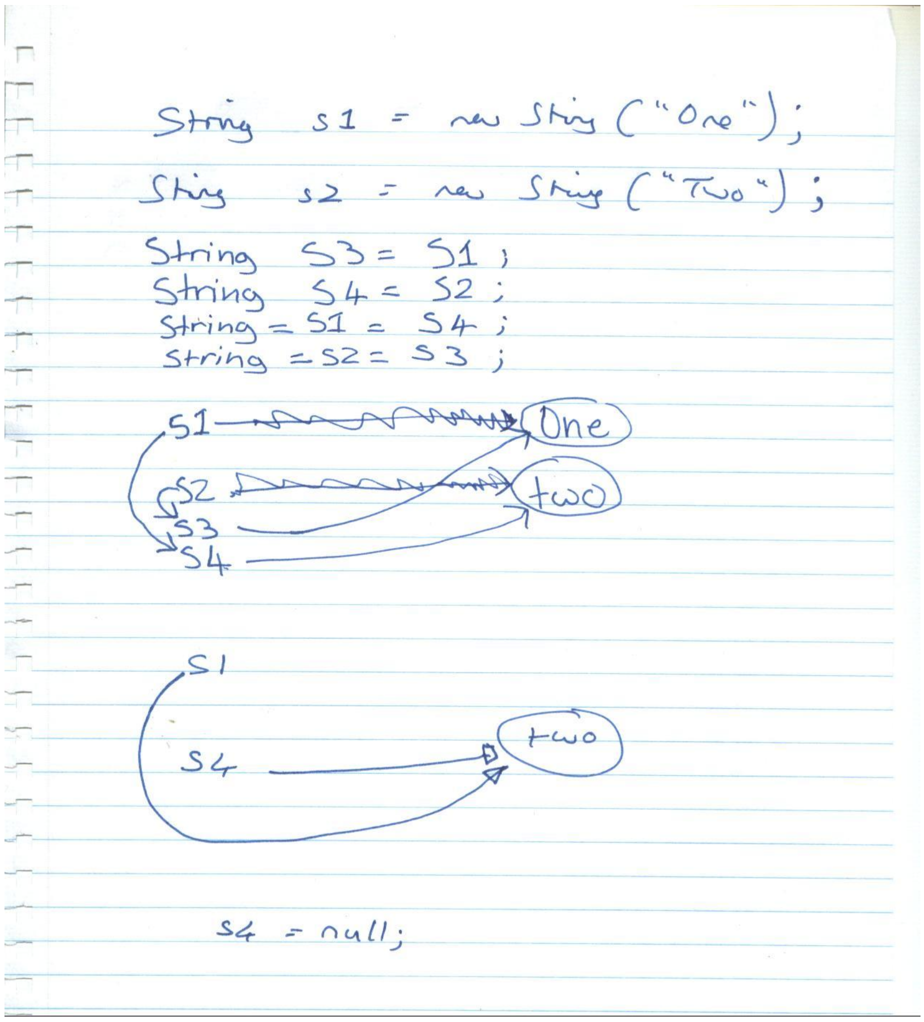


Figure 9.7 String exchange

Phillip talks about this as he draws it, then says

Oh you could have done it with just one string actually.

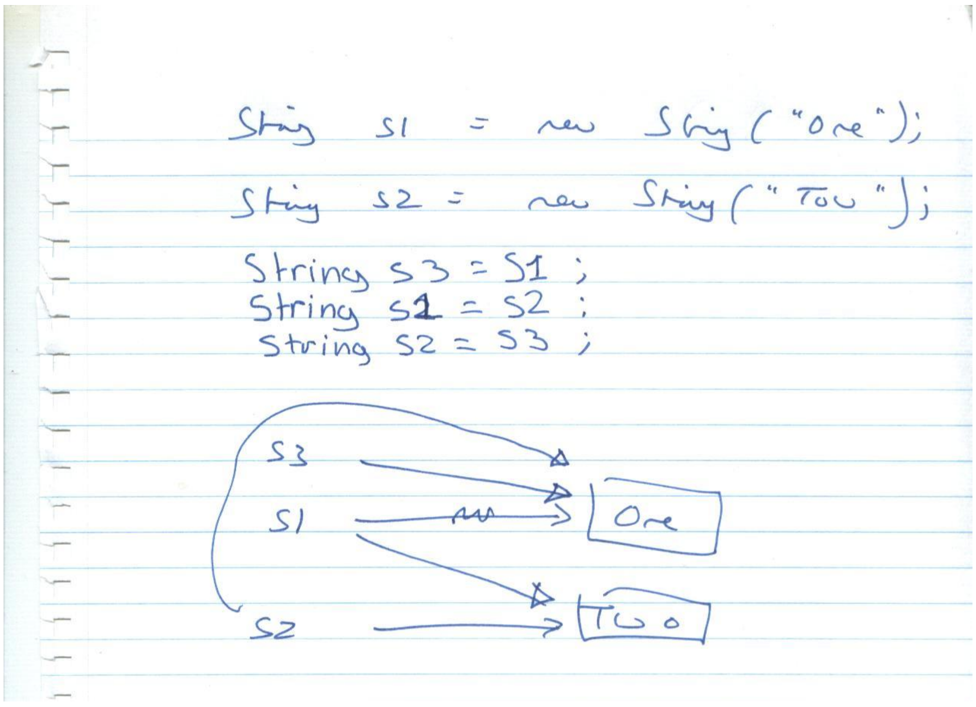
OK let's do it as you started, to start with.

Two points are interesting about this. Firstly, he has very rapidly grasped the way these diagrams are signifying what the code does, in contrast with other work (Thomas, Ratcliffe and Thomasson 2004) which has suggested that such diagrams were little help. Secondly, he makes the same mistake as Jennifer and Matthew (8.4.4 and 8.5.6) in that

```
s1 = s4;
```

makes s1 point to *s4*, when it should point to *what s4 points to*.

Phillip then suggests he can do this with just one extra variable, and writes this:



```
String s1 = new String("One");
String s2 = new String(" Two ");
String s3 = S1 ;
String s1 = S2 ;
String s2 = S3 ;
```

Figure 9.8 String exchange

Phillip then describes how the student class has been working on an assignment which inputs miles and gallons and calculates miles per gallon. He said that variablea for miles and gallons were declared as local to main(), rather than being instance variables (data members). Consequently his experience has been with procedural programming in Java, rather than any work using or defining classes and instantiating objects. This explains why at the start of the interview he did not recognise the term 'data member' or its synonyms.

Interview Summary

1. At this point he has been programming procedurally in Java and has little experience of classes and objects

2. His concept of 'class' was a program. His idea of 'object' was initially the code in the class, but this developed through the interview.

3. The term 'reference type' was new to him, but he quickly picked it up.

4. He had not experienced object diagrams before, but was immediately able to use them in the context of the variable value exchange task.

9.4 The Professor

To obtain an 'expert' view, I interviewed a Professor of Computer Science and asked him how he saw objects and classes:

> *Objects I see as machine representations of real-world objects and the way things are set up so objects will have some data which will represent the state of the real-world object, and they'll have some operations like the real-world objects have, the abilities, the capabilities that the objects can execute, um, so that's what an object is, and a class is a type of such things.*
>
> OK so a class is a type of such things, so
>
> *For example the table, the class, the class table is the type of some particular types of tables.*
>
> OK how would you think of it if you had a situation where the class was static, so there were no instantiated objects?
>
> *I think of that as a kludge (laughs). OK so classes in a real programming language are a mixture of different concepts I think. So classes in Java not only serve the purpose that I've mentioned, but they also serve the purpose of modules, so a class without any instances is really a module and we are really re-using the class concept to implement modules*
>
> Ah so for the Math class - why why is it a kludge?

Well they could have had a separate module concept which had nothing to do with object orientation

Ah so a kludge in that it could then be said that this is a pure object-oriented language, when in fact it isn't, because Math is like a module in C, for example. Is that right?

Mm

OK that's interesting. So, a class has several kinds of aspects to it, or several concepts put together, yeah, so you have type of object, and you're also having a module to kind of yeah define a namespace, would it be? So that you put the things together?

OK I mean from a programming point of view modules just give you a namespace, but from a system architecture point of view a module is some kind of collection of related things, which might have some state as well, most often its some capabilities put together for modularisation purposes.

OK

Now in object-oriented languages, since they already had a class concept, they started using that for modules as well. Now classes, to be fair, I think classes also had static methods and static fields, don't remember where they came from, but it would be natural to say that there are some aspects which are which really belong to the class itself rather than to individual objects, so that's where I think the idea of static fields came from, and then once you have classes with static fields you might also think of a class which only has static fields, and it would feel like a module, but conceptually its far from the original class concept.

..

and once you start doing sub-classing, sometimes classes look like they're just interfaces, you know you might just have a class with nothing defined in it at all, just placeholders

Yeah if its an abstract class?

Abstract class, yes. And so there again the separation between behavioural type and structural type you know is not a sharp division, once you start using abstract classes there is a continuum of how abstract a class is - an interface is a completely abstract class and a concrete class would be completely concrete and there would be lots of mixtures in between

..

Yes maths is a good parallel, so in maths when they define structures like a group or a field structure etc, is it a class that you're defining or is it an interface, or is it a type? Mathematicians I think don't really pause to think of that issue, but there the same issues come up. So a group, for instance, is really like an interface because it has something called a binary operation, and something called the unit, and so at that level it is like an interface, but then there is also but you might also think of the unit as a data object, as a datum, so in that sense it's also a bit more concrete, not just a type.

So this starts with an OOP object as being a model of a real-world object, and a class as a type of such objects. But this sits in a three dimensional view of the class/object concept. One axis ranges from theoretical notions to implementations in real programming languages. The implementations might involve kludges which alter aspects of the 'theoretical' concept.

A second axis is from behavioural to structural. A classic class models the behaviour of some real-world type of object, and at the other end of the spectrum, a purely static class performs the function of a module, supporting the purpose of dividing a large code base into smaller re-usable units which contain data and methods which have something in common. OOP already has the idea of encapsulation, implemented in Java by access control modifiers and accessor methods. Applying this to purely static classes means this will also work with code modules which do not really model objects - like the Math class.

The third axis is abstraction. This ranges from interfaces and purely abstract classes where we only have place-holders, through partly abstract classes, to completely concrete classes. An analogy is drawn to a group. A group might be defined without specifying what the elements and the binary operation of the group were, except for what a+b was for all a and b elements of the group. More concretely, we might give groups where the 'meanings' were specified, such as a symmetry or permutation group. Or we might be more abstract, by defining no more than what a mathematical group was.

9.5 Summary

This chapter has followed the progress of two more students, with weak academic backgrounds, and a Professor of Computer Science. It has shown the enormous range of ways of grasping OOP ideas. Aaron struggles to keep up, with a very weak grasp of basic ideas of program and variable, Phillip starts with very non-normative self-constructed ideas of class and object, but makes very fast progress. And the Professor works at the level of OOP principles, above that of the implementation of those ideas in an actual language such as Java.

Chapter Ten - Analysis

We shall not cease from exploration
And the end of all our exploring
Will be to arrive where we started
And know the place for the first time.

from Little Gidding b y T. S. Eliot

10.1 Introduction

Preceding chapters have presented appropriate theoretical perspectives and a large body of data about students' ideas about programming. This chapter analyses the data. It presents a set of propositions in brief, then goes on to expand on them and point out the supporting evidence.

10.2 A structured set of conclusions

Data presented in the previous chapters support the following five conclusions:

Language is typically non-literal, because concepts are grasped non-literally. This includes the language used in the programming expert community, so programming concepts are non-literal. In other words, the way in which the OOP concepts of *class, object, method* and so on are used by Java 'experts' is metaphorical and not literal.

Students construct their own versions of the ideas presented to them. This is not a profound remark challenging the nature of reality. It simply means that students try to deduce what the teacher is talking about, on the evidence of what they say.

The construction is based on signs - spoken, written, diagrams and so on. Some students have a limited window of awareness, meaning the construction is severely hampered.

Thinking about programs involves the use of fictive mental spaces. The mental spaces are structured by frames. There are 3 mental spaces relevant to novice programmers - program text, state, and I/O

OOP comprises a further hierarchy of spaces with structuring frames. The acquisition of these frames has various difficulties.

10.3 Language use is typically non-literal

For example

The IBM PC revolutionised computing

This easily-understood sentence with only 5 words or acronyms has three major non-literal aspects. Firstly, *the IBM PC* is a compression, in the sense of Fauconnier, of the millions of actual IBM PCs, into something which we think of as a single machine. Secondly, *revolve* literally refers to the process where a physical object rotates, and hence presents a different aspect. This is then a metaphor for a revolution as a quick and major change applicable to systems of thought or organisational structures such as governments. Thirdly *computing* is literally the action of using of a computer, but here it is again a compression of many items of hardware, software and patterns of use into a single item.

The evidence to support this contention that humans have a metaphorical understanding of many concepts is provided in many sources, such as Lakoff and Johnson (1983), where countless examples are given. Evidence that this is also true of the subset of human endeavour which we are concerned with here, namely computing, is given in Douce (2004). Five examples of the use of non-literal language in programming follow – ideas of a computer, a variable, the use of ‘belongs’, the loop compression, and the Listener metaphor.

10.3.1 The computer blend

The responses to 'what is a computer?' and 'how do computers work?' make sense if we suppose that our culture conceptualises a computer as a blend of person and machine, as in figure 10.1.

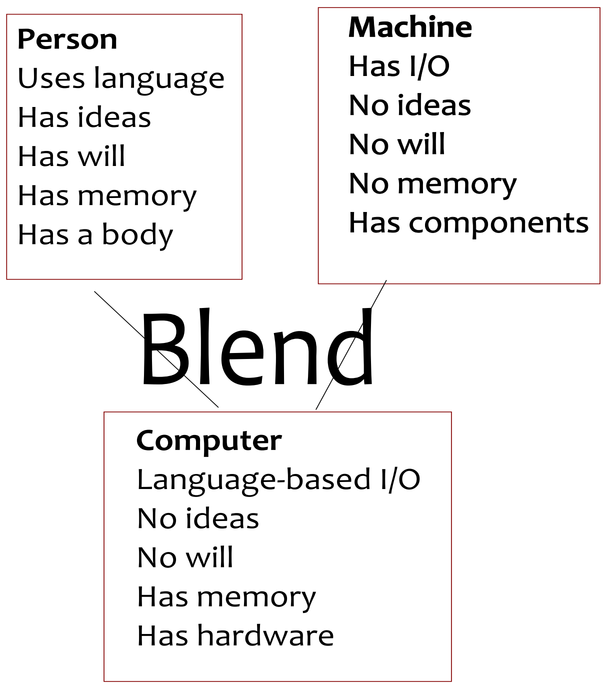


Figure 10.1 The computer blend

Our culture is familiar with machines, like clocks, lathes and ovens, and we all interact with people all the time. Both people and machines have attributes which can be related, as above. For some attributes, either the person form or the machine form carries over into the computer, while for some there is something which modifies both. A clock has input (such as time adjustment and alarm setting) and output (the displayed time). A person uses language. In the blend, a computer has language-based input output. However the person understands ideas, as expressed in language, but the machine has no such grasp of ideas. This leads to the notion that the computer can understand English but not understand ideas:

> (F0A) So does the computer understand what we are telling it to do?

Mm

It does understand?

Yes

Is it conscious?

No

No, its not conscious. Does it understand English?

Ha! Yes, of course! Yes! Well, the computer itself doesn't understand, like a general idea, but if we put software in, it will understand.

In the past our culture has used a corresponding blend between a person and a machine, into a slave or servant. The language we use with reference to computers shows this. For example, a computer program is made of "instructions". But not literally. An instruction is what one person gives to another person - only when carried over into the blend do we speak of a metaphorical program instruction. Similarly a computer user will issue "commands" to the operating system such as to save a file. But literally a command is what a one person gives to another person in a servant or slave role. For example, programming is seen as:

(F1A) *To write the programs, its to tell the computer what to do, its not something vaguely*

Yes

You can tell them what to do, what not to do

Small interpreted computer programs are often known as "scripts". A script was originally anything written, but the specialisation relevant here is a script for an actor telling them what to say and do.

Again, computers have memory. But they do not remember. Computers have a functionality called "memory" which is only partly analogous to human memory.

This blend is held by our culture, but the weighting of that blend varies between individuals, which is why different subjects give different responses. For example

> (M2F) What do you think a computer is?
>
> *A computer, um, I think a computer in today's terms is a device that computes signals and forms an output based on inputs*

shows a blend which is heavily weighted on the machine side.

The language we use in connection with computers is based on this blend. We have already seen textbook exanples of this (Horstmann 2005 page 67).

10.3.2 The variable blend

The ‘variable’ concept, as a blend of the two input spaces of mathematics and hardware, was outlined from a theoretical perspective in section 4.3.1.

Attributes from each input space (mathematical variable and memory location) combine in various ways to form the parts of the output space (program variable). Mathematical variables have names (x, y, theta) and memory locations have addresses. In most systems a 'symbol table' relates names to locations. In the output space we refer to the variable name and usually ignore the memory location.

Mathematical variables have a domain (such as the natural numbers or the rationals) while memory locations have a representation format, which might be binary-coded decimal, IEEE floating point, two's complement signed integer and so on. This format specifies how to translate a given bit pattern into an appropriate value. These attributes (domain and format) are related, since some are compatible (such as the signed integers to two's complement) and some are not (rational numbers to ASCII). Domain and format map in the output space to the programming notion of 'type'. This can be thought of in many ways - novice programmers think of type as integer or string or float, whereas Stroustrup (1997 page 223), the creator of C++ says 'a type is a concrete representation of a concept', which is a far more abstract notion, and is correct, since for example an OOP class can be seen as a type. This shows the complexity of the domain-representation-type relationship.

Mathematical variables sometimes have a value, and this corresponds to the memory location contents, which will be a set of bits. These map to the value of a program variable.

This analysis is an attempt to show how the community of 'expert variable users' conceptualises the notion of a program variable. Novice students with limited experience of variables in mathematics or programming just show a few features of this complex blend, and the evidence for this has been presented in the previous chapters, and in particular section 6.3. Further, in 9.2.3 we saw Aaron showing that he did not have the concept of variable, as distinct from value.

10.3.3 The metaphor 'belongs'

In 8.5.8 there was a discussion of the word 'belong', with three aspects of meaning - association, type and set membership. For example a static method belongs to a class, and a non-static method belongs to an object, in the first sense. An object belongs to a class, in the second sense. And an object does *not* belong to a class, in the third sense, in that a class is not a *set* of objects, although novices sometimes think of it that way.

Sometimes a class *is* designed to hold all its instances, such as this (an explanation follows):

```
class MyClass // define a class called MyClass
{
```

```
static Vector<MyClass> objects = new Vector<MyClass>();
..
public MyClass // the constructor, to make a new object
{
..
objects.add(this); // put the new object into the
                    //collection
}
}
```

A Vector is like an array, except that it can grow and shrink at runtime. MyClass has a static data member called 'objects', which will contain all the objects which are instances of MyClass. This is because in the constructor, we add the new object into it. So MyClass.objects contains all the instances of the class.

But this is unusual - a class is always a *type* and rarely a *container*.

At the heart of this is the use of the word 'belong', without definition, which risks students constructing incorrect ideas of class and object. In normal use, the word 'belong' is used in relation to mutually-understood items, such as

My heart belongs to Glasgow

where it is used in the sense of association, and 'heart' is used as a metonymic metaphor for 'me'. But for a novice programmer, 'class' and 'object' are not well understood, and the use of the word 'belong' may be mis-interpreted.

10.3.4 The loop compression

It is suggested that statements in a loop are handled cognitively as compressions of the statements in the unrolled loop (Milner 2008). For example, consider

```
x = 0
input n
```

```
while n is not equal to -99
{
if n > x then x = n
input n
}
output x
```

and the response of a student struggling with this:

(M0C) Because when the 7 goes in at the end, 7 is greater than 0, then input the next number, which is 8, x becomes, sorry x becomes 7, x takes the value of n, 8 is greater than x, so x takes the value of 8, then n takes the value of 4, 4 is not greater than 8, so x does not take the value of n..

There are two differences between the code and this - we have numbers, not variable names, and *we have no loop, because it has been 'unrolled' into a linear sequence*. To an experienced programmer this is so obvious as to be hard to see, and also pointless, since it is possible to get much more powerful conclusions from the rolled up loop than from the unrolled version. However the student struggles to handle it.

This unrolling consists of transforming our given code into

```
x=0
input n
check loop end
if n..
input n
check loop end
if n..
input n
check loop end
if n..
input n
..
output x
```

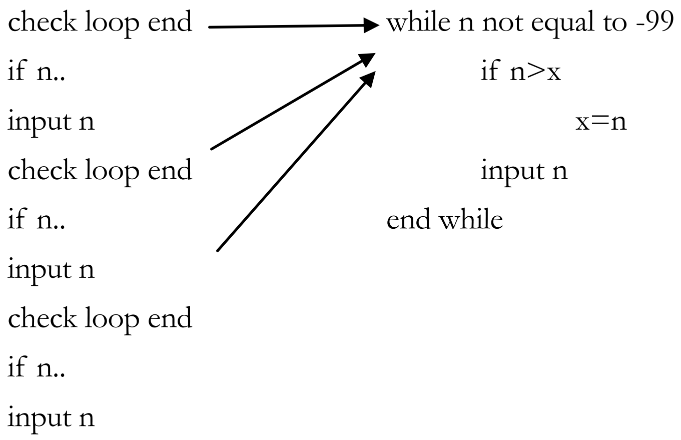
For each traced run of the program, the student must generate this sequence from the input, and work out what happens (that is, does it stop and when) and what the output is. Doing this once yields nothing much ('it outputs x'). However doing it several times provides a set of experiences which the student can then compress into a single item, so that 'what does it do?' is a meaningful question.

The student is not just doing inductive reasoning here - that is, he is not saying 'the first time it found the largest, and the second time, and the third, so it will always find the largest.' This is because the student can also provide an explanation of the mechanism involved in the program:

> *(M0C) It would increase normally here, but then when n is 97, its greater than x which is the previous value, which is 23, then none of the others are greater than that.*

At this point, the student has been able to generalise the action, and is seeing the general in the particular example.

As shown above, the novice unrolls the loop in this way for each traced execution run. Doing this one or more times provides the required experience to roll up the loop again, where each of the multiple executions are compressed back to the orginal loop:



```
check loop end          while n not equal to -99
if n..                          if n>x
input n                                 x=n
check loop end                  input n
if n..                  end while
input n
check loop end
if n..
input n
```

Figure 10.2 The loop rolled up

What does a single execution of the if statement signify? When the if is done once, what can anyone deduce? Nothing. But when it is seen as applying to the *compressed* n and x, the student can see that it means that x will be the maximum of all previous inputted values. This vision is what happens when the novice comes to know what the program does.

This can be related to the notion of a loop invariant. It is always true for this loop that x will be the maximum of all previous input values (provided they are greater than 0), and consequently this must also be true when the loop terminates. This is in the domain of logic, and we have here a true proposition. This is very useful if we are concerned with the proof of program correctness - but we are not, and our concern is instead with how a novice conceptualises things like this program fragment. New conceptualisations are not like propositions - as shown by the students' words above, they are at first impossible to handle, then there is some uncertain grasp which comes and goes, then it becomes 'easy', then the conceptualisation becomes so familiar that it again becomes hard to see.

10.3.5 The listener metaphor

Another example of difficulties relating to metaphor is that of the 'listener' interface. Understanding this requires some knowledge of how this is used in Java.

A typical task is to put a button on a window, and have something happen when the user clicks the button. This works as follows. The window would normally be an instance of a standard class called JFrame. There is also a standard class called JButton which models a user-interface button. We make one by, for example,

```
JButton button = new JButton("Click me");
```

This constructs a button, in memory, with the label 'Click me'. We make a window show the button by saying

```
add(button);
```

in the constructor for the window. We prepare to deal with the clicks by saying

```
button.addActionListener(this);
```

This means the window will listen for clicks on the button. To make this work, the window must have a method named actionPerformed. The code would look like

```
public void actionPerformed(ActionEvent e)
{
..// this code is executed when the button
..// is clicked
}
```

Now when the user clicks on the button, the method called actionPerformed is executed, and this is programmed to do whatever is required. This is usually called an *event-handler*.

This is clearly metaphorical - the window has no ears, and does not really listen. However this is how the community of Java experts think of it, and so it is how students are taught.

For example:

> Can you remember anything about ActionListener?
> *Well it um it um like takes note of what the user, how the user interacts with the program, and it does certain things based on what how they interact with it, like through um mouse clicks and things*

The bizarre point about this (Milner 2009) is that it is a broken metaphor - nothing listens to anything, even metaphorically. To listen would require the window to have some code to do the listening, all the time, even when the Java program is doing something else - which is impossible (without multi-threading).

In fact, the *window* does not listen. The *button* maintains a list of objects which need to be notified when it is clicked. The call to addActionListener adds the window to this list. When the user clicks the button, the button calls the actionPerformed method of the objects on that list - in our case, the window.

So the community understands this metaphorically, even though the mechanism is not even metaphorically true.

10.4 Students construct their own versions of the ideas presented to them

This relates to the school of thought referred to as constructivism, which some regard as controversial since in its radical form it challenges conventional notions of the nature of reality. But what is being asserted here is nothing so profound - it is simply that students meeting new concepts in a formal educational setting try to deduce how the teacher (or textbook or other source) is conceiving the concept.

This process is required of the student because of the previous point - that language is *not* used literally, so the student is obliged to construct a metaphor which matches with the way the teacher appears to use the metaphor.

For example, consider the term 'class'. We have seen that people are aware that this term is polysemic (5.3). They have learned that in a formal educational setting, familiar words (like 'class') are often used to denote concepts peculiar to that domain, and they are obliged to deduce the concept that the word is being used for. Sometimes the constructed version is largely identical to the version held by the teacher and the wider 'expert' community. Sometimes it is not - clearly an unfortunate situation, which is made worse because it leads the student to make incorrect deductions about consequent concepts.

In the case of Java, many examples of this have been seen. Consider the case of *static*. Almost all the students interviewed here had problems with *static* at least once. In 5.2.7 static is seen as ‘cannot be changed’. In 8.4.5 Jennifer used this idea, even though she had previously showed the normative meaning (in 8.4.2). In 8.5.7 Matthew also said that *static* meant this, and his grasp of static is still unsure in 8.5.8. A student who has the normative meaning (5.2.7):

> Static belongs to class the method belongs to class ..belongs to class not to to individual objects

uses the term ‘belong’, aspects of the meaning of which have already been discussed.

How can this be analysed? The keyword *static* is seen by students in their first Java program, since *main* is *static*. One pedagogic approach to this is to say ‘ignore it for now – you’ll understand it later’. This is used in 4.2.4. But students in fact do not ignore it, and instead construct their own version, which might be based on the everyday meaning of static, denoting fixed in some sense. However the picture is not simple – in 8.3.1 Fiona confuses *static* with *abstract*, since they both contain a ‘no object’ aspect. Nevertheless Fiona is constructing her own version of the concept.

A second example concerns the central OOP concepts of class and object. In 5.2.6 the student, when asked how many objects are present when a program runs, says

> what the definition of an object is.... um I mean I suppose in my mind I when I think of an object I'm thinking of some thing and the only things that is a thing..

This implies he has no OOP-specific object concept, and is using the everyday notion of object as thing. In the same section another student takes data members to be objects. Because the everyday notion of object, as thing, is so very general, the student is liable to construct their own concept in many different ways – here, for example, a data member, which is a category of information, is seen as a thing.

In 8.5.9 Matthew provides two examples in one interview – he guesses that in OOP overload is the same as overflow, and that over-riding might be the same as over-writing (the previous contents of a memory cell with a new value). Both show the student seeking to construct a new concept on the basis of what has been seen before. Later in the same section, the student constructs another (this time normative) concept of over-riding, on the evidence of what the interviewer is saying.

10.5 The construction is based on the perception of symbolic content

Symbolic content is material such as lecture content, diagrams, discussions, textbook content, program texts and so on. The student is obliged to perceive the meaning of this material, to ‘grasp’ its significance.

Content could be placed on two dimensions, that of scale and generality:

		Generality	
		Particular	General
Scale	small	Single program statement	Type, variable, loop, conditional etc
		Fragment	Algorithm
		Application	System modelling
	large	Java concepts	OOP and other paradigms

Most of the material in Chapter Seven, and elsewhere where ‘what does this program do’ is asked, is referencing program fragments. Much of the material in Chapter Eight (such as ‘what does *abstract* mean?’) concerns Java concepts, which are large scale, but are particular to the Java language. In Chapter Nine we saw the Professor considering programming language concepts in a variety of paradigms, with Java being just an attempt to implement one of them.

Two instances follow. The first is about understanding a simple program fragment, and the problem is the student’s limited window of awareness. The second is about assignments with primitive and reference types, and is a consequence of using language in a non-Gricean way.

10.5.1 Understanding fragments

A student trying to understand program code must be able to cope with it at a variety of levels. For example, a =a + 1 must be understood at the single execution level of doing something like changing a from 0 to 1. If it is in a loop, it must be seen as a compression, such that it will increment the variable a. And if it is in a loop, the role of that incrementation must be comprehended in the loop. Those multiple levels of comprehension can only be achieved by having the ability to be aware of several things at the same time, together with the ability to compress several items into a single one.

As an example look at a student trying to deal with:

```
a = 0
```

```
b = 0
input c
while a not equal to c
     b = b + a
     a = a + 1
output b
```

Go around, well, while a is not equal to c, 1 is not equal to 5, so we carry on, b=b+a, so b becomes how big?

Er b equals zero, no no sorry

b is zero, a is 1, so b+a equals?

5 no sorry 1

OK so b becomes 1, because it says b=b+a

Yeah

So its 1 OK? Then a=a+1,

It should be 1, it should be 2

a becomes 2, it is 1, then we add 1 to it so it becomes 2

Because its adding in numbers like

The values of the variables are being written down as they are being established, but the student still struggles to see what b=b+a and a=a+1 will do for *a single execution*. In the case of b=b+a, the student must recall the current values of b and a, add them, and see that b acquires this new value. This would seem to be straight-forward, but the transcription for this student, who was typical, shows that it is not. But this is just the first level of comprehension required. The second is to see how this functions in relation to the loop, which as described in the previous section requires an understanding that this is a compression. Only then is it possible to see that

```
while a not equal to c
..
a=a+1
..
```

means the loop will iterate c times. But this student can only be aware of a small part of this. Consequently he cannot start to comprehend what the program will do.

10.5.2 Primitive and reference type assignments

While loops and conditionals are clearly not trivial, students can construct inappropriate ideas even about simple assignments like

z = x + y;

Three problems have been seen. The first (6.2.3) is the idea that assignments like this establish a functional relationship between z and x and y, such that if x or y change *at any time*, then z will also. This is not unreasonable, since in effect this is what happens in a spreadsheet - it just happens that languages like Java do not work like this.

The second problem relates to primitive and reference types. We have three students (8.4.4, 8.5.6 and 9.3.3) who make the same mistake. In

```
String s1 = new String("One);
String s2 = new String("Two");
String s3;
s3=s1;
```

all three strudents draw object diagrams which show that s3=s1 makes s3 point to the *variable* s1, rather than pointing to *what s1 points to*. This shows a conflation between a variable and the value of the variable.

In 'Alice in Wonderland', Lewis Carrol has the White Knight distinguish between what the name of the song is called (Haddock's Eyes), what its name is (The Aged Aged Man), what the song is called (Ways and Means), and what the song is (A-sitting On A Gate).

In the case of Java code, we have the difficulty that we cannot distinguish between the variable and the name of it, since we can only refer to it by means of its name. And when we do refer to it, we usually mean the value of it, not the variable 'itself'. So

.. = s1;

in Java means (the value of) (the value named) s1;

and

s3=..

means change (the value of) (the variable named) s3 to the right hand side.

Why do these students make the same mistake? The White Knight talks like a logician, whereas Alice talks like a 'normal' person, and consequently uses Gricean language. To maximize efficiency, when we use the name of something, there is an assumption that we are talking about the *something*, not the *name*. In the rare situations where we are talking about the name, that often has to be stated explicitly.

In Java this is also true for primitive types. So when we say for example

```
int x;
x=3;
```

we typically think of x as *being* an integer variable, rather than being the *name* of an integer variable. In fact it does not matter. If we say

```
x=y;
```

we are not saying that x is now the name of a different variable. It still names the same variable, but its value has changed. The previous '3' value does not persist.

But for reference types this is not true. If we say

```
String s1;
```

then we are declaring that s1 is a name for an object of type String - but it does not actually name anything. This is different from normal Gricean language, where you very rarely have names divorced from what the names denotes.

If we go on to

```
s1 = new String("Hello");
```

then s1 might be thought of now as the name of that String object. But that is not a wise way of thinking, since we can say

```
String s2 = new String("Goodbye");
s1=s2;
```

and now the name s1 is the name of a different thing. This is like stopping using Paris to be the name of the capital of France, and instead using it to name a small Welsh mountain.

So Java variables of primitive type have names which can be thought of as being the variable, but this is not true of reference types. This is why the three students got it wrong in the same way.

The third problem is the same as the second but the other way round. Aaron (9.2.3) can get it wrong by applying correct reference type reasoning to primitives. This was when we were trying to exchange data values. We had

```
int x = 4;
int y = 5;
x = y;
```

So that x=y - what does it actually make happen?

Stop getting the answer from there and get it from there.

He sees two memory locations, one containing 4 and the other 5. Initially x labels the 4 and y labels the 5. But the x=y is treated as if they were references, so the values 4 and 5 persist, and x is now a reference to the 5.

10.6 Thinking about programs involves the use of fictive mental spaces

Thinking about computer programs (and many other areas of 'scientific' thought) involves the use of *fictive mental spaces*. For example, we must consider what happens *if* this program runs, or *if* a particular data value is input. Someone familiar with the mental space uses a frame to structure it - in other words the frame provides names for the expected parts of the mental space, and provides roles for what those parts are expected to do. These frames do not have an objective, independent reality – they are personal to each individual student. The elements of the frame – the concepts of programming – are constructed by the student as described previously. There is no ‘real’ or ‘correct’ frame. But there is a frame (which is a complex hierarchical blend) which is shared by the expert programming community. The task of 'learning Java' involves constructing frames which are congruent with the frames the expert community uses.

The frames comprise a hierarchy, the basic ones being so obvious they are usually not explicit, and the highest ones going above actual programming language implementations. There are three basic mental spaces with associated structuring frames - the actual program text, the state space which the computer moves through as execution proceeds, and the I/O space of input data.

10.6.1 The program text and execution spaces

The evidence to support this is mostly in Chapter Seven, and depends on the relationship between program text and execution. There was only one ‘program’ in Chapter Seven which all 15 students could simply read and say what it would do, namely (‘program one’):

```
a=4
b=5
c=a+b
output c
```

For the similar ‘program three’

```
input a
input b
c=a+b
output c
```

12 out of 15 could read the text and say what it would do. For the three other programs, on only 3 out of 45 times could a student do this. In other words, there is a basic difference between the way these two are dealt with, compared to the rest. This difference is that for these, there is a one-to-one mapping between program lines and execution states. Program one has five lines, and execution rolls through five states. That means that reading the program text and following execution is the same process. By contrast, programs with loops or conditionals do not have this one-to-one mapping. For example in program two:

```
a = 0
b = 0
while a not equal to 4
     b = b + a
     a = a + 1
output b
```

the line ‘b=b+a’ in present in the text once, but is executed four times. This is obvious, if you are aware of the execution frame. But only two out of 15 students even attempted to carry out a mental trace of what would happen, and 12 could not understand even after being prompted to carry out a trace on paper. We can explain this if we accept the fact that those students were not aware of the execution frame.

Similarly for

```
a = 0
```

```
repeat 5 times
     input b
     if b>a then a = b
output a
```

whether the assignment a=b, always present in the text, is executed, obviously depends on the values of a and b, so again there is not a one-to-one mapping between program text and execution states, and the student must explicitly think about the execution space. Only two out of 15 students could read the text and say what it would do.

It might be said that this is simply showing that some programs are 'harder' than others. It does show that, but it also provides insight into why that is the case. It could also be said that these students on a vocational course are just less 'able'. That ignores the differentiation – nearly all the students could deal with two of the programs, and very few could deal with the rest. It is also true that some academic students (6.2) had problems tracing similar programs.

10.6.2 The I/O frame

This is not trivial. For example, (9.3.2) we saw that:

```
a=0
repeat 5 times
  input a
```

led the student to think that a value of 0 had to be input 5 times. Or (7.5) the student who could only understand:

```
input a
input b
c = a + b
output c
```

when actual input values were suggested. Or the student (6.3) who sees user input in terms of playing a game:

And then depending upon what the user was doing it could be in a game etc. and then the moves of the user would determine the values of n, and then it could be that the output was something for the computer to do.

These students do not have the frame which structures the I/O fictive mental space. They do not see that we must imagine what would happen if the 'user' (or other source of input data) enters any data values, valid or invalid, typical or extreme. A full picture of this involves ideas about validation and program testing, and that is obviously non-trivial, but for these students with effectively no I/O frame they cannot start to consider 'what will this program do?' except for programs with no input.

There is some correspondence between this and the idea of a generalized number, as in $f(x) = x^2$, where x is used metonymically to mean 'any' real number - a compression. This is a simple idea if you grasp it. If not, you might respond to this by trying to solve it to find the value of x.

10.6.3 Features of the blended space

With some experience, students progress to thinking about a blended space which has program text, execution space and I/O as input spaces. Spaces are structured by frames. What are the frames of this blend?

Firstly, program text is executed by default from the ‘top’ down. This is arbitrary, and derives from the metaphor of program text as normal prose, which is read in sequence from the top of the page. It is also symbolic – when program text is in memory, in whatever form (source code, intermediate form or native machine code), there is no longer any ‘top’.

Secondly, execution sequence is modified from the default by features such as loops and conditionals (or even GOTOs). On a larger scale, execution sequence is modified by structures like function and procedure calls. Looking ahead to Java code, this produces a major departure from the top-down sequence, in that Java code consists of *class definitions*, and there is no significance to the sequence of the parts of this. In other words it makes no difference what order data members or methods are declared, only the sequence *within* a method body. Further, the use of inheritance means for example a class might have methods which are not visible in the textual definition of the class, but are inherited from a super-class defined elsewhere.

A third feature of this blend is the assumption that values of variables are changed by execution of the code, and by nothing else. For example, in

```
b=5
c=a+b
```

the value of b when the second line executes is still 5. In the multi-threading frame this is not always true. A different thread might execute a statement at a time between the first and second lines here, and that statement might alter the value of b. The point of this is to show that this familiar blend has features which are so implicit and 'obvious' that they are rarely explicitly stated, yet in other frames they cease to be true.

10.7 OOP Frames

What are the frames which the Java expert community uses, versions of which students must construct? There seem to be three – *object*, *class* and *inheritance hierarchy*.

An *object* is a computer model of a particular thing. An *object* has localised state. This relates to the idea of the computer state, in that the values of the *data members* of an *object* correspond to the values of the variables in a non-OOP program. But it is localised in that the state applies only to one particular *object*, and other objects will be in different states. By contrast, in non-OOP the whole system has just one state. As well as data members, objects have methods which code what the object can do, and constructors which specify how an object is made.

While an *object* is a model of a thing, a *class* is a *type* of thing.

The *inheritance hierarchy* supports the idea that real world types of things are hierarchical – a rectangle is a type of trapezium, which is a type of quadrilateral, which is a type of polygon. In OOP a *class* which is a derived version of another *class* is a *sub-class*.

Inheritance means that *base-class* features are also possessed by the *sub-class*, while *polymorphism* means that in the *sub-class*, inherited features can be changed or new features introduced. This provides a rich set of ideas to model type hierarchies, and leads to the practical convenience of code re-use.

We can logically have *objects* without *classes*, since this is possible in JavaScript, where *objects* can be set up as *object literals*, containing property-value pairs and functionality. Here *objects* are set up individually, and there is no way to indicate that two *objects* share the same structure. But this is not possible in Java, where the *class* is paramount, and an *object* can only be constructed as an instance of a *class*.

This therefore is one reason why some students find this difficult, in that they must acquire the *class* and *object* frames as a dual, if they are to read or write Java applications. Section 8.5 above provides a detailed portrait of Matthew as a student who spends many months studying Java without being able to answer 'What is an *object*?'.

Again we could logically have the class-object dual without an *inheritance hierarchy*. But in actual Java we cannot, since all *classes* are at least a subclass of the Object *class*, and in practice most *classes* sit several levels below Object. In the early stages students can be left ignorant of this, but in many courses *inheritance* and the keyword *extends* is introduced within the first year.

The introduction of inheritance leads to a complex three-dimensional picture. Along one dimension there is the sequence of class hierarchy, starting with Object. A second dimension is that of instantiation, with objects being instances of classes. Thirdly we have the structure of each object, containing data members and fields.

This is therefore a mixture of partonomy-taxonomy as described by Tversky (1989). A *data member* is part-of an *object*, and a *class* is a type-of *object*. But the situation is different from classification and composition structures in other situations. For example in Java we can instantiate classes high up in the hierarchy, such as

```
Object obj = new Object();
```

But in the natural world we cannot have say a mammal, in a general sense, unless it is also a creature belonging to a particular species. Java can model this, if a *class* is declared as being *abstract*. A second difficulty is that often an *object* will be part-of a data structure, such as an array, and so there is the issue of 'belongs', already discussed, so that the *object* belongs to the array and belongs to a *class*, but in different senses.

Chapter Eleven - Reflections

11.1 The journey

The previous chapter has analysed the data collected, presented a structured set of implications, and shown how these are supported by the data. This chapter is a set of reflections.

This thesis constitutes a long journey. It starts in the classroom, with a naïve attitude to the process of learning and the nature of understanding –namely that it is based on literal language. It travels through philosophical, psychological and linguistic domains to obtain some insights into what 'understand' means. It works its passage through five sets of data collection, talking to students, listening and thinking about what they say. At the end of the journey there is a new insight into the nature of learning, and a return to the classroom.

11.2 Aspects of Java

The first chapter considered some aspects of Java which appeared to be relevant to the research questions. This section re-visits these in the light of the analysed data.

The broad idea that some students find Java and OOP difficult is confirmed. Jennifer is an example of a student who can construct OOP concepts with little difficulty. But more typically, Matthew is seen to be an intelligent and resourceful student, yet he shows his struggle to grasp the OOP paradigm, finding it hard to answer the repeated question 'what is an object' at any point during the two semesters.

The fact that Java was designed to be a language which would be learnt by professional programmers accustomed to concepts such as variable and loop and the syntax of C can now be seen as an aspect of the 'objects first' question. This issue, whether students should be first exposed to ideas of object and class, or alternatively to structured programming notions of variables of primitive type, conditionals and loops, follows from this. The professionals for whom Java was designed were already familiar with structured programming; our novice students are not, and so this raises the question of the sequencing problem of introducing those ideas, when the central task is dealing with the OOP paradigm.

In fact the data collected here shows the image of learning as a simple sequence of the acquistion of discrete concept ‘packets’ is naïve and does not correspond to what actual students say or do. We have seen students who start by expecting to put to one side their previous ideas about what words mean and how they connect to ideas, and think a teaching course will enable them to construct new concepts. We have also seen that if this fails, during a course students will revert to ‘met-befores’. And they will also construct normative ideas, loose them, and re-gain them. This means that sequencing is not crucial, since constructed concepts are being continually re-visited and developed further in an organic manner. What is more important is how the material is presented, since it is on the basis of that experience than the construction takes place.

This also partly explains why the tight integration of OOP ideas, and the fact that they have to be present in the simplest Java program, is in fact not a major problem. The keyword phrase ‘public static void..’, which must be encountered in the first program, is treated by most students simply as ‘something I’ll learn later’, and confirms their expectation that they will be introduced to new ways of thinking – and correspondingly, new meanings for words.

However this is only a partial mitigation, since the student must subsequently deal with the problem that OOP ideas are not decomposable, and he or she must deal with the concepts of object, class and member at the same time. This is clearly one reason why some students find OOP difficult.

The fact that Java programs typically ‘say what they do’ in fact is no more than something that researchers and teachers need to deal with. For example, a student can easily guess that a program fragment which says ‘ .. = new ..’ is making a new object. To counter this superficial approach, the teacher and researcher need to take this slowly and to some depth, relating the keyword ‘new’ to what an object is, what making one would involve, and how this relates to the idea of constructors.

The layers of metaphor and blends which comprise Java, OOP and computing are the central theme of this thesis, and have been considered in some depth in the previous chapter.

11.3 The contribution of the thesis

The foundation of this work is not original. It is based on the work of Lakoff, Fauconnier and others, described primarily in Chapter Three, who have shown that humans grasp concepts in a manner outlined by conceptual integration networks. This is supported by a very large set of examples across natural languages, and a demonstration of the internal consistency of this at a deep conceptual level rather than just a superficial lexical one.

The original step is to apply this notion beyond ‘everyday’ ideas to ‘scientific’ ideas, in particular that of OOP in Java, and students’ acquistion of those ideas in an educational context. This step solves the problem of why some students cannot understand OOP ideas, after they have been ‘told’ in apparently literal language what these ideas are and how they relate. The solution of that problem is that understanding is not based on literal language.

This makes a profound difference to the way that we perceive the normative conceptions of the expert community, and to the way that novices learn the subject. The way the expert community holds concepts of Java and OOP is generally thought to be captured in a simple way in the explicit definitions of Java syntax, and the surface content of textbooks, lectures, instructional websites and so on. But we have seen that an analysis of this discourse shows that Java and OOP, and the underlying ideas of programming and computers, is intensely metaphorical or otherwise non-literal.

In turn, the learning process should be seen not simply as students being ‘told things’, but experiences which provide them with evidence upon which they can construct corresponding non-literal concepts. The early interviews with first-year undergraduates showed how they had developed metaphorical views of computers and programming before they started their degree courses, in ‘normal life’. Those were the foundations for learning the theory and practice of Java and OOP, in a context where the participants regarded the discourse as literal. As noted in 11.2, this is particularly significant in the case of Java, since it is both implicitly and explicitly metaphorical.

11.4 Is there any place for literal language?

As an example, consider the following question asked by a student in the Facebook Java group:

> *“If ResultSet is an interface then how can we call methods like next() isLast()etc. without defining the method body anywhere in our program? As I understand it in an interface methods are only declared but not defined.”*

The author’s response was:

> *“In the Java Language Specification (JLS), it is written:*
>
> *4.12.2 A variable of an interface type can hold a null reference or a reference to any instance of any class that implements the interface.*
>
> *In other words, if you say:*
>
> *SomeInterface something;*
>
> *you are telling the compiler that 'something' will be a reference to an object which belongs to a class which implements SomeInterface. It does NOT say that 'something' IS an interface, but that the 'something' object will be able to DO the methods in SomeInterface.*
>
> *In this case*
>
> *RecordSet rs = .. executeQuery(...;*
> *then the executeQuery makes an object and returns a reference to it. We don't know (or care) what class that object is (we could find out if we wanted to, using reflection) , but we do know that it can do the RecordSet methods.”*

The student replies:

"..Thanks for welcoming and a very elaborated answer,that was very helpful...."

The student has some knowledge of Java concepts, but has identified an apparent conflict – something is seemingly declared as an instance of an Interface and its methods are used, yet Interfaces do not have methods defined. The student has constructed a frame which includes the concept of interface, but has now encountered a situation which seems to involve evidence which is inconsistent with the frame. The conflict is resolved by looking at the formal specification of the Java language, which shows that interface variables are not references to an interface, but to an object of a class which implements the interface methods.

So at first sight it appears that explicit literal definitions can help students' understanding. The caveat is that since almost all conceptualisation is figurative, apparently purely literal statements are not what they seem. For example JLS 4.12.2 is

A variable.. can hold..

and the term 'hold' is metaphorical.

11.5 Pedagogical Implications

This is not a reductionist theory, and many factors are involved in determining the extent to which a student understands a topic. Matthew was asked about the general progress of the class, and he answered that most students who attended lectures grasped it quite well. Clearly motivation, previous experience, the ability to integrate experiences over a short period of time, and contingent factors (such as Aaron's financial problems) are all significant in determining a student's progress.

Neither is it a necessary theory, in the sense that courses have been taught without it. Some students (such as Jennifer in Chapter 8) readily acquire OOP ideas, even though their teachers are unaware of CINs and such like. Many approaches to teaching have been developed pragmatically, with teachers using techniques which work and dropping ineffective ones. However this thesis explains why these approaches are effective.

11.5.1 Explicit literal statements are insufficient

This follows from the basic thesis. For example, while 'an object is an instance of a class' is true, and concisely summarises the situation, it is intensely metaphorical, and of little use to novices. It does not help a student construct ideas of what OOP classes and objects are, unless they know already, in which case it is nothing more than a little phrase which is useful in encapsulating the situation.

11.5.2 Numerous carefully chosen examples are needed

Examples are a traditional part of a teaching programme. This thesis provides an explanation of why that is the casse. The simple statement of a concept, in apparently literal language, provides insufficient evidence for the student to construct their own versions of something which will inevitably be a complex non-literal blend.

However the choice of examples is difficult, and we can relate this to the various notions of concept examined in Chapter Two. An example should be

- Minimalistic, in the sense of not having aspects which are not essential features of the concept, and

- Typical, possessing features which are usually found when the concept is used.

The first point relates to the feature list aspect of a concept. If an example contains an aspect which is incidental, we risk the student incorrectly assuming it is mandatory. The second corresponds to concepts not having defining criteria, but having indistinct boundaries of items which are more or less typical.

Unfortunately these two points often conflict. For example in a class hierarchy the base class is often abstract. If an example is chosen where this is not the case, we meet the first criteria, in that we are omitting something which is not essential. But such an example is not typical. On the basis of either choice, we risk the student concluding that a base class *cannot* be abstract, or that it *must* be.

Another consideration is whether ideas are introduced one at a time in sequence, or several in parallel. If we give the typical example of a hierarchy where the base class is abstract, we are expecting the student to construct at the same time the ideas of inheritance, and the distinct idea corresponding to the keyword ‘abstract’. This is another case of the tight integration of OOP concepts.

A second example relates to how class definitions are introduced. A class definition will include data members or fields, methods, and constructors. One path through this is to first exemplify classes which only have data members. These correspond to *structs* in C and *records* in Pascal, so this might be an accessible route for students who know these languages. A second round of examples might include methods, and one method might be an *init* method which initialises an object. The third round of examples would include constructors, as a development of the *init* idea.

This gives a simple step-wise approach introducing one idea at a time. However it means the student first sees examples which are very non-typical, and also forces the student to radically revise their picture of what an object is, from the data-only form, through methods as well, and then to include constructors. It also risks the student thinking a constructor is a special type of method, which is not true, since constructors are not inherited.

Some mitigation of these problems derives from presenting numerous examples. Giving only one example risks the student thinking the example *is* the concept. An example of this is using the database ‘autonumber’ idea to show *static*. At least one student thought static meant auto-number. Several examples also provide students with more evidence to construct their concepts from. If example A has feature X, but example B does not, the student can work out that X is allowed but not mandatory.

11.5.3 Common mis-conceptions should be explicitly addressed

Non-normative concept constructions often arise as a consequence of met-befores –such as ‘static means constant’ which has been seen in very many students. Others are that a class is a collection of objects, or that a class is a ‘program’. These ideas should be explicitly refuted, in lectures, demonstration classes or textbooks. Such material is a very efficient way of supporting students’ construction of normative concepts.

11.5.4 Hidden frames should be made explicit

By ‘hidden frames’ is meant concept complexes which are foundational and typically assumed to be ‘obvious’, and perhaps not normally identified. Three examples in the domain of programming are described next.

The three basic frames of program text, state space and input. The idea that program code normally executes step-wise, that each step alters machine state, and that the effect of each step is a function of current state and input, is often treated as already understood (although not in these terms). Many students (Chapter Seven) have been seen who were in fact not aware of this.

The intricacy of the notion of ‘program variable’. This is often treated metonymically, and in different ways – name for whole, value for whole, type for whole. Reference and primitive types is another related difficulty, compounded by the fact that the useful met-before of ‘pointer’ is usually omitted from a Java course.

Code as sequence and code as definition. Typical procedural code is a sequence with a significant order of execution step, and the same is true of Java code *inside* a method. However program code in a Java class file is a definition, and the textual sequence of constructors, methods and data members signifies nothing. Explicitly addressing this would avoid a great deal of student mystification.

11.6 Further Work

This thesis offers a description of what ‘understanding’ means in an educational context. As such it provides a framework within which to understand the process of learning. However the description covers a great deal of ground, including the role of frames in structuring mental spaces, the use of metaphors and other CINs, the development of these over time, and the contrast between learning in ‘every-day’ contexts and in formal education. Each of these requires deeper investigation.

The thesis has shown that thinking about simple program fragments is consistent with three spaces each with their own frames – program text, state transition and I/O. Can other domains be seen in a similar way? For example, can the understanding of elementary classical mechanics, or electricity and magnetism, be seen to have such structuring frames? A corresponding investigation would be how non-literal is the normative language associated with these domains. The terms ‘current’ and ‘field’ suggest metaphor is used here. But mechanics is as curious as Java, although in reverse – mechanics is apparently concerned with non-abstract quantities some of which, such as velocity and force, are almost directly perceivable. Does this affect non-literal conceptions?

This thesis has been mostly concerned with the character and nature of the concepts which students have, and only in a few instances has it been possible to consider the development of those concepts as a change through time. However analysis here has to take into account the fact that every-day life concepts are acquired over several years, whereas in formal education this takes place over a few months. For example children develop the mental frame of ‘supermarket’ over several years, whereas a student must learn OOP concepts in one term or semester.

So this opens out two further lines of work. The first is the development over time of children’s mental frames of everyday concepts, such as ‘supermarket’, ‘airport’ or Fillmore’s paradigmatic ‘commercial transaction’ frame. A phenomenographic approach might be appropriate here.

The second is a focus on the development of frames in undergraduates in formal education through the duration of a module. This would involve an examination of the changes in how students think of a concept over time, correlated with their learning experiences in lectures, workshops, assignments and so on. In programming, a possible target would be the idea of ‘variable’.

These suggestions would lead to a wider and deeper way of thinking about the process of learning.

APPENDICES

Appendix One – The Game

The Game

A computer game is played between the computer and a human user. The game takes place in two stages.

In the first stage, both the player and the computer each choose a number, either 1 or 2. The player chooses through keyboard input, whilst the computer chooses at random. They might choose the same, or differently. For example, both might choose 2, or the player might choose 1 and the computer 2.

In the second stage, 2 further random numbers are produced, in the range 1 to 50 inclusive. The player is associated with one of these, as a 'score', depending on their choice in the first stage. So if the player chose 1, they get the first number, while if they chose 2, they get the second number.

The same applies to the computer.

This means if they both choose the same number in stage 1, they'll get the same score.

Then the winner is decided, as follows. If a score is over 40, it is 'too large'. Then:

if both scores are too large, it is a draw. For example, player 45 and computer 49, it's a draw.

if just one score is too large, the other wins. For example, player 12 and computer 42, player wins

if neither is too large, but they are equal, it's a draw. For example player 21 and computer 21, result draw

otherwise higher score wins. For example, player 21 and computer 29, computer wins.

OOP Data model

What classes would you have in a Java OOP program for this?

What objects would you have?

What would be the attributes (data members or fields) of those classes?

What methods would they have?

Appendix Two – Source code population modelling

```
package disease;

import java.util.Vector;

// Model of a population of organisms
// susceptible to a disease, with a vaccine available.
//  We  can  change  the  organism  characteristics,  and  follow
the
// change in population size over 100 'years'

public class Main {

    public static void main(String[] args) {
        // start off with just 10 of them
        for (int i = 0; i < 10; i++) {
            Organism.population.add(new Organism());
        }
        // then follow them for 100 years
        for (int year = 0; year < 100; year++) {
            // show how many we've got
            System.out.println("Year   "   +   year   +   "   :
Population size " + Organism.population.size());

            // spread infection?
            for  (int  i  =  0;  i  <  Organism.population.size();
i++)  {
                Organism.population.elementAt(i).infect();
            }
            // reproduce?
            for  (int  i  =  0;  i  <  Organism.population.size();
i++)  {

Organism.population.elementAt(i).reproduce();
            }
            // die?
            for (int i = 0; i < Organism.population.size();
i++) {
                Organism.population.elementAt(i).die();
            }
        }
    }
}

class Organism { // models our organism

    boolean vaccinated = false;
    boolean infected = false;
    // fraction of new ones that are vaccinated
    final static double VACCINATION_RATE = 0.2;
    //  chance  that  this  will  infect  another  in  one
generation, and that
    // new one starts infected
    final static double INFECTION_RATE = 0.8;
    // chance of reproducing in one year
    final static double REPRODUCTION_RATE = 0.05;
    // chance of dying in one year if infected
    final static double DEATH_RATE = 0.9;

    //  population  is  a  Vector,  which  holds  all  the
organisms
    // 'alive' at the moment
    static     Vector<Organism>     population     =     new
Vector<Organism>();

    Organism() { // construct a new one
        if (Math.random() < VACCINATION_RATE) {
            vaccinated = true;
        }
        if  (Math.random()  <  INFECTION_RATE  &&  !vaccinated)
{
```

```
            infected = true;
        }
    }

    void infect() // maybe infect another one
    {
        if (!infected) { // can't infect if this not
infected
            return;
        }
        if (Math.random()>INFECTION_RATE) // throw the dice
            return;
        // identify the one to infect
        int otherIndex = (int) (Math.random() *
population.size());
        Organism other = population.elementAt(otherIndex);
        // infect it if it is not vaccinated
        if (!other.vaccinated) {
            other.infected = true;
        }
    }

    void reproduce() // maybe reproduce
    {
        if (Math.random() < REPRODUCTION_RATE) {
            population.add(new Organism());
        }
    }

    void die() // maybe die
    {
        if (infected && Math.random() < DEATH_RATE) {
            population.remove(this);
        }
    }
```

```
}
```

Appendix Three - Employee and related classes

Version One

```
public class Employee
{
// fields
     private String name;          // person's name
     private int grade;       //   seniority:   1   =   most
junior, 7 = most senior

     public Employee(String name, int grade)
     {  // This is the constructor
this.name=name;
this.grade=grade;
     }

     public void promote()
     { // increase their grade
     if (grade<7)
          grade++;
     return;
     }

     public void display()
     {    // display name and grade on the screen
     System.out.println(name+"  .. Grade"+grade);
     }

}
___________________________________________________________

Code to use this class….
```

```

Employee person1=new Employee("Joe",3);
person1.display();
person1.promote();
person1.display();
```

Version Two

```
public class Employee
{
// fields
     private String name;           // person's name
     private int grade;       //   seniority:   1   =   most
junior, 7 = most senior

     public Employee(String name, int grade)
     {  // This is the constructor
this.name=name;
this.grade=grade;
     }

     public void promote()
     { // increase their grade
     if (grade<7)
          grade++;
     return;
     }

     public void display()
     {    // display name and grade on the screen
     System.out.println(name+"  .. Grade"+grade);
     }
     public Employee getMoreSenior(Employee someoneElse)
     {    // compare this person to someoneElse and
          // return the more senior person
     if (this.grade>someoneElse.grade)
          return this;
     else
          return someoneElse;
     }
}
___________________________________________________________

Code to use this class….

Employee person1=new Employee("Joe",3);
Employee person2=new Employee("Jane",5);
Employee another = person1.getMoreSenior(person2);
another.display();
```

Version 3

```
public class Employee
{
// fields
     private String name;          // person's name
     private int grade;       //   seniority:   1   =   most
junior, 7 = most senior

     public Employee(String name, int grade)
     {  // This is the constructor
 this.name=name;
 this.grade=grade;
     }

     private void promote()
     { // increase their grade
     if (grade<7)
          grade++;
     return;
     }

     public void display()
     {    // display name and grade on the screen
     System.out.println(name+"  .. Grade"+grade);
     }

     private Employee getMoreSenior(Employee someoneElse)
     {    // compare this person to someoneElse and
          // return the more senior person
     if (this.grade>someoneElse.grade)
          return this;
     else
          return someoneElse;
     }

       public static void main(String[] args)
       {
        Employee[] employees = new Employee[3];
        employees[0] = new Employee("Joe",1);
        employees[1] = new Employee("Jane",3);
        employees[2] = new Employee("Jim",2);
        for (int i=0; i<employees.length; i++)
          employees[i].display();

       }

}
```

Version 4

```
public class Employee
{
// fields
     private String name;          // person's name
     private int grade;       //   seniority:   1   =   most
junior, 7 = most senior

     public Employee(String name, int grade)
     {  // This is the constructor
this.name=name;
this.grade=grade;
     }

     private void promote()
     { // increase their grade
     if (grade<7)
          grade++;
     return;
     }

     public void display()
     {    // display name and grade on the screen
     System.out.println(name+"  .. Grade"+grade);
     }

     private Employee getMoreSenior(Employee someoneElse)
     {    // compare this person to someoneElse and
          // return the more senior person
     if (this.grade>someoneElse.grade)
          return this;
     else
          return someoneElse;
     }

        public    static    Employee    mostSenior(Employee[]
somePeople)
          {
          Employee boss = null;
          int highestGradeSoFar=0;
          for (int i=0; i<somePeople.length; i++)
               {
               Employee whichOne=somePeople[i];
               if (whichOne.grade>highestGradeSoFar)
                    {
                    boss=whichOne;
                    highestGradeSoFar=whichOne.grade;
                    }
               }
          return boss;
          }



       public static void main(String[] args)
       {
        Employee[] employees = new Employee[3];
        employees[0] = new Employee("Joe",1);
        employees[1] = new Employee("Jane",3);
        employees[2] = new Employee("Jim",2);
        Employee boss= Employee.mostSenior(employees);
        boss.display();
       }
}
```

Version Five

```
public class Employee
{
// fields
     String name;        // person's name
     int grade;          // seniority: 1 = most junior, 7 =
most senior
```

```
     public Employee(String name, int grade)
     {  // This is the constructor
 this.name=name;
 this.grade=grade;
     }

     private void promote()
     { // increase their grade
     if (grade<7)
          grade++;
     return;
     }

     public void display()
     {    // display name and grade on the screen
     System.out.println(name+"  .. Grade"+grade);
     }

     private Employee getMoreSenior(Employee someoneElse)
     {    // compare this person to someoneElse and
          // return the more senior person
     if (this.grade>someoneElse.grade)
          return this;
     else
          return someoneElse;
     }

        public    static    Employee    mostSenior(Employee[]
somePeople)
          {
          Employee boss = null;
          int highestGradeSoFar=0;
          for (int i=0; i<somePeople.length; i++)
               {
               Employee whichOne=somePeople[i];
               if (whichOne.grade>highestGradeSoFar)
                    {
                    boss=whichOne;
                    highestGradeSoFar=whichOne.grade;
                    }
               }
          return boss;
          }



       public static void main(String[] args)
       {
        ManagerialStaff        person1        =        new
ManagerialStaff("Rehana",5,1200.00);
        person1.pay();
        HourlyPaid         person2         =         new
HourlyPaid("Freddy",2,6.50);
        person2.setHoursThisMonth(150);
        person2.pay();
       }


}


class ManagerialStaff extends Employee
  {
  private double getsPaid; // salary per month
  public  ManagerialStaff(String  name,  int  grade,  double
salary)
  {
     super(name, grade);
     this.getsPaid=salary;
  }

  public void pay()
  {
     System.out.println(name+" gets paid "+ getsPaid) ;
  }


  }


class HourlyPaid extends Employee
  {
  private double getsPaid; // earns this per hour
  private int hoursThisMonth; // how many hours they've
worked this month

  public HourlyPaid(String name, int grade, double
payPerHour)
  {
     super(name, grade);
     this.getsPaid=payPerHour;
  }

  public void setHoursThisMonth(int hours)
  {
     hoursThisMonth=hours;
  }

  public void pay()
  {
     System.out.println(name+" gets paid "+
getsPaid*hoursThisMonth);
  }
}
```

Version Six

```
import java.util.ArrayList
public abstract class Employee
```

```
{
// fields
     String name;        // person's name
     int grade;          // seniority: 1 = most junior, 7 =
most senior

     public Employee(String name, int grade)
     {  // This is the constructor
this.name=name;
this.grade=grade;
     }

     public abstract void pay();

     private void promote()
     { // increase their grade
     if (grade<7)
          grade++;
     return;
     }

     public void display()
     {    // display name and grade on the screen
     System.out.println(name+"  .. Grade"+grade);
     }

     private Employee getMoreSenior(Employee someoneElse)
     {    // compare this person to someoneElse and
          // return the more senior person
     if (this.grade>someoneElse.grade)
          return this;
     else
          return someoneElse;
     }
```

```
        public    static    Employee    mostSenior(Employee[]
somePeople)
          {
          Employee boss = null;
          int highestGradeSoFar=0;
          for (int i=0; i<somePeople.length; i++)
               {
               Employee whichOne=somePeople[i];
               if (whichOne.grade>highestGradeSoFar)
                    {
                    boss=whichOne;
                    highestGradeSoFar=whichOne.grade;
                    }
               }
          return boss;
          }



       public static void main(String[] args)
       {
       WorkForce workforce = new WorkForce();
      workforce.recruit("Lee",6,2000,true);
      workforce.recruit("Alex",1,10.00,false);
      workforce.display();
      workforce.pay();
       }

}

class ManagerialStaff extends Employee
  {
  private double getsPaid; // salary per month
  public  ManagerialStaff(String  name,  int  grade,  double
salary)
  {
     super(name, grade);
     this.getsPaid=salary;
  }

  public void pay()
  {
     System.out.println(name+" gets paid "+ getsPaid) ;
  }


  }

class HourlyPaid extends Employee
  {
  private double getsPaid; // earns this per hour
  private int hoursThisMonth=40; // how many hours they've
worked this month

  public   HourlyPaid(String   name,   int   grade,   double
payPerHour)
  {
     super(name, grade);
     this.getsPaid=payPerHour;
  }

  public void setHoursThisMonth(int hours)
  {
     hoursThisMonth=hours;
  }

  public void pay()
  {
     System.out.println(name+"      gets      paid      "+
getsPaid*hoursThisMonth);
  }
  }


   class WorkForce
   {
   private     ArrayList<Employee>     people     =     new
ArrayList<Employee>();

   public void display()
   {
     for (Employee e : people)
          e.display();
   }

   public void recruit(String name, int grade, double pay,
boolean manager)
   {
     if (manager)
     {
     ManagerialStaff  person  =  new  ManagerialStaff(name,
grade, pay);
     people.add(person);
     }
     else
     {
     HourlyPaid person = new HourlyPaid(name, grade, pay);
     people.add(person);
     }
   }

   public void pay()
   {
   for (Employee e : people)
          e.pay();
   }
   }
```

Appendix Four – Consent Form

Invitation to participate in research - Participant Copy

Dear participant

I am currently carrying out some research into how students learn OOP in Java. I am studying how people develop their ideas of the concepts involved in OOP, how those ideas develop over time and as they attend classes and work on the topic.

The research will involve taking part in interviews where an audio recording is made. Please note the following:

- Participation in the research is *entirely voluntary*. If you do not wish to take part you do not need to give a reason and there will be no consequences.
- If you do take part, you *can withdraw* at any time if you wish to, again without giving a reason.
- The assessment of your progress and attainment is completely unaffected by your participation or otherwise in this research.
- You will be kept informed about outcomes from the research.
- Your responses will be held securely and will be confidential.
- Extracts from a written transcript of the interview may be published, but they can be anonymised so it will be impossible to identify yourself or your institution as a participant.

If you are willing to take part, please complete the sections below in duplicate. I will keep one copy, and you keep the other.

I am happy to take part in this research as described above

Subject name:…………………………………………….. (please print)

Subject signature……………………………………….

Date:……………………………………………………

Researcher name:………………………………………

Researcher signature:…………………………………..

Date:……………………………………………………

(subject copy)

BIBLIOGRAPHY

Anderson, M.L. (2003) Embodied Cognition : A field guide, in *Artificial Intelligence* 149 (2003) 91-130

Apostle, H. G. (1980) *Aristotle's Categories and Propositions* (De Interpretatione) Grinnell Iowa: Peripatetic Press

Armstrong, D. (2006) The Quarks of object-oriented development. In *Comms. of the ACM* February 2006/Vol. 49, No. 2

Arnold K., Gosling J. and Holmes D. (2001) *The Java Programming Language* Third Edition Boston: Addison-Wesley

Asiala M., Cottrill J., and Dubinsky E, (1997) The Development of Students' Graphical Understanding of the Derivative *Journal of Mathematical Behavior* 16 (4) 399-431

Barnes D. J. & Kolling M. (2006) *Objects First with Java Third Edition* Harlow: Pearson Education

Beck, K. (2000) *Extreme Programming Explained: Embrace Change*, Reading, Mass. Addison-Wesley

Ben-Ari, M. (1998) Constructivism in computer science education. *SIGCSE Bull. 30, 1* (Mar. 1998), 257-261

Ben-Ari, M. and Sajaniemi, J. (2004) Roles of variables as seen by CS educators. In *Proceedings of the 9th Annual SIGCSE Conference on innovation and Technology in Computer Science Education* (Leeds, United Kingdom, June 28 - 30, 2004). ITiCSE '04. ACM, New York, NY, 52-56.

Bennedsen, J. and caspersen, M. (2008) Failure Rates in Introductory Programming. In *Inroads - The SIGCSE Bulletin*, Vol 39 Number 2 pp. 32-36

Beynon, W.M. and Russ, S.B. (1991) The Development and Use of Variables in Mathematics & Computer Science in *The Mathematical Revolution Inspired by Computing*, IMA Conf Series 30, 285-95, 1991.

Biggs, J. (1999) *Teaching for Quality Learning at University*. Buckingham: Open University Press

Boden, M. A. (1982) Is Equilibriation important? A view from artificial intelligence. In *British Journal of Psychology* (1982) 73, 165-173

Bogart, C. (2009) Private communication

Bourne, L. E. (1970) Knowing and using concepts. *Psychol. Rev.* 77, 546 - 556 (1970).

Brooks, R. (1983) Towards a theory of the comprehension of computer programs, in *Int. J. Man-Machine Studies* (1983) 18,543-554

Brown A., DeVries D., Dubinsky E. and Thomas K. (1997) Learning Binary Operations, Groups and Subgroups *Journal of Mathematical Behavior* 16 (3) 187-239

Bruce, K.B., Danyluk, A., and Murtagh, T. (2005) Why Structural Recursion Should Be Taught Before Arrays In CS1, *36th. Technical Symposium on Computer Science Education*, St. Louis Missouri

Bruner, J. S., Goodnow, J. J., & Austin, G. A. (1956). *A Study of Thinking.* New York: Wiley.

Burr, V. (2003) *Social Constructionism*, Routledge, New York

Byckling, P., Gerdt, P. and Sajaniemi, J. (2005) Roles of Variables in Object-oriented Programming in *Companion to the 20th annual ACM SIGPLAN conference on Object-oriented programming, systems, languages, and applications*, San Diego

Cann, R. (1993) *Formal Semantics.* Cambridge: Cambridge University Press

Catania, A. C. and Harnad, S. R. (1988) *The Selection of Behaviour: the operant behaviourism of B. F. Skinner.* Cambridge : Cambridge University Press

Croft, W. & Cruse, D. A. (2004) *Cognitive Linguistics* Cambridge : Cambridge University Press

Collins, A. M., & Quillian, M. R. (1969) Retrieval time from semantic memory. *Journal of Verbal Learning and Verbal Behavior*, 1969, 8, 240-248.

Dennet, D. C. (1991) *Consciousness Explained.* Boston: Little, Brown and Co

Détienne, F. (1997) Assessing the cognitive consequences of the object-oriented approach: A survey of empirical research on object-oriented design by individuals and teams' in *Interacting with Computers* 9 (1997) 47-72

DFES (2009) *Handbook for teachers Part 2 Summarising* http://publications.teachernet.gov.uk/eOrderingDownload/DfES%200035-2205G%2011.pdf retrieved 24 August 2009

Dijkstra E W (1968) Go To Statement Considered Harmful *Communications of the ACM*, Vol. 11, No. 3, March 1968, pp. 147-148

Donald, M. (2001) *A Mind So Rare.* New York: Norton

Douce, C. (2004) Metaphors We Program By. In *Proceedings of the 16th Workshop of the Psychology of Programming Interest Group*, Carlow, Ireland

Dreyfus H. and Dreyfus S. (1986) *Mind over machine: The power of human intuition and expertise in the era of the computer* New York: Free Press

du Boulay, B. (1989) Some difficulties of learning to program. In Soloway E. and Spohrer J. C. (eds) *Studying the Novice Programmer.* Hillsdale NJ: Lawrence Erlbaum

Dubinsky E. (1991) Reflective Abstraction in Advanced Mathematical Thinking in *Advanced Mathematical Thing* (ed. D. Tall) Dordrect Kluwer Academic Publishers

Eckerdal, A., McCartney, R., Mostrom, J. E., Ratcliffe, M., Sanders, K. and Zander, C. (2006) Putting Threshold Concepts into Context in Computer Science Education. In *Proceedings of the 11th conference on Information Technology in Computer Science Education* (ITiCSE 2006), Bologna, Italy

Eckerdal, A. , McCartney, R. , Moström, J.E., Sanders, K. , Thomas, L, Zander C. (2007) ICER '07: From Limen to Lumen: computing students in liminal spaces. *Proceedings of the third international workshop on Computing education research*

Eckerdal, A. and Thuné, M. (2005) Novice Java programmers' conceptions of 'object' and 'class' and variation theory. In *ITiCSE-05* pages 89-93

Eco, U, Santambrogio, M., and Violi, P. (1988) *Meaning and Mental Representations.* Bloomington: Indiana University Press

Ehrlich, K. and Soloway, E. (1984) An Empirical Investigation of the Tacit Plan Knowledge in Programming. In: Thomas, J.C. and Schneider, M.L. (eds) *Human Factors in Computer Systems.* Norwood NJ: Ablex Publishing

Ennis, R.H. (1978) Conceptualization of Children's Logical Competence. In Siegel L. and Brainerd C. J. (eds) *Alternatives to Piaget.* New York: Academic Press

Evans, V., Bergen, B. and Zinken, J. (2007) *The Cognitive Linguistics Reader.* London: Equinox

Evans, V. & Green, M. (2006) *Cognitive Linguistics : An Introduction* New Jersey : Lawrence Erlbaum

Fauconnier, G. (1994) *Mental Spaces: Aspects of Meaning Construction in Natural Language.* Cambridge: Cambridge University Press

Fauconnier, G. (2008) How Compression Gives Rise to Metaphor and Metonymy. *9th Conference on Conceptual Structure, Discourse and Language* (CSDL)
Meaning, Form and Body . Case Western Reserve University

Fauconnier, G. and Turner, M. (2002) *The Way We Think*. New York: Basic Books

Feldman, J. (2000). Minimization of Boolean complexity in human concept learning. *Nature,* 407, 630-633

Fillmore, C. (1982) Frame Semantics, in Linguistic Society of Korea (ed) *Linguistics in the Morning Calm*. Seoul: Hanshin Publishing

Fincher, S. and Petre, M. (2004) *Computer Science Education Research*, Routledge Falmer

Fleury, A. (2000) Programming in Java: Student-constructed rules. in *SIGCSE Bulletin* 32(1): 197-201 Austin Texas

Fodor, J. (1976) *The Language of Thought.* Sussex: Harvester Press

Fodor, J. (1987) *Psychosemantics*. London: MIT Press

Fodor, J. (1994) *The Elm and the Expert.* London: MIT Press

Gamma, E., Helm, R., Johnson, R.., and Vlissides, J. (1995) *Design Patterns: Elements of Reusable Object-Oriented Software* Reading, Mass. : Addison-Wesley

Garner, S., Haden, P., and Robins, A. 2005. My program is correct but it doesn't run: a preliminary investigation of novice programmers' problems. In *Proceedings of the 7th Australasian Conference on Computing Education* - Volume 42 (Newcastle, New South Wales, Australia). A. Young and D. Tolhurst, Eds. ACM International Conference Proceeding Series, vol. 106. Australian Computer Society, Darlinghurst, Australia, 173-180

Götschi, T., Sanders, I. and Galpin, V. (2003) Mental Models of Recursion *34th. SIGCSE Technical Symposium on Computer Science Education*, Reno, Nevada

Grice, H.P. (1981) "Presupposition and Conversational Implicature", in P. Cole (ed.), *Radical Pragmatics*, Academic Press, New York, pp. 183–198.

Griffiths, R., Holland, S. and Edwards, M. (2007) Sense before syntax: a path to a deeper understanding of objects. In *ITALICS* Vol. 6 Issue 4 October 2007

Harnad, S. (1990) The Symbol Grounding Problem. *Physica* D 42: 335-346.

Hoc, J.M., Green, T.R.G., Samurçay, R., and Gillmore, D.J. (Eds.) (1990) *Psychology of Programming*. London: Academic Press

Holland, S., Griffiths, R. and Woodman, M. (1997) Avoiding Object Misconceptions. In *SIGCSE-97*, pages 131-134

Holmboe, C. (1999) A cognitive framework for knowledge in informatics: the case of object-orientation. *SIGCSE Bull.* 31, 3 (Sep. 1999), 17-20.

Horstmann C. S. (2005) *Java Concepts* New Jersey: Wiley

Horstmann C. S. & Cornell G. (1997) *Core Java 1.1 Volume 1 Fundamentals New Jersey* : Sun Microsystems Press

Hu, C. (2006) When to Use an Interface in *SIGCSE Bulletin* Vol. 38 No. 2

Hume, D. (1998) *An Enquiry concerning Human Understanding.* ed Tom Beauchamp. Oxford: Oxford University Press

Inhelder, B. and Piaget, J. (1958) *The Growth of Logical Thinking.* New York: Basic Books

Johnson, M. (1987) *The Body in the Mind.* Chicago: University of Chicago Press

Johnson, O. (1978) *Skepticism and Cognitivism* Los Angeles: University of California Press

Johnson-Laird, P.N. (1983) *Mental Models* . Cambridge: Cambridge University Press

Kahney, K. (1985) What do novice programmers know about recursion? in Soloway, E, and Spohrer J.C. (Eds.) (1981) *Studying the Novice Programmer.* Hillsdale NJ: Lawrence Erlbaum

Kernighan B W and Ritchie D M (1988) *The C Programmming Language* Second Edition Prentice Hall

Keeling, S.V. (1978) *Descartes* Second edition, Oxford: Oxford University Press

Knuth, D. (1997) *The Art of Computer Programming*, Volume 2: Seminumerical : Algorithms, Third Edition. Boston: Addison-Wesley,

Kölling, M., Quig, B., Patterson, B. and Rosenberg, J. (2003) The BlueJ system and its pedagogy *Journal of Computer Science Education*, Special Issue on Learning and Teaching

Object Technology, Vol 13, No 4, Dec 2003

Kölling, M. and Rosenberg, J. (2001) Guidelines for Teaching Object Orientation with Java in *Proceedings of the 6th conference on Information Technology in Computer Science Education* (ITiCSE 2001), Canterbury, 2001

Kripke, S. (1979) A Puzzle about Belief, In *Meaning and Use*, edited by A. Margalit. Dordrecht and Boston: Reidel

Kripke, S. (1980). *Naming and Necessity.* Cambridge, MA: Harvard University Press.

Kuittinen, M. and Sajaniemi, J. (2004) Teaching Roles of Variables in Elementary Programming Courses, in *Proceedings of the 9th conference on Information Technology in Computer Science Education* (ITiCSE 2004), Leeds

Kuittinen, M. and Sajaniemi, J. (2003) First results of an experiment on using roles of variables in teaching. In *EASE and PPIG 2003*, Papers from the Joint Conference at Keele University pp 347-357

Lahtinen, E., Ala-Mutka, K. and and Järvinen, H (2005) A Study of the Difficulties of Novice programmers. In *Proceedings of the 10th conference on Information Technology in Computer Science Education* (ITiCSE 2005), Portugal

Lakoff, G. (1996) Sorry I'm Not Myself Today : The Metaphorical System for Conceptualizing the Self. In Fauconnier, G. and Sweetser, E. (eds) *Spaces Worlds and Grammar*. Chicago: University of Chicago Press

Lakoff, G. (2008) Idea Framing, metaphors and your Brain. http://www.youtube.com/watch?v=S_CWBjyIERY&feature=fvw accessed 2 September 2009

Lakoff, G. (1973) Hedges: A study in meaning criteria and the logic of fuzzy concepts . In *Journal of Philosophical Logic* Volume 2, Number 4 Pages 458-508

Lakoff, G. (1982) *Categories and Cognitive Models : Series A Paper No. 96*, Linguistics Dept, UCLA Berkeley

Lakoff, G. (1987a) Cognitive models and prototype theory. In Neisser, U. (ed) *Concepts and Conceptual Development*. Cambridge: Cambridge University Press

Lakoff, G. (1987b) *Women, Fire and Dangerous Things*. Chicago: University of Chicago Press

Lakoff, G. and Johnson, M. (1983) *Metaphors We Live By*. Chicago: Chicago University Press

Langacker, R.W. (1987) *Foundations of Cognitive Grammar* Vol. 1 : Theoretical Pre-requisites Stanford: Stanford University Press

Lawson, A.E. (1979) Relationships among performances on group-administered items of formal reasoning. *Perceptual and Motor Skills*, 48, 71-78.

Lewis, J. (2000) Myths about Object-Orientation and Its Pedagogy *31st SIGCSE Technical Symposium on Computer Science Education*, Austin, Texas

Lister, R., Box, I., Morrison, B., Teneberg, J., Westbrook, D. S. (2004) The Dimensions of Variation in the Teaching of Data Structures *9th Annual ITiCSE Conference*, Leeds UK

Lister, R., Adams, E. S., Fitzgerald, S., Fone, W., Hamer, J., Lindholm, M., McCartney, R., Moström, J. E., Sanders, K., Seppälä, O., Simon, B., and Thomas, L. 2004. A multi-national study of reading and tracing skills in novice programmers. *SIGCSE Bull. 36*, 4 (Dec. 2004), 119-150.

Lui, A.K., Kwan, R., Poon, M. and Cheung, Y.H.K. (2004) Saving Weak programming Students: Applying Constructivism in a First Programming Course. *SIGCSE Bulletin* Volume 36 Number 2, June 2004

Marton, F. (1994) Phenomenography. In *The International Encyclopedia of Education*. Second edition , Volume 8. Eds. Torsten Husén & T. Neville Postlethwaite. Pergamon 1994, pp. 4424 - 4429.

Marton, F. and Saljo, R. (1976) On Qualitative Differences in Learning. *British Journal of Educational Psychology* 1976, 46, 4-11

McCartney, R. and Sanders, K. (2005) What are the 'threshold concepts' in computer science? *Proceedings of the Koli Calling* 2005 Conference on Computer Science Education

McCawley, J. D.(1993). *Everything that Linguists Have Always Wanted to Know about Logic* (But Were Ashamed to Ask). Chicago : University of Chicago Press

McCracken, M., Almstrum, V., Diaz, D., Guzdial, M., Hagan, D., Kolikant, Y. B., Laxer, C., Thomas, L., Utting, I., and Wilusz, T. 2001. A multi-national, multi-institutional study of assessment of programming skills of first-year CS students. SIGCSE Bull. 33, 4 (Dec. 2001), 125-180.

Medin, D. L. and Schaffer, M.M. (1978) Context theory of classification learning. *Psychological Review*, 85, 207-238

Medin, D. L., Wattenmaker, W.D, and Hampson, S.E. (1987) Family Resemblances, conceptual cohesiveness, and category construction. *Cognitive Psychology*, 19, 242-279

Meyer, D. E. (1970) On the representation and retrieval of stored semantic information. *Cognitive Psychology*, 1970, 1, 242-299.

Meyer, J. and Land, R. (2003) Threshold Concepts and Troublesome Knowledge: Linkages to Ways of Thinking and practising within disciplines. *ETL Project Occasional Report 4*

Milner, W. W. (2008) A loop is a compression *Proceedings of the 20th. annual meeting of the Psychology of Programming Interest Group* Lancaster UK

Milner, W.W. (2009) A broken metaphor in Java *Inroads SIGCSE Bulletin* Volume 41 Number 4 December 2009

Minsky, M. (1974) *A Framework for Representing Knowledge* MIT-AI Laboratory Memo 306

Montague, R. (1974) *Formal Philosophy: Collected Papers of Richard Montague* Ed. by R. H. Thomason. New Haven: Yale University Press

Montangero, J. and Maurice-Naville, D. (1997) *Piaget, or The Advance of Knowledge*. MahWah, New Jersey: Lawrence Erlbaum

Murphy, G. L. (2002) *The Big Book of Concepts*. Cambridge, Mass.: MIT Press

Murphy, G. L. and Medin, D.L. (1985) The Role of Theories in Conceptual Coherence. In *Psychological Review* 92, 3 289-316

Newell, A. and Simon, H. (1972) *Human Problem Solving* . Englewood Cliffs, NJ: Prentice-Hall

Neisser, U. (1989) From direct perception to Conceptual Structure. In *Concepts and Conceptual Development* (ed Neisser). Cambridge: Cambridge University Press

Norman, D. A. (1983) Some Observations on Mental Models. In Genter, D and Stevens, A. L. (eds) *Mental Models*. Hillsdale, New Jersey: Lawrence Erlbaum

Or-Bach, R. and Lavy, I., (2004) Cognitive Activities of Abstraction in Object Orientation: An Empirical Study *SIGCSE Bulletin* Volume 36 Number 2, June 2004

Papert, S. (1980) *Mindstorms: Children, Computers, and Powerful Ideas* New York: Basic Books

Parsons, C. (1960) 'Inhelder and Piaget's The Growth of logical thinking". In *British Journal of Psychology,* 51

Pea, R. D. (1986) Language-Independent Conceptual Bugs In Novice Programming. In *Journal of Educational Computing Research* Vol. 2(1) (1986)

Pears, A., Seidman, S., Eney, C., Kinnunen, P., Malmi, L. (2005) Constructing a Core Literature for Computing Education Research, *SIGCSE Bulletin* Volume 37, Number 4

Pears, A., Seidman, S., Malmi, L., Mannila, L., Adams, E., Bennedsen, J., Devlin, M., Paterson, J. (2007) A Survey of Literature on the Teaching of Introductory Programming, *12th Annual ITiCSE Conference*, University of Canterbury Kent UK

Piaget, J. (1953). *Logic and Psychology*. Manchester: Manchester University Press

Piaget, J. (1955) *The Child's Construction of Reality* trans. Margaret Cook London: Routledge and Kegan Paul

Piaget, J. (1973) *The Child's Conception of the World* trans. J. and A. Tomlinson. St. Albans: Paladin

Piaget, J. and Inhelder, B. (1958) *The Growth of Logical Thinking*. London: Routledge and Kegan Paul

Portner, P. and Partee, B. H. (2002) *Formal Semantics* Oxford: Blackwell Publishers

Putnam, H. (1973) Meaning and Reference. In *The Journal of Philosophy*, Vol. 70, No. 19

Putnam, H. (1979) *Philosophical Papers Vol.2 Mind language and Reality* Cambridge: CUP

Quine, W. V. O. (1960) *Word and Object.* Cambridge Mass: MIT Press

Quine, Willard Van Orman. 1969. Natural Kinds. in *Ontological Relativity and Other Essays*: Columbia Univ. Press.

Rapp, A., Leube, D. Erb, M. Grodd, W. and Kircher, T. (2004) Neural correlates of metaphor processing. In *Cognitive Brain Research*, Volume 20, Issue 3, August 2004, Pages 395-402

Ragonis, N. and Ben-Ari, M. (2005) 'A long-term investigation of the comprehension of OOP concepts by novices', *Computer Science Education*,15:3,203 -221

Reddy, Michael J. (1979). "The conduit metaphor: A case of frame conflict in our language about language", in *Metaphor and Thought*, ed. Andrew Ortony, Cambridge: Cambridge University Press, 1993

Reddy, U. S. 2002. Objects and classes in Algol-like languages. *Information and Computation*. 172, 1 (Feb. 2002), 63-97

Rips, L.J., Shoben E.J. Smith E. E. (1973) Semantic Distance and the verification of semantic relations *Journal of Verbal Learning and Verbal Behaviour*, 12, 1-20

Rist, R. S. (2004) Schema Creation, Application and Evaluation in Fincher, S. and Petre, M. (eds) *Computer Science Education Research*. London: Taylor and Francis

Roberts, E., Bruce, K., Cutler, R., Cross, J., Scott Grissom, S., Karl Klee, K., Rodger, S., Trees, F., Utting, I., Yellin, F. (2006) *The ACM Java Task Force Report.* http://jtf.acm.org accessed September 2008

Robins, R., Rountree, J., and Rountree, N. (2003) Learning and Teaching programming: A Review and Discussion. *Computer Science Education* Vol. 13 Number 2

Rosch E. and Mervis C. B. (1975) Family resemblance: Studies in the internal structure of categories *Cognitive Psychology*, 7, 573-605

Rumelhart, D. E., Lindsay, P. H., & Norman, D. A. (1972) A process model for long-term memory. In E. Tulving & W. Donaldson (Eds.), *Organization and memory*, New York: Academic Press

Sackman, H. (1970) *Man-computer problem solving.* Princeton, NJ: Auerbach

Sackman, H., Erikson, W.J. and Grant E. E. (1968) Exploratory Experimental Studies Comparing Online and Offline Programmmg Performance. *Communications of the ACM* Vol. 11 Number 1

Sajaniemi, J. (2002a) An Empirical Analysis of Roles of Variables in Novice-Level Procedural Programs, in *Proceedings of the IEEE 2002 Symposia on Human Centric Computing Languages and Environments*

Sajaniemi, J. (2002b) Visualizing Roles of Variables to Novice Programmers *14th Workshop of the Psychology of Programming Interest Group*, Brunel University, June 2002

Säljö, R. (1979) 'Learning in the learner's perspective. I. Some common-sense conceptions', *Reports from the Institute of Education*, University of Gothenburg, 76.

Sanders, K. and Thomas, L. (2007) Checklist for Grading Object-oriented CS1 Programs: concepts and misconceptions. In *ITiCSE'07*, Dundee, Scotland

Searle, J. (1980) "Minds, Brains and Programs" in *The Behavioral and Brain Sciences.*3, pp. 417-424.

Shank, R. C. and Abelson, R. (1977) *Scripts Goals Plans and Understanding* Hillsdale, NJ: Erlbaum

Sheil, B. (1981) The Psychological Study of Programming. *Computing Surveys* Volume 13 Number 1

Searle, J. R. (1983) *Intentionality, an essay in the philosophy of mind.* Cambridge: Cambridge University Press

Skemp, R. (1987) *The Psychology of Learning Mathematics.* Hillsdale NJ: Lawrence Earlbaum

Smith E. E. and Medin D. L. (1981) *Categories and Concepts* London: Harvard University Press

Smith, E. E., Shoben, E.J. and Rips, L.J. (1974) Structure and process in semantic memory: A featural model for semantic decisions. In *Psychological Review* 1974, Vol. 81, No. 3, 214-241

Soloway, E. and Spohrer, J. (1989) Novice Mistakes: Are The Folk Wisdoms Correct?. In Soloway, E, and Spohrer J.C. (Eds.) *Studying the Novice Programmer.* Hillsdale NJ: Lawrence Erlbaum

Soloway, E., Ehrlich, K. and Bonar, J. (1982) Tapping Into Tacit Programming Knowledge. In *Proceedings of the Conference on Human Factors in Computing Systems*, 1982: Gaitherburg, Maryland

Spohrer, J.C., Soloway, E. and Pope, E. (1989) A Goal/Plan Analysis of Buggy Pascal Programs. In Soloway, E, and Spohrer J.C. (Eds.) *Studying the Novice Programmer*. Hillsdale NJ: Lawrence Erlbaum

Steffe, L. (1991) The Constructivist Teaching Experiment: Illustrations and Implications. In von Glasersfeld (ed) *Radical Constructivism in Mathematics Education*. Dordrecht: Kluwer Academic Publishers

Stroustrup, B. (1997) *The C++ Programming Language* Third Edition Reading Mass: Addison-Wesley

Tall, D. (2009). Cognitive and social development of proof through embodiment, symbolism & formalism. *ICMI Conference on Proof*, May 2009, Taipei.

Tall, D. (2011) *How Humans Learn to Think Mathematically* (in press)

Tall, D and Vinner, S (1981) Concept image and concept definition in mathematics, with special reference to limits and continuity, *Educational Studies in Mathematics*, 12 151-169

Thomasson, B., Ratcliffe, M., and Thomas, L. (2006) Identifying novice difficulties in object-oriented design. In *ItiCSE '06* pages 28-32

Thurston, W.P. (1990) Mathematics Education. In *Notices of the AMS* 37 (1990), 844–

850

Tulving, E. (1972) .Episodic and semantic memory. In E. Tulving & W. Donaldson (Eds.), *Organisation and memory*. New York: Academic Press,

Tversky, B. (1989). Parts, partonomies, and taxonomies, *Developmental Psychology*, Vol. 25, No. 6, pp. 983--995

Vagianou, E. (2006) Program Working Storage: A Beginner's Model in *Proceedings of the Koli Calling Conference on Computer Science Education*

von Glasersfeld, E. (1984) An Introduction to Radical Constructivism. In Watzlawick, P. (ed) *The Invented Reality*. New York: W.W.Norton

von Glasersfeld, E. (1991) *Radical Constructivism in Mathematics Education*. Dordrecht: Kluwer Academic Publishers

Weinberg, G.M, (1971) *The Psychology of Computer Programming* New York: Van Nostrand Reinhold

Wiser, M. and Carey, S. (1983) When Heat and Temperature Were One. In Genter, D and Stevens, A. L. (eds) *Mental Models*. Hillsdale, New Jersey: Lawrence Erlbaum

Wick, M.R., Stenvenson, D. E. and Phillips, A. T. Seven Design Rules for Teaching Students Sound Encapsulation and Abstraction of Object properties and Member Data in 35th *SIGCSE Technical Symposium on Computer Science Education*, Norfolk, Virginia

Winslow LE (1996) Programming Pedagogy - A psychological overview, *SIGCSE Bulletin*, 28, 17-22

Wittgenstein L. (1958) *Philosophical Investigations* trans: G E M Anscombe Oxford: Blackwell

Young R. M. (1983) Surrogates and mappings: Two Kinds of Conceptual Models for Interactive Devices in Gentner, D., and Stevens, A.L. (eds) *Mental Models*. Hillsdale, NJ : Lawrence Erlbaum