In this chapter you will learn to:

- identify the needs that led to the development of different paradigms
- recognise the issues associated with using an imperative approach to solve some problems such as Artificial Intelligence (AI) and computer gaming
- recognise representative fragments of code written using the logic paradigm
- recognise the use of the logic paradigm concepts in code
- interpret a fragment of code written using the logic paradigm, and identify and correct logic errors
- modify fragments of code written using the logic paradigm to incorporate changed requirements
- code and test appropriate solutions in a language using the logic paradigm
- assess the appropriateness of a software solution written using the logic paradigm against a solution written using an imperative approach
- recognise representative fragments of code written using the object oriented paradigm
- recognise the use of the object oriented concepts in code
- interpret a fragment of code written using the object oriented paradigm, and identify and correct logic errors
- modify fragments of code written using the object oriented paradigm to incorporate changed requirements
- code and test appropriate solutions in a language using the object oriented paradigm
- assess the appropriateness of a software solution written using the object oriented paradigm against a solution written using the imperative approach
- describe the strengths of the imperative, logic and object oriented paradigms
- identify an appropriate paradigm relevant for a given situation
- evaluate the effectiveness of using a particular paradigm to solve a simple problem

Which will make you more able to:

- differentiate between various methods used to construct software solutions
- explain the implications of the development of different languages
- explain the interrelationship between emerging technologies and software development
- identify needs to which software solutions are appropriate
- apply appropriate development methods to solve software problems
- select and apply appropriate software to facilitate the design and development of software solutions

In this chapter you will learn about

Development of the different paradigms

- limitations of the imperative paradigm
  - difficulty with solving certain types of problems
  - the need to specify code for every individual process
  - difficulty of coding for variability
- emerging technologies
- simplifying the development and testing of some larger software projects
- strengths of different paradigms

Logic paradigm

- concepts
  - variables
  - rules
  - facts
  - heuristics
  - goals
  - inference engine
  - backward/forward chaining
- language syntax
  - variables
  - rules
  - facts
- appropriate use, such as:
  - pattern matching
  - AI
  - expert systems

Object oriented paradigm

- concepts
  - classes
  - objects
  - attributes
  - methods/operations
  - variables and control structures
  - abstraction
  - instantiation
  - inheritance
  - polymorphism
  - encapsulation
- language syntax
  - classes
  - objects
  - attributes
  - methods/operations
  - variables and control structures
- appropriate use, such as
  - computer games
  - web-based database applications

Issues with the selection of an appropriate paradigm

- nature of the problem
- available resources
- efficiency of solution once coded
- programmer productivity
  - learning curve (training required)
  - use of reusable modules
  - speed of code generation
  - approach to testing

9

OPTION 1
PROGRAMMING PARADIGMS

The evolution and history of programming languages is traditionally presented as a chronological list commencing with first generation and progressing through second, third, fourth and concluding with fifth generation languages. This technique presents programming languages in order of their emergence and increasing user friendliness. Although this is a reasonable technique for the study of the evolution of programming languages, it does not present the different paradigms used in a coherent manner. In this chapter we will focus on programming languages in terms of different paradigms.

Paradigm
A philosophical or theoretical framework. A different approach under which laws, theories and generalisations are produced.

A paradigm can be thought of as a philosophical or theoretical framework. This framework is then used as the basis of a new way of viewing problems. Theories, laws and generalisations are developed in support of the initial paradigm. Eventually, practical and workable techniques emerge for operating under the new paradigm. In this chapter, we consider and investigate different programming paradigms, why they came into existence and what problems they aim to solve. We concentrate on the major features of two paradigms – the logic and object oriented paradigms. We learn to write some simple source code using languages based on each paradigm as we assess the strengths (and weaknesses) of each. We also compare the logical, object oriented and imperative paradigms in terms of appropriate use and programmer's productivity.

Consider the following:

Many problems can be solved using a variety of different strategies. Some of these strategies and techniques will be useful for the solution of various other problems; some may be of little further use. Different paradigms provide a structure for solving problems from different angles. Lateral thinking involves generating new associations using seemingly unrelated areas, and using these associations in new and original ways. A paradigm is a description of the new association so it can be formalised and reused.

GROUP TASK Activity and Discussion
Everyone in the class is to calculate the sum of the first 1000 counting numbers e.g. 1 + 2 + 3 +… + 1000. Discuss all the different techniques used to accomplish this task. Is any one technique the best? What makes one technique better than another?

DEVELOPMENT OF THE DIFFERENT PARADIGMS

Until relatively recently, the architecture of computer hardware had driven the development of programming languages. Although many different computer architectures are currently being developed, by far the most common is still the traditional von Neumann architecture first developed to calculate trajectories for bombs and shells during World War II. The von Neumann architecture was implemented in the ENIAC (Electronic Numerical Integrator and Computer). Probably every computer you have seen or used is based on the von Neumann concept (also known as the stored program concept). Essentially, this architecture separates data and processing. Data is sent to the CPU for processing and the result is sent back to memory for storage. The CPU is a sequential device; instructions are processed one at a time in a predetermined sequence. The von Neumann computer has lead to the development of procedural or imperative languages. In this course, we have studied the imperative paradigm in detail, as it remains widely used and many of its methodologies are useful foundations to mastering other paradigms.

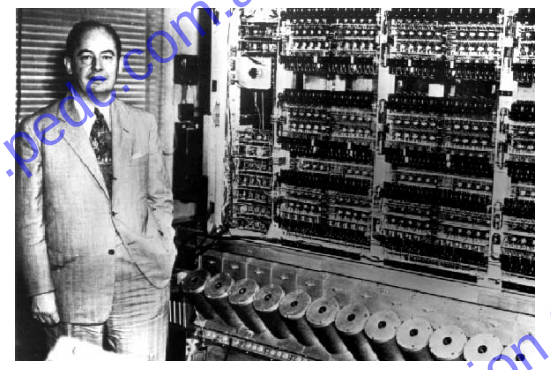


Fig 9.1
John von Neumann (1903-57) with ENIAC.

Imperative languages use sequencing, decisions and repetition as their main problem solving methods. Data and processing are separated with imperative languages. We create variables for data storage and then we perform processes on them. Each program has a beginning and a distinct end. This is not the only way of doing things.

Why would we want to do things differently? There are various motivations that have lead to the development of different paradigms. In the next section we consider some of these reasons.

LIMITATIONS OF THE IMPERATIVE PARADIGM

Although new imperative programming languages and user-friendly GUI interfaces have made the development of software an easier task, the resulting products are still performing the same tasks using essentially the same techniques. The difference is they are just doing it a lot faster! Imperative languages require the developer to understand all details of the problem and to be able to solve the problem completely. Many types of problems do not have precise solutions and developing an algorithm which solves such problems in not feasible.

The imperative or procedural paradigm on which languages such as Pascal, C, Fortran and Basic are based, restrict the developer in many ways. For instance, a function can only accept data as its inputs and it can only return data as its output. Individual programs are designed to solve a particular problem and they must be modified to solve related problems. The human brain can use the solution techniques used on past problems to assist in the resolution of new seemingly unrelated problems. As human beings we do not think in terms of data and processes; our brains are able to readily connect the two. The human brain is able to make inferences and deductions based on experience and intuition. Surely if we can create programming languages that can simulate the brain more accurately then productivity gains will result.

Difficulty solving certain types of problems

Computers were initially designed to solve mathematical and arithmetical problems. The finance and motivation for the development and production of early computers came from the military. This was soon followed by statistical analysis of census data and then by large business applications. These problems were best solved using mathematical techniques.

Many real life problems do not have definite answers or they have a variety of answers. Other problems can be solved using strategies that cannot be stated easily in strict mathematical terms. For example, doctors diagnose diseases in patients using symptoms and a certain amount of intuition gained from experience with other patients. Driving a car involves far more than knowing how to operate the controls. Good drivers are able to become an integral part of the vehicle; they sense potential dangers before they become problems. Communication between people involves multiple signals. Tone, volume and inflexions in voice have different subtle meanings; facial expressions and body language convey meaning. These types of problems are unable to be solved efficiently using imperative languages. A different set of rules governs the solution of these types of problem. A new programming paradigm that could help software developer's work towards solutions to these problems would be invaluable.

GROUP TASK Discussion

How would you explain precisely what a dog looks like to an alien? Make up a list of features that could be used to describe a dog. Would the alien then be able to differentiate a dog from a cat using your list? Can you program a computer to recognise a dog? In your experience are imperative languages suited to this problem?

The need to specify code for every individual process

The imperative paradigm requires that every part of the problem must be solved in complete detail before the final software will operate. For example, when using top-down design thorough testing cannot commence until the final lowest level subroutines have been written. Imagine the subroutine required to sort was not present within a database management system (DBMS). The whole application would fail even though this sort subroutine is a very minor part of the larger application.

Now imagine it was not necessary to specify details of every single process. Instead imagine the software could infer a suitable method of solution at runtime. If this were the case then software will be able to react to the changing needs of users without the need for modification. This is a significant aim of the logic paradigm and is why the logic paradigm is used to develop many artificial intelligence applications.

Difficulty of coding for variability

Many problems solved by software developers are similar in many ways or involve similar solution strategies. In chapter 4, we examined a number of standard algorithms for searching and sorting. There are countless other examples where programming tasks are repeated as part of the solution to multiple problems. Well-structured imperative programs can be designed so that modules of code can be readily reused; however this is not an integral part of the imperative paradigm. It would be preferable if code could be used to solve related problems without the need to be rewritten. This

reusability should be an integral part of the paradigm in much the same way as our brain uses past experience to solve new related problems.

The development of software products has now become one of the fastest growing industries. This was not the case a mere 20 to 30 years ago. It makes sense that software should be designed so that code can be reused in different contexts to solve different problems. Is there a paradigm that will encourage the reuse of code? If so, then the task of software developers will be simplified and the quality of the resulting products improved. Many other industries reuse components in a variety of contexts. For example, the padding in a lounge chair can also be used in the seat of a car or to provide sound- proofing in a studio. A 12-volt light bulb may be used in a torch, on a boat, in an airplane or in a car. Sheet aluminium is used to produce soft drink cans, lithographic printing plates and to line the outside of air transport crates. As software developers, we would benefit from programming languages that allowed us to reuse software components in a similar way.

GROUP TASK Discussion

Consider the problem solving methods you would use to plan a trip to a new location you have never visited before. Think about how you go about deciding on the best method of transport and then booking the trip so it will actually happen. Discuss aspects of your thoughts and actions that are known in every detail (imperative) and aspects which are not.

EMERGING TECHNOLOGIES

The first programmers had to program in machine language; a series of binary digits. The programmer was forced to think like the machine and the language was dependent on the processor. John von Neumann was 'using' his students to convert his code into binary machine language for input using punched cards. Apparently one of his students asked why the machine could not perform the conversion. Von Neumann replied that the resources of the computer where far too valuable to be wasted on such menial tasks. This must have made his students feel particularly valuable! Of course, eventually assembler languages emerged to perform this precise task. At the time, assembler languages were viewed as a way of removing the programmer from the technical aspects of implementation so they could focus on solving the problem. Compared to today's modern languages assembler is extremely primitive.

The ever-increasing speed of the technology underpinning computer hardware has allowed programming languages to further remove the programmer from the machine code. The emphasis is on creating programming tools to assist in the development of software products without the need for the programmer to directly interact with lower-level CPU processes. The final software product may not always be as efficient as a similarly developed machine code product, however its development is certainly a simpler and more intuitive process.

The imperative paradigm has evolved over the years from lower-level languages that directly accessed the computer's hardware functions. Computers are now powerful enough that different ways of viewing and solving problems can be utilised. In the next section, we look at some different approaches that are emerging to assist in the solution of problems.

GROUP TASK Investigation
The evolution of programming languages is often viewed in terms of generations; first, second, third, fourth and fifth. Using the Internet or other computer texts make up a table to explain the essential differences between each generation.

GROUP TASK Investigation
Hardware technologies have progressed at an astounding speed. Often the history of hardware is also viewed in terms of first, second, third, fourth and fifth generations. Make up a table outlining the major features of each of these hardware generations.

GROUP TASK Discussion
Examine your results from the above two investigations. Is there a correlation between the programming language generations and the hardware generations? Can you see how advances in hardware technologies have influenced software development? Discuss.

INTRODUCTION TO DIFFERENT PARADIGMS

The traditional imperative programming paradigm separates input, output, data, control and processing as distinct components. It is convenient to split computers into these components but is it the best or most appropriate way of doing things? The natural world, and in particular the human thought process, does not make this distinction. When we make a decision to get out of our chair and get a drink from the fridge we do use input, processing, output, data and control, however we do not separate them into distinct components rather we utilise all these components as a total integrated package. A fish is a complete system. Its component parts operate together transparently. The act of swimming combines input, processing, output, data and control into a network of actions culminating in the propulsion of the fish through the water. We need to consider new ways of combining and viewing components of computer systems. Perhaps there are better ways of integrating components that will assist software developers in the production of more 'natural' products.

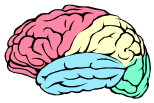
Consider the following:

Suppose we wish to organise a dinner party for four friends. We need to know who is able to come so we phone each of our four friends. The final guest list is data, however the processes involved in obtaining this data are quite involved. For instance, perhaps you wish to ask Sue but if Sue is able to come then it's not appropriate to ask Bill. The process of asking Sue involves knowing her contact details, then contacting her. How do you contact her? By phone or by mail? If you choose mail then time constraints arise. Will there be time to ask Bill if Sue declines the invitation? You need all the components required to write a letter; paper, pen, envelope and stamp. You need to know what to write, how to get to the post box, some idea of the time it takes for a letter to be delivered, etc… We could continue on and on, however the point is that each process we perform is made up of a collection of building blocks that are combined to perform some task. Most tasks include building blocks that themselves are comprised of other building blocks.

Let us try to develop a list of building blocks used when we make decisions and solve problems such as the one outlined above. The building blocks for a particular task include differing types of components. They include knowledge or things we know. They include facts as well as how these facts are linked. They also include methods for manipulating our knowledge. In imperative computing terms, we call these processes, functions or procedures. We also have rules to follow. Often these rules are not stated but are understood e.g. it is not appropriate to ask Bill if Sue is coming.

GROUP TASK Activity
We have a set of components made up of facts, rules, processes and physical components. Make up a table with these building blocks as headings. Using the dinner party scenario complete the table by listing items under each of the building blocks.

There are various ways of approaching the solution to a problem. Each of these requires a set of components or building blocks upon which the approach or paradigm is based. The imperative paradigm we are all familiar with is based on variables and control structures. In this section, we introduce two other paradigms; logic and object oriented. These paradigms are built on different ways of solving problems using different components. To explain the concepts underpinning each of these paradigms let us consider a particular problem. Be aware that we are examining one particular problem and not all the paradigms are particularly suited to this problem's solution; nevertheless our example will serve as a worthwhile introduction.



Consider the following:

The Towers of Hanoi is a well-known problem, which was first published in 1883 by the French mathematician Edouard Lucas. The object of the game is to move all the disks to a different pole. You can only move one disk at a time and a bigger disk can never be placed on a smaller disk. There are three poles and initially the disks are all on one pole.

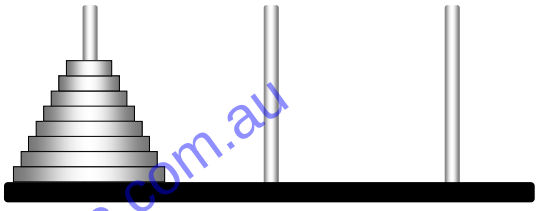


Fig 9.2
Initial set-up for the Towers of Hanoi problem using 8 disks.

The original problem allegedly contained 64 gold disks and it was thought that once all 64 disks had been moved, the universe would come to an end. The problem was to calculate an efficient manner of moving the disks to minimise the total number of moves. Once the total number of moves required is calculated then the life of the universe will be known.

Our aim in this section is not to come up with an answer but rather to illustrate different paradigms or approaches that could be used to arrive at a solution. Later in this chapter, we will examine how these paradigms are implemented using programming languages.

Imperative

If we were looking for an imperative method of solution we first need to consider variables. In our problem we would require variables to represent each pole, each disk and the pole on which each disk is currently sitting. We then require a series of control structures (sequences, decisions and repetitions) that could manipulate these

variables in such a way that the problem would be solved. Our choice of data structure used to represent each variable will have a large impact on the way we arrange our control structures into an algorithm, yet these two activities are separate.

We could create an array called Disk where for example Disk(3)=1 means the third smallest disk is currently on the first post. Another array called Post maintains the order of disks currently held on each post. Post(1,3)=5 might mean that disk 5 is currently on post 1 and is third from the bottom.

An algorithm is then created using control structures. The algorithm controls the movement of the disks to eventually solve the problem. When writing the algorithm we must have a precise understanding of how to solve the problem.

Logic

From a logic approach we focus on facts and rules. Some initial facts for this problem are that there are three poles, all the disks are on the first pole and each disk is a different size. The rules for this problem are that a larger disk can never be placed on a smaller disk, only one disk can be moved at a time and disks can only be moved from one pole to another.

Facts
- ⇨ There are three poles
- ⇨ Initially all disks are on the first pole
- ⇨ Each disk is a different size

Rules
- ⇨ Larger disks can't be placed on smaller disks
- ⇨ Only one disk can be moved at a time
- ⇨ Disks are moved from one pole to another

Fig 9.3
The logic paradigm is based on facts and rules. New unproven rules (Rules of thumb) develop from these facts and rules.

We could commence our exploration using the fact that all the disks are on the first pole, therefore our first move must involve moving the top disk. As a disk can only be moved from one pole to another, then this disk must be placed on the second or third pole. None of our rules help us to decide which pole so we take a guess and place it on the second pole. What move is next? Moving the second disk seems to make sense. It can't go on the second pole as it contains a smaller disk; one of the rules says this is not allowed. It can go on the third pole so we place it there. What options do we have now? Can we move the third disk? No. Can we move the smallest disk from the middle pole? Yes, it can move to either pole. Can we move the disk on the third pole? We could move it back to the first pole. Moving the smallest disk makes sense but to which pole? We consider each of these two possibilities. Eventually, moving the small disk to the first pole leads to a dead end so we continue down the other branch. We can learn from the dead ends and introduce 'rules of thumb' that will enable us to reduce the dead ends encountered.

GROUP TASK Discussion
The imperative and logic paradigms provide different approaches to problem solution. Discuss the essential concepts used in each of these paradigms.

Object oriented

Using an object oriented approach, involves considering each component of the problem as a self-contained unit. Each unit is able to remember things (data/attributes) and do things (methods/operations). There are two types of objects in this problem, disks and poles. We can approach the problem from the point of view of each of these objects.

If you were one of the disks then what do you need to know and what do you need to be able to do? Disks need to know how big they are, they need to know what pole they are on and if they are on top of the pole. They also need to ask poles if they can move to them and then know how to move to poles. The size of the disk, the pole it is on and whether it is on top is the data. Asking poles if they can move to them and moving to other poles are the methods.

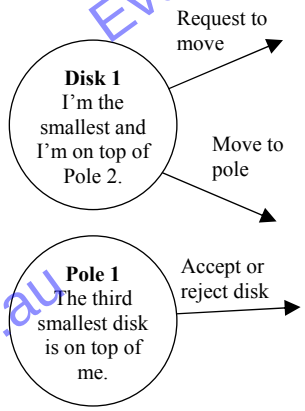


Fig 9.4
Diagrammatic view of the Disk and Post objects showing their data and methods.

Let us now look at a pole. Poles need to know what disks they have. In particular, they need to know the size of their top disk. This is their data. They also need to be able to reject disks trying to move to them that are larger than their top disk. This is a method.

The interaction of the disks and poles results in the eventual solution of the problem. Let us consider the interactions that may occur at the commencement of the solution using an object oriented approach. The top disk sends a message to the second pole asking if it can move there. The pole responds by checking its top disk. In this case, the pole tells the disk it can move. As a result the smallest disk moves to the second pole. The new top disk on the first pole realises it is now at the top so it attempts to move to the second pole. It is rejected. It then requests a move to the third pole and is accepted. The processing continues in this fashion until the final objective is achieved.

Our explanation above implies that methods are carried out in a sequential fashion. This is not the case. A number of disks can be simultaneously requesting moves. This is one of the aspects of the object oriented paradigm that makes it so powerful; each object is able to operate on its own. As soon as a disk reaches the top of a pole it can start asking other poles for permission to move. Each disk is in control of itself. Similarly, each pole may be receiving requests from a variety of disks. It is up to the pole to either accept or reject the requests.

GROUP TASK Discussion
CPUs designed using the von Neumann architecture process instructions in a strict sequence. How then can software written using object oriented programming languages be executed on these machines? Discuss.

GROUP TASK Research
The object oriented paradigm is the basis of many popular programming languages. Create a list of programming languages that are based on the object oriented paradigm.

SET 9A

1. A paradigm can best be described as
   (A) a new way of programming.
   (B) a philosophical or theoretical framework.
   (C) Darwin's theory of evolution.
   (D) a problem solving technique.

2. Which paradigm is based on variables and control structures?
   (A) Evolution
   (B) Logic
   (C) Imperative
   (D) Object oriented

3. Productivity can best be described as
   (A) a measure of the output produced from a given set of inputs.
   (B) the number of lines a programmer can write.
   (C) the collection and analysis of information.
   (D) a revised performance standard.

4. Which paradigm has evolved directly from lower level languages that directly accessed the computer's hardware functions?
   (A) Imperative
   (B) Object oriented
   (C) Logic
   (D) All of the above

5. Homer is writing a piece of code that is based on previously known facts and rules. Which paradigm would Homer most likely use?
   (A) Logic.
   (B) Functional.
   (C) Imperative.
   (D) Object oriented.

6. The imperative paradigm is also referred to as
   (A) the operational paradigm.
   (B) the repetitive paradigm.
   (C) the algorithmic paradigm.
   (D) the procedural paradigm.

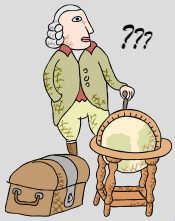
7. Which programming language is based on the object oriented paradigm?
   (A) Basic.
   (B) Fortran.
   (C) C++.
   (D) Pascal.

8. Roland owns a software development company that specialises in researching and creating family trees. Which paradigm would be most suited to specifying and determining relationships between people?
   (A) Functional.
   (B) Object oriented.
   (C) Logical.
   (D) Imperative.

9. What is another term commonly used for the von Neumann concept?
   (A) Stored program concept.
   (B) Data processing concept.
   (C) Sequential data concept.
   (D) Variable data concept.

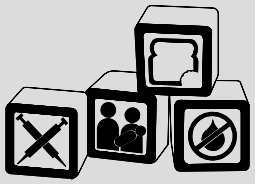
10. Imperative languages use which components for solving problems?
    (A) data and methods.
    (B) facts and rules.
    (C) attributes and rules.
    (D) variables and control structures.

11. For thousands of years, man believed the earth was flat. We now know that this is not the case. In effect a paradigm shift affecting the entire population occurred. Explain the term 'paradigm shift'. Use the 'flat earth' scenario to assist with your explanation.



12. Most modern computers are based on the 'Stored program concept'. John von Neumann initially developed this concept. What is the stored program concept and how has it influenced the evolution of programming languages?

13. The imperative, logical and object oriented paradigms utilise different sets of components or building blocks. List and describe the building blocks used with each of these paradigms.



Use the following scenario to answer questions 14 and 15.

A man died leaving all his money to be divided amongst his widow, four daughters and three sons. He stipulated that each daughter should receive three times as much as each son and each son should receive twice as much as their mother. If the exact amount left was $7936, how much should the widow receive?

14. Which paradigm seems the most suited to the solution of this problem? Justify your choice by explaining why the other paradigms would be less suitable.

15. Using the paradigm you selected in question 14, attempt to solve the above problem. Concentrate on the process used to solve the problem rather than just the final solution.

PARADIGM SPECIFIC CONCEPTS

In this section, we examine the logic and object oriented paradigm in more detail. The important features of each paradigm are examined, together with examples using programming languages based on the paradigm. The aim being to illustrate how each paradigm is implemented in a programming environment.

Before we commence, a more detailed understanding of the term 'paradigm' should serve as a worthwhile introduction. Understanding new paradigms involves a shift in the way we think; often the term 'paradigm shift' is used to describe this change. A paradigm shift can be described as a global change in thinking. To achieve a paradigm shift involves coming to a new realisation through new insights. This is often not an easy task, particularly when old ways of thinking about a problem are thoroughly entrenched. We, as humans thought the world was flat, we now know this is not the case. We thought the earth was the centre of the universe; of course it is not. These are examples where the entire population of the world has undergone a paradigm shift.

Consider the following:

Charles Darwin did not release his book 'Origin of the Species' for many years after he had developed and refined his theory of evolution. He was concerned about the reaction he would receive from the general public. When Darwin eventually did go public, the majority of the population were not only slow to accept his theory but were outraged. Darwin's supporters were publicly humiliated and ostracised. Darwin's theory involved a paradigm shift, a new way of looking at creation.

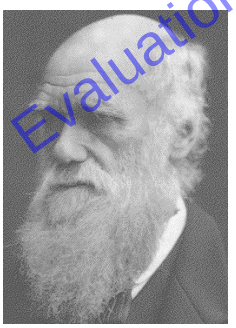


Fig 9.5
Charles Darwin's theory of evolution resulted in a paradigm shift.

Over time, a paradigm shift has occurred where evolution is now an accepted theory. As school students we are taught Darwin's theory of evolution almost as though it were fact; even the most sceptical of us accept it as a possibility.

GROUP TASK Discussion

Imagine if a new theory arose in regard to creation. How receptive do you think the population would be? Just because a paradigm is accepted this does not make it the only possibility nor is it necessarily the best. Discuss.

As software developers, we need to be able to choose a paradigm most suited to the current problem. Software development is primarily a problem solving activity; problem solving is a creative process requiring original thought and new realisations. Programmers need to equip themselves with a tool kit of ideas and strategies they can draw upon when embarking on the development of new products. Knowledge and understanding of programming languages that are based on alternative paradigms adds to this tool kit.

In the next section, you will need to have an open mind to the ideas presented. Try to view these new concepts in their own right yet balance their virtues with their short-comings. The aim is to enable you to select a programming language, or selection of languages, that provides the best tools to solve a particular problem. You will probably need to read much of what follows a number of times to grasp these concepts.

LOGIC PARADIGM

We saw in the previous section that the logic paradigm uses facts and rules as its basic building blocks. In this section, we extend our logic paradigm idea further to include concepts central to logical programming languages. The logic paradigm is used to develop programs in the area of artificial intelligence. Rather than just explain the concepts, we'll create a small program using Prolog and use this example to illustrate the concepts. Following this example, we examine expert system shells and how they can be used to create expert systems.

Prolog is perhaps the most common logical programming language hence examples throughout this section will use this language. Prolog is short for **PRO**gramming in **LOG**ic. Prolog was developed as a language in the early 1970s. One of the major influences on the nature of the Prolog language was the need to process natural language. Prolog is used primarily in the areas of artificial intelligence including natural language processing. There are various versions of Prolog in common usage. In this text, we will use 'Strawberry Prolog', a product developed by Dimiter Dimitrov Dobrev in Bulgaria. At the time of writing, a demonstration version was available for free download on the Internet. SWI-Prolog is another popular version which is freely available. For our purposes the particular implementation or version is not critical so we will just call it Prolog.

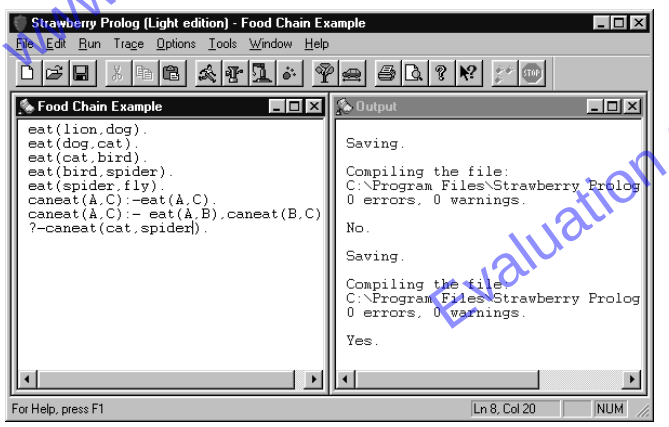
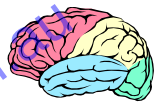


Fig 9.6
The Strawberry Prolog editor and user interface.



Consider the following:

Let us assume we have a simple food chain where lions eat dogs, dogs eat cats, cats eat birds, birds eat spiders and spiders eat flies. In Prolog we write the following facts:

```
eat(lion,dog).
eat(dog,cat).
eat(cat,bird).
eat(bird,spider).
eat(spider,fly).
```

In Prolog `eat` is called a predicate; to be more precise it is a user defined predicate. We just made up the name `eat`; we could have used `devour` or almost any other name. The animal's names are atoms. These are simple data items that are known in Prolog as atoms. Atoms must commence with a lower case character. A fact is a definite known item e.g. cats eat birds or in Prolog `eat(cat,bird).`

GROUP TASK Activity
Design a set of facts, using Prolog syntax, describing the components of a computer system as either input, processing or output components. Remember that some components are both input and output devices.

Querying these facts is simple. Say we wish to determine if a spider can eat a dog. We enter the line `?-eat(spider,dog).` When we run our program (or query), the Prolog inference engine responds with `No`, where as if we enter the query `?-eat(dog,cat)` the output is `Yes`. In Prolog a query is known as a goal, i.e. our goal was to find out if a dog could eat a cat. When we just have facts in our database and no rules then Prolog merely searches through the facts for a match. If a match occurs then our goal has been fulfilled and `Yes` is output. If no match is found then our goal fails and `No` is output.

Goal
A query that can result in either being fulfilled, in which case the result is Yes, or not being fulfilled, in which case the result is No.

GROUP TASK Activity
Write down two goals that will be fulfilled and two goals that will not. What do you think would be the result of the following goals?
`?-eat(pig,table)` `?-eat(dog,bird)`

Let us introduce a general fact into our program to say that everyone likes eating flies. Rather than having to write a long list of new facts, we can use a variable, say `X` to represent any value. Our new fact would be `eat(X,fly).` Variables must have an upper case first letter. Now if our goal is `?-eat(dog, fly)` or `?-eat(lion,fly)` or `?-eat(egg,fly)` the result will always be true.

Let us further generalise by introducing a rule into our system. Say that if a lion can eat a dog and a dog can eat a cat then a lion must be able to eat a cat. By continually applying this rule we see that all animals will be able to eat flies so we can remove `eat(X,fly)` from our facts. Generalising this statement we get something like this: If A can eat B and B can eat C then A can eat C, in Prolog we write this as: `eat(A,C):-eat(A,B),eat(B,C).` Notice that A, B and C are in upper case so they are variables. Facts and rules combine to form a knowledge base. During execution of a goal Prolog interrogates the knowledge base in an attempt to fulfil the goal.

Knowledge base
In terms of logical programming, a knowledge base is a database containing all the facts and rules.

As an example, we wish to know if dogs eat spiders. So we execute the goal `?-eat(dog,spider)` and it yields the result `Yes` as expected. How is Prolog executing this goal? Firstly, the list of facts is examined in search of a direct match; none is found. Next the rules are examined. In our example, only one rule exists. This rule allows us to reduce our goal into two sub-goals, `eat(dog,X)` and `eat(X,spider)`. We then examine `eat(dog,X)` and compare it to our facts. The first, and in this case the only match is `eat(dog,cat)`, the sub-goal `eat(dog,X)` is fulfilled. For our original goal to be fulfilled we require the sub-goal `eat(cat,spider)` to be fulfilled. Breaking this goal down using our rule, we get

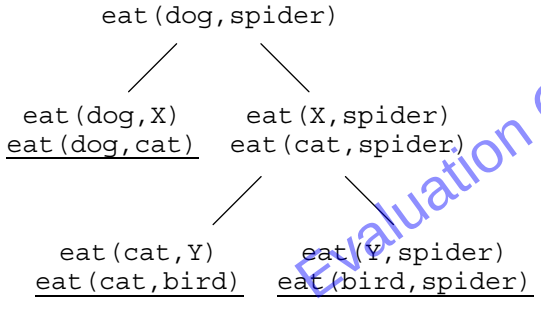


Fig 9.7
Tree showing a trace of the fulfilment of a goal.

Pages 374 to 393

Not included in this sample chapter

The Java keyword this is used to specify the current object. The keyword super is used to refer to public items in the super class. As constructor methods do not have specific names then using the keyword super on its own refers to a constructor in the super class. Methods within the super class could also be executed from within a method in the subclass using the super keyword. For example to execute the move() method in the Character super class from within a method in the Baddie class we would write super.move().

Our constructor for the baddie class is sufficient for our needs, however we'll often need to create goodies with agility and health equal to 5 and also goodies where all attributes have different initial values. We add two constructors to the goodie class which pass on the agility and health inputs to the character class constructor.

```
public class Goodie extends Character {

    private int personality;
    private int intelligence;

    public Goodie(int a, int h, int p, int i){
       super(a, h);
       if (p<0 || p>10)
           p = 0;
       this.personality = p;
       if (i<0 || i>10)
           i = 0;
       this.intelligence = i;
    }
    public Goodie(int p, int i){
       this(5, 5, p, i);
    }
    public void getTreasure(){
    }
}
```

The statement Goodie mrNice = Goodie(4, 7, 3, 8) creates a goodie object called mrNice where agility=4, health=7, personality=3 and intelligence=8. Don't be too concerned about the detail of the Java syntax. In examinations we are more concerned about understanding the concepts of OOP rather than learning a particular language.

GROUP TASK Discussion
Why aren't constructors required when using imperative languages? What techniques are used to perform a similar function to constructors in imperative languages? Discuss.

GROUP TASK Discussion
Each class could also contain a destructor. In Java destructor methods are called finalize() and in C++ using a tilde "~" followed by the class name. What do you think is the purpose of a destructor?

Examine the goodie class definition above. To those of us familiar with imperative languages, the fact that the goodie class now contains two methods with the same name should be of concern. i.e. we have two methods (or functions) called Goodie within the goodie class definition. Although in this example the Goodie methods are constructors it is possible for any method to have multiple definitions within a single class and not only is it possible it is also very common. In object oriented programming this is a valuable and deliberate inclusion in all OO languages. The

compiler recognises different methods and functions not only by their names but also by the number of parameters and the type of those parameters. This concept is known as *overloading*. In our Goodie class the first version of Goodie contains four integer parameters a, h, p and i whereas the second contains just two integer parameters named p and i.

Overloading
Methods which have the same name but accept a different number of parameters or parameters of different types.

Let us expand our concept of overloading by considering how an object's methods are described within a program. We'll also learn how we can get our characters to actually perform their methods. Let's consider the getTreasure method which is part of each of the goodie characters. Usually when a goodie gets some treasure he feels better and more confident and as a consequence, his health and personality are both incremented by 1. Some treasure items scattered throughout the game are nasty; they can cripple a goodie causing their agility to be set to 0.

Within the Goodie class we write two different versions of our getTreasure method. Both versions have a single parameter however one is an integer and the other a char. Here's our first attempt:

```
public class Goodie extends Character {

    private int personality;
    private int intelligence;

   ...

   void getTreasure(int a) {
       super.health++;        //this line causes an error
       this.personality++;
   }
   void getTreasure(char a) {
       super.agility = 0;        //this line also causes an error
   }
};
```

Two errors occur as we are unable to access the health and agility attributes from within the Goodie class. This is because they are encapsulated within the parent Character class. We need methods within the parent Character class which allow us to set the value of these attributes. We note that the data validation logic required is the same as was used in the Character class constructor; hence some reworking of the code is performed to reduce repetition and improve maintainability of the code. Later we would like to have an array of characters so it makes more sense to store the name of each character as an attribute of each object. In addition a simple print method is written to assist with testing. The updated character class is as follows:

```
public class Character {
   private String name;
   private int agility;
   private int health;
   public Character(String n, int a, int h) {
      this.name=n.trim();
      this.setAgility(a);
      this.setHealth(h);
    }
```

```
    public void setAgility(int a){
        if (a<0 | | a>10)
            a = 0;
        this.agility = a;
    }
    public int getAgility(){
        return this.agility;
    }
    public void setHealth(int h){
        if (h<0 | | h>10)
            h = 0;
      this.health = h;
    }
    public int getHealth(){
        return this.health;
    }
    public String getName(){
        return this.name;
    }
    public void move() {
    }
    public void fight() {
    }
    public void print(){
        System.out.println("My name is "+this.name);
        System.out.println("Agility = "+this.agility);
        System.out.println("Health = "+this.health);
    }
}
```

The Goodie class is also updated to include more robust getTreasure() methods and to add a print() method to assist with testing:

```
public class Goodie extends Character {
    private int personality;
    private int intelligence;
    public Goodie(String n,int a, int h, int p, int i){
        super(n,a,h);
        if (p<0 | | p>10)
            p = 0;
        this.personality = p;
        if (i<0 | | i>10)
            i = 0;
        this.intelligence = i;
    }
    public Goodie(String n, int p, int i){
        this(n, 5, 5, p, i);
    }
    public void setPersonality(int p){
        if(p<0 | | p>10)
          p = 0;
        this.personality = p;
    }
```

```
    public void getTreasure(int a){
        int h = this.getHealth();
        this.setHealth(h++);
        int p = this.personality;
        this.setPersonality(p++);
    }
    public void getTreasure(char a){
        this.setAgility(0);
    }
    public void print(){
        System.out.println("Goodie");
        super.print();
        System.out.println("Personality = " + personality);
        System.out.println("Intelligence = " + intelligence);
    }
}
```

Suppose someone is playing our adventure game and they are controlling MrNice. They come across a particular piece of treasure so they press the appropriate key. Somewhere in your code the command mrNice.getTreasure(188); gets executed (the value 188 could have been any integer). This statement sends a message to the getTreasure method of the mrNice object. Because 188 is an integer rather than a character, MrNice's health and personality are incremented. If the command had been MrNice.getTreasure("k") then MrNice's agility would have been set to 0. The compiler sees the two getTreasure functions as different because their parameters are of different types; one is an integer (int) and the other a character (char). When using goodie objects to code other parts of our game, we just view the getTreasure method as a single method; we say that getTreasure is *overloaded*.

Overriding
Methods (or properties) which replace an inherited method (or property).

Overriding is a further OOP concept related to overloading. Overriding occurs when a subclass includes a different version of a method (or attribute) which it inherited from its parent class. The version within the subclass replaces or overrides the inherited parent version. Overriding only occurs when the number and type of parameters is the same in both versions. Overriding allows the programmer to change the behaviour of subclass objects compared to the behaviour of the parent class. For example, our Character class contains a move() method which is currently inherited by both goodies and baddies. If we decide we'd like baddies to move differently then we can write another move() method within the baddies class. This means that whenever a baddie moves the code within the baddie class will execute. However when goodies move the original inherited move() code from the parent Character class will continue to be executed.

GROUP TASK Discussion
Examine the above class definitions to locate and distinguish between examples of overloading and of overriding.

GROUP TASK Activity
Determine the output from the following lines of code:

```
Character myGoodie = new Goodie("Luke", 6, 8);
myGoodies.getTreasure(123);
myGoodie.print();
```

Changes are also made to the Baddie class. The updated Baddie class definition is reproduced below:

```
public class Baddie extends Character {
   private int aggression;
   private int badType;
   public Baddie(String n, int a, int b) {
      super(n, 5, 5);
      if (a<0 | | a>10)
         a = 0;
      this.aggression = a;
      if (b<0 | | b>10)
         b = 0;
      this.badType = b;
   }
   public void makeNoise() {
   }
    public void print(){
      System.out.println("Baddie");
      super.print();
      System.out.println("Aggresion = "+aggression);
      System.out.println("BadType = "+badType);
   }
}
```

GROUP TASK Discussion

Currently code similar to the following lines are repeated in each of the Character, Goodie and Baddie classes:

```
if (a<0 | | a>10)
    a = 0;
```

Suggest modifications so this code only appears once.

GROUP TASK Activity

Currently if the new value for an attribute is outside the range 0 to 10 the attribute is set to 0. Alter the code so that new values less than 0 cause the attribute to be set to 0 and new values greater than 10 cause the attribute to be set to 10.

The final OOP concept we need to discuss is *polymorphism*. Generally *polymorphism* is the ability to appear in more than one form. Poly means many and morph refers to different forms of an object. A butterfly can be described as polymorphic; it was once in a different form, that of a caterpillar and before that it was an egg. The butterfly is still the same species of insect; it just takes on various different forms or morphs.

In terms of OOP, *polymorphism* refers to the ability of the same command to process objects from the same super class differently depending on their subclass. At runtime the system chooses the precise method to execute based on the subclass of each particular object being processed. Polymorphism allows programmers to process objects from a variety of different subclasses together efficiently within loops. There are many other advantages of polymorphism which help to reduce the

Polymorphism

The ability to appear in many forms. In OOP this means at runtime a method can process data differently depending on the circumstances.

complexity of code. Because at runtime the system decides which particular method to execute then the programmer does not need to include decisions (often "if" statements) within their code to make such decisions. This results in cleaner and more maintainable code and also faster code execution because the decisions are being made by a built-in part of the system rather than the programmers logic.

GROUP TASK Discussion
Polymorphism is not an easy concept to understand. Create a list of animals that could be viewed as polymorphic. Explain your answers.

As an example of polymorphism consider the execution of the following Java code:

```
1    Character[] myCharacters = new Character[3];

2    myCharacters[0]= new Goodie("Mr Nice",8,2,3,4);
3    myCharacters[1] = new Baddie("Grizzly",5,6);
4    myCharacters[2] = new Character("Crazy Tree",7,8);

5    for(int i=0; i<3; i++) {
6       myCharacters[i].print();
7    }
```

The above code produces the following output:

```
Goodie
My name is Mr Nice
Agility = 8
Health = 2
Personality = 3
Intelligence = 4

Baddie
My name is Grizzly
Agility = 5
Health = 5
Aggresion = 5
BadType = 6

My name is Crazy Tree
Agility = 7
Health = 8
```

In line 1 `Character[] myCharacters` is telling the compiler that the variable `myCharacters` will hold an array of objects of type `Character`. The right hand side code `= new Character[3]` creates an array of `Characters` with 3 elements and initialises the variable `myCharacters` with this array. In Java the 3 elements are indexed beginning with 0.

Line 2, 3 and 4 are instantiating and assigning actual objects to each of the array elements. Note that both `Goodie` and `Baddie` objects are also `Character` objects because both `Goodie` and `Baddie` classes inherit from the `Character` class. It is true to say that a `Goodie` is a `Character` and also that a `Baddie` is a `Character`. Because this is true we are able to store `Goodie`, `Baddie` and also `Character` objects within variables declared to be of type `Character`. This is critical to understanding the concept of polymorphism.

Lines 5, 6 and 7 iterates through the `myCharacters` array executing the `print()` method for each object. This is where the benefits of polymorphism can be seen. If the object is a `Goodie` then the `print()` method from the `Goodie` class executes, if the object is a

Baddie then the Baddie class print() method executes and finally for Character objects the Character.print() method executes. These decisions about the most appropriate method to execute occurs despite the array being initialised to hold Character objects.

GROUP TASK Activity
Work through the code above to confirm the output is indeed correct. Pay particular attention to the decisions made by the runtime system as it decides which print() method should be executed.

GROUP TASK Discussion
Inheritance, abstraction and overriding are critical to the operation of polymorphism. Discuss the connection between inheritance, abstraction, overriding and polymorphism.

HSC style questions:

Question 1.

Consider the following in relation to object-oriented paradigm concepts.

All cars, buses and bicycles are vehicles. All vehicles have travelled a certain distance since new, have a particular number of wheels and seats, use some type of fuel, are able to move and can turn.

However, moving and turning cars and buses is quite a different process to moving and turning a bicycle. As a vehicle moves it travels a certain distance, and the distance travelled since new is stored within the vehicle's odometer. The only way to alter the value in the odometer is to move the vehicle.

Based on the above information:

(a) Describe an example of a class including its data and methods.

(b) Identify an example of inheritance and use your example to explain how inheritance improves the productivity of programmers.

(c) Describe an example of polymorphism.

(d) Identify an example of encapsulation and explain how encapsulation improves the ability to reuse code.

Suggested solutions

(a) A class could be created called Vehicle which would include the data (or attributes) Wheels, Seats, Fuel and Odometer. Methods (or operations) would include Turn and Move.

(b) Three sub-classes (or child classes), say Car, Bus and Bicycle could inherit data and methods from the parent class Vehicle. This reduces the need to write the sub-classes from scratch as the code within the Vehicle class is reproduced within the sub-classes. In addition, once the code has been thoroughly tested it can be relied upon each time it is reused. The Bicycle class would have its own distinct Move and Turn methods which would override those inherited from the parent Vehicle class.

(c) Assume one object from each of the classes Car, Bus and Bicycle has been created (instantiated). As all objects are Vehicles then the methods Move and Turn can be used to manipulate all three. However at runtime the Move method for the Bicycle object performs quite different processing to the Move method of the Car and Bus objects. Similarly for the Turn method. Both the Move and Turn methods are said to be polymorphic as the system chooses the appropriate method to execute at runtime.

(d) Encapsulation means the value of data (attributes) of an object can only be altered by that object's methods. In the example the Odometer attribute can only be altered by the Move method of that object. Encapsulation allows programmers to create robust objects that are suitable for reuse. As data can only be changed by methods the programmer can test and validate all inputs thoroughly within the objects methods before any of the object's data is changed. This means unexpected inputs can always be dealt with appropriately by the object rather than causing errors to occur within the larger program.

Question 2.

The following Java code defines 3 classes:

```
class Animal {
  private String name = "not set";
  private static int numAnimals = 0;
  public Animal() {
     numAnimals++;
  }
  public void setName(String n) {
     if (n.length() != 0) name = n;
  }
  public String getNumAnimals() {
     return (numAnimals + " Animals");
  }
  public String getName() {
     return (name + " is an animal");
  }
}

class Bird extends Animal {
  private static int numBirds = 0;
  public Bird() {
     super();
     numBirds++;
  }
  public String getNumBirds() {
     return (numBirds + " Birds");
  }
  public String getName() {
     return (super.getName() + " and a bird");
  }
}

class Parrot extends Bird {
  private static int numParrots = 0;
  public Parrot() {
     super();
     numParrots++;
  }
  public String getNumParrots() {
     return (numParrots + " Parrots");
  }
  public String getName() {
     return (super.getName() + " and a
parrot");
  }
}
```

The following Java code is executed:

```
class MyAnimalApp {
  public static void main(String[] args) {
     Animal a = new Animal();
     a.setName("Dumbo");
     System.out.println(a.getName() );
     Animal b = new Bird();
     b.setName("Flighty");
     System.out.println(b.getName() );
     Animal p1 = new Parrot();
     p1.setName("Cracker");
     System.out.println(p1.getName() );
     Parrot p2 = new Parrot();
     p2.setName("Polly");
     System.out.println(p2.getName() );
     System.out.println(p2.getNumAnimals() );
     System.out.println(p2.getNumBirds() );
     System.out.println(p2.getNumParrots() );
  }
}
```

The following output is produced when the above code is executed:

```
Dumbo is an animal
Flighty is an animal and a bird
Cracker is an animal and a bird and a parrot
Polly is an animal and a bird and a parrot
4 Animals
3 Birds
2 Parrots
```

(a) Identify an example of each of the following within the above code.

(i) Class

(ii) Instance of an object

(iii) Class variable (or attribute)

(iv) Instance variable (or attribute)

(v) Method

(vi) Instantiation

(b) The code below only occurs within the Animal class definition, however it is executed using objects of type Animal, Bird and Parrot.

```
public void setName(String n) {
    if (n.length() != 0) name = n; }
```

Identify and outline the object oriented concept that makes this possible.

(c) Identify and describe an example of *polymorphism* within the above code.

(d) Identify examples within the code that illustrate the concept of *encapsulation*.

Suggested solutions

(a) *Only one of each of the following is required*

(i) Class: Animal, Bird, Parrott, MyAnimalApp

(ii) Instance of an object: a, b, p1, p2

(iii) Class variable (or attribute): numAnimals, numBirds, numParrots

(iv) Instance variable (or attribute): name

(v) Method: Animal(), setName(String n), getNumAnimals(), Bird(), getNumBirds(), Parrot(), getNumParrots(), getName(), main(String[] args)

(vi) Instantiation: new Animal(), new Bird(), new Parrot()

(b) Inheritance is the OO concept that makes this possible. The super class Animal implements the method setName(String n). The Bird and Parrot classes are subclasses of the Animal super class, as Bird extends Animal and Parrot extends Bird. This means that the subclasses inherit all the attributes and methods of their super class (Animal) which includes the setName (String n) method.

(c) An example of polymorphism is the getName() method found in the subclasses Bird and Parrot. Although this method has the same name (and call signature) in each Class it performs different processing. The getName() method is found in the super class Animal, but the Bird and Parrot subclasses override and redefine this method, to create their own implementation. For example, Bird redefines the getName method to call the super class (Animal) implementation of getName and then also perform additional operations.

In addition Parrot is a Bird and Bird is a Animal, therefore both Bird and Parrot are also Animal objects. This means that a, b and p1 can be declared to be Animals and then assigned objects of either Animal, Bird or Parrot type. When the overridden method getName() is executed the version of the method from the appropriate class is executed. This is an example of polymorphism in action.

(d) Examples illustrating encapsulation are the private attributes in the Animal, Bird, and Parrot classes combined with methods being the only way to alter these attributes.

For example, the numBirds class variable within the Bird class. This variable has the "private" identifier meaning that the variable can only be accessed through a public method within the class. Continuing with this example, this variable can only be altered using the constructor (initialisation method) Bird() and to get the value you need to use the method getNumBirds(). This is designed to protect the attribute from being altered and accessed inappropriately from code outside the class, thus protecting the integrity of the attribute's data.

Similarly the name instance variable present in the Animal class, which is inherited by the Bird and Parrot classes, cannot be altered or accessed directly. One must use the setName() method to alter or the getName() method to access the name instance variable. The setName() method performs some limited validation on the value passed to the method before the actual name variable is altered. This protects the attribute from inappropriate changes.

Notes

- This question includes class variables and instance variables which are both types of attributes. A class variable exists once for each particular class therefore when an object changes a class variable the changed value is reflected across all objects of that class. Instance variables exist within each individual object. That is each object has its own set of instance variables which are separate to other object's instance variables. If an object changes the value of one its instance variables this has no effect on the value of any other object's variables.

GROUP TASK Activity
Perform a desk check of the code in question 2 to ensure you are able to reproduce the output provided in the question.

GROUP TASK Activity
Enter the Java code from the question into a Java editor. Create further classes for Finches, Mammals, Monotremes and Marsupials which include similar attributes and methods as the existing classes. Create an array of at least 10 animals of all different types and efficiently print out the data returned by each animal's getName() method.

SET 9C

1. What is the term used to describe programs that react to particular user inputs?
   (A) Object oriented.
   (B) Event driven.
   (C) Procedural.
   (D) Functional.

2. An object's data is hidden and can only be altered via one of its public methods. This is a description of
   (A) encapsulation.
   (B) abstraction.
   (C) inheritance.
   (D) polymorphism.

3. Abstraction includes the process of
   (A) hiding an object's data and processes from it's environment.
   (B) coding a précis or summary of the objects.
   (C) selecting the common features of objects and procedures.
   (D) taking on the attributes of another object.

4. In terms of OO programming, what provides the interface that allows objects to communicate with other objects?
   (A) The object's attributes.
   (B) The object's data items.
   (C) The object's class.
   (D) The object's methods.

5. The term Class can best be described as
   (A) a category of objects.
   (B) a category of attributes.
   (C) a category of methods.
   (D) a category of processes.

6. Inheritance is the ability of objects to take on the characteristics of
   (A) child classes.
   (B) sub classes.
   (C) parent classes.
   (D) Both (A) and (B).

7. A constructor is used to
   (A) initialise an object during its creation.
   (B) assign the object to its applicable class.
   (C) create the object's class definition.
   (D) None of the above.

8. Overriding can be described as
   (A) a method with the same name but a different number of parameters as another method.
   (B) the ability to inherit the attributes of another class.
   (C) A subclass method with identical name and parameters as its parent class.
   (D) the ability of an object to create a second mirror-image copy of itself.

9. Creating an object based on a class is called
   (A) instantiation.
   (B) abstraction.
   (C) polymorphism.
   (D) encapsulation.

10. Which of the following can be described as an object oriented language?
    (A) Java.
    (B) C++.
    (C) Smalltalk.
    (D) All of the above.

11. Objects contain both attributes and methods. It is not possible to alter an object's attributes directly from another object. What term is used to describe this concept? Wouldn't it be better if attributes could be altered directly? Discuss.

12. A frog progresses through three stages. First it is an egg, then a tadpole and then finally a frog. Although all forms are the same species and are genetically identical they appear and behave quite differently. Which object oriented concept could be used to describe this scenario? Give an example of the use of this term as it is used in OOP.

13. Inheritance, encapsulation and abstraction are all vital aspects of all object oriented languages. Explain the meaning of each of these terms. How do these concepts improve the reusability and maintainability of code?

Use the following scenario to answer questions 14 and 15.

A program is being developed using an object oriented programming language. This program is designed for use by pre-school children. A simulated adventure playground is the backdrop for the game. The aim of the game is to try to please both the children and the animals within the playground. The pre-schooler does this by moving children and animals to more favourable positions within the playground. The interaction of items provides the incentive for children to rearrange items into more appropriate positions. There are three different types of item within the playground, animals, play equipment and children. Individual items can interact both with the user and with other items. For example, a child may play on a piece of play equipment or a dog could look at the user and bark. Children and animals are able to move freely around the playground, play equipment cannot.

14. Develop a hierarchical system of classes that could be used within this project. Describe the attributes and methods you envisage may be required within each class.

15. Create a set of at least six objects that might exist within the playground. Write down the initial attributes of each of your items. What will you use to determine the result of interactions with other objects? Explain your answer using examples.

PROGRAMMER PRODUCTIVITY

The overriding motivation for introducing programming languages based on different paradigms is to increase the productivity of programmers. Different problems require different sets of tools to enable the production of efficient and reliable solutions. The effect of using languages more suited to particular tasks has an effect on productivity. Some of these affects will improve productivity and others may reduce productivity. Management must weigh up the positive and negative effects when deciding on a choice of language to be used for particular projects. Some areas influencing productivity include:

- speed of code generation
- approach to testing
- effect on maintenance
- efficiency of solution once coded
- learning curve (training required)

Let us consider these areas in more detail.

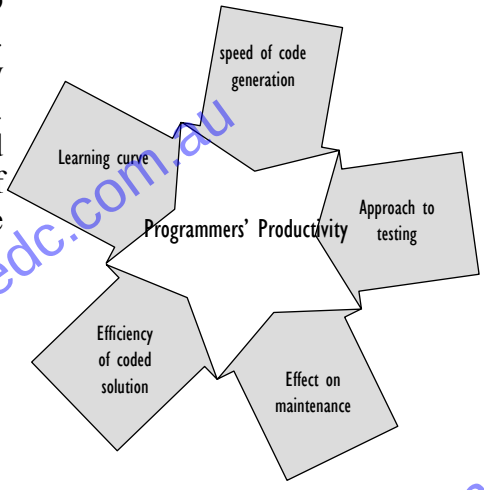


Fig 9.17
New languages affect programmers' productivity in various ways.

Speed of code generation

The speed at which code can be generated is a traditional method of measuring a programmer's productivity. Many software companies still pay programmers based on the number of lines of code they write. Thankfully, this method of payment has been largely phased out; encouraging programmers to write long-winded source code is certainly not the way to develop efficient and elegant programs. Nevertheless, the speed at which a programmer can create code to solve a problem is a valuable measure of productivity.

Programming languages that increase the speed of code generation must increase the productivity of programmers. Similarly, using a language based on a more suitable paradigm will also increase the speed of code generation. Different problems are suited to solutions using different paradigms. Choosing the most suitable paradigm can make the process of software design and code generation more efficient and will result in a more elegant and useable final solution. Let us examine an example to illustrate:

Consider the following:

A software development company is working on a new computer game called *Teenager's Revenge*. This game has various characters that interact with each other; each character is either a teenager or an adult. The teenagers possess qualities in common and similarly the adults have qualities in common. Teenagers are able to argue with adults and adults are able to attempt to reason logically with teenagers. Adults are split into sub-types; parents, relatives, teachers and friends. Of course, the teachers are the most understanding and logical, whereas the parents are the least logical but the most responsible from the teenager's viewpoint.

This product suits development using an object oriented paradigm. Each character within the game would be an object with data and methods. For example, a parent would be an adult object containing low levels of logic combined with a high level of responsibility and a medium level of understanding. Each parent object would have a method attached to explain how they reason with teenagers of different types. Teachers would be adult objects with high levels of understanding and logic and medium levels of understanding. They too would have methods attached to describe how they interact with teenagers. Similar objects could be created for each of the teenage characters.

GROUP TASK Discussion
Could the imperative or logical paradigm be used to develop the above product? Why would these paradigms be less suitable than the object oriented paradigm?

Approach to testing

In chapter 6, we examined the testing phase of the software development cycle. Much of what we learnt is relevant to testing software produced using languages based on any of the paradigms.

Unit testing, or testing individual modules, is perhaps where the largest productivity gains can be made. This is particularly the case when using object oriented languages. These languages force programmers to write self-contained classes. These classes (or the objects produced) can be thoroughly tested without requiring reference to other components. Imperative languages do not force programmers to develop such self-contained units. Classes that have been thoroughly tested can be treated as black boxes; the internal processing details need not be retested.

Once a class has been created it is possible to create new classes that inherit the attributes and methods from the original. This ability means that much of the code produced within new software projects will be tried and tested code from previous efforts. The previous section on OOP explained these concepts in more detail. As a consequence, testing is simplified resulting in productivity increases.

It is difficult to discuss testing without mentioning test data. As we said in chapter 6, the creation of test data sets is a vital stage of the testing process. Languages that can encapsulate their data in such a way that it cannot be altered by other processes will greatly reduce the magnitude and breadth of the final test data sets. OOP languages discussed in the previous section not only encourage data encapsulation, in some cases they enforce it.

GROUP TASK Discussion
Explain how encapsulation and inheritance can assist in the process of testing the source code of a newly developed software product.

GROUP TASK Discussion
Testing should be an ongoing process throughout the development of a software product. In what ways do each of the new paradigms assist in the ongoing testing process? Discuss.

Effect on maintenance

Maintenance is primarily about the ability of the code to be modified to meet changed requirements. To modify code requires locating the section of code that requires modification and then actually implementing the changes within the code. This requires thorough documentation throughout the software development cycle but particularly at the design and implementation stages.

Isolating the precise location of the code to be altered can be a time consuming and difficult task, particularly for large applications. Languages that force programmers to modularise their work greatly assist maintenance programmers to locate particular procedures and functions within the source code. Object oriented languages do this well, imperative and logical languages certainly provide the facility but it is not enforced. Ultimately no matter what language is used, programmers can write well-structured, elegant and readable code and they can also write rubbish.

Large software companies are reluctant to develop products using languages that are not widely understood. This is a reasonable and responsible reaction given the limited number of programmers available to maintain the code. If a large bank decided to convert all its applications to a new language then they must ensure maintenance personnel are available to maintain the product.

Robust, bug-free code is the desire of all software companies for it minimises maintenance issues. Many of the concepts we have covered in this chapter encourage or enforce the development of such code. Logical languages such as Prolog operate at a higher conceptual level thus removing the programmer from the technical details at the CPU level. The responsibility for managing memory and CPU operations is removed from the programmer and handed to the language and it's developers.

GROUP TASK Discussion

In terms of fault maintenance, what are the advantages and disadvantages of using higher-level languages? Justify your answers.

Efficiency of solution once coded

Efficiency of software once it is coded is often measured in terms of speed. How fast can an application retrieve a record, process it and store the result? How many CPU cycles does it take for a word processor to prepare a ten-page document for printing? These types of tests are designed to test the efficiency of the final solution.

Imperative languages are based on the von Neumann architecture. They have evolved in line with the developments in hardware technologies. Many of these languages e.g. C enable the programmer to work closely with the CPU's operations. Well written C programs will out-perform most similar programs written using OOP or logical programming languages. Unfortunately, this is a consequence of the development and evolution of hardware technologies rather than software technologies. Essentially, languages using other non-imperative paradigms are, in a sense, crippled by the hardware. They are not able to compete on an even playing field.

The good news is that these languages can considerably reduce development times. In addition, hardware, although not designed for these paradigms, is now capable of executing applications at such a speed that efficiency concerns are often of reduced importance. This is particularly true of many popular OOP languages such as Java and C++.

GROUP TASK Discussion
Explain the reasons why applications developed with Prolog are often unable to operate with the speed of similar products produced with more traditional languages.

GROUP TASK Research
Java and also .NET languages use an intermediate code which then runs on a virtual machine. Research to determine advantages and disadvantages of intermediate code such as Java's bytecode and .NET's CIL code.

Learning curve (training required)

Programming languages based on the logical programming paradigm (and other paradigms such as the functional paradigm no longer studied in SDD) have not gained wide acceptance amongst the general software development community. They are often viewed as obtuse and specialised. Many programmers view them as interesting oddities and hence they are only used in specialised areas. It can be argued that intensive and ongoing training and marketing is required to lift the status of these languages. It is a difficult task to thoroughly learn languages based on new paradigms and often the learning curve is steep. The benefits however are often greater than first envisaged.

Object oriented languages, on the other hand, are accepted and used by a large proportion of software development companies. Most universities now teach object oriented languages as part of their computing and engineering degrees. It is true that the languages we first learn are the ones with which we feel most comfortable. Because many programmers are experiencing and learning OOP techniques early in their careers these languages have gained wide acceptance.

Try searching the internet for tutorials on particular imperative, logical, object oriented and functional languages. You'll most likely find hundreds on C, Basic, Pascal, Fortran and various other imperative languages. You'll find an almost equal quantity on C++, Java and other OOP languages. What about Prolog, or functional languages such as APL, Haskell or Lisp? Your search will of course reveal them, but specialist interest groups will have written most with not many available as commercial material.

It is not an easy task to learn any new programming language, but when the paradigm on which it is based is also new the problem is compounded. There is no doubt that software developers and the greater community will benefit if they can find the time to examine and learn new ways of doing things.

GROUP TASK Discussion
Imagine you are a member of a logical programming language's international interest group. What recommendations would you make to this group in regard to training new users? Discuss.

GROUP TASK Discussion
Many of the concepts introduced in this chapter are new and require concentration and perseverance to master. Share your experiences with the class. Use the classes combined input to develop a set of training recommendations for future students doing this topic.

HSC style questions:

Question 1.

Imperative languages use control structures of sequence, selection and repetition together with local and global variables. Explain how logic languages such as Prolog achieve the same outcomes without all of the control structures, local and global variables.

Question 2.

(a) Describe ONE significant reason for the development of the *object oriented* paradigm.

(b) Explain the use of *encapsulation* within the *object oriented* paradigm and provide an example.

(c) Discuss features of the *object oriented* paradigm that affect the programmers' productivity with particular reference to speed of code generation.

Question 3.

A scientist is trying to classify various rock materials. Many experiments have been performed to determine the features and characteristics of each sample. Features and characteristics include colour, atomic weight and structure, abundant elements plus location found.

With all of this information the scientist has discovered that it possible to classify any rock material into one of the categories.

The scientist wants his assistants to be able to carry out the classification process without him present and therefore requires a system to be developed which accepts the necessary information about the sample and classifies it.

Recommend the most appropriate paradigm for the solution of this problem. Explain why other paradigms would be less appropriate.

Suggested solutions

Question 1.

Imperative paradigms use the repetition control structure to repeat statements whereas the logical paradigm uses recursion. The predicate is called initially and for each subsequent call the argument is modified until it reaches the termination condition. Logical languages have a stopping (or terminating) condition in their code. Selection in Prolog is implemented by having different versions of each predicate which will match with different facts or rules.

With regards to variables, imperative languages allow the use of global variables resulting in a situation where a value can be seen and hence modified anywhere within the program. With logical languages values can only be seen in a predicate if they are passed as parameters. Prolog variables are used to match or bind with known facts during the goal seeking process. This is different to imperative language variables which are assigned values specifically by the programmer or by the detailed logic of the solution.

Question 2.

(a) One significant reason for the development of OOP was that it simplifies the reuse of code. This reduces repetitive programming tasks and is achieved through the use of classes of objects and their associated data and methods. Once a class has been defined it can be reused in any programming project without the need to redefine the data structures and operations.

(b) The principle of encapsulation is essential to the OOP. It establishes a firewall between the user of an object and the code implementing it, thus achieving information hiding. In particular the user does not need to know how the data or attributes of an object are defined. If we had a Stack class with pushing and popping operations the stack could be implemented as an array, a list or whatever, since the representation of the Stack is unknown outside of the object. The only way to interface with the object is via the methods.

(c) OOP increases programmers' productivity because of the inheritance feature. Once a class has been defined and tested it can be used by the programmer. If the programmer wishes to extend the class into a sub class they do not need to rewrite, and hence test, all of the methods needed to operate an object of the sub class but can call the methods from its super or parent class, this means only additional methods need to be defined and tested which increases code generation time.

Question 3.

The logic paradigm would be the most appropriate as the scientist has come up with some rules on how to classify the rock material. These rules as well as all the known facts about existing rocks can be entered into a Prolog system or perhaps into an expert system shell. The assistant will then be able to enter the facts about the current sample to be classified and the logic paradigm (or expert system) will then use its rules to determine the correct classification.

The imperative and OOP paradigms are unsuitable. Both these paradigms require algorithms and then code which explicitly describes how to solve the classification problem. A different algorithm (and hence subprogram) would be needed to solve each different classification method. Therefore, any OOP or imperative program written to solve this problem will be long and complicated with lots of cases listed. Subsequently, it can be seen that the most appropriate paradigm is logic.

Notes

- Questions which ask about the most appropriate paradigm do not necessarily have a single correct answer. It is often possible to describe a suitable method of implementing a solution using various different paradigms.
- The logic and object oriented paradigms can be used together as the concepts involved are very different and meet very different purposes. There are object oriented versions of Prolog which attempt to combine the advantages of both logic and object oriented programming.

CHAPTER 9 REVIEW

1. Some areas that have an influence on productivity include
   (A) effect on maintenance.
   (B) approach to testing.
   (C) speed of code generation.
   (D) all of the above.

2. If you as a programmer were considering variables and control structures for the design of your computer program, which paradigm would you have most likely chosen to work with?
   (A) Imperative.
   (B) Logic.
   (C) Object oriented.
   (D) Functional.

3. A goal in Prolog terms, can best be described as
   (A) a fact contained in the knowledge database.
   (B) a rule contained in the knowledge database.
   (C) a query that results in either being fulfilled or not being fulfilled.
   (D) a query that results in being fulfilled.

4. You are writing a computer program and have created a class that contains multiple methods with exactly the same name. This does not concern you because you are coding in
   (A) an object oriented language.
   (B) an imperative language.
   (C) a logical language.
   (D) any of the above.

5. Which term refers to the ability to run code in response to a user's action?
   (A) Imperative.
   (B) Event driven.
   (C) Object oriented.
   (D) Logic.

6. You have been employed by a company that specialises in the development of artificial intelligence systems. With which paradigm will you most likely have to become proficient?
   (A) Relational.
   (B) Imperative.
   (C) Object oriented.
   (D) Logic.

7. With regards to software programs, maintenance can be described as
   (A) the ability of the code to be modified to meet changed requirements.
   (B) the development of robust code.
   (C) modularised coding.
   (D) a language that encapsulates data.

8. Which type of languages are based on the von Neumann architecture?
   (A) Logic.
   (B) Imperative.
   (C) Event driven.
   (D) Object oriented.

9. Backward and forward chaining can best be described as
   (A) two common strategies used to resolve goals.
   (B) two facts contained in the knowledge base used to resolve goals.
   (C) two rules contained in the knowledge base used to resolve goals.
   (D) none of the above.

10. The process of designing objects by breaking them down into component classes is the process called
    (A) encapsulation.
    (B) inheritance.
    (C) polymorphism.
    (D) abstraction.

11. In this topic, we have studied three programming paradigms; imperative, logical and object oriented. List and describe the main components or building blocks used in each of these paradigms.

12. The choice of programming paradigm can have a significant effect on programmers' productivity. In fact the main reason for choosing a particular paradigm is often to increase programmers' productivity. List and discuss five effects on programmers' productivity.

Consider the following problem when answering questions 13, 14 and 15.

Snakes and ladders is a common children's board game. Each player takes it in turns to throw a pair of dice. The player then moves his or her piece the number of squares indicated on the dice. If they land on a square containing the bottom of a ladder then their piece is moved to the square at the top of the ladder. If a player's piece lands on the top or head of a snake then their piece moves down to the square indicated by the end of the snake's tail. The first player to reach the last square on the board wins.

Imagine you are considering creating a computer-based simulation of this game.

13. Explain how the game is played from an imperative paradigm point of view. Would this be a suitable paradigm in which to develop this game?

14. Explain how the game is played from a logical paradigm point of view. Would this be a suitable paradigm in which to develop this game?

15. Explain how the game is played from an object oriented paradigm point of view. Would this be a suitable paradigm in which to develop this game?