Introduction to Computer Science

Mark McIlroy

Contents

1. Prelude

A computer program is a set of statements that is used to create an output, such as a screen display, a printed report, a set of data records, or a calculated set of numbers.

Most programs involve statements that are executed in sequence.

A program is written using the statements of a programming language.

Individual statements perform simple operations such as printing an item of text, calculating a single value, and comparing values to determine which set of statements to execute.

Simple instructions are performed in hardware by the computer's central processing unit.

Complex instructions are written in programming languages and translated into the internal instruction set by another program.

Computer memory is generally composed of bytes, which are data items that contain a binary number. These values can range from 0 to 255.

Memory locations are referred to by number, known as an address.

A memory location can be used to record information such as a small number, data from a graphics image, part of a memory address, a program instruction, and a numeric value representing a single letter.

Program instructions and data are stored in memory while a program is executing.

2. Program Structure

2.1. Procedural Languages

Programs written in procedural languages involve a set of statements that are performed in sequence. Most programs are written using procedural languages.

2.1.1.   Data

2.1.1.1. Data Types

Basic data types include numeric values and strings.

A string is a short text item, and may contain information such as a name or a report heading.

Numeric data may be stored internally as a binary number, which is a distinct format from a set of individual digits stored in a text format.

Several numeric data types may be available. These may include integer data types, floating point data types and other formats.

Integers are whole numbers and integer data types cannot record fractional numbers. However, operations with integer data types are generally faster than operations with other numeric data types.

Floating point data types store the digits within a number separately from the magnitude, and can store widely varying values such as 2430000000 and 0.0000002342.

Some languages also support a range of other numeric data types with varying range and precision.

Dates are supported as a separate date type in some languages.

A Boolean data type is a type that records only two values, "true" and "false". Boolean data types and expressions are used in checking conditions and performing different actions in different circumstances.

The language COBOL is used in data processing. Data items within cobol are effectively fields within database records, and may contain a combination of text and numeric digits.

Individual positions within a data field in COBOL can be defined as holding an alphabetic, alphanumeric or numeric character. Calculations can be performed with numeric fields.

2.1.1.2. Type Conversion

Languages generally provide facilities for converting between data types, such as between two different numeric data types, or between numeric data in binary format and a text string of digits.

This may be done automatically within expressions, through the use of an operator symbol, or through a subroutine call.

When different numeric data types are mixed within an expression, the value with the lower level of precision is

generally promoted to the higher level of precision before the calculation is performed.

The details of type promotion vary with each language.

2.1.1.3. Variables

A variable is a data item used within a program, and identified by a variable name.

Variables may consist of fundamental data types such as strings and numeric data types, or a variable name may refer to multiple individual data items.

Variables can be used in expressions for calculations, and also for comparisons to perform different sections of code under different conditions.

The value of a variable can be changed using an assignment statement, which changes the value of a variable to equal the value of an expression.

2.1.1.4. Constants

Constants such as fixed numbers and strings can be included directly within program code.

Constants can also be given a name, similar to a variable name, and used in several places with the program.

The value of a constant is fixed and cannot be changed without recompiling the program.

2.1.1.5. Data Structures

Variables can be defined as a collection of individual data items.

An array is a variable that contains multiple data items of the same type. Each item is referred to by number.

A structure type, also known as a record, is a collection of several different data items.

An object is an element of object orientated programs. An object is referred to by name and contains individual data items. Subroutines known as methods are also defined within an object.

Arrays can contain structures, and structures can contain arrays and other structures.

Some languages support other data structures such as lists.

2.1.1.6. Pointers & References

A pointer is a variable that contains a reference to another variable. The second variable can be accessed indirectly by referring to the pointer variable.

Pointers are used to link data items together, when data structures are dynamically created as a program executes.

In some languages, pointers can be increased and decreases to scan through memory and access different elements within an array, or individual bytes within a block of data.

A reference to a variable is also known as an address, and refers to the location of the variable in memory.

The value of a pointer variable can be set to the address of another data item by using a reference operator with the data item.

The data item that a pointer points to can be accessed by using a de-referencing operator.

2.1.1.7. Variable Scope

Individual variables can only be accessed within certain sections of a program.

Global variables can be accessed from any point within the code.

Local variables apply within a single subroutine. An independent copy of the local variables is created each time that a subroutine is called.

Where a local variable has the same name as a global variable, the name would refer to the variable with the tightest scope, which in that case would be the local variable.

Parameters are data values or variables that are passed to a subroutine when it is called. Parameters can be accessed from within the subroutine.

Some languages have multiple levels of scope. In these cases, subroutines may be defined within other subroutines, and variables may be defined within inner code blocks.

Variables within the current level of scope and outer levels of scope can be accessed, but not variables within an inner level of scope or in an independent part of the system.

Modules and objects may have public and private subroutines and variables. Public variables are accessible outside the module, while private variables are only accessible within the module.

The use of global variables can lead to interactions between different parts of the code, which may make debugging and modifying the code more difficult.

2.1.1.8. Variable Lifetime

Global variables exist for the period of time that the program is running.

Local variables are created when a subroutine is called, and expire when the subroutine terminates.

Static variables may have a scope that applies within a single subroutine, however they have a lifetime that exists for the full period that the program is executing, and they retain their value from one call to the subroutine to the next.

Dynamically created data items exist until they are freed. Dynamic memory allocation involves creating data items while a program is running.

This may be done explicitly, or it may occur automatically when the last remaining variable that points to the item is assigned a different value, or expires as its level of scope terminates.

2.1.2. Execution

2.1.2.1. Expressions

An expression is a combination of constants, variables and operators that is used to calculate a value.

An assignment operation involves a variable name and an expression. The expression is evaluated, and the value of the variable is changed to equal the result of the expression.

Expressions are also used within control flow statements such as "if" statements and loops.

Numeric expressions include the standard arithmetic operations of addition, subtraction, multiplication and division and exponentiation.

The basic string operations are concatenating two strings to form a single string, extracting a substring, and comparing strings.

String expressions may include constant strings, string variables, and operators such as a concatenation operator.

Boolean variables and expressions have only two possible values, "true" and "false".

An expression containing a relational operator, such as "<=", is a Boolean expression. For example, "5 < 3" has the value "false".

The Boolean operators "and", "or" and "not" can also be used in expressions. An "and" expression has the value "true" when both parts are true, an "or" expression has the value "true" when either value is true, and a "not" expression reverses the value.

Boolean expressions are used within "if" statements to execute code under certain conditions and within loops to repeat a series of statements while a condition remains true.

2.1.2.2. Statements

2.1.2.2.1. Assignment Statements

An assignment statement contains a variable name, an assignment symbol such as an "=" sign, and an expression.

The expression is evaluated, and the value of the variable is set to equal the result of the expression.

Some languages are expression-focused rather than statement-focused. In these languages, an assignment operation may itself be an expression, and may be used within other expressions.

2.1.2.2.2. Control Flow

2.1.2.2.2.1. If Statements

An "if" statement contains a Boolean expression and an associated block of code. The expression is evaluated, and if the result is true then the statements within the block are executed, otherwise they are skipped.

An "if" statement may also have a block of code attached to an "else" section. If the expression is false, then the code within the "else" section is executed, otherwise it is skipped.

2.1.2.2.2.2. Loops

A loop statement may contain a Boolean expression. The expression is evaluated, and if it is true then the code within the block is executed. The control flow then returns to the beginning of the loop, and the cycle repeats the loop each time that the condition evaluates to true.

Other loop statements may also be available, such as statements that specify a fixed number of iterations, or statements that loop through all items in a language data structure.

2.1.2.2.2.3. Goto

Some languages support a "goto" statement. A goto statement causes a jump to a different point in the program to continue execution.

Code that uses goto statements can develop very complex control flow and may be difficult to debug and modify.

Some languages also support structured goto operations, such as a statement that terminates the current loop mid-way through the loop code.

These operations do not complicate the control flow to the same extent as general goto statements, however these operations can be easily missed when code is being read.

For example, a statement in an early part of a complex loop may result in the loop being exited when it is executed. This statement complicates the control flow and may make interpreting the loop code more difficult.

2.1.2.2.2.4. Exceptions

In some languages, exception handling subroutines and sections of code can be defined.

These code sections are automatically executed when an error occurs.

2.1.2.2.2.5. Subroutine Calls

Including the name of a subroutine within a statement causes the subroutine to be called. The subroutine name may be part of an expression, or it may be an individual statement.

When the subroutine is called, program execution jumps to the beginning of the subroutine and execution continues at that point. When the code in the subroutine has been executed, or a termination statement is performed, the subroutine terminates and execution returns to the next statement following the original subroutine call.

2.1.2.3. Subroutines

Subroutines are independent blocks of code that are referred to by name.

Programs are composed of a collection of subroutines.

When execution reaches a subroutine call the program execution jumps to the beginning of the subroutine.

Control flow returns to the point following the subroutine call when the subroutine terminates.

Subroutines may include parameters. These are variables that can be accessed within the subroutine. The value of the parameters is set by the calling code when the subroutine call is performed.

Calling code can pass constant data values or variables as the parameters to a subroutine call.

Parameters are passed in various ways. "Call-by-value" passes the value of the data to the subroutine. "Call-by-reference" passes a reference to the variable in the calling routine, and the subroutine can alter the value of a parameter variable within the calling routine.

Call by value leads to fewer unexpected effects in the calling routine, however returning more than one value from a subroutine may be difficult.

Subroutines may also contain local variables. These variables are accessible only within the subroutine, and are created each time that the subroutine is called.

In some languages, subroutines can also call themselves. This is known as recursion and does not erase the previous call to the subroutine. A new set of local variables is created, and further calls can be made.

This process is used for functions that involve branching to several points at each stage in a process. As each subroutine call terminates, execution returns to the previous level.

2.1.2.4. Comments

Comments are included within program code for the benefit of a human reader. Comments are identified as separate text items, and are ignored when the program is compiled.

Comments are used to include additional information within the code that is relevant to a particular calculation or process, and to describe details of the function within a complex section of code.

2.2. Declarative Languages

A declarative program defines structures and patterns, and may contain a set of information and facts.

In contrast, procedural code specifies a set of operations that are executed in sequence.

Declarative code is not executed directly, but is used as input to other processes.

For example, a declarative program may define a set of patterns, which is used by a parser to identify patterns and sub-patterns within a set of input data.

Other declarative systems use a set of facts to solve a problem that is presented.

Declarative languages are also used to define sets of items, such as records within data queries.

Declarative programs are very powerful in the operations that can be performed, in comparison to the size and complexity of the code.

For example, all possible programs can be compiled using a definition of the language grammar.

Also, a problem solving engine can solve all problems that fall within the scope of the information that has been provided.

Facts may include basic data, and may also specify that two things are equivalent.

For example:

$$x + y = z * 2$$

Month30Days = April OR June OR September OR November

FieldName = 342-???-453

expression: number "+" expression

The first example is a mathematical statement that two expressions are equivalent, the second example specifies that "Month30Days" is equal to a set of four months, the third example matches the set of field names beginning with 342 and ending with 453, and the fourth example specifies a pattern in a language grammar.

Patterns may be recursively defined, such as specifying that brackets within an expression may contain an entire expression, with potentially infinite levels of sub-expressions.

Declarative code may involve patterns, which have a fixed structure, and sets, which are unordered collections of items.

2.2.1.　Code Structure

Declarative code may contain keywords, names, constants, operators and statements.

Keywords are language keywords that may be used to separate sections of the program and identify the type of information that is recorded.

The names may identify patterns, while the operators may be used to create a new pattern from other patterns.

Statements may be entered in the form of specifying that two expressions are equivalent.

The chain of connections is defined by the appearance of names within different statements. There is no order within a statement or from one statement to the next.

2.3. Other Languages

Programming languages appear in a wide variety of forms and structures.

In the language LISP, for example, all processing is performed with lists, and a LISP program consists of multiple brackets within brackets defining lists of data and instructions

3. Topics from Computer Science

3.1. Execution Platforms

3.1.1.   Hardware

Computer hardware executes a simple set of instructions known as machine code.

Machine code includes instructions to move data between memory locations, perform basic calculations such as addition, and jump to different points in the code depending on a condition.

Only machine code can be directly executed. Programs written in programming languages are converted to a machine code format before they are executed.

Machine code instructions and data are stored in memory while a program is running.

3.1.2.   Operating systems

An operating system is a program that manages the operation of a computer. The operating system performs a wide range of functions, including managing the screen display and other user interface components, implementing the disk file system, managing execution of processes, and managing memory allocation and hardware devices.

Generally programs a developed to run on a particular operating system and significant changes may be required to run on other operating systems. This may include changing the way that screen processing is handled, changing the memory management processes, and changing file and database operations.

3.1.3. Compilers

A compiler is a program that generates an executable file from a program source code file.

The executable file contains a machine code version of the program that can be directly executed.

On some systems, the compiler produces object code files. Object code is a machine code format however the references to data locations and subroutines have not been linked.

In these cases, a separate program known as a linker is used to link the object modules together to form the executable file.

Fully compiled code is generally the fastest way to execute a program.

However, compilation is a complex process and can be slow in some cases.

3.1.4. Interpreters

An interpreter executes a program directly from the source code, rather than producing an executable file.

Interpreters may perform a partial compilation to an intermediate code format, and execute the intermediate code internally.

This approach is slower than using a fully compiled program, and also the interpreter must be available to run the program. The program cannot be run directly in a stand-alone environment.

However, interpreters have a number of advantages.

An interpreter starts immediately, and may include flexible debugging facilities. This may include viewing the code, stepping through processes, and examining the value of data variables. In some cases the code can be modified when execution is halted part-way through a program.

3.1.5. Virtual Machines

A virtual machine provides a run-time environment for program execution. The virtual machine executes a form of intermediate code, and also provides a standard set of functions and subroutine calls to supply the infrastructure needed for a program to access a user interface and general operating system functions.

Virtual machines are used to provide portability across different operating platforms, and also for security purposes to prevent programs from accessing devices such as disk storage.

An extension to a virtual machine is a just-in-time compiler, which compiles each section of code as it begins executing.

3.1.6. Intermediate Code Execution

A run-time execution routine can be used to execute intermediate code that has been generated by compiling source code.

Programs may be written using a language developed specifically for an application, such as formula evaluation system or a macro language.

The system may contain a parser, code generator and run-time execution routine.

Alternatively, the code generation could be done separately, and the intermediate code could be included as data with the application.

3.1.7.   Linking

In some environments, subroutine libraries can be linked into a program statically or dynamically.

A statically linked library is linked into the executable file when it is created. The code for the subroutines that are called from the program are included within the executable file.

This ensures that all the code is present, and that the correct version of the code is being used.

However, executable files may become large with this approach. Also, this prevents the system from using updated libraries to correct bugs or improve performance, without using a new executable file.

Static linking may only be available for some libraries and may not be available for some functions such as operating system calls.

Dynamic linking involves linking to the library when the program is executing. This allows the program to use facilities that are available within the environment, such as operating system functions.

Dynamically linked libraries can be updated to correct bugs and improve performance, without altering the main executable file.

However, problems can arise with different versions of libraries.

3.2. Code Execution Models

3.2.1. Single Execution Thread

Programs execution is generally based on the model of a single thread of execution.

Execution begins with the first statement in the program and continues through subroutine calls, loops and "if" statements until the program finally terminates.

At any point in time, the current instruction position will only apply to a single point within the code.

A system may include several major processes and threads, but within each major block the single execution thread model is maintained.

3.2.2. Time Slicing

In order to run multiple programs and processes using a single central processing unit, many operating systems implement a time slicing system.

This approach involves running each process for a very short period of time, in rapid succession. This creates the effect of several programs running simultaneously, even though only a single machine code instruction is executing at any point in time.

3.2.3. Processes and Threads

On many systems, multiple programs may be run simultaneously, including more than one copy of a single program.

An executing program is known as a process. Each running program is an independent process and executes concurrently with the other processes.

A program may also start independent processes for major software components such as functional engines.

Some systems also support threads. A thread is an independently executing section of code. Threads may not be entire programs however they are generally larger functional components than a single subroutine.

Threads are used for tasks such as background printing, compacting data structures while a program is running and so forth.

On systems that support multiple user terminals with a central hardware system, users can start processes from a terminal. Multiple processes may operate concurrently, including multiple executing copies of a single program.

3.2.4.   Parallel Programming

Languages have been developed to support parallel programming.

Parallel programming is based on an execution model that allows individual subroutines to execute in parallel.

These systems may be extremely difficult to debug. Synchronisation code is required to prevent conflicts when two subroutines attempt to update the same section of data, and to ensure that one task does not commence until related tasks have completed.

Parallel programming is rarely used. Total execution time is not reduced by the parallel execution process, as the total CPU time required to perform particular task is unchanged.

3.2.5.   Event Driven Code

Event driven code is an execution model that involves sections of code being automatically triggered when a particular event occurs.

For example, selecting a function in a graphical user interface environment may lead to a related subroutine being automatically called.

In some systems several events could occur in rapid succession and several sections of code could run concurrently.

This is not possible with a standard menu-driven system, where a process must complete before a different process can be run.

Event driven code supports a flexible execution environment where code can be developed and executed in independent sections.

3.2.6.   Interrupt Driven Code

Interrupt driven code is used in hardware interfacing and industrial control applications. In these cases, a hardware signal causes a section of code to be triggered.

Interfacing with hardware devices is generally conducted using interrupts or polling. Polling involves checking a data register continually to check whether data is available. An interrupt driven approach does not required polling, as the interrupt handling routine is triggered when an interrupt occurs.

3.3. Data structures

3.3.1.   Aggregate data types

3.3.1.1. Arrays

3.3.1.1.1. Standard Arrays

Arrays are the fundamental data structure that are used for storing collections of data.

An array contains multiple data items of the same type. Each item is referred to by a number, known as the array index.

Indexes are integer values and may start at 0, 1, or some other value depending on the definition and the language.

Arrays can have multiple dimensions. For example, data in a two-dimensional array would be indexed using two independent numbers. A two dimensional array is similar to a grid layout of data, with the row and column number being used to refer to an individual data item.

Arrays can generally contain any data type, such as strings, integers and structures.

Access to an array element may be extremely fast, and may be only slightly slower than accessing an individual data variable.

Arrays are also known as tables.

This particularly applies to an array of structures, which may be similar to a table with rows of the same format but different data in each column. A table also refers to an array of data that is used for reference while a program executes.

In some cases the index entry of the array may represent an independent data value, and the array may be accessed directly using a data item.

In other cases an array is simply used to store a list of items, and the index value does not have any particular significance.

In cases where the array is used to store a list of data, the order of the items may or may not be significant, depending on the type and use of the data.

The following diagram illustrates a two–dimensional array.
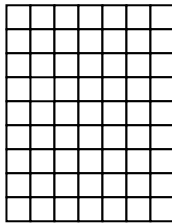
3.3.1.1.2. Ragged Arrays

Standard arrays are square. In a two-dimensional case, every row has the same number of columns, and every column has the same number of rows.

A ragged array is an array structure where the individual columns, or another dimension, may have varying sizes.

This could be implemented using a one-dimensional array for one dimension and linked lists for each column.

Alternatively, a single large array could be used, and the row and column positions could be calculated based on a table of column lengths.

The following diagram illustrates a ragged array.
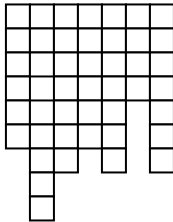
3.3.1.1.3. Sparse Arrays

A sparse array is a large array that contains many unused elements.

This can occur when a data item is used as an index into the array, so that items can be accessed directly, however the data items contain gaps between individual values.

Where entire rows or columns are missing, this structure could be implemented as a compacted array.

Alternatively, the index values could be combined into a single text key, and the data items could be stored by key using a structure such as a hash table or tree.

Another approach may involve using a standard array for one dimension, and linked lists to stored the actual data and so avoid the unused elements in the second dimension.

A sparse array is shown below

		x		x			
x							x
	x		x				
				x			x
	x				x		x

3.3.1.1.4. Associative Arrays

An associative array is an array that uses a string value, rather than an integer as the index value.

Associative arrays can be implemented using structures such as trees or hash tables.

Associative arrays may be useful for ad-hoc programs, as code can quickly and easily be written using an associative array that would require scanning arrays and other processing using standard code.

However, due to the use of strings and the searching involved in locating elements, these structures would have slower access times than other data structures.

3.3.1.2. Structures

A structure is a collection of individual data items. Structures are also known as records in some languages.

A programming structure is similar in format to a database record.

Arrays of structures are visually similar to a grid layout of data with each row having the same type, but different columns containing different data types.

3.3.1.3. Objects

In object orientated programming, a data structure
known as an object is used.

An object is a structure type, and contains a collection of
individual data items.

However, subroutines known as methods are also
defined with the object definition, and methods can be
executed by using the method name with a data variable
of that object type.

3.3.2. Linked Data Structures

Linked data structures consist of nodes containing data
and links.

A node can be implemented as a structure type. This may
contain individual data items, together with links that are
used to connect to other nodes.

Links can be implemented using pointers, with
dynamically created nodes, or nodes could be stored in
an array and array index values could be used as the
links.

Using dynamic memory allocation and pointers results in
simple code, and does not involve defining the size of the
structure in advance.

An array implementation may result in more complex
code, although it may be faster as allocating and
deallocating memory would not be required.

Unlike dynamic data allocation, the array entries are
active at all times. Entries that are not currently used

within the data structure may be linked together to form a free list, which is used for allocation when a new node is required.

3.3.2.1. Linked Lists

A linked list is a structure where each node contains a link to the next node in the list.

Items can be added to lists and deleted from lists in a single operation, regardless of the size of the list. Also, when dynamic memory allocation is used the size of the list is not fixed and can vary with the addition and deletion of nodes.

However, elements in a linked list cannot be accessed at random, and in general the list must be searched to locate an individual item.
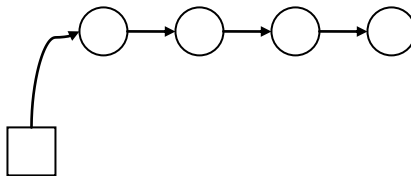
3.3.2.2. Doubly Linked Lists

A doubly linked list contains links to both the next node and the previous node in the list.

This allows the list to be scanned in either direction.

Also, a node can be added to or deleted from a list be referring to a single node. In a singly linked list, a pointer to the previous node must be separately available in order to perform a deletion.
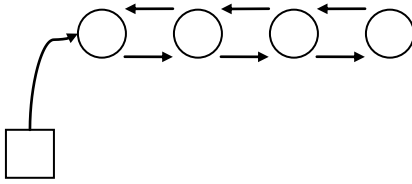
3.3.2.3. Binary Trees

A binary tree is a structure in which a node contains a link to a left node and a link to a right node.

This may form a tree structure that branches out at each level.

Binary trees are used in a number of algorithms such as parsing and sorting.

The number of levels in a full and balanced binary tree is equal to $\log_2(n+1)$ for "n" items.
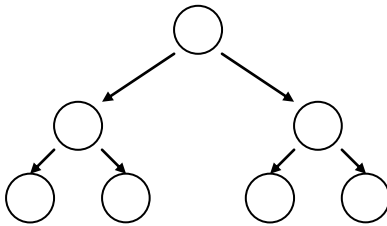
3.3.2.4. Btrees

A B-tree is a tree structure that contains multiple branches at each node.

A B-tree is more complex to implement than a binary tree or other structures, however a B-tree is self balancing when items are added to the tree or deleted from the tree.

B-trees are often used for implementing database indexes.
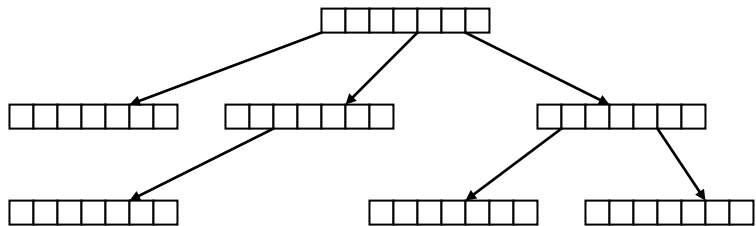
3.3.2.5. Self-Balancing Trees

A self-balancing tree is a tree that retains a balanced structure when items are added and deleted, and remains balanced regardless of the order of the input data.

3.3.3.   Linear Data Structures

3.3.3.1. Stacks

A stack is a data structure that stores a series of items.

When items are removed from the stack, they are retrieved in the opposite order to the order in which they were placed on the stack.

This is also known as a LIFO, Last-In-First-Out structure.

The fundamental operations with a stack are PUSH, which places a new data item on the top of the stack, and POP, which removes the item that is on the top of the stack.

A stack can be implemented using an array, with a variable recording the position of the top of the stack within the array.

Stacks are used for evaluating expressions, storing temporary data, storing local variables during subroutine calls and in a number of different algorithms.
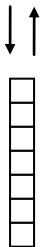
3.3.3.2. Queues

A queue is used to store a number of items.

Items that are removed from the queue appear in the same order that they were placed into the queue.

A queue is also known as a FIFO, First-In-First-Out structure.

Queues are used in transferring data between independent processes, such as interfaces with hardware devices and inter-process communication.
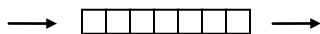
3.3.4.   Compacted Data Structures

Memory usage can be reduced with data that is not
modified by placing the data in a separate table, and
replacing duplicated entries with a single entry.

3.3.4.1. Compacted Arrays

A compacted array can sometimes be used to reduce
storage requirements for a large array, particularly when
the data is stored as a read-only reference, such a state
transition table for a finite state automaton.

In the case of a two dimensional array, a additional one-
dimensional array would be created.

Entries such as blank and duplicated rows could be
removed from the main array, and the remaining data
compacted to remove the unused rows. This may involve
sorting the array rows so that adjacent identical rows
could be replaced with a single row.

The second array would then be used as an indirect index
into the main array. The original array indexes would be
used to index the new array, which would contain the
index into the compacted main array.

An indirectly addressed compacted array is shown below
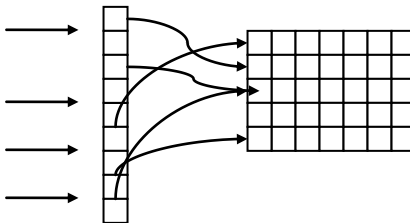
3.3.4.2. String Tables

For example, where a set of strings is recorded in a data structure, a separate string table can be created.

The string table would be an array containing the strings, with one entry for each unique string. The main data table would then contain an index into the string table.
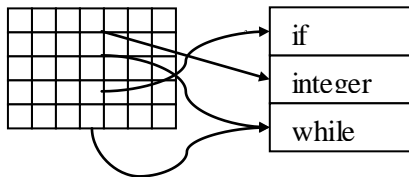
3.3.5.   Other Data Structures

3.3.5.1. Hash tables

A hash table is a data structure that is designed for storing data that is accessed using a string value rather than an integer index.

A hash table can be implemented using an array, or a combination of an array and a linked structure.

Accessing an entry in a hash table is done using a hash function. The hash function is a calculation that generates a number index from the string key.

The hash function is chosen so that the indexes that are generated will be evenly spread throughout the array, even if the string keys are clustered into groups.

When the hash value is calculated from the input key, the data item may be stored in the array element indexed by

the hash value. If the entry is already in use, another hash value may be calculated or a search may be performed.

Retrieving items from the hash table is done by performing the same calculation on the input key to determine the location of the data.

Accessing a hash table is slower than accessing an array, as a calculation is involved. However, the hash function has a fixed overhead and the access speed does not reduce as the size of the table increases.

Access to a hash table can slow as the table becomes full.

Hash tables provide a relatively fast way to access data by a string key. However, items in a hash table can only be accessed individually, they cannot be retrieved in sequence, and a hash table is more complex to implement than alternative data structures such as trees.

3.3.5.2. Heap

A heap is an area of memory that contains memory blocks of different sizes. These blocks may be linked together using a linked list arrangement.

Heaps are used for dynamic memory allocation. This may include memory allocation for strings, and memory allocated when new data items are created as a program runs.

Implementing a heap can be done using pointers and a large block of memory. This requires accessing the memory as a binary block, and creating links and spaces within the block, rather than treating the memory space as a program variable.

Unused blocks are linked together to form a free list, which is used when new allocations are required.
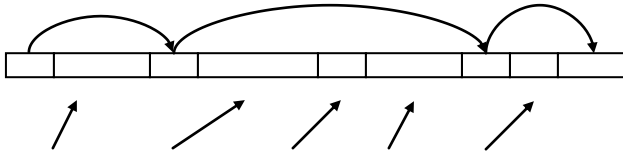
3.3.5.3. Buffer

A buffer is an area of memory that is designed to be treated as a block of binary data, rather than an individual data variable.

Buffers are used to hold database records, store data during a conversion process that involves accessing individual bytes within the block, and as a transfer location when transferring data to other processes or hardware devices.

Buffers can be accessed using pointers. In some languages, a buffer may be handled as an array definition with the array containing small integer data types, with the assumption that the memory block occupies a contiguous section of memory.

3.3.5.4. Temporary Database

Although databases are generally used for the permanent storage of data, in some cases it may be useful to use a database as a data structure within a program.

Performance would be significantly slower than direct memory accesses however the use of a database a program element would have several advantages

A database has virtually unlimited size, either strings or numeric variables can be used as an index value, random accesses are rapid, large gaps between numeric index values are automatically handled and no code needs to be written to implement the system.

3.3.6.   Language-Specific Structures

Some languages include data structures within the syntax of the language, in addition to the commonly implemented array and structure types.

In the language LISP, for example, all data is stored within lists, and program code is written as instructions contained within lists.

These lists are implemented directly within the syntax of the language.

3.3.7. Data Structure Comparison

Structure	Access Method	Random Access Time	Addition & Deletion Time	Full Scan	Memory Usage
Array	Direct Index	1	1	Yes	1 item
	Search (sorted)	$\text{Log}_2(n) - 1$	$n/2$		
	Search (unsorted)	$n/2$	1		
Linked List	Search	$n/2$	1	Yes	1 item + 1 link
Binary Tree	Search (Fully Balanced)	$\log_2(n) - 1$	$\log_2(n) - 1$ (addition)	Yes	1 item + 2 links
	Search (Fully Unbalanced)	$n/2$	$n/2$ (addition)		
Hash Table	String	1 hash function	1 hash function	No	1 item + implementation overhead

3.4. Algorithms

An algorithm is a step by step method for calculating a particular result or performing a process.

For example, the following steps define the sorting algorithm known as a bubble sort.

1. Scan the list and select the smallest item.
2. Move the smallest item to the end of the new list.
3. Repeat steps 1 and 2 until all items have been placed into the new list.

In many cases several different algorithms can be used to perform a particular process. The algorithms may vary in the complexity of implementation, the volume of data used or generated, and the execution time needed to complete the process.

3.4.1. Sorting

Sorting is a slow process that consumes a significant proportion of all processing time.

Sorting is used when a report or display is produced in a sorted order, and when a processing method or algorithm involves the processing of data in a particular order.

Sorting is also used in data structures and databases to store data in a format that allows individual items to be located quickly.

A range of different sorting algorithms can be used to sort data.

3.4.1.1. Bubble Sort

The bubble sort method involves reading the list and selecting the smallest item. The list is then read a second time to select the second smallest item, and so on until the entire list is sorted.

This process is simple to implement and may be useful when a list contains only a few items.

However, the bubble sort technique is inefficient and involves an order of n^2 comparisons to sort a list of "n" items.

Sorting an array of one million data items would require a trillion individual comparisons using the bubble sort method.

When more than a few dozen items are involved, alternative algorithms such as the quicksort method can be used.

3.4.1.2. Quicksort

These algorithms involve using an order of $n*\log_2(n)$ comparisons to complete the sorting process. In the previous example, this would be equal to approximately 20 million comparisons for the list of one million items.

The quicksort algorithm involves selecting an element at random within the list. All the items that have a lower value than the pivot element are moved to the beginning of the list, while the items with a value that is greater than the pivot element are moved to the end of the list.

This process is then applied separately to each of the two parts of the list, and the process continues recursively until the entire list is sorted.

```
subroutine qsort(start_item as integer, end_item as
integer)

    pivot_item as integer
    bottom_item as integer
    top_item as integer

    pivot_item = start_item + (Rnd * (end_item -
start_item))
    bottom_item = start_item
    top_item = end_item

    while bottom_item < top_item

        while data(bottom_item) < data(pivot_item)
            bottom_item = bottom_item + 1
        end

        if bottom_item < pivot_item
            tmp = data(bottom_item)
            data(bottom_item) = data(pivot_item)
            data(pivot_item) = tmp
            pivot_item = bottom_item
        end

        while data(top_item) > data(pivot_item)
            top_item = top_item - 1
        end

        if top_item > pivot_item
            tmp = data(top_item)
            data(top_item) = data(pivot_item)
            data(pivot_item) = tmp
            pivot_item = top_item
        end

    end

    if pivot_item > start_item + 1
        qsort start_item, pivot_item - 1
    end

    if pivot_item < end_item - 1
        qsort pivot_item + 1, end_item
    end
end
```

3.4.2.   Binary Tree Sort

A binary tree sort involves inserting the list values into a binary tree, and scanning the tree to produce the sorted list.

Items are inserted by comparing the new item with the current node. If the item is less than the current node, then the left path is taken, otherwise the right path is taken.

The comparison continues at each node until an end point is reached where a sub-tree does not exist, and the item is added to the tree at that point.

Scanning the tree can be done using a recursive subroutine. This subroutine would call itself to process the left sub tree, output the value in the current node, and then call itself to process the right sub-tree.

A binary tree sort is simple to implement. When the input values appear in a random order, this algorithm produces a balanced tree and the sorting time is of the order of $n*\log_2(n)$.

However, when the input data is already sorted or is close to sorted, the binary tree degenerates into a simple linked list. In this case, the sorting time increases to an order of n^2.

```
subroutine insert_item

    if insert_value < current_value
       if left_node_exists
          next_node = left_node
       else
          insert item as new left node
       end
    else
       if right_node_exists
          next_node = right_node
       else
          insert item as new right node
       end
    end
end
```

```
subroutine tree_scan

   if left node exists
      call tree_scan on left node
   end

   output current node value

   if right node exists
      call tree_scan on right node
   end
end
```

3.4.3. Binary Search

A search on a sorted list can be conducted using a binary search.

This is a fast and simple technique that requires approximately $\log_2(n)$-1 comparisons to locate an item. In a list of one million items, this corresponds to approximately 19 comparisons.

In contrast a direct scan of the list would require an average of half a million comparisons.

A binary search is performed by comparing the search string with the item in the centre of the list. If the search string has a lower value than the central item, then the first half of the list is selected, otherwise the second half is selected.

The process then repeats, dividing the selected half in half again. This process is repeated until the item is located.

```
Subroutine binary_search

   found as integer
```

```
    top_item as integer
    bottom_item as integer
    middle_item as integer

    found = False
    bottom_item = start_item
    top_item = end_item

    while not found And bottom_item < top_item
        middle_item = (bottom_item + top_item) / 2

        if search_val = data(middle_item)
            found = True
        else
            if search_val < data(middle_item)
                top_item = middle_item - 1
            else
                bottom_item = middle_item + 1
            end
        end
    end

    if not found Then
        if search_val = data(bottom_item)
            found = True
            middle_item = bottom_item
        end
    end

    binary_serach = middle_item

end
```

3.4.4.   Date Data Types

Some languages do not directly support date data types,
while other languages support date data types but
implement a restricted data range.

Dates may be recorded internally as text strings, however
this may make comparisons between data values
difficult.

Alternatively, data variables may be implemented as a
numeric variable that records the number of days
between a base date and the data value itself.

When a date variable is implemented as a two byte signed integer value, this date value covers a maximum data range of 89 years.

Depending on the selection of the base date, the earliest and latest dates that can be recorded may be less than 30 years from the current date.

Dates implemented in this way cannot be used to represent a date in a long series of historical data, and these date ranges may be insufficient to record long-term calculations in some applications.

The Julian calendar is based on the number of days that have elapsed since the 1st of January, 4713 BC.

Julian data values can be stored in a four-byte integer variable.

Integer variables are convenient to use and operations with integer data types execute quickly. Two dates stored as Julian variables can be directly compared to determine whether one date is earlier than the other date.

Conversion between a julian value and a system date using a two byte value can be done by substracting a number equal to the number of days between the system base date and the julian base date.

The following algorithm can be used to calculate a julian date.

```
Jdate = 367 * year - int( 7 * (year
        + int((month + 9) / 12)) / 4)
        -   int( 3 * (int(( year + (month
            - 9) / 7)
            / 100) + 1) / 4)
        + int( 275 * month / 9) + day +
1721028.5
```

3.4.5.   Solving Equations

In some cases, the value of a variable in an equation cannot be determined by direct calculation.

For example, in the equation "$y = x + x^2$", the value of "x" cannot be calculated directly from the equation.

In these cases, an iterative approach can be used.

This involves using an initial guess of the solution, and then repeatedly calculating the result and determining a more accurate estimate of the solution with each iteration.

The following method uses two estimates of the result, and calculates a straight line between the values to determine an improved estimate of the solution.

This process continues, with the two most-recent values being carried forward as new estimates are produced.

Given reasonable initial guesses, this method may generate a solution with an accuracy of six significant figures within five to ten iterations.

This method does not use the derivative of the function or estimate the slope of the line from individual values.

When a curve displays a jagged shape, problems can arise with methods that use the slope of the curve.

Jagged curves have a smooth shape at large scales, but the detail of small sections of the curve may display sharp movements.

This can occur in practical situations where the curve is derived from a large number of individual values that are related in a broad way, but where small changes in the pattern of values may result in small random movements in the curve.

The following code outlines a subroutine using this method.

```
' y = f(x) is the function being evaluated.
'
' Ensure that x=0 or some other value for 'x' does
not
' generate a divide-by-zero
'
' y_result is the known "y" value
' x_result is the value of "x" that is calculated
for "y_result"


subroutine solve_fx( y_result as floating_point,
x_result as floating_point)

    define attempts as integer
    define x1, x2, x3, y1, y2, y3, m, c as
floating_point

    constant MAX_ATTEMPTS = 1000

    attempts = 0

        ' use estimates that are reasonable and are
likely
        ' to be on either side of the correct result

    x1 = 1
    x2 = 10

    y1 = f(x1)
    y2 = f(x2)


    while (absolute_value( y_target - y2 ) > 0.000001
                          AND attempts <
MAX_ATTEMPTS)

            ' line between x1,y1 and x2,y2

        If x2 - x1 <> 0 then
            m = (y2 - y1)/(x2 - x1)
            c = y1 - m * x1
        else
                ' unstable f(x), x1=x2 but y1<>y2

            attempts = MAX_ATTEMPTS
        end

            ' calculate a new estimate of 'x'

        x3 = (y_target - c) / m
```

51

```
        y3 = f(x3)

            ' roll over to the two latest points

        x1 = x2
        y1 = y2

        x2 = x3
        y2 = y3

        attempts = attempts + 1
    end

    if attempts >= MAX_ATTEMPTS then  ' failed to
find solution
        solve_fx = false
        x_result = 0
    else
        solve_fx = true
        x_result = x2
    end

end
```

3.4.6. Checksum & CRC

Checksums and CRC calculations can be used to determine whether a block of data has changed.

This may be used in applications such as data transfers through data links, checking whether a block of memory has been altered during a debugging process, and verification of data within hardware devices.

A checksum may involve summing the individual binary values within the block and recording the total.

The same calculation could then be performed at a future time, and a different result would indicate that the data had been changed.

A checksum is a simple calculation that may detect some changes, but it does not detect changes such as two values being exchanged.

A CRC (Cyclic Redundancy Check) calculation can detect a wider range of changes, including values that have been transposed.

A checksum or CRC calculation cannot guarantee that the data is unchanged, as this would only be possible with a random data block by comparing the entire block with the original values.

However, a 4 byte CRC value can represent over four billion values, which implies that a random change to the data would only have a one in four billion chance of generating the same CRC value as the original calculation.

These figures would only apply in the case of a random error. In cases where differences such as transposing values may occur, this would cause problems with some calculations such as checksums that would generate the same result if the data was transposed.

3.4.7. Check Digits

In the case of structured number formats such as account numbers and credit card numbers, additional digits can be added to the number to detect keying errors and partially validate the number.

This can be done by calculating a result from the number, and storing the result as additional digits within the number.

For example, the digits may be summed and the result included as the final two digits within the number.

A more complex calculation would normally be used that could detect digits that were transposed, as transposition is a common error and is not detected by a simple sum of the values.

Verifying a number would be done by performing the calculation with the main digits, and comparing the calculated result with the remaining digits in the number.

3.4.8. Infix to Postfix Expression Conversion

3.4.8.1. Infix Expressions

Mathematical equations and formulas are generally presented in an infix format. Binary operators within infix expressions appear between the two values that they operate on.

In this context, the term binary does not refer to binary numbers, but refers to operators that take two arguments, such as addition.

Arithmetic expressions use arithmetic precedence, so that some operations, such as multiplication, are performed before other operators such as addition.

The standard levels of arithmetic precedence are:

1. Brackets
2. Exponentiation x^y.
3. Unary minus   Negative value such as -3 or -(2*4)
4. Multiplication, Division
5. Addition, Subtraction

Brackets may be used to group operations and change the order of operations.

Due to the issue of operator precedence, and the use of brackets, an infix expression cannot be directly evaluated by performing the operations in a direct order, such as from left to right in the expression.

Infix expressions must be parsed before they can be evaluated. This can be done by using a parser such as a recursive descent method, and evaluating the expression as it is parsed or generating intermediate code.

3.4.8.2. Postfix Expressions

A postfix expression is an alternative format for expressing an expression, that places the operators after the values that they operate on.

Using this format, brackets are not required, and operator precedence does not need to be applied to the expression as the precedence is implied in the order of the symbols.

For example, the infix expression "2 + 3 * 5" would be converted to a postfix expression of "3 5 * 2 +"

Postfix expressions can be evaluated directly from left to right.

This can be done using a stack, where a value in the expression is pushed on to the stack, and an operator pops the arguments from the stack, calculates the result, and pushes the result on to the stack.

When a valid expression is evaluated, a single result should remain on the stack after the expression evaluation is complete, and this should equal the result of the expression.

Expressions may be stored internally in a postfix format, so that they can be directly evaluated.

Code generation effectively generates code to evaluate expressions in a postfix order.

3.4.8.3. Infix to Postfix conversion

Conversion from an infix format to a postfix format can be done using a binary tree.

During the parse, a tree is built of the expression containing a node for each operator and value. A binary operator node would have two subtrees, with one argument appearing in the left sub tree and one argument appearing in the right sub tree.

These sub trees may themselves be complete expressions.

56

The parse tree can be built during the parse, with a node created at each level and returned to the next highest level to be connected as a subtree. This results in the tree being built using a bottom-up approach.

Generating the postfix expression can be done by using a recursive subroutine to scan the tree. This subroutine would call itself to process the left sub tree, then call itself to process the right sub tree, then output the value in the current node.

The output could be implemented as a series of instruction stored in a table.

3.4.8.4. Evaluation

The expression can be evaluated by reading each instruction in sequence. If the instruction is a push instruction, then the data value is pushed on to the stack. If the instruction is an operator, then the operator pops the arguments from the stack, calculated the result, and pushes the result on to the stack.

For example, the following infix expression may be the input string

```
x = 2 * 7 + ((4 * 5) - 3)
```

Parsing this expression and building a bottom-up parse tree would produce a structure similar to the following diagram.
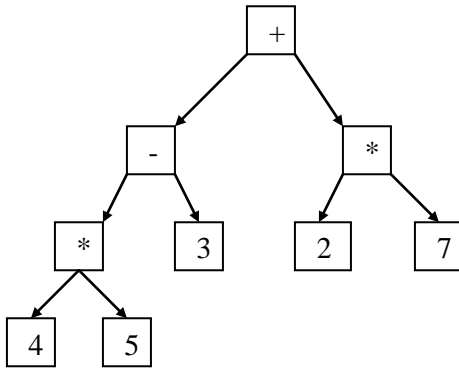
Generating the postfix expression by scanning the parse tree leads to the following expression.

```
x = 4 5 * 3 - 2 7 * +
```

This expression could be directly translated into instructions, as in the following list

```
push 4
push 5
multiply
push 3
subtract
push 2
push 7
multiply
add
```

Executing the expression would lead to the following sequence of steps. In this example the stack contents are shown with the item on the top of the stack shown at the left side of the column.

```
Operation      Stack contents

push 4
                  4
push 5
                  5 4
multiply
                  20
push 3
                  3 20
subtract
                  -17
push 2
                  2 -17
push 7
                  7 2 -17
multiply
                  14 -17
add
                  -3
```

This process ends with the stack containing the result -3, which is the correct result of the original expression.

3.4.9. Regular Expressions

A regular expression is a text pattern-matching method.

Regular expressions form a simple language and can be translated into a finite state automaton. This allows the patterns within the input text to be identified in a single pass, regardless of the complexity of the text patterns.

The operators within a regular expression are listed below.

a	The letter A (or whichever letter or phrase is selected)
[abc]	Any one of the letters a, b or c (or other letters within brackets)
[^abc]	Any letter not being a, b or c (or other letters within brackets)
a*	The letter "a" repeated zero or more times (or other phrase)

a+	The letter "a" repeated one or more times (or other phrase)
a?	The letter "a" occurring optionally (or phrase)
.	Any character
(a)	The phrase or sub-pattern "a"
a-z	Any letter in the range "a" to z" (or other range)
a|b	The phase "a" or "b" (or other phrase)

For example, the pattern specifying a variable name within a programming language may be defined using the following regular expression

[a-zA-Z_][a-zA-Z0-9_]*

This would be interpreted as an initial character being a letter in the range a-z or A-Z, or an underscore character, followed by a character in the range a-z, A-Z, 0-9 or an underscore, repeated zero or more times.

This pattern would match text items such as "x", "_aa", "d3", but would not match patterns such as "3dc" or "a%s".

Regular expressions can also be used in text searching. For example, the following expression would match the words "text scanning" or "scanning text", separated by any characters repeated zero or more times.

(text.*scanning)|(scanning.*text)

As another example, a search for the word "sub" in program code may exclude words such as "subtract" and "subject" by using a pattern such as "sub[^a-z]". This would match any text that contained the letters "sub" and was followed by a character that was not another letter.

3.4.10.   Data Compression

Data compression is used to reduce storage space, and to increase the rate of data transfer through communication channels.

A wide range of data compression techniques and algorithms are used, ranging from the trivial to the highly complex.

Data compression approaches include identifying common patterns within data, and replacing common patterns with a smaller data items.

In compressing text, run length encoding involves replacing a string of identical characters, such as spaces, with a single character and a number specifying the number of occurrences.

Within a text document, entire words could be replaced with number codes.

Huffman encoding involves replacing fixed character sizes with variable bit codes. In standard text, characters may be represented as eight-bit values. In a section of text, however, some characters may occur more often than others.

In this case, frequent characters could be replaced with 5 or 6 bit codes, with less frequent characters replaced with 10 and 11 bit codes.

Compression techniques used with sampled data such as graphics images and sound falls into two categories.

Lossless techniques preserve the original data when they are decompressed. This could involve replacing a repeating section of the data, such as an area containing a single colour, with a single value and codes representing the location of the area.

Within data such as video sequences, multiple identical frames could be replaced with a single frame and a count

of the number of occurrences, and frames that differ slightly could be replaced with a single frame and information identifying the difference to the next frame.

Compaction techniques could involve storing data such as six bit values across byte boundaries, rather than storing each six bit value within a standard eight bit byte and leaving two bits unused.

Lossy techniques offer a higher compression ratio, but with a loss in detail of the data. Data compressed using a lossy method permanently loses detail and cannot be restored to the original data.

Lossy methods include reducing the number of bits used to record each data item, replacing adjacent similar areas with a single value, and filtering data to remove components such as barely visible or barely audible information.

Fractal techniques may involve very high compression ratios. A fractal is an equation that can be used to generate repeating structures, such as clouds and fern leaves. Fractal compression involves filtering data and defining an alternative set of data that can be used to generate a similar image or information to the original data.

3.5. Techniques

3.5.1.   Finite State Automaton

A finite state automaton is a model of a simple machine. The machine works by receiving input characters, and changing to a new state based on the current state and the input character received.

This is a simple but very powerful technique that can be used in a wide range of applications.

Finite state machines are able to detect complex patterns within input data. Due to their simple operation, a finite state machine executes extremely quickly.

The FSA consists of a loop of code, and a state transition table that specifies the next state to change to, based on the current state and the next input character.

A complex model increases the size of the data table, however the code remains unchanged and the execution requires only a single array reference to process each input character.

Parsing program code can be performed by defining a grammar of the language structure, and using an algorithm to convert the grammar definition into a finite state automaton.

Text patterns can be specified using regular expressions, which can also be translated into an FSA.

An example of a finite state automaton is the following description of a state transition table that identifies a certain pattern within text.

This is a pattern that defines program comments that begin with the sequence "/*" and end with the sequence "*/".

State	Next Character	Next State	Within a comment
1	not "/""	1	No
	"/"	2	
2	not "*" or "/"	1	No
	"/"	2	
	"*"	3	
3	not "*"	3	Yes
	"*"	4	
4	not "*" or "/"	3	Yes
	"*"	4	
	"/"	1	
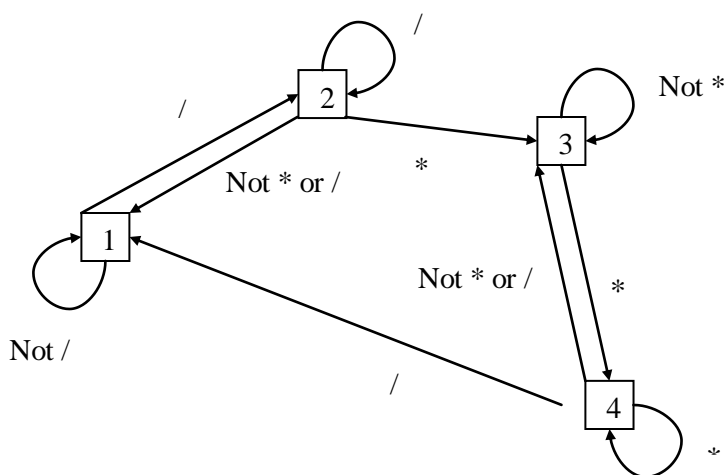
The system begins in state 1, and each character is read in turn. The next state is determined from the current state and the input character.

For example, if the system was in state 2 at a certain point in the processing, and the next character was a "/", then the system would remain in state 2. If the character

was a "*", the system would change to state 3, and for any other character the system changes to state 1.

The current state could be stored as the value of an integer variable.

The process would continue, changing state each time a new character was read until the end of the input was reached.

During processing, any time that the current state was state 3 or state 4, this would indicate that the processing was within a comment, otherwise the processing would be outside a comment.

This process could be used to extract comments from the code.

No backtracking is required to handle sequences such as "/*/**/" that may appear within the text

3.5.2.   Small Languages

In some applications a language may be developed specifically for a single application.

This may involve developing a macro language for specifying formulas and conditions, where the language code could be stored in a database text field or used within an application.

Another example may involve a language for defining the chemical structure of molecules and compounds. This would be a declarative language and would not involve generating code and execution, however it would involve lexical analysis and parsing to extract the individual items and structures within the definition.

A language can be defined with statements, data objects and operators that are specific to the task being performed. For example, within a database management system a task language could be defined with data types representing a record, index node, cache table entry etc, and operators to move records between buffers, data pages and disk storage.

Routines could then be written in the task language to implement procedures such as updating a record, creating a new index and so forth.

The broad steps involved in implementing a small language are:

- Lexical analysis
- Parsing
- Code Generation
- Execution

3.5.2.1. Lexical Analysis

Lexical analysis is the process of identifying the individual elements within the input text, such as numbers, variable names, comments, and operators such as "+" and "<=".

This process can be done using procedural code. Alternatively, the patterns can be defined as regular expression text patterns, and an algorithm used to convert the regular expressions into a finite state automaton that can be used to scan the input text.

The lexical analysis phase may produce a series of tokens, which are numeric codes identifying the elements in the input stream.

The sequence of tokens can be stored in a table with the numeric code for the token, and any associated data

value such as the actual variable name or number constant.

3.5.2.2. Parsing

Parsing or syntax analysis is the process of identifying the structures within the input text.

The language can be defined using a grammar definition which would specify the structure of the language.

One approach is to use an algorithm to convert the grammar to a finite state automaton for parsing, however this may be a complex process.

Another alternative is to use a recursive descent parser, which is a fast and simple parsing technique that is relatively easy to implement.

3.5.2.3. Direct Execution

When the language consists of expressions only, and does not involve separate statements, the value of the expression can be calculated as it is being parsed.

This would involve calculating a result at each point in the recursive descent parser and passing the result back up to the previous level of the expression.

An advantage of this method is that it is simple to implement, and no further stages would be required in the processing.

However, this method would involve parsing the expression every time it was evaluated. If a single formula was applied to many data items then this

approach would be slower than continuing the process to a code generation stage.

Also, this method may not be practical when the language includes statements such as conditions and loops.

3.5.2.4. Code Generation

Code generation may involve generating intermediate code that is executed by a run-time processing loop.

The intermediate code could consist of a table of instructions, with each entry containing an instruction code and possibly other data, such as a variable name or variable reference.

Instructions would be executed in sequence, and branch instructions could be added to jump to different points within the code based on the results of executing expressions within "if" statements and loop conditions.

As each structure within the code is parsed, intermediate code instructions could be generated. This could involve adding additional entries to the table of intermediate code.

Using a stack to execute expressions, the basic operations of a simple language could be implemented using the following instructions.

Variable name reference

   PUSH variable name   (pushes a new value on to the stack)

Assignment operator

POP variable name   (removes a value from the stack and stores

                                    it in the variable)

Operator (e.g. multiplication)

MULT   (e.g. multiply the top two stack values

                                    and replace with the result)

"if" statement

Allocate label position X
Generate code for the "if" condition
Generate a TEST jump instruction to jump to label X if stack result is false
Generate the code for the statement block within the "if" statement
Define the position of label X as this point in the code

Loop

Allocate label position X
Allocate label position Y
Define this position in the code as the position of label Y
Generate code for the loop expression condition
Generate a TEST jump instruction to jump to label X if stack result is false
Generate the code for the loop statements
Generate a jump instruction to label Y
Define label X as this position in the code

When the code has been generated, a second pass can be done through the code to replace the label names with the positions in the code that they refer to.

In this code the MULT operator is used as an example and similar code could be generated for each arithmetic operation, for comparison operators such as "<", and for Boolean operators such as "AND".

No processing is required for brackets, as this is automatically handled by the parsing stage. The parsing

operation ensures that the code is generated in the correct sequence to be executed directly using a stack.

3.5.2.5. Run-Time Execution

When the intermediate code has been generated, it can be executed using a run time execution routine.

This routine would read each instruction in turn and perform the operation. The operations could include retrieving a variable's value and pushing it on to the stack, popping a result from the stack and storing it in a variable, performing an operation, such as MULT or LESSTHAN, and placing the result on the stack, and changing the position of the next instruction based on a jump instruction.

In the case where all values were numeric, including instructions, jump locations and variable location references, execution speed could be quite high.

3.5.3.   Recursion

Recursion is a simple but powerful technique that involves a subroutine calling itself.

This subroutine call does not erase the previous subroutine call, but forms a chain of calls that returns to the previous level as each call terminates.

Recursion is used for processing recursively defined data structures such as trees. Many algorithms are also recursive or include recursive components.

For example, the quicksort algorithm sorts lists by selecting a central pivot element, and then moving all items that are less that the pivot element to the start of

the list, and the items that are larger than the pivot element to the end of the list.

This operation is then performed on each of the two sublists. Within each sublist, two other sublists will be created and so the process continues until the entire list is sorted.

This algorithm can be implemented using a recursive subroutine that performs a single level of the process, and then calls itself to operate on each of the sub parts.

The recursive descent parsing method is a recursive method where a subroutine is called for each rule within the language grammar, and subroutines may indirectly call themselves in cases such as brackets, where an expression may exist within another expression.

A scan of a binary tree would generally be done using a recursive routine that processes a single node and then calls itself to process each of the sub trees.

Recursive code can be significantly simpler than alternative implementations that use loops.

In some languages a subroutine call erases the previous value of the subroutine's variables and recursion is not directly supported.

In these cases, a loop can be used with a stack data structure implemented as an array. In this case a new value is pushed on to the stack when a subroutine call would occur, and popped from the stack when a subroutine call would terminate.

3.5.4. Language Grammars

Some programming languages use a syntax that is based on a combination of specific text layout details, and formal structures for expressions.

In other cases, the language structure can be completely specified by using a formal structure such as a BNF (Backus Naur Form) grammar.

This is a simpler and more flexible approach than using a layout definition, both for parsing and for using the language itself.

The use of a BNF grammar has the advantage that a parser can be produced directly from the grammar definition using several different methods.

For example, a recursive descent parser can be constructed by writing a subroutine for each level in the grammar definition.

A BNF grammar consists of terminal and non-terminal items. Terminals represent individual input tokens such as a variable name, operator or constant.

Non-terminals specify the format of a structure within the language, such as a multiplication expression or an "if" statement.

Non-terminals may be defined with several alternative formats. Each format can contain a combination of terminal and non-terminal items.

The definitions for each rule specify the structures that are possible within the grammar, and may result in the precedence of different operators such as multiplication and addition being resolved though the use of different non-terminal rules.

The example below shows a BNF grammar for a simple language that includes expressions and some statements.

```
statementlist:    <statement>
                  <statement> <statementlist>
```

```
statement:       IF <expression> THEN
<statementlist> END
                 WHILE <expression>
<statementlist> END
                 variable_name = <expression>

expression:      <addexpression> AND
<expression>
                 <addexpression> OR
<expression>

addexpression:   <multexpression> +
<addexpression>
                 <multexpression> -
<addexpression>

multexpression: <unaryexpression> * <
multexpression >
                 <unaryexpression > / <
multexpression >

unaryexpression: constant
                 variable_name
                 - <expression>
                 ( <expression> )
```

3.5.5.   Recursive descent parsing

Recursive descent parsing is a parsing method that is
simple to implement and executes quickly.

This method is a top-down parsing technique that begins
with the top-level structure of the language, and
progressively examines smaller elements within the
language structure.

Bottom-up parsing methods work by detecting small
elements within the input text and progressively building
these up in to larger structures of the grammar.

Bottom-up techniques may be able to process more complex grammars than top-down methods but may also be more difficult to implement.

Some programming languages can be parsed using a top-down method, while others use more complex grammars and require a bottom-up approach

Recursive descent parsing involves a subroutine for each level in the language grammar. The subroutine uses the next input token to determine which of the alternative rules applies, and then calls other subroutines to process the components of the structure.

For example, the non-terminal symbol "statement" may be defined in a grammar using the following rule:

```
statement:      IF <expression> THEN
<statementlist> END
                WHILE <expression>
<statementlist> END
                variable_name = <expression>
```

The subroutine that processes a statement would check the next token, which may be an "IF", a "WHILE", a variable name, or another token which may result in a syntax error being generated.

After detecting the appropriate rule, the subroutines for "expression" and "statementlist" would be called. Code could then be generated to perform the jumps in the code based on the conditions in the "if" statement or loop, or in the case of the assignment operator, code could be generated to copy the result of the expression into the variable.

Grammars can sometimes be adjusted to make them suitable for top-down parsing by altering the order within rules without affecting the language itself.

For example, changing the rule for an "addexpression" in the following way could be used to make the rule suitable for top-down parsing without altering the language.

Initial rule:

```
addexpression:   <addexpression> +
<multexpression>
```

Adjusted rule

```
addexpression:   <multexpression> +
<addexpression>
```

In the adjusted rule the next level of the grammar, "multexpression", can be called directly.

When a language is being created, adjustments can be made to the grammar to make it suitable for parsing using this method. This may include inserting symbols in positions to separate elements of the language.

For example, the following rule may not be able to be parsed as the parser may not be able to determine where an "expression" ends and a "statementlist" begins.

```
statement:       IF <expression>
<statementlist> END
```

This language could be altered by inserting a new keyword to separate the elements in the following way:

```
statement:       IF <expression> THEN
<statementlist> END
```

A recursive descent parse may involve generating intermediate code, or evaluating the expression as it is being parsed.

When the expression is directly evaluated, the calculation can be performed within each subroutine, and the result passed back to the previous level for further calculation.

The structure of the subroutine calls automatically handles issues such as arithmetic precedence, where multiplications are performed before additions.

3.5.6. Bitmaps

Bitmaps involve the use of individual bits within a numeric variable.

This may be used when data is processed as patterns of bits, rather than numbers. This occurs with applications such as the processing of graphics data, encryption, data compression and hardware interfacing.

Typically data is composed of bytes, which each contain eight individual bits. Integer variables may consist of 1, 2, or 4 bytes of data.

In some languages a continuous block of memory can accessed using a method such as an array of integers, while in other languages the variables in the array are not guaranteed to be contiguous in memory and there may be spaces between the individual data items.

Bitmaps can also be used to store data compactly. For example, a Boolean variable may often be implemented as an integer value, but in fact only a single bit is required to represent a Boolean value.

Storing several Boolean flags using different bits within a variable allows 16 independent Boolean values to be recorded within a 2-byte integer variable.

Storing flags within bits allows several flags to be passed into subroutines and data structures as a single variable.

Bit manipulation is done using the bitwise operators AND, OR and NOT.

The bitwise operators have the following results

AND	1 if both bits are set to 1, otherwise the result is 0
OR	1 if either one bit or both bits are set to 1, otherwise the result is 0
NOT	1 if the bit is 0 and 0 if the bit is 1
XOR	1 if one bit is 1 and the other bit is 0, otherwise the result is 0

The operations performed on a bit are testing to check whether the bit is set, setting the bit to 1, and clearing the bit to 0.

A bit can be checked using an AND operation with a test value that contains a single bit set, which is the bit being tested.

For example

Value	0101**1**011
Test Value	0000**1**000
Value AND Test Value	0000**1**000

If the result is non-zero then the bit being checked is set.

Constants may be written using hexadecimal numbers or decimal numbers.

In the case of decimal values, the value would be two raised to the power of the position of the bit, with positions commencing at zero in the rightmost place and increasing to the left of the number.

In the case of hexadecimal numbers, each group of four bits corresponds to a single hexadecimal digit.

The following numbers would be equivalent

Binary	01011011
Hexadecimal	5B

A bit can be set using the OR operator with a test value than contains a single bit set.

For example

Value	01011011
Test Value	00000**1**00
Value OR Test Value	01011**1**11

A bit can be cleared by using a combination of the NOT and AND operators with the test value

For example

Value	01011**1**11
Test Value	00000**1**00
NOT Test Value	11111**0**11
Value AND NOT Test Value	01011**0**11

3.5.7.   Genetic Algorithms

Genetic algorithms are used for optimisation problems that involve a large number of possible solutions, and where it is not practical to examine each solution independently to determine the optimum result.

For example, a system may control the scheduling of traffic signals. In this example, each signal may have a variable duty cycle, being either red or green for a fixed proportion of time, ranging from 10% to 90% of the time. This represents nine possible values for the duty cycle.

One approach would be to set all traffic signals to a 50% duty cycle, however this may impede natural traffic flow.

In a city-wide traffic network with 1000 traffic signals, there would be 9^{1000} possible combinations of traffic signal sequences.

This is an example of an optimisation problem involving a large number of independent variables.

A genetic algorithm approach involves generating several random solutions, and creating new solutions by combining existing solutions.

In this example, a random set of signal sequences could be created, and then a simulation could be used to determine the approximate traffic flow rates based on the signal sequences and typical road usage.

After the random solutions are checked, a new set of solutions is generated from the existing solutions.

This may involve discarding the solutions with the lowest results, and combining the remaining solutions to generate new solutions.

For example, two remaining solutions could be combined by selecting a signal sequence for each traffic signal at random from one of the two existing solutions.

This would result in a new solution which was a combination of the two existing solutions.

New random solutions could then be generated to restore the total number of solutions to the previous level.

This cycle would be repeated, and the process is continued until a solution that meets a threshold result is found, or solutions may be determined within the processing time available.

In some cases a genetic algorithm approach can locate a superior solution in a shorter period of time than alternative techniques that use a single solution that is continually refined.

3.6. Data Storage

3.6.1.　Individual Files

Software tools often use individual files for storing data. For example, an engineering design model may be stored as an individual file.

This approach allows files to be copied, moved and deleted individually, however this approach becomes impractical when a large number of data objects are involved.

3.6.2.　Document Management Systems

Document storage systems are used in environments that involve a large number of documents. These may simply be mail messages, memos and workflow items, or they could also store large volumes of scanned paper documents.

Document management systems include facilities for searching and categorising documents, displaying lists of documents, editing text and mailing information to other locations.

These systems may also include programming languages that can be used to implement applications such as workflow systems.

3.6.3.　Databases

3.6.3.1. Database Management Systems

A database management system is a program that is used to store, retrieve and update large volumes of data within structured data files.

Database management systems typically include a user interface that supports ad-hoc queries, reports, updates and the editing of data.

Program interfaces are also supported to allow programs to retrieve and update data by interfacing with the database management system.

Some operating systems include database management systems as part of the operating system functionality, while in other cases a separate system is used.

3.6.3.2. Data Storage

Databases typically store data based on three structures.

A field is an individual data item, and may be a numeric data item, a text field, a date, or some other individual data item.

A record is a collection of fields.

A table is a collection of data records of the same type. A database may contain a relatively small number of tables however each table may contain a large number of individual records.

Primary data is the data that has been sourced externally from the system. This includes keyed input and data that is uploaded from other systems.

Derived data is data than has been generated by the system and stored in the database.

In some cases derived data can be deleted and re-generated from the primary data.

Links between records can be handled using a pointer system or a relational system.

In the pointer model, a parent record contains a pointer to the first child record. Following this pointer locates the first child record, and pointers from that record can be followed to scan each of the child records.

In the relational model, the child record includes a field that contains the key of the parent record. No direct links are necessary, and each record can be updated individually without affecting other records.

The relational model is a general model that is based simply on sets of data, without explicit connections between records.

In the relational model, the child record refers to the parent record, which is the opposite approach to the pointer system, in which the parent record refers to the child record.

3.6.3.2.1. Keys

The primary key of a record is a data item that is used to locate the record. This can be an individual field, or a composite key that is derived from several individual fields combined into a single value.

A foreign key is a field on a record that contains the primary key of another record. This is used to link related records.

Keys fields are generally short text fields such as an account number or client code. Composite keys may be

composed of individual key fields and other fields such as dates.

The use of a short key reduces storage requirements for indexes, and may also avoid some problems when multiple layers of software treat characters such as spaces in different ways.

3.6.3.2.2. Related Records

Records in different tables may be unrelated, or they may be related on a one-to-one, one-to-many, or many-to-many basis.

A one-to-one link occurs when a single record in one table is related to a single record in another table.

Multiple record formats are an example of this, where the common data is stored in a main table, and specific data for each alternative format is stored in separate tables.

This link can be implemented by storing the primary key of the main record as a foreign key field in the other record.

A one-to-many link occurs when a single record in one table is related to several records in another table.

For example, a single account record may be related to several transaction records, however a transaction record only relates to one account.

This is also known as a parent-child connection.

A one-to-many link can be implemented by storing the primary key of the parent record as a foreign key field in each of the child records.

A many-to-many link occurs between two data entities when each record in one table could be related to several records in the other table.

For example, the connections between aircraft routes and airports may be a many-to-many connection. Each route may be related to several airports, while each airport may be related to several routes.

A many-to-many connection can be implemented by creating a third table that contains two foreign keys, one for each of the main tables.

Each record in the third table would indicate a link from a record in one table to a record in the other table.

The record may simply contain two keys, or additional information could be included, such as a date, and a description of the connection.

The following diagram illustrates a one-to-one database link.
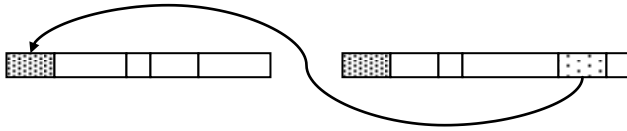
The following diagram illustrates a one-to-many database link.
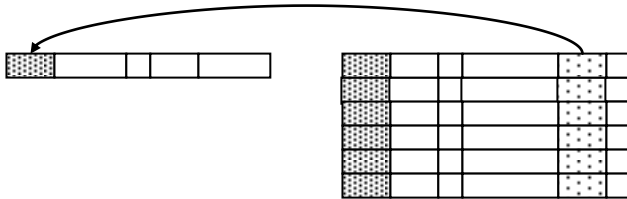
The following diagram illustrates a many-to-many database link.
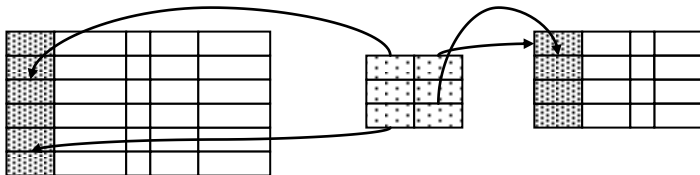
3.6.3.2.3. Referential Integrity

A problem can occur when a primary key of a record is updated, but the foreign keys of related records are not updated to match the change. In this case and link between the records would be lost.

This situation can be addressed by using a cascading update. When this structure is included in the data definition, the foreign keys will automatically be updated when the primary key is updated.

A similar situation arises with deleting records. If a record is deleted, then records containing foreign keys that related to that record will refer to a record that no longer exists.

Defining a cascading delete would cause all the records containing a matching foreign key to be deleted when the main record was deleted.

3.6.3.2.4. Indexes

Databases include data structures known as indexes to increase the speed of locating records. An index may be a self-balancing tree structure such as a B-tree.

The index is maintained internally within the database. In the case of query languages the index is generally transparent to the caller, and increases access speed but does not affect the result.

In other cases, indexes can be selected in program code and a search can be performed directly against a defined index.

Indexes are generally defined against all primary keys.

Also, where records will be accessed in a different order to the primary key, a separate index may be defined that is based on the alternative processing order.

Indexes may also be defined on other fields within the record where a direct access is required based on the value of another field.

3.6.3.3. Database design

In general each separate logical data entity is defined as a separate database table.

Also, data that is recorded independently, such as an individual data item that is recorded on different dates, may also be defined as a separate table

3.6.3.3.1. Data Types

Data types supported by database management systems are similar to the data types used in programming languages. This may include various numeric data types, dates, and text fields.

Some database systems support variable-length text fields although in many cases the length of the field must be specified as part of the record definition, particularly if the field will be used as part of a key or in sorting.

Unexpected results can occur if numeric values and dates are stored in text fields rather than in their native format.

For example, the text string "10" may be sorted as a lower value than "9", as the first character in the string is a "1" and this may be lower in the collating sequence than the character "9".

3.6.3.3.2. Redundant Data

When a data value appears multiple times within a database, this may indicate that the data represents a separate logical entity to the table or tables that it is recorded in.

In this case, the data could be split into a separate data table.

This would result in a one-to-many link from the new table to the original table. This would be implemented by storing the primary key of the new record as a foreign key within the original records.

This process would generally be done as part of the original database design, rather than appearing due to actual duplicated information.

For example, if a single customer name appeared on several invoices, this would be redundant data and would indicate that the customer details and invoice details were independent data entities.

In this case a separate customer table could be created, and the primary key of a customer record could be stored as a foreign key in the invoice record.

3.6.3.3.3. Multiple Record Formats

In some cases a single logical entity may include more than one record format, such as a client table which includes both individual clients and corporate clients.

Some details would be the same, such as name and address, while others would differ such as date of birth and primary contact name.

In this case, the common fields can be stored in the main record, and a one-to-one link established to other tables containing the specific details of each format.

However, this structure complicates the database design and also the processing, replacing one table with three with no increase in functionality.

In cases where the number of specific fields is not excessive, a simpler solution may be to combine all the fields into a single record.

This would result in some fields being unused, however this would simplify the database design and may increase processing speed slightly.

This may also reduce the data storage size when the overhead of extra indexes and internal structures is taken into account.

Some database management systems allow multiple record formats to be defined for a single table. In these cases, the primary key and a set of common fields are the same in each record, while the specific fields for a particular record format use the same storage space as the alternative fields for other record formats.

3.6.3.3.4. Coded & Free Format Fields

A coded field is a field that contains a value that is one of a fixed set of codes. This data item could be stored as a numeric data field, or it may be a text field of several characters containing a numeric or alphabetic code.

Free format fields are text fields that store any text input.

Coded fields can be used in processing, in contrast to free-format fields which can only be used for manual reference, or included in printed output.

Where a data item may be used in future processing, it should generally be stored in a numeric or coded field, rather than a general text field.

3.6.3.3.5. Date-Driven Data

Many data items are related to events that occur on a particular date, such as transactions. Also, many data series are based on measurements or data that forms a time series of information.

In these cases the primary key of the data record may consist of main key and a date. For example, the primary key of a transaction may be the account number and the date.

In cases where multiple events can occur on a single day, a different arrangement, such as a unique transaction number, can be used to identify the record.

Histories may also be kept of data that is changed within the database.

For example, the sum insured value of an insurance policy may be stored in a data-driven table, using the policy number and the effective date as the primary key. This would enable each different value, and the date that it applied from, to be stored as a separate record.

A history record may be manually entered when a change to a data item occurs, or in other cases the main record is modified and the history record is automatically generated based on the change.

Histories are used to re-produce calculations that applied at previous points in time, such as re-printing a statement for a previous period.

Histories may be stored of data that is kept for output or record-keeping purposes only, such as names and addresses, but history particularly applies to data items that are used in processing.

Date-driven data is also used when a calculation covers a period of time.

For example, if the sum insured value of an insurance policy changed mid-way through a period, the premium for the period may be calculated in two parts, using the initial sum-insured for the first part of the period and the updated sum-insured for the second part of the period.

In some cases, changes to a main data record are recorded by storing multiple copies of the entire data record.

In other cases each data item that is stored with a history is split into an individual table. This can be implemented by creating a primary key in the history record that is composed of the primary key of the main record, and the effective date of the data.

3.6.3.3.6. Timestamps

A timestamp is a field containing a date, time and possibly other information such as a user logon and program name.

Timestamps may be stored as data items in a record to identify the conditions under which a record was created, and the details of when it was last changed.

Timestamps can be used for debugging and administrative purposes, such as reconciling accounts,

tracing data problems, and re-establishing a sequence of events when information is unavailable or conflicting.

3.6.3.4. Data Access

3.6.3.4.1. Query Languages

Some database management systems support query languages such as SQL.

These languages allow groups of records to be selected for retrieval or updating.

Query languages are generally declarative, not procedural languages. Although the language may contain statements that are executed in sequence, the actual selection of records in based on a definition statement, rather than a set of steps.

The use of a query language allows many of the details of the database implementation to become transparent to the calling program, such as the indexes defined in the data base, the actual collection of fields in a table, and so on.

Fields can be added to tables without affecting a query that selected other existing fields, and indexes can be added to increase access speed without requiring a change in the query definition.

In theory a database system could be replaced with an alternative system that supports the same query language, such as when data volumes grow substantially and an alternative database management system is required.

However, there are also disadvantages with query languages

The major disadvantage relates to performance.

As the query language is a full text-based language, there may be a significant overhead involved in compiling the query into an internal format, determining which indexes to use for the operation, and performing the actual process.

This delay may be reduced in some cases by using a pre-compiled query, however this is still likely to be slower that a direct subroutine call to a database function.

This delay may be a particular issue for algorithms that involve a large number of individual random record searches.

3.6.3.4.2. Index-Based Access

Some database management systems support direct record searches based on indexes. This may involve selecting a table and an index, creating a primary key and performing the search.

This approach is less flexible than using a query language. The structure of the indexes must be included in the code, and the code may be more complex and less portable.

However, due to the lower internal overhead involved in a direct index search, the performance of index-based operations may be substantially faster than data queries.

3.6.3.4.3. User Interfaces

Many database management systems include a direct user interface. This may allow data to be direct edited,

and may support ad-hoc queries, reporting and data updates.

Queries may be performed using a query language, or using a set of menu-driven functions and report options.

Some systems also include an internal macro language that can be used to perform complex operations, including developing complete applications.

3.7. Numeric Calculations

3.7.1.   Numeric Data Types

Many languages support several different numeric data types.

Data types vary in the amount of memory space used, the speed of calculations, and the precision and range of numbers that can be stored.

Integer data types record whole numbers only.

Floating point variables record a number of digits of precision, along with a separate value to indicate the magnitude of the number.

Integer and floating point data types are widely used. Calculations with these data types may be implemented directly as hardware instructions.

Fixed point data types record a fixed number of digits before and a fixed number of digits after the decimal point.

Scaled data types record a fixed number of digits, and may use a fixed or variable decimal point position.

Arbitrary precision variables have a range of precision that is variable, with a large upper limit. The precision may be selected when the variable is defined, or may be automatically selected during calculations.

COBOL uses an unusual format where a variable is defined with fixed set of character positions, and each position can be specified as alphabetic, alphanumeric, or numeric

3.7.2.   Numeric Data Storage

Integers are generally stored as binary numbers.

Floating point variables store the mantissa and exponent of the value separately. For example, the number 1230000 is represented as 1.23×10^6 in scientific notation, and 0.00456 would be 4.56×10^{-3}.

In the first example, a value representing the 123 may be stored in one part of the variable, while a value representing the 6 may be stored in a separate part of the variable. Actual details vary with the storage format used.

The precision of the format is the number of digits of accuracy that are recorded, while the range specifies the difference between the smallest and largest values that can be stored.

This format is based on the scientific notation model, where large and small numbers are recorded with a separate mantissa and exponent.

For example

6254000000	=	6.254×10^9
0.00000873	=	8.73×10^{-6}

Examples of integer data types:

Storage size	Number range
2 bytes	-32,768 to +32,767
4 bytes	-2,147,483,648 to +2,147,483,647

Examples of floating point data types:

Storage Size	Precision	Range
4 bytes	approx 7 digits	approx 10^{-45} to 10^{38}
8 bytes	approx 15 digits	approx 10^{-324} to 10^{308}

Fixed point numbers may also be stored as binary numbers, with the position of the decimal point taken into account when calculations are performed.

BCD (Binary Coded Decimal) numbers are stored with each decimal digit recorded as a separate 4-bit binary code.

Other numeric types may be stored as text strings of digits, or as structured binary blocks containing sections for the number itself and a scale if necessary.

3.7.3.   Memory Usage and Execution Speed

Generally calculations with integers are the fastest numeric calculations, and integers use the least amount of memory space.

Floating point numbers may also execute quickly when the calculations are implemented as hardware instructions.

Both integers and floating point numbers may be available in several formats, with the range and precision of the data type related to the amount of memory storage used.

When the calculations for a numeric data type are implemented internally as subroutines, the performance may be significantly slower than similar operations using hardware-supported data types.

3.7.4.   Numeric Operators

Although the symbols and words vary, most programming languages directly support the basic arithmetic operations:

+	addition
-	subtraction
*	multiplication
/	division
mod	modulus
^	Exponentiation, y^x

When calculations are performed with integer values, the fractional part is usually truncated, so that only the whole-number part of the result is retained.

Languages that are orientated towards numeric processing may include other numeric operators, such as operators for matrix multiplication, operations with complex numbers, etc.

3.7.5.   Modulus

The modulus operator is used with integer arithmetic to capture the fractional part of the result.

The modulus is the difference between the numerator in a division operation, and the value that the numerator would be if the division equalled the exact integer result.

For example, 13 divided by 5 equals 2.6. Truncating this result to produce an integer value leads to a result of 2.

However, 10 divided by 5 is exactly equal to 2.

In this example, the modulus would be equal to the difference between 13 and 10, or in this case 3.

This could be expressed as

$$13 \bmod 5 = 3$$

An equivalent calculation of "a MOD b" is "a – b * int(a / b)", where the "int" operation produces the truncated integer result of the division.

This operator is used in a wide range of algorithms.

For example, if "total-lines" was the total number of lines in a long document, then the following result would occur:

    page-number = total-lines / lines-per-page   (integer division)
    line-on-page = total-lines Mod lines-per-page

In this example, the "total lines" value increases through the document, while the "line-on-page" value would reset to zero as it crossed over into each new page.

3.7.6.   Rounding Error

Some numbers cannot be represented as a finite series of digits. One divided by three, for example, is ⅓ when expressed as a fraction, but is

    0.333333333….

when expressed as a decimal. Programming languages do not generally support exact fractions, apart from some specialist mathematical packages.

In this case, the 3 after the decimal point repeats forever, but the precision of the stored number is limited. One-third plus two thirds equals one. Using numbers with 15 digits of precision, however, the result would be

$$
\begin{array}{ll}
& 0.333333333333333 \\
+ & 0.666666666666666 \\
= & 0.999999999999999
\end{array}
$$

If a check was done in the program code to see whether the result was equal to 1, this test would return false, not true, as the number stored is 0.999999999999999, which is different from 1.000000000000000.

The maximum rounding error that can occur in a number is equal to one-half of the smallest value represented in the mantissa.

For example, with 15 digits of precision, the maximum rounding error would be a value of 5 in the sixteenth place.

Rounding problems can be reduced if some of the following steps are taken.

- A number storage format with an adequate number of digits of precision is used.

- Chaining of calculations from one result to the next is avoided, and results are calculated independently from input figures where possible.

- The full level of precision is carried through all intermediate results, and rounding only occurs for printing, display, data storage, or when the

figure represents an actual number, such as whole cents.

- Calculations are structured to avoid subtracting similar numbers. For example, "3655.434224 – 3655.434120 = .000104", and ten digits of precision drops to three digits of precision.

When calculations have been performed with numbers, comparing values directly may not generate the expected results.

This issue can be addressed by comparing the numbers against a small difference tolerance.

For example

```
   If (absolute_value( num1 – num2 ) <
0.000001 then
```

Errors in numeric calculations may also appear due to processing issues that are not directly related to the storage precision.

For example, one system may round numbers to two decimal places before storing the data in a database, while another system may display and print numbers to two decimal places but store the numbers with a full precision.

In these cases, comparing totals of the two lists may result in rounding errors in the third decimal place, rather than the last significant digit.

3.7.7.   Invalid Operations

If a calculation is attempted that would produce a result that is larger than the maximum value that a data type can store, then an overflow error may occur.

Likewise, if the result would be less than the minimum size for the data type, but greater than zero, then an underflow error may occur.

Dividing a number by zero is an undefined mathematical operation, and attempting to divide a number by zero may result in a divide-by-zero error.

The handling of invalid numeric operations varies with different programming languages, but frequently a program termination will result if the error is not trapped by an error-handling routine within the code.

These problems can be reduced by checking that the denominator is non-zero before performing a division or modulus, and by using a data type that has an adequate range for the calculation being performed.

3.7.8.   Signed & Unsigned Arithmetic

In some languages numeric data types can be defined as either signed or unsigned data types.

Using unsigned data types allows a wider range of numbers to be stored, however this is a minor difference in comparison to the variation between different data types.

Mixing signed and unsigned numbers in expressions can occasionally lead to unexpected results, especially in cases where values such as -1 are used to represent markers such as the end of a list.

For example, if a negative value was used in an unsigned comparison, it would be treated as a large positive value,

as negative values are the equivalent bit-patterns to the largest half of the unsigned number range.

In the case of a 2 byte integer, the range of numbers would be as shown below.

Data Type	Range
2 bytes signed	-32,768 to +32,767
2 bytes unsigned	0 to +65536

3.7.9.   Conversion between Numeric Types

When numeric variables of different types are mixed, some languages automatically convert the types, other languages allow conversion with an operator or function, and other languages do not allow conversions.

If two items in an arithmetic expression have different numeric types, the value with the lower level of precision may be converted to the same level of precision as the other value before the calculation is performed. However, the detail of data type promotion varies with each language.

3.7.10.   String & Numeric conversion

Numeric data types cannot be displayed directly and must be converted to text strings of digits before they can be printed or displayed.

Also, Information derived from screen entry or text upload files may be in a text format, and must be converted to a numeric data type before calculations can be performed.

Languages generally include operators or subroutines for converting between numeric data types and strings.

This conversion is a relatively slow operation, and performance problems may be reduced if the number of conversions is kept to a minimum.

3.7.11. Significant Figures

The number of significant figures in a number is the number of digits that contain information. For example, the numbers 5023000000 and 0.0000005023 both contain four significant digits. The figures "5023" specify information concerning the number, while the zeros specify the number's size.

When a wide range of numbers are displayed in a narrow space, a floating decimal point can be used with a fixed number of significant figures. For example, the numbers " 3204.8" and " 3.7655" both contain five significant digits.

3.7.12. Number Systems

Numbers are used for several purposes with programs.

One use is to record codes, such as a number that represents a letter, or a number that represents a selection from a list of options.

Numbers are also used to represent a quantity of items. The decimal number 12, the Roman numeral XII, and the binary number 1100 all represent the same number, and would represent the same quantity of items.

3.7.12.1. Roman Numerals

In the Roman number system, major numbers are represented by different symbols. I is 1, V is 5, X is 10, L is 50, C is 100 and so on.

Numerals are added together to form other numbers. If a lower value appears to the left of another value, it is added to the main value, otherwise it is subtracted.

The first ten roman numerals are

I	1	
II	2	
III	3	
IV	4	five minus one
V	5	
VI	6	five plus one
VII	7	five plus two
VIII	8	five plus three
IX	9	ten minus one
X	10	

Although this system can record numbers up to large values, it is difficult to perform calculations with numbers using this system.

3.7.12.2. Arabic Number Systems

An Arabic number system uses a fixed set of symbols. Each position within a number contains one of the symbols.

The value of each position increases by a multiple of the base of the number system, moving from the right-most position to the left.

The value of each position in a particular number is given by the value of the position itself, multiplied by the value of the symbol in that position.

For example

	31504	=	3×10^4	=	3×10000
			$+ 1 \times 10^3$		$+ 1 \times 1000$
			$+ 5 \times 10^2$		$+ 5 \times 100$
			$+ 0 \times 10^1$		$+ 0 \times 10$
			$+ 4 \times 10^0$		$+ 4 \times 1$

The decimal number system is an Arabic number system with a base of 10. The digit symbols are the standard digits 0, 1, 2, 3, 4, 5, 6, 7, 8 and 9.

Other bases can be used for alternative number systems. For example, in a number system with a base of three, the number 21302 would be equivalent to the number 218 in decimal.

21302_3	=	2×3^4	=	$2 \times 81_{10}$
		$+ 1 \times 3^3$		$+ 1 \times 27_{10}$
		$+ 3 \times 3^2$		$+ 3 \times 9_{10}$
		$+ 0 \times 3^1$		$+ 0 \times 3_{10}$
		$+ 2 \times 3^0$		$+ 2 \times 1$

$$= 218_{10}$$

In this notation, the subscript represents the base of the number. The base-three representation 21302 and the base-ten representation 218 both represent the same quantity.

3.7.12.2.1. Binary Number System

The Arabic number system with a base of 2 is the known as the binary number system. The digits used in each position are 0 and 1.

In computer hardware, a value is represented electrically with a value that is either on or off. This value can represent two states, such as 0 and 1.

Each individual storage item is known as a bit, and groups of bits are used to store numbers in a binary number format.

Computer memory is commonly arranged into bytes, with a separate memory address for each byte. A byte contains eight bits.

An example of a binary number is

$$
\begin{array}{lcl}
101101_2 = & 1 \times 2^5 & = \quad 1 \times 32_{10} \\
 & + 0 \times 2^4 & \quad + 0 \times 16_{10} \\
 & + 1 \times 2^3 & \quad + 1 \times 8_{10} \\
 & + 1 \times 2^2 & \quad + 1 \times 4_{10} \\
 & + 1 \times 2^1 & \quad + 0 \times 2_{10} \\
 & + 1 \times 2^0 & \quad + 1 \times 1 \\
 & & \quad = 45_{10}
\end{array}
$$

In this example, the binary number 101101 is equivalent to the decimal number 45.

As the base of the binary number system is two, each position in the number represents a number that is a multiple of 2 times larger than the position to its right.

The value of a number position is the base raised to the power of the position, for example, the digit one in the fourth position from the right in a decimal number has a

109

value of $10^3 = 1000$. In a binary number, a digit in this position would have a value of $2^3 = 8$.

3.7.12.2.2. Hexadecimal Numbers

The hexadecimal number system is the Arabic number system that uses a base of 16.

This is used as a shorthand way of writing binary numbers. As the base of 16 is a direct power of two, each hexadecimal digit directly corresponds to four binary digits.

Hexadecimal numbers are used for writing constants in code that represent a pattern of individual bits rather than a number, such as a set of options within a bitmap.

Due to the fact that the base of the number system is 16, there are 16 possible symbols in each position in the number.

As there are only ten standard digit symbols, letters are also used.

The characters used in a hexadecimal number are 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E and F. The characters A through to F have the values 10 through to 15.

For example,

$$8D3F_{16} \quad = \quad 8 \times 16^3 = \quad\quad 8 \quad \times 4096_{10}$$

$$+ D \times 16^2 \quad\quad + 13_{10} \times 256_{10}$$

$$+ 3 \times 16^1 \quad\quad + \quad 3 \quad \times 16_{10}$$

$$+ F \times 16^0 \quad\quad + 15_{10} \times 1$$

$$=$$

$$36159_{10} \quad\quad = 1000110100111111_2$$

110

There is a direct correspondence between each hexadecimal digit and each group of four binary digits.

For example,

$$
\begin{array}{cccc}
8 & D & 3 & F \\
1000 & 1101 & 0011 & 1111
\end{array}
$$
=

In this example, the second group of bits has the value 1101, which corresponds to 13 in decimal and D in hexadecimal.

The following table includes the first eight powers of two and some other numbers, represented in several different bases.

Hexadecimal Value	Binary	Decimal	
01	00000001	1	2^0
02	00000010	2	2^1
04	00000100	4	2^2
08	00001000	8	2^3
10	00010000	16	2^4
20	00100000	32	2^5
40	01000000	64	2^6
80	10000000	128	2^7
0F	00001111	15	
F0	11110000	240	
FF	11111111	255	

3.8. Speed & Efficiency

3.8.1.   Execution Speed

3.8.1.1. Accessing data

A significant proportion of processing time is involved in accessing data. This may include searching lists locate a data item, or executing a search function within a data structure.

3.8.1.1.1. Access Methods

The fundamental access methods include the following.

3.8.1.1.1.1. Direct Access

Direct access involves using an individual data variable or indexing an array element. This is the fastest way to access data.

In a fully compiled program, both these operations require only a few machine code instructions.

In contrast, a search of a list of 1,000 items may involve thousands of machine code operations.

3.8.1.1.1.2. Semi-Direct Access

Data can be accessed directly using small integer values as array indexes.

However, data cannot be accessed directly by using a string, floating point number or wide-ranging integer value to refer to the data item.

Data stored using strings and other keys can be accessed in several steps by using a data structure such as a hash table.

Access time to a hash table entry is slower than access to an array, as the hash function value must be calculated. However, this is a fixed delay and is not affected by the number of items stored in the table.

Access times can slow as a hash table becomes full.

3.8.1.1.1.3. Binary Search

A binary search can locate an item in a sorted array in an average of approximately $\log_2(n)-1$ comparisons.

This is generally the simplest and fastest way to locate data using a string reference.

Unless the data size is extremely large, such as over one million array elements, then the overhead in calculating a hash table key is likely to be higher than the number of comparisons involved in a binary search.

A binary search is suitable for a static list that has been sorted.

However, if items are added to the list or deleted from the list, this approach may not be practical. Either a complete re-sort would be required, or an insertion into a sorted list could be used which would involve an n/2 average update time.

In these cases, a hash table or a self-balancing tree could be used.

3.8.1.1.1.4. Exhaustive Search

An exhaustive search involves simply scanning the list from the first item onwards until the data item is located.

This takes an average of n/2 comparisons to locate an item.

3.8.1.1.1.5. Access Method Comparison

The average number of comparisons involved in locating a random element is listed in the table below.

Entries	Direct Index	Binary Search (sorted)	Full Scan (unsorted)
100	1	5.6	50
1,000	1	9.0	500
10,000	1	12.3	5,000
100,000	1	15.6	50,000
1,000,000	1	18.9	500,000
10,000,000	1	22.3	5,000,000

The direct index method can be used when the item is referenced by an index number. When the item is referenced by a string key, an alternative method can be used.

The hash table access is not shown. However, it has a fixed access overheard that could be of the approximate order of the 20-comparison binary search operation.

The binary search method can only be used with a sorted list. Sorting is a slow operation. This method would be

suitable for data that is read at the beginning of program execution and can be sorted once.

The full search is extremely slow and would not be practical for lists of more than a few hundred items where repeated scans were involved.

3.8.1.1.2. Tags

Searching by string values can be avoided in some cases by assigning temporary numeric tags to data items for internal processing.

For example, the primary and foreign keys of database records may be based on string fields, or widely-ranging numeric codes.

When values are read into arrays for internal calculations, each entry could be assigned a small number, such as the array index, as a tag.

Other data structures would then use the tag to refer to the data, rather than the string key. This would allow that data to be accessed directly using a direct array index.

3.8.1.1.3. Indirect Addressing

In cases where a data item cannot be accessed directly using a numeric index or a binary search on a sorted array, an indirect accessing method may be able to be used.

This approach would apply when there was a need to search a table by more than one field, or when a numeric index value did not map directly to an array entry, such as an index into a compacted table.

This method involves defining an additional array that is indexed by the search item, and contains an entry that specifies the index into the main array.

In the case of an indirect index, such as a mapping into a compacted table, the array would be indexed by the original index and would contain the index value for the compacted table.

In the case of a string search, the array would be sorted by the string field. This would be searched using a binary search method and would contain the index into the main table.

This approach could also be used when the main table contained large blocks of data, and moving the data during a sort operation would be a slow process.
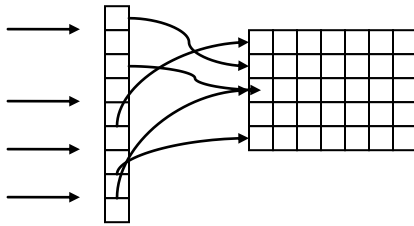

3.8.1.1.4. Sublists

In cases where searching lists is unavoidable, the search time may be reduced if the data is broken into sub-sections.

This is effectively a partial sort of the data.

For example, a set of data may contain ten samples from ten locations, for a total of 100 data items. Searching this data would involve an average of 50 comparisons.

However, if the samples were allocated to individual locations, then searching the locations would take an average of 5 comparisons, and searching the samples would also involve an average of five comparisons, for a total of 10 average comparisons.

Additionally, although one of the other approaches such as direct indexing may not be practical for the full data set, it may be applicable to one of the stages in the sub-list structure.
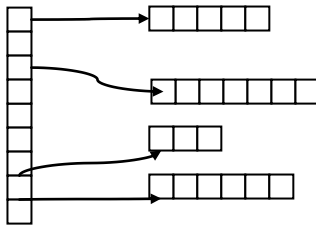
3.8.1.1.5. Keys

Where data is located by searching for several different data items, the data items can be combined into a single text key.

This key can then be used to locate the data using one of the string index methods, such as a hash table or a binary search on a sorted array.

3.8.1.1.6. Multiple Array Indexes

Where data is identified by several independent numeric items, a multiple dimension array can be used to store the data, with each numeric item being used to index one of the array dimensions.

117

This allows for direct indexing, however a large volume of memory may be used. The memory consumed is equal to the size of the array items multiplied by the size of each of the dimensions. For example, if the first dimension covered a range of 3000 values, the second dimension covered 80 values and the third dimension covered 500 values, then the array would contain 120 million entries.

This problem may be reduced by compacting the table to remove blank entries and using a separate array to indirectly index the main table, or by using numeric values for some dimensions and locating other values by searching the data.

3.8.1.1.7. Caching

Caching is used to store data temporarily for faster access.

This is used in two contexts. Caching may be used to store data that has been retrieved from a slower access storage, such as storing data in memory that has been retrieved from a database or disk file.

Also, caching can be used to store previously calculated values that may be used in further processing. This particularly applies to calculations that required reading data from disk in order to calculate the result.

A structured set of results can be stored in an array. Where a large number of different types of data are stored, a structure such as a hash table can be used.

Caching of calculated results can also be stored in temporary databases.

Where input figures have changed, the cached results will not longer be valid and should be discarded,

otherwise incorrect values may be used in other calculations.

3.8.1.2. Avoiding Execution

Execution speed can be increased by avoiding executing statements unnecessarily.

3.8.1.2.1. Moving code

Moving code inside "if" statements and outside loops may increase execution speed.

If the results of a statement are only used in certain circumstances, then placing an "if" statement around the code may reduce the number of times that the code is executed.

This could also involve moving the statement inside an existing "if" statement.

Moving code outside a loop has a similar effect. If a statement returns the same result for each iteration of the loop, then the code could be moved outside the loop to a previous position.

This would result in the statement being executed only once, rather than every time that the loop was repeated.

3.8.1.2.2. Repeated Calculation

In some cases a calculation is repeated but it cannot be moved outside a loop.

This can occur with nested loops for example.

Where a statement within the inner loop refers to the inner loop variable, it cannot be moved outside the inner loop as its value may be different with each iteration of the loop.

However, if this statement does not refer to the outer loop variable, then the entire cycle of values will be repeated for each iteration of the outer loop.

This situation can be addressed by including a separate loop before the nested loop, to cycle through the inner loop once and store the results of the statement in a temporary array.

These array values could then be used in the inner nested loop.

This may have significant impact on the execution time when the statement includes a subroutine call to a slow function and the outer loop is executed a large number of times.

3.8.1.3. Data types

3.8.1.3.1. Strings

Processing with strings is significantly slower than processing with numeric variables.

Copying strings from one variable to another, and performing string operations may involve memory allocations and copying individual characters within the string

In contrast, setting a numeric value can be done with a single instruction.

In general execution speed may be significantly increased if numeric values are used for internal processing rather than string codes.

For example, dates, tags to identify records, flags and processing options could all be represented using numeric variables rather than string codes.

Numeric data and dates read into a system may be in a string format, in which case the data can be converted to a numeric data type for storage and processing.

3.8.1.3.2. Numeric Data

Integer data types generally provide the fastest execution speed.

Floating point number calculations may also be fast, as hardware generally supports floating point calculations directly.

However, many languages support other numeric data types that may be quite slow, if the arithmetic is implemented internally as a set of subroutines.

The numeric data types that are compiled into direct hardware instructions varies with the language, hardware platform and language implementation.

3.8.1.4. Algorithms

An algorithm is a step-by-step process for achieving a result.

The choice of algorithm can have a significant impact on execution speed. In cases where this available memory is

restricted, the choice of algorithm can also impact on the volume of memory used for data storage.

In some cases, alternative algorithms to produce the same result may differ in execution speed by many orders of magnitude.

Sorting is an example of this. Sorting an array of one million items using the bubble sort algorithm would involve approximately one trillion operations, while using the quicksort algorithm may involve approximately 20 million operations.

In many cases several different algorithms can be used to calculate a particular result.

Algorithms that scan input data without backtracking may be faster than algorithms that involve backtracking.

For example, a test searching method that used a finite state automaton to scan the text in a single pass may be faster than an alternative algorithm that involved backtracking.

Some algorithms make a single pass through the input data, while others involve multiple passes.

If the complete processes can be performed during a single pass then faster execution may be achieved, particularly when database accesses are involved. However, in some cases a process with several simple passes may be faster than a process that uses a single complex pass.

3.8.1.5. Run-time processing

In some cases, a structure can be compiled or translated into another structure that can be processed more quickly.

For example, a text formula may specify a particular calculation. Processing this formula may involve identifying the individual elements within the formula, parsing the string to identify brackets and the order of operators, and finally calculating the result.

If the calculation will be applied to many data items, then the expression can be translated into a postfix expression and stored in an internal numeric code format.

This would enable the expression to be directly executed using a stack. This would be a relatively fast operation and would be significantly faster than calculating the original expression for each data item.

This process could be extended to the case of a macro language using variables and control flow, by generating internal intermediate code and executing the code using a simple loop and a stack.

Another example of separate run-time processing would involve algorithms that generate finite state automatons from patterns such as language grammars or text search patterns.

Once the finite state automaton has been generated, it can be used to process the input at a high rate, as a single transition is performed for each input character and no backtracking is involved.

3.8.1.6. Outdated Code

When a section of code has been subject to a large number of changes and additions, the processing can become very complex. This may involved multiple nested loops, re-scanning of data structures and complex control flow. These changes may lead to a significant drop in execution speed.

In these cases, re-writing the code and changing the data structures may lead to a drastic reduction in complexity and an increase in execution speed

3.8.1.7. Numeric Processing

Numeric processing is generally faster when the data is loaded into program arrays before the calculation commences. Data may be loaded from databases, data files, or more complex program data structures.

Execution speed may be increased if calculations are not repeated multiple times. This situation can also arise within expressions.

For example, an expression of the form "x = a * b * c" may appear within an inner processing loop. However, if part of the expression does not change within the inner loop, such as the "b * c" component, then this calculation can be moved outside the loop.

This may lead to an expression of "d = b * c" in an outer loop, and "x = a * d" in the inner loop.

While optimising compilers can perform some of these enhancements automatically, this is no replacement for the human programmer writing high-quality code.

Where a calculation within an inner loop is effected by the inner loop but not the outer loop, then a separate loop can be added before the main loops to generate the set of results once, and store then in a temporary array for use within the inner loop.

Special cases within the structure of the calculations and data may enable the number of calculations to be reduced.

For example, if a matrix is triangular, so that the data values are symmetrical around the diagonal, then only the data on one side of the diagonal needs to be calculated.

As another example, some parts of the calculation may be able to be performed using integers rather than floating point operations. However, the conversion time between the integer and floating point formats would also affect the difference in execution times.

If the data is likely to contain a large number of values equal to zero or one, then these input values can be checked before performing unnecessary multiplications or additions.

Integer operations that involve multiplications or divisions by powers of two can be performed using bit shifts instead of full arithmetic operations.

An integer data item can be multiplied by a power of two by shifting the bits to the left, by a number of places that is equal to the exponent of the multiplier.

In languages that support pointer arithmetic, stepping through an array by incrementing a pointer may be faster than using a loop with an array index.

In cases where an extremely large number of calculations are performed on a set of numeric data, several steps may lead to slight performance increases.

Accessing several single-dimensional arrays may be slightly faster than accessing a multiple dimensional array, as one less calculation is involved in locating an element position.

When multiple dimensional arrays are used, access may be slightly faster if the array dimensions sizes are powers of two. This is due to the fact that the multiplication required to locate an element position can be performed

using a bit shift, which may be faster than a full multiplication.

3.8.1.8. Language and Execution Environment

In extreme cases an increase in execution speed may involve changing the program to a different programming language or development environment.

Fully compiled code that produces an executable file is generally the fastest way to execute a program.

Some languages and development platforms are based on interpreters or use partial compilation and a run-time interpreter to execute the code.

These approaches may be substantially slower than using a full compiler.

Compilers often implement optimising features to increase execution speed by performing actions such as converting multiplication by powers of two into bit shifts, automatically moving code outside loops, and generating specific machine code instructions for specific situations.

The level and type of optimisation may be selectable when a compilation is performed.

3.8.1.9. Database Access

Processes that are structured to avoid re-reading database records will generally execute more quickly than processes that read records more than once.

Where a record would be read many times during a process, the record can be read once and stored in a record buffer or program variables.

When a one-to-many link is being processed, if the child records are read in the order of the parent record key, not the child record key, then each parent record would only need to be read once.

Processing the child records in the order of the child record key may require re-reading a parent record for every child record.

Selecting a processing order that requires only a single pass through the data would generally result in faster execution that an alternative process that used multiple passes through the data, or that re-read records while processing one section of the data.

3.8.1.10. Redundant Code

In code that has been heavily modified, there may be sections of code that calculate a result or perform processing that is never used.

Additionally, some results may be produced and then overwritten at a later stage by other figures, without the original results having been accessed.

Removing redundant code can simplify the code and increase execution speed.

3.8.1.11. Bugs

Performance problems may sometimes be due to bugs. A bug may not affect the results of a process, but it may result in a database record being re-read multiple times, a loop executing a larger number of times than is

necessary, cached data being ignored, or some other problem.

Using an execution profiler or a debugger may identify performance problems in unexpected sections of code due to bugs.

3.8.1.12. Special Cases

In some calculations or processes, there is a particular process than could be done more quickly using an alternative method.

In these cases, the alternative approach may also be included, and used for the particular cases that it applies to.

For example, a subroutine to calculate x^y could implement a standard numerical method to calculate the result.

However, this may be a slow calculation, and in a particular application it may be the case that the value of "y" is often 2, to calculate the square of a number.

In this case the value of "y" could be checked in the subroutine, and if it is equal to 2 then a simple multiplication of "y * y" could be used in place of the general calculation.

Another example concerns sorting. In a complex application it may be the case that a list is sometimes sorted when it is already in a sorted order.

The sort routine could check for the special case that the input data was already sorted, in which case the full sorting process would not be necessary.

Special cases may also apply to caching, rather than a particular figure or set of data. For example, a subroutine to return the next item in a list after a specified position could retain the value of the last position returned, as in many cases a simple scan of the list would be done and this would avoid the need to re-establish the starting position.

3.8.1.13. Performance Bottlenecks

In some cases the majority of the processing time may be spent in a small section of code.

This may occur with nested loops for example. If an outer loop iterates 1000 times and the inner loop also iterates 1000 times, then the code within the inner loop will be executed one million times.

Indirect nested loops can occur when a subroutine is called from within a loop, and the subroutine itself contains a loop.

This would initially appear to be two single-level loops, however because the subroutine is called from within one of the loops, the number of times that the code would be executed would be the product of the number of loop iterations, not the sum.

In some development environments an execution profiling program can be used to determine the proportion of execution time that is spent within each section of the code.

In some cases a nested loop may be avoided by changing the order of nesting.

For example, if an array is scanned using a loop and another array is scanned within that loop, if the first array can be directly accessed using an array index, then a

nested loop could be avoided by scanning the second array first, and using the direct index to access the first array.

Also, sub-lists could be expanded to provide a single complete set of data combinations, rather than including a main set of data and multiple sub-lists of related information.

3.8.2. Memory Usage

3.8.2.1. Data Usage

3.8.2.1.1. Individual variables

Numeric variables use less memory space than strings. Where possible, replacing string values in tables and arrays with a number code may reduce memory usage.

Integer numeric variables generally occupy less memory space than other numeric data types.

Also, floating point data types may be available in several levels of precision. Changing an array of double-precision floating point values to single-precision variables may halve the memory usage, although this would only be practical when the single-precision format contained an adequate number of digits of precision for the data being stored.

Data types such as dates may be able to be stored as numeric variables rather than using a different internal format that consumes more memory.

3.8.2.1.2. Bitmaps

Where several Boolean flags are used, these can be stored as individual bits using a bitmap method, rather than using a full integer variable for each individual flag.

In some applications, such as graphics processing, individual data items may not use a number of bits that is a multiple of a standard eight-bit byte.

For example, eight separate values can be represented using a set of three bits. If a large volume of data consisted of numbers with three-bit patterns, then several data items could be stored within a particular byte.

This would involve a calculation to determine the location of a three-bit value within a block of data, and the use of bitmaps to extract the three-bit code from an eight-bit data item.

3.8.2.1.3. Compacting Structures

In cases where structures contain duplicated entries or blank entries, the structure may be able to be compacted.

This can be done by replacing the multiple duplicated entries with a single entry, removing blank entries, and using a separate array to map the original array indexes to the index value into the compacted array.

This is an indirect indexing method.

3.8.2.1.4. String tables

In cases where a particular string value may appear multiple times within a set of data, the strings can be stored in a separate array, and the entries in the original data structure can be changed to numeric indexes into the string table array.

This may also increase processing speed when entries are moved and copied in the main data structure.

4. The Craft of Programming

4.1. Programming Languages

4.1.1.   Assembler

Machine code is the internal code that the computer's processor executes. In provides only basic operations, such as arithmetic with numbers, moving data between memory locations, comparing numbers and jumping to different points in the program based on zero/non-zero conditions.

Assembly code is the direct representation of machine code in a readable, text format. Assembler has no syntax and is simply a sequence of instructions.

Assembly code is the fastest way to implement a procedure however it has some severe limitations.

Each processor uses a different set of machine code instructions, so an assembly language program must be completely re-written to run on a computer that uses a different internal processor.

Assembler is a very basic language and a large volume of code must be written to accomplish a particular function.

Assembly language is sometimes used in controlling hardware and industrial machine control, and in applications where speed is critical, for example the routing of data packets in a high-speed network.

4.1.2.   Third Generation Languages

Third-generation languages (3GL), refer to a range of programming languages that are similar in structure and are in widespread use.

These languages operates at the level of data variables, subroutines, "if" statements, loops etc.

4.1.2.1. Fortran

Fortran and Cobol were the first widely-used programming languages, with Cobol being used for business data processing and Fortran for numeric calculations in engineering and scientific applications.

Fortran is an acronym for "FORmula TRANslator" Fortran has strong facilities for calculation and working with numbers, but is less useful for string and text processing.

4.1.2.2. Basic

Basic was initially designed for teaching Fortran. Basic is an acronym for "Beginners All Purpose Symbolic Instruction Code".

Basic has flexible string handling facilities and reasonable numeric capabilities.

4.1.2.3. C

C was original developed as a systems-level language and was associated with the development of the UNIX operating system.

C is fast and powerful, and makes extensive use of pointers. C is useful for developing complex data

structures and algorithms. However, processing strings in C is cumbersome.

4.1.2.4. Pascal

Pascal was initially developed for teaching computer science courses. Pascal is named after the mathematician Blaise Pascal, who invented the first mechanical calculating machine of modern times.

Pascal is a highly structured language that has strict type checking and multiple levels of subroutine and data variable scope.

4.1.3. Business Orientated Languages

4.1.3.1. COBOL

COBOL operates at a similar level to third-generation languages, however it is generally grouped separately as the structure of COBOL is quite different from other languages.

COBOL is an acronym for "COmmon Business Orientated Language".

COBOL is designed for data processing, such as performing calculations and generating reports with large volumes of data. Vast amounts of COBOL code have been written and are particularly used in banking, insurance and large data processing environments.

In COBOL data fields are defined as a string of individual characters, and each position in the variable may be defined as an alphabetic, alphanumeric or numeric character.

Arithmetic can be performed on numeric items.

Processing is accomplished with procedural statements. COBOL statements use English-like sentences and COBOL code is easy to read, however a COBOL program can fill a larger volume of text space that an equivalent program in an alternative language.

COBOL is closely intertwined with mainframe operating systems and database systems, resulting in efficient processing of large volumes of data.

4.1.3.2. Fourth Generation Languages (4GL)

Following the increasing complexity of computer systems in the 1970's and 80's, attempts were made to define new higher-level languages that could be used to develop data processing applications using far less code than previous languages.

These languages involve defining screen and report layouts, and include a minimum amount of code for specifying formulas.

Fourth-generation languages, also known as application generators, are used for the development of business applications in large-scale environments.

However, while third-generation languages can be adapted to any programming task, fourth-generation languages are specific to business data processing applications.

4.1.4.   Object-Orientated Languages

Object-orientated languages have existed since the earliest days of computing in the 1960's, however they only grew into widespread use in the 1990's.

Many object-orientated languages are extensions of third-generation languages.

4.1.4.1. C++

C++ is a major OO language. C++ is an extension of C.

Some of the facilities of C++ include the ability to define objects, known as classes, containing data and related subroutines, known as methods. These classes may operate at several levels and sub-classes can be defined that include the data and operations of other class objects.

C++ also supports operator overloading, which allows operators such as addition to be applied to newly-created data types.

4.1.4.2. Java

Java is an object orientated language developed for use in internet applications. It has a syntax that is similar to C, but is not based on a previous language design.

Java includes dynamic memory allocation for creating and deallocating objects, and a strictly defined set of class libraries (subroutines) that is intended to be portable across all operating environments.

Java generally runs in a "virtual machine" environment for portability and security reasons.

4.1.5.   Declarative Languages

4.1.5.1. Prolog

Prolog is a declarative language. A prolog program consists of a set of facts, rather than a set of statements that are executed in order.

Prolog is used in decision-making and goal-seeking applications.

Once the program has been defined, the prolog interpreter then uses the defined facts in an attempt to solve the problem that is presented.

For example, a prolog program may include the moves of a chess game. The prolog interpreter would then use the facts that were defined in the program to determine the moves in a computer chess game.

4.1.5.2. SQL

SQL, Structured Query Language, is a data query language that is used in relational databases. SQL is a declarative language and groups of records are defined as a set, which can be retrieved or updated.

SQL statements may be executed in sequence however the actual selection of records is based on the structure of the selection statement.

4.1.5.3. BNF Grammars

A Backus Naur Form grammar is a language that is used to specify the structure of other languages.

The syntax of some programming languages can be fully specified using a set of statements written as a BNF grammar.

BNF grammars are declarative and specify the structure and patterns of a language.

4.1.6.   Special-Purpose Languages

4.1.6.1. Lisp

Lisp is a highly unusual language that was developed early in the history of computing, and used in artificial intelligence research.

All data items in Lisp are stored as lists. All processing in Lisp involves scanning lists.

The syntax of Lisp is very simple, but involves massive amounts of brackets as lists are defined within lists which are within other lists.

The program code itself is also stored in lists, as well as the data items.

4.1.6.2. APL

"A Programming Language", APL is a language that was developed for actuarial calculations involving insurance and finance calculations.

APL is highly mathematical and includes operators for matrix calculations etc. APL code is very difficult to read and the APL character set includes a wide range of special characters, such as reverse arrows ←, that are not used in other programming languages or general text.

APL makes extensive use of a large number of operators and symbols and contains few keywords.

4.1.6.3. Symbolic Calculation Languages

Some packages for evaluating and displaying mathematical functions include symbolic calculation languages.

These packages recognise mathematical equations, and can re-arrange equations and solve the equations for a particular variable.

In contrast, standard programming languages do not recognise full mathematical equations, but recognise expressions instead.

For example, the statement "$y = x * 2 + 5$" would be interpreted as "determine the value of "x", multiply it by two, add five and then copy the result into the variable "y" in most programming languages.

The equals sign is deceptive as it does not represent a statement of fact, as in an equation, but it represents an action to be performed.

The "=" sign is the assignment operator. In some languages a different symbol such as ":=" is used for the assignment operator.

Although the expression on the right hand side of the "=" sign is evaluated as an expression, the value on the left-hand side of the "=" sign is simply a variable name.

True mathematical equations such as "$y - 5 = x * 2$" are not recognised by standard programming languages.

However, a language that supports symbolic calculation would recognise this equation and would be able to calculate the value of any variable when given the value of the other variables.

Also, some symbolic calculation languages support calculations with true fractions.

In most programming languages ⅓ is treated as one divided by three, with a result of 0.333333. Multiplying this result by 3 would produce 0.999999, not 1.

Using true fractions, however, ⅓ * 3 = 1.

4.1.7.   Hosted Languages

Hosted languages are compiled and executed within an application.

The application may provide a development environment, which may include editing and debugging facilities.

The application also supplies the infrastructure necessary to display screen information, print, load data etc. A range of built-in functions would also be included, depending on the particular functions supported by the application

4.1.7.1. Macro Languages

Applications may include a language for specifying formulas or developing full program code within macros.

These languages may be specifically developed for the application, or they may be implementations of standard programming languages.

Accessing data and functions in other parts of the application may be slow, as multiple layers of software may be involved. However, accessing data and variables within the program itself may be reasonably fast,

depending on the interpreter used to run the code and the level of compilation used.

4.1.7.2. Database Languages

Some database systems include programming languages that can be used to develop complete applications.

In some cases these languages are interpreted and may execute relatively slowly, while in other cases a full compiler is available.

4.1.8. Execution Script languages

Running programs, coping files and performing other operating system functions can be done with languages such as shell scripts and job control languages.

Statements in script languages include program and file names, and may also support variables and basic operations such as "if" statements. Script languages may include pattern-matching facilities for searching text and matching multiple filenames.

Although compilers may be available in some cases, in general a script language is interpreted and run on a statement-by-statement basis by the operating system or command line interpreter.

4.1.9. Report Generators

Report generators allow a report to be automatically produced once the layout and fields have been defined.

Formulas can be added for calculations, however code is not required to produce the report itself.

Report generators are useful for producing reports that appear in a row and column format.

Document output, such as a page with graphs and tables of figures, could be produced by using a standard language in a graphical printing environment.

Alternatively, the document could be hosted in an application such as a word processor, with macro code used to update data from external sources.

4.2. Development Environments

4.2.1.   Source Code Development

The three fundamental elements of a development environment are an editor, a compiler and a debugger.

An editor is used to view and modify the source code.

Compilers compile the source code and produce an executable file. Alternatively, interpreters or virtual machines may provide the run-time infrastructure needed to execute the program.

A debugger is a program that is used to assist in debugging. Debuggers allow the program to be stopped during execution, and the value of data variables can be examined.

In some environments all three programs may be integrated into a single user interface, while in other environments separate programs are used.

4.2.2. Version Control

Version control or source-code control systems perform two main functions.

They generally allow previous versions of a file to be accessed, so that changes can be checked when debugging or developing new code.

Also, these systems allow a source file to be locked, so that problems do not occur where two people attempt to modify the same source code file at the same time.

4.2.3. Infrastructure

Programs are generally developed using other code facilities in addition to the language operations.

This could include subroutine libraries, functional engines, and access to operating system functions.

The code elements may be sourced from previous projects, a set of common routines across projects, standard language libraries and from external sources.

4.2.3.1. Changes to Common Code

Changes to existing common code can be handled in several ways.

For systems that are currently in development, or where future changes are expected, the systems using the subroutine or function can be updated to reflect the change.

For old systems where a large number of changes are not expected, a separate copy of the common code may be stored with the system. This avoids the need to continually update the system to simply reflect changes in the common code.

Alternatively, in some cases the existing code can be left intact and a new subroutine can be added with a slightly different name. This is impractical when a large number of changes are made, but may be used when external programs use the interface and the existing definition must be left unchanged for compatibility reasons.

4.2.3.2. Development of Common Functions

Common functions are usually reserved for functions that perform general calculations and operations that are likely to be used in other projects.

Future changes can be reduced if the subroutine supports the full calculation or function that is involved, rather than the specific cases that may be used in the current project.

Error handling in common code may involve returning error conditions rather than displaying messages, as the code may be called in different ways in future projects.

Also, the code may avoid input and output functions such as file access, screen display and printing. This would allow the functions to be used by different projects in different circumstances.

4.3. System Design

System design is largely based on breaking connections, and using different approaches for different components of the design.

This includes connections between the following components:

- Processes performed and the internal system design.
- The system design and the coding model.
- The user interface structure and the code control flow model.
- The application data objects and the system objects.
- Database entities and program structures, and the specific data.
- The sequence of function calls and the outcome.
- The outcome of a subroutine call and previous events.
- The outcome of a subroutine call and external data.
- Connections between major code modules.

4.3.1.   Conceptual Basis

A system will generally be built on the concepts that underlie a particular field or process.

For example, a "futures" transaction is a financial-markets transaction that involves fixing future prices for buying or selling a commodity.

An accounting system may treat this as a contract, with a set of payments and a finishing date.

An investment management system may treat this as an asset that had a trading value. This value would rise and fall each day as the financial markets moved.

This same instrument is recorded in completely different ways under different conceptual models.

One model records a value that rises and falls daily, while the other model records a set of payments over time.

In some cases the basic concepts and structures are clearly defined, and a system could be directly built on the same foundations.

However, designing a separate set of concepts for the system may lead to a more flexible internal structure that could be more easily adapted to different processing.

In other cases, a system may include functions that relate to more than one conceptual framework. In these cases, a separate set of concepts and models may need to be developed to implement both frameworks.

For example, in the previous case of the futures transaction, a system that performs both accounting an investment management functions would need to use an internal concept of a futures transaction that allowed the implementation of both the accounting view, of regular payments, and the investment view, of a traded asset.

For example, one internal approach would be to break the link between "accounts" and "transactions", and simply record cashflows. An account could be a collection of cashflows that matched a certain criteria.

Also, the link between assets and trading values could be broken, resulting in a model based on "actual cashflows" (historical transactions), "predicted casflows" (forecast payments) and "potential cashflows" (market value).

4.3.2.   Level of Abstraction

One approach to systems design is to directly translate the processes, data objects and functions that the application supports into a systems design and code.

The leads to straightforward development, and code that is clear and easily maintained.

However, this approach can also lead to a large volume of code, and produce a system structure that is difficult to modify.

A higher level of abstraction may be produced by defining a smaller number of more general objects, and using these objects and functions to implement the full application.

This approach may reduce code volumes and create a more flexible internal structure. However, the initial design and development may be more complex and the system may be more difficult to debug.

For example, rather than developing a set of separate reports or processes, a general report function or processing function could be developed, with the actual result being dependant on the flags and options selected.

At a higher level of abstraction, an implementation may include internal macro languages for calculations, fundamental operations on basic objects such as calculations on blocks of data, and a complete disconnection between the application objects and processes and the internal code structure.

This process may lead to even smaller code volumes and more flexible internal operations, with the disadvantage of longer initial development times and complex

transformations between internal objects and application processes.

Low levels of abstraction are useful when there is little overlap between processes, while high levels of abstraction are useful for systems such as software tools, and for processing systems that include a large number of similar operations.

Low levels of abstraction involve mainly application coding and little infrastructure development, while high levels of abstraction involve mainly infrastructure development with little application coding.

Output is generally more consistent with highly abstracted systems. In low abstraction systems, minor differences in formatting and calculation methods can arise between different processes.

4.3.3. Data Information vs. Structure Information

In general, system structures are simpler and more flexible, code volumes are smaller, and code is simpler when information is recorded within a system as data rather than as structure.

For example, a system may involve several products with a separate database table and program structure for each product.

The structure of the database tables and program objects would specify part of the information concerning the application elements.

A simpler approach would be to use a single product table and structure, and record the different products as data records and table entries.

A further generalisation would be to remove product-specific fields and processing, and store a set of general fields that could be used to specify options for a general set of processing.

In some cases an application may contain a large number of structures, objects and functions, while the database definition and code structure may implement a small number of general structures.

The application details would then relate to the data stored within the database and code tables.

In general a simpler system may result from using the minimum number of database tables and program structures.

4.3.4.   Orthogonality

When a set of options is available for a process, functionality may be increased if the options are independent and all combinations of the options are available.

For example, a report module may define a sort order, layout type and a subtotalling method.

If each option was implemented as a separate stage in the code, rather than implementing each combination individually, this would lead to orthogonal functionality.

In the case of two sort orders, five layout types and three subtotalling methods, there would be three major sections of code and $2+5+3 = 10$ total code sections.

However, the number of possible output combinations would be $2*5*3 = 30$ output report formats.

This function may have been derived from five separate report formats.

By combining the report formats, overlapping code would be eliminated leading to a reduction in code volume, and the number of output formats would rise from 5 to 30.

As another example, if a graphics engine supported three types of lights and three types of view frames, then in an orthogonal system any light could be used with any view frame.

In practice some combinations may not be supported, either because they would be particularly difficult to implement, they would take an extremely long time to execute, or because they specified logical contradictions.

For example, a list can be sorted in ascending or descending order, but cannot be sorted in both orders simultaneously.

When selected options conflict, one outcome may be selected by default.

4.3.5.   Generality

Generality involves functions that perform fundamental operations, rather than specific processes.

For example, a calculation routine that evaluated any formula would be a more general facility than a calculation method that implemented a fixed set of formulas.

A data structure that stored details of products would be a more general structure than creating a separate data structure for each product.

Performing processing based on attributes, flags and numbers is a more general approach than implementing specific combinations of options within the code.

For example, storing a formula on a record would be a more general approach than storing a variable that indicated that calculation method A should be used.

In cases where the same formula may appear on multiple records, a separate table could be created to represent a different database object, rather than including fixed options within the code.

Generality also applies to functions that accept variable input for processing options, rather than a value that is selected from a fixed list of options.

For example, a processing routine that accepted a number of days between operations would be a more general routine than an alternative that implemented a fixed set of periods.

In broad terms, a general process or function is one that is able to perform a wider range of functions than an alternative implementation.

4.3.6.   Batch Processes

Batch processes are processes that perform a large amount of processing, and are generally designed to be run unattended.

This may include reading large numbers of database records, or processing large amounts of numeric data.

When data problems occur during processing, this may be handled by writing an error message to a log file and continuing processing.

More serious errors, such as internal program errors or missing critical data may result in terminating the process.

4.3.6.1. Input Checking

Input data can be checked using a range of checks. For example,

- Data that is negative, zero or outside expected ranges.
- Data that is inconsistent, such as percentages that do not sum to 100.
- Data that is greatly different from previous values.

4.3.6.2. Output Checking

Interacting processes may involve a manual check of the results, however in the case of a batch process, large volumes of data may be updated to a database.

Output results can be checked using similar checks to the input data, to detect possible errors before incorrect data is written to a file.

4.3.6.3. Performance

Performance may be a significant issue with batch processes.

Data that will be read multiple times can be stored in internal program arrays or data record buffers.

The order in which the processing is performed can also have a significant impact on execution speed.

When a one-to-many database link is being processed, if the child records are read in the order of the parent key then each parent record would only need to be read once.

For example, if account records where processed in account number order, then the customer record would have to be read for each new account. However, if the account records were processed in customer number order, then the accounts for each customer would be processed in sequence and the customer record would only have to be read once.

4.3.6.4. Rewriting

Batch processing code is often very old, as it performs basic functions that change little over time.

Rewriting code may have lead to a reduction in execution time, particularly if the code has been heavily modified or a long time has elapsed since it was first written.

When the code has been heavily modified, the code may calculate results that are never used, read database records that are not accessed and re-read records unnecessarily.

This may occur when code in a late part of the process is changed to a different method, however the earlier code that calculated the input figures remains unchanged.

Re-reading records may occur as the control flow within the module becomes more complex.

Also, re-writing the code may enable structures in the database and the code to be used that were not available when the module was originally written.

4.3.6.5. Data Scanning

A batch processes may also be written purely to scan the database, and apply checks such as range checks to identify errors in the data.

4.4. Software Component Models

A software component is a major section of a program that performs a related set of functions.

For example, the user interface code handling screen display and application control flow may be separated into a major section of the program.

Calculation code may also be grouped into a separate section of code.

Software components may consist of a section of related code, or the functions performed by the component may be grouped into a clearly defined functional interface.

4.4.1.   Functional Engines

An engine is an independent software component that uses a simple interface, and has complex internal processing.

Database management systems are an example of a functional engine.

Other engines may include calculation engines, printing engines, graphics engines etc.

In some cases an engine executes as an independent process, and communication occurs through a program interface.

4.4.2.   Abstract Objects and Functions

Software components such as processing engines are generally more flexible and useful when they deal with a

number of fundamental objects and functions, rather than a set of fixed processes.

For example, a calculation engine may calculate the result of any formula that is presented in the specified format, and apply formulas to blocks of data.

This would be a more useful component than an engine that calculated the results of a fixed set of formulas.

4.4.3.   Implementation

The model of data and functions that appears on one side of the interface is independent of the internal structure of the code on the other side of the interface, which may use the same objects or may be structured differently.

For example, the code on the other side of the interface could implement the objects and processes directly, or translate function calls into definitions and attributes, and perform actual processing when data or output was requested.

Also, the interface objects themselves could be implemented within the software component using a smaller set of concepts. These objects could be used to implement the full range of objects and functions that were defined in the interface.

4.4.4.   Single Entry and Exit Points

Software components are generally more flexible when they operate through a single entry point.

Designing a single entry point interface may also lead to a clearer structure for both the interface and the internal code.

This involves defining a set of functions and operations that are supported by the component, and passing instructions and data through a single interface.

In some cases a single exit point could also be used.

In this case, the subroutine calls that are made within the component would be re-routed through a single function that accessed external operations. This function would then call the relevant external functions for database operations, screen interfacing etc.

This approach would allow the internal code of the component to be developed completely independently. Changing interfaces to external code, and using the component in alternative situations, could be done by modifying the external service routine.

4.4.5.   Code Models

Separating a system into major functional sections allows a different code model to be used in each section.

4.4.6.   Sequence Dependance

Components are more flexible when independent operations can be called in any order. For example, a graphics interface may define that textures, positions and lighting must be defined in that order, or alternately that the three elements could be specified in any order.

4.4.7.   Orthogonality

Orthogonality within an interface allows each option and function to be selected independently, and all combinations would be available.

This increases the functionality of the interface, and may also make program development easier through less restrictions of use of the software component.

4.4.8. Generality

A general functional interface is one that implements a wide range of functions.
This may involve passing continuous data values as input, rather than selecting from a pre-defined list of options.

Fundamental functions and operations may be more general than a software component interface that includes a range of application-specific functions and data objects.

4.5. System Interfaces

4.5.1. User Interfaces

4.5.1.1. Character Based Interfaces

Character based user interfaces involve the use of a
keyboard for system input, and fixed character positions
for video display.

Running functions may be done using menu options, or
through a command line interface.

4.5.1.2. Command Line Interfaces

A command line interface is based on a dialog based
system. This generally uses a character based interface
with keyboard input and a fixed character display.

Commands are typed into the system. The particular
function is performed, and processing returns to the
command line to accept further input.

Command line interfaces are often available with
operating systems, for performing tasks such as running
programs and copying files.

Some applications also include a command line
interface.

4.5.1.3. Graphical User Interfaces

Graphical user interfaces involve a graphical layout of windows and data, and the use of a mouse and keyboard to select items and input data.

Environments that support graphical user interfaces are generally multi-tasking, and allow multiple windows to be open and programs to be running at a particular time.

4.5.1.4. Other User Interfaces

In other circumstances, different devices may be used for both input from the user and the display or output of information.

Input devices could include control panels, speech recognition etc. Output devices could include multiple video displays, indicator panels, printed list output etc.

4.5.1.5. Application Control Flow

Control flow with applications may follow a menu system. This is used in both character-based and graphical environments.

Selecting an option from a menu may perform the specified function, or display a sub-menu.

Menu systems can be hierarchical, with sub menus connected directly to main menus. Functions may appear in one place or they may appear on multiple sub-menus.

In some cases sub-menus may be cross-connected and a submenu could jump directly to another submenu.

Other control flow models include event driven systems. In this model, functions can be performed by selecting

application objects or command buttons and performing the action required to run the function.

Application control flow may be process-driven or function-driven.

Process-driven control flow would involve selecting menu options to perform various processes, and may be used for data processing systems.

Function-driven systems focus on data objects rather than processes. This approach is commonly used for software tools.

In a function driven system, the object would first be selected, and then the function to run would be invoked.

Program code can be structured in the same way as the application control flow. This approach leads to a direct implementation, with the structure of the internal code following the structure of the application itself.

However, in other cases an indirect link may be more suitable. For example, a process-driven system with a large amount of overlap between processes could be implemented internally using a function-driven code structure, while a function-driven application may be implemented internally as a set of processes.

4.5.2. Program Interfaces

Program interfaces are used when a program connects to another program or software component.

This includes interfaces to subroutine libraries, database management systems, operating system functions, processing engines and other major modules within the system.

4.5.2.1. Narrow Interfaces

Narrow interfaces occur when the interface consists of a small number of fundamental operations and data objects.

The use of a narrow interface enables the code on each side of the interface to be changed independently.

Also, the interface itself may be easier to use.

This is a loosely-coupled interface, as in theory the application program could be coupled to a different software component that performed similar functions, possibly using a translation layer.

This may occur, for example, in porting an application to another operating system that used different screen or printing operations. An alternative screen or printing engine could be developed for the alternative operating system, and coupled to the main program.

An example of a narrow interface is a data query language, with statements passed to a database engine. The data retrieval and update operations are specified by the statement contained in the query text string.

A subroutine library that contained a large number of subroutines, each performing a simple function, would be a wide interface.

The functions in subroutine library could represent a closely-coupled interface, as the interface would contain a large number of links and interactions. This would prevent the code on either side of the interface from being changed independently, as this would also require significant changes on the other side of the interface. Also, the code could not be replaced with an alternative

library that used a different internal structure, as the
structure itself would be part of the interface.

Many program interfaces are implemented as subroutine
calls, however in the case of a narrow interface this may
consist of a few subroutines providing a link that passes
instructions and data through the interface.

A narrow functional interface does not necessarily imply
a slow data transfer rate, as would be the case in a
narrow bandwidth network.

A narrow functional interface may support the transfer of
large blocks of data, in contrast to a narrow data channel
which involves transferring data serially, with small data
items transferred in sequence, rather than a parallel
transfer of all the data in a single large block.

4.5.2.2. Passing Data

Data can be passed though interfaces using parameters to
subroutines, or pointers to data objects stored in memory.

Another approach involves the use of a handle.

A handle is a small data item that identifies a data object.
The handle is returned to the calling program, however
the data object itself remains within the software
component that was called.

For example, calling a graphics engine to load a graphics
object may result in the object being loaded into the
graphics engine's memory space, and a handle being
returned to the calling program.

The handle could then be used to identify the object
when further calls were made to the graphics engine.

4.5.2.3. Passing Instructions

Rather than containing a large number of subroutine calls to perform individual functions, a narrow program interface works by passing instructions through the interface.

This may take the form of a simple language, such as a data query language. A calculation engine may receive text strings containing formulas or simple statements for applying formulas to data objects.

Instructions can also be passed as a set of numeric operation codes.

For example, an engine for displaying wireframe images of structures could implement instructions to define an element, define connections between elements, define a view position and generate an image.

These functions could be implemented in an interface by using a numeric value to represent the operation to be performed, and a handle to indicate the data object.

This approach would lead to a simple interface that was easily extended, and would also support data transfer through network connections or inter-process communication channels.

4.5.2.4. Non-Data Linking

In general, a program interface passes data between processes, and subroutines and data variables on the other side of the interface cannot be directly accessed.

This allows the interface to be clearly defined, and prevents problems due to interactions between sections of code on either side of the interface.

Additionally, if instructions and data are passed through an interface channel, then the two code sections can execute independently, and may operate in separate memory spaces, through inter-process communication or through a network communication channel.

However, in some cases callback functions are implemented.

This occurs where the engine calls a subroutine within the main program.

In some cases callback functions are unavoidable. For example, a general sorting routine that applied to any type of object would need to call a subroutine in the main program to compare two objects, so that the order of the sorting could be determined.

4.6. System Development

4.6.1.   Infrastructure and Application Development

System development can be broken into two separate phases.

Infrastructure development involves developing facilities that are used internally within the program.

This could include developing data structures and the subroutines for operating on them, developing subroutine libraries, developing processing engines, developing common modules in process-orientated systems, and developing object models in object-orientated systems.

Application development involves writing code that performs the processing required by the system, handles the user interface, and performs the functions and operations that the application supports.

In some projects, the majority of design and coding may involve application coding. This may be based on straightforward processing, or it may be based on an existing infrastructure.

The infrastructure used could include facilities developed for earlier projects, or a combination of operating system functions, standard language libraries, and externally-sourced libraries.

In other projects the majority of the development process may involve infrastructure development, with the application comprising a collection of simple code that calls general processing facilities.

This may occur in processing applications with a large degree of overlap between similar functions, and also with function-driven applications such as software tools, where the application itself may be largely a user interface around a set of general facilities.

The development of some infrastructure can result in a system that uses less code, is more flexible, and is more consistent across different parts of the system.

However, developing infrastructure code is time consuming. Also, a complex infrastructure may make system development more difficult if the complexity of the infrastructure is greater than the complexity of the underlying application requirements.

4.6.2. Project Systems Development

Developing a system on a project basis involves a set of stages.

Typically a development process could include the following steps

- Requirements Analysis
- Systems Analysis
- Systems Design
- Coding
- Testing & Debugging
- Documentation

4.6.2.1. Requirements Analysis

Requirements analysis involves defining the broad scope of a system, and the basic functions that it will perform.

This may be based on a document that is supplied as an input to the systems development process.

Alternatively, the requirements analysis can be done at an early stage of the project, prior developing the system.

Ideally the requirements specification should outline the basic functions and operations that the system should provide, along with a set of desirable but non-essential functions.

4.6.2.2. Systems Analysis

Systems analysis involves defining the functions that the system should perform, including details of the calculations, processing, and the data that should be stored.

This process results in the production of a functional specification.

The functions and data required can be established by determining the outputs that the system should produce, and working backwards to the input data and the calculations that are necessary to produce the outputs.

Some additional data may be stored purely for record-keeping purposes, however this process would identify the fundamental data that was needed for the processing to be completed.

The functions specified may include interactive facilities and batch processing functions.

Exceptions and special cases are a minor issue in manual processing and calculation environments, but they are a significant issue in systems design.

For example, if one formula applied to 10,000 clients and a different formula applied to 1 client, then double the amount of code would have to be written to cater for the 1 client that used a different formula.

Processing exceptions arise when discontinued products or operations remain active, when separate arrangements apply to conversion from another product or situation, and when non-standard arrangements are entered into.

Special cases can be handled by consolidating a range of situations into a number of product features and options, including override values on individual records that override default processing values, and including user-defined formulas in database fields that can be added to a list of available formulas.

4.6.2.3. Systems Design

Systems design involves designing the internal structure of the system to provide the application facilities that are specified in the functional specification.

In some cases, the system design and code structure may follow the structure of the functional specification and the application control flow.

This may lead to a process-driven code structure for a data processing system, or a function-driven code structure for a software tool.

In other cases, a different code structure could be used.

For example, where there was a large amount of overlap between different processes, implementing a process-driven application using a function-driven code structure may result in a large reduction in code volume.

As another example, application modules may be grouped by process type, while the code may be grouped by internal function, such as screen display, calculation routines etc.

4.6.2.3.1. Future Expansion

In some cases, assumptions and restrictions in the functional specification could be relaxed to allow for future expansion.

For example, a many-to-many link could be implemented when the data is currently limited to one-to-many combinations, but future changes may be likely.

Also, a general calculation facility could be implemented, rather than programming a fixed set of existing formulas.

4.6.2.4. Code Development

In some cases a detailed system design document may be produced.

In other cases, code development may follow directly from the functional specification, or from a code structure that has been selected for implementing the system.

Testing and debugging follows a cycle of testing, correcting problems, releasing new versions of the test system and continuing testing.

4.6.2.5. Formal Development Methods

171

Formal development methodologies involve a highly structured set of steps in the development process. The processes and outputs of each stage are clearly defined.

This approach is used is some data processing development environments.

Formal methods define the stages in the development process. This includes the processes that are performed during each stage and the documents and diagrams that are produced.

Formal methods have advantages in managing and developing large data processing systems.

The steps are clearly defined, and the results of the development processes, both in terms of the time period involved and that shape of the final system can be clearly identified at the beginning of the project.

However, there are several disadvantages with formal methods.

Formal methods generate large volumes of paper documentation, and may involve several layers of meticulous development to produce a final section of code that may be quite straightforward.

A more serious disadvantage relates to the flow-through structure of the development process. There is a strong linking flow-through from the initial requirements, through to systems analysis and design, and finally coding.

This may result in the production of a system that closely matches the existing process model and data objects. While this system may perform the processing functions without difficulty, it may also be more complex and inflexible than alternative implementations.

A system design based on a range of general objects and functions may lead to a simpler data, code and user interface structure than a system that directly implements existing processes.

4.6.3.   Large System Development

A large system may be straightforward in design, but involve a large number of processes, database tables, reports and screens.

Also, a large development may involve a smaller amount of code, but may implement a complex set of operations.

4.6.3.1. Separation of Modules

Large system development may be more effective when major modules are clearly separated, and communicate through defined programming interfaces.

This enables each section to be developed independently. Also, where one section is not complete, another section can continue development by referring to the defined interface when including processes that require a link to the other module.

4.6.3.2. Consistency

Consistency in design, use of language features and coding conventions is important in large systems.

When code is consistently developed, different sections of the code can be read without the need to adjust to different conventions.

Also, consistency reduces the chance of problems occurring within interfaces between modules.

4.6.3.3. Infrastructure

Large developments generally involve the development of some common functions and general facilities.

In too little infrastructure is developed, a large amount of duplicated code may be written. This may result in a system that has large code volumes, is rigid in structure, and does not include internal facilities to enable future modules to be developed more easily.

Also, if too much infrastructure is developed, long delays may be involved and the final system may involve unnecessarily complex code.

4.6.4.   Evolutionary Systems Development

Evolutionary systems development involves the continual development of systems and releasing new versions on a regular basis.

Software tools are frequently developed and extended on an evolutionary basis.

The sequence of evolutionary development follows a pattern of adding a range of new features and facilities, and then consolidating a range of features into a new abstract facility.

This leads to minor and major changes, as features are continually added, and then a major upgrade occurs as the design is re-structured to include new concepts and structures that have arisen from the individual features.

Systems that are not restructured on a regular basis can become extremely complex, with execution speed and reliability decreasing as time passes and more individual features are added.

The sequence of minor and major changes applies to the internal code structure, and also the use of the application itself, and the objects and facilities that it supports.

Evolutionary systems generally become more complex as time passes. Existing features are rarely removed from systems, however new features and concepts may be continually added.

In some cases, this results in the system becoming so large and complex that it effectively becomes unusable and unmaintainable, and is eventually abandoned.

The life span of an evolutionary system can be extended by making major structural changes when the system design has become more complex than the functions that it performs, and by removing previous functions from the system as the system evolves in different directions.

4.6.5.   Ad-hoc System Development

Ad-hoc system development involves creating systems for unstructured and short-term purposes.

Examples include test programs, conversion programs and productivity tools.

In some cases ad-hoc developments may be done using a macro language that is hosted within an application such as a spreadsheet.

This allows a simple process to be performed without the overhead involved in developing a complete program.

The application provides the environment for loading data, displaying information, printing etc.

An alternative to using an application program as a host would be to develop a small temporary module within an existing system. This would allow the process to use the internal facilities of the main program to perform calculations and processing.

Execution speed may be less important in an ad-hoc development than in a permanent system.

This may allow simpler methods to be used to write the code, such as the use of strings, simple algorithms and hard-coded values.

In cases where the system is expanded in size and continues in use on a long-term basis, the original design and coding practices may be unsuitable for long-term operation.

This could be addressed through a major consolidation and re-design, and commencing on an evolutionary development path.

In other cases, particularly when the programming language or development environment itself is unsuitable for long term operation, a complete redevelopment of the system could be conducted.

4.6.6.   Just-In-Time System Development

Just-in-time techniques are based on responding to problems as they arise, rather than relying on forward planning to predict future conditions.

Just-in-time system development may involve developing and releasing sections of a system in response to processing requirements.

This approach involves responding to changes in processing requirements and adapting the system to changing conditions, rather using project developments over long timeframes that rely on an expectation of future conditions.

A Just-in-time development approach may be suitable for rapidly changing environments.

This approach avoids several problems that are inherent in project developments.

The requirement for a system often arises before the system is developed, not at the time that the development will be completed.

As some large development projects may take several years, this means that other methods must be found to perform the functions during the development period.

This may involve large volumes of manual processing, development of temporary systems, patchwork changes to existing systems, and populating databases with information that later proves to be in an unsuitable format.

Large development projects may be abandoned or never fully implemented, due to unmanageable complexity, flaws in design, or circumstances having changed so much by the time the project is completed that the system is not of practical use.

Disadvantages with just-in-time development include the fact that the process is deliberately reactive, and so an effective and flexible system design may not appear unless the process is combined with regular internal

redesign, and proactive changes based on a medium-term view.

This approach is similar to the evolutionary development approach. However, the evolutionary approach involves continuous development of a system, while the just-in-time approach involves a stable system that changes as required in response the changes in the processing environment.

4.6.6.1. Advantages

Using a just-in-time approach, systems are available at short notice. This may avoid problems with using temporary systems, and storing data in ways that later proves unsuitable for historical processing.

As the development adapts to changing conditions, the system may be more closely matched to the processing requirements that a system developed over a long period, and based on assumptions about the type of processing required

4.6.6.2. Disadvantages

A just-in-time approach could lead to a system that is structured in a similar way to the processing itself.

This would not generally be an ideal structure, as this would lead to a rigid code and data design that was difficult to modify.

An alternative would be to retain a separation between the system structure and the processing requirements, and adapt the system structure independently of the processing changes.

Testing could become an issue with just-in-time development, as frequent releases of new versions would increase the amount of testing involved in the development process.

If structural changes were not made regularly, a just-in-time process could degenerate into a maintenance process involving patchwork changes to a system.

This could occur when changes were implemented directly, rather than adapting the system structure to a new type of process.

If the development process is tied in too closely with the use of the system, then alternative system designs and structures may be lost and the system may be limited to the current assumptions and methods used in processing.

4.6.6.3. Implementation

In general, just-in-time development may be more effective when the system structure is adapted to suit a new process, rather than implementing the change directly.

For example, a change to a calculation could be implemented by adding a new facility for specifying calculation options, rather than including the specific change within the system.

This process would allow the system to retain a flexible and independent structure, based on a range of general facilities rather than fixed processes.

When a large number of changes had been made, internal structures could be changed to better reflect the new functions and processes that were being performed.

4.6.7. System Redevelopments

Many systems developments are based on replacing an existing system.

A system that has been heavily modified and extended often becomes complex, unstable and difficult to use and maintain.

Also, the functions and processes performed may change and a system may not include the functions required for the particular task being performed.

Redevelopments may follow the standard project development steps.

In some cases, a better result may be achieved by avoiding a detailed review of the existing system.

This may occur because the existing system will be based on a set of structures and assumptions that may not be the ideal structure to use for a new development.

Performing a detailed review of the existing system may lead to replacing the system with a new version based on the same structure, rather than designing a new structure based on the actual underlying requirements.

However, a review of the existing system after the design has been completed may highlight areas that have been missed, and particular problems that need to be addressed.

System redevelopments generally involve a data conversion of the data from the previous system.

This can be done by writing conversion programs to extract the fundamental data, and insert the data into the new database structure.

4.6.8.   Maintenance and Enhancements

System maintenance involves fixing bugs and making minor changes to allow continued operation.

This may include adding new database fields and additional processing options, or modifying an existing process due to changed circumstances.

In some cases, a system may appear to operate normally, however invalid data may occasionally appear in a database.

In these cases, error checking code can be added into the system to trigger a message when the invalid results are produced. This may enable the problem to be located by trapping the error as it occurs, enabling the data inputs and processing steps to be identified.

Alternately, in a batch process the data could be written to a log file when the invalid condition was detected.

Maintenance changes can have a high error rate, particularly in function-driven rather than process-driven systems.

This is related to the fact that the code may be unfamiliar.

Also, the system design is frozen, and changes must be made within the existing structure. This is in contrast to general development, when the structure of the code is fluid and can evolve as new code is added to the system.

In general, maintenance changes may be safer when they use the same coding style and structures as the existing code.

Also, implementing the minimum number of changes required to correct the problem is less likely to introduce new bugs than if existing code is also re-written.

Testing that a maintenance change has not disturbed other processes can be done by running a set of processes and producing data and output before the change is made.

The same processes can be re-run after the change to check that other problems have not appeared.

Enhancements are larger changes that add new features to the system.

These changes may be done on a project basis and include the full analysis and design stages.

At a coding level, problems are less likely to occur if the existing system structures are used, and the code is structured in a similar way to the existing code within the system.

An example would be adding a module for a new report to a system.

In this case, the code structures used for the other report modules could also be used for the new module.

In general, minor bugs and problems should be resolved where possible. A bug that appears to be a minor problem may be the cause of a more serious problem in another part of the system, or may have a greater impact in other circumstances.

When a large number of maintenance changes have been made to a system, the control flow and interactions in the code can become very complex. This can result in a change that fixes one problem, but leads to a different problem appearing.

This problem can be addressed by re-writing sections of code that have been subject to a large number of maintenance changes.

4.6.9.   Multiple Software Versions

In some systems, multiple versions of the system must be developed and maintained concurrently.

This may apply when customised versions of the program are supplied to different clients, or when the system is provided on several operating platforms.

Multiple versions can be implemented by using conditional compilation, which allows different sections of code to be compiled for different versions, keeping multiple versions of source code files, and using configuration parameters in files or databases to select processing options.

Extending the general functionality of a system to cover the range of different uses may be a better solution in some cases.

Maintaining multiple versions can lead to problems with tacking source code and executable files, and may involve maintaining larger volumes of code.

4.6.10. Development Processes Summary

Development Process	Description
Infrastructure Development	The development of general functions, common routines and object structures.
Application Development	Development of code to implement application specific functions and processes.
Project Development	Development of a system from analysis through to completion on a project basis.
Evolutionary Development	Continual development of a system.
Ad-hoc Development	Development of temporary and unstructured systems, such as testing programs and productivity tools.
Just-In-Time Development	Adapting a system to meet changing requirements, and implementing modules as they are developed.
System Redevelopment	Developing a system on a project basis to replace an existing system.
Maintenance	Minor changes to an existing system to allow continued operation.
Enhancements	Additional functions added to an

existing system.

4.7. System evolution

4.7.1.   Feature & Structural growth

System evolution involves continually adding new features to a system.

These features may include an expansion in the types of data that can be stored, an increase in the range of calculations and functions that can be performed, and an increase in the type of reports and displays that can be generated.

These new features increase the complexity and functionality of the system, however they do not affect the structure of the system or the concepts on which it is based.

As the system evolves, changes to the structure and underlying concepts of the system may also be made.

This includes abstraction, which combines specific objects into general concepts.

Also, the usage and purpose of the system may also evolve over time. Many systems are used for tasks that are very different from their original uses.

This can be addressed by changing the basic concepts on which the system rests to other structures.

For example, a system that was originally a processing system may evolve into a design tool, or vice versa, which would involve different code structures and basic functions.

When changes are made to system functions and the system structure no longer matches the functions that the

system performs, the system can become complex and unstable.

In this situation, re-aligning the internal structure with the functions that are performed may lead to a significant reducing in code volume, and an increase in processing speed and reliability.

Changes occur when the system is applied to different types of processing or environments, and also when techniques and approaches change.

This applies at both the coding and design level, and at the level of the objects and processes that the application uses.

4.7.2.   Abstraction

Abstraction is the process of combining several specific objects or concepts into a single, more general concept.

This applies within the code, within the objects and functions that are used in the application, and in database design.

The systems evolution process typically involves continually adding features, and periodically combining a range of features into a new abstract concept.

For example, in the code a number of statistical functions may be added over time. As the number and complexity grows, these functions could be replaced with a generate data set object that implemented a range of functions on sets of data.

In an application, formatting features could be added to a range of objects and processes. As the formatting features become increasingly complex, the individual formatting features could be replace with a single

187

"format" object that could be attached to any other
object.

This change could reduce the complexity of the code,
consolidate data items within the database and make the
system easier to use and more flexible.

However, implementing this change could also involve
changes in a large number of different code sections, and
may also require a database or file format conversion.

In a database structure example, a database may initially
contain a "customer" table. As the system changes, a
"supplier" and "shipping agent" table could be added.

These tables contain related information, and could be
consolidated into a single table using a general concept
of "counterparty".

4.7.3.   Changing the Code Model

As a system evolves there may be sections of code that
are more suited to a different code model than the
existing model.

For example, process-driven code could be changed to
function-driven code when a large amount of duplication
had appeared within the code.

An example would be writing a general report-
processing routine to replace a large number of similar
reports.

Function-driven code could be changed to process-driven
code when the functions had become complex, and could
be implemented more easily using a process approach.

An example would be a data conversion and upload
facility that had become extremely complex, and could

be re-written using a number of separate process-orientated routines.

Table driven code could be implemented when a large number of similar operations had appeared within the code, such as menu item definitions and definitions of data objects.

Object orientated code could be included in a function-driven section of code.

In some cases the object–orientated structure model may not be effective in process-driven code, and standard procedural operations may be simpler and clearer.

In some cases, a change to a different code model may commence, and during the re-writing it may become apparent that the new code model would not be an improvement on the existing structure. In this case, the newly-written code can be abandoned leaving the existing structure in place.

In other cases, a drastic reduction in code size and complexity can be achieved by changing to a different code structure.

4.7.4.   Rewriting Code

When code is initially implemented, the structure may follow the design and ideas behind the development.

After the code has been written, it may be able to be consolidated by rewriting sections to simplify the structure in the light of the fully written section of code.

This process can also be applied to code that has been modified several times.

In this case, the code may develop a complex control flow pattern and a large number of interactions between different sections of the code. This can make maintaining the code difficult and lead to an increased frequency of bugs and general instability.

This situation can be addressed by rewriting sections of code to simplify the structure and control flow.

Reviewing code may be beneficial if a significant period of time has elapsed since the code was developed. In an evolutionary system, there may be a range of subroutines and facilities that could be used to simplify the code that were not available when the code was originally developed.

4.7.4.1. Combining Functions and Modules

Within an application, and also within the code, changes to functions may result in two modules or application functions performing a similar process.

In these cases, a single process could be used to replace the previous processes and simplify the design.

4.7.5. Growing Data Volumes

Data volumes generally increase in systems over time, sometimes at dramatic compounding rates.

As data volumes grow, processes that were previously performed manually may be automated. This requires data to specify the processing options and calculation inputs.

In general, small data volumes may allow for free-format entry of customised information, while large data

volumes generally require fixed fields that can be used to perform automated processing.

4.7.6. Database Development

As the code and system features change, the database structure may also change.

Similar problems can arise with a number of database tables storing similar data, and inconsistencies between formats and structures in different parts of the database.

This can be addressed through changing the database definition to simplify the structure. This may involve merging related tables and using field attributes to identify separate types, or creating new tables to specify independent concepts that have arisen within existing tables.

4.7.7. Release Cycles

Systems that are continually modified are generally released as new versions on a periodic basis.

Minor upgrades would include bug fixes and minor enhancements, while major upgrades may include significant new functionality and possibly require database or file format conversions.

Some systems that are continually developed are effectively live at all times, with changes being introduced as soon as they are made.

This approach can be used with maintenance changes. However, problems can occur when larger and more frequent changes are introduced in this way.

Testing a full system is not feasible when each individual change is updated separately to the production version of the system.

This may lead to bugs occurring, and problems that lead to the incorrect updating of data may be difficult to unwind.

4.8. Code Design

4.8.1.   Code Interactions

Reducing interactions between different parts of a system can have several benefits.

Systems development and maintenance may be easier, as one section of code can be independently modified without the need to refer to other sections of code.

Bugs may be reduced, as a cause-and-effect that is widely separated through a system can produce problems when one part of the code is changed without related code sections being updated.

Portability is enhanced and independent modules can be used in other projects or rewritten for other environments.

In general, using data and operations that pass through interfaces between modules, rather than including interactions between one code section and a distant code section, results in a system that is more flexible, easier to debug and easier to modify.

4.8.2.   Localisation

Localisation involves grouping the processing that involves a particular variable or concept within a small section of code.

This may involve determining conditions within a region of code, and then passing flags to other subroutines, rather than passing the data value itself.

For example, if a variable is passed through several layers of subroutines and is then used in a condition, there is a wide separation between the cause and effect of the original definition of the variable, and its use in the condition.

This increases the interactions between different sections of the code, which can make reading and modifying the code more difficult.

If the condition is tested in the original location, however, and a Boolean variable is passed through, then the processing of the original variable is contained within the original section of code.

The subroutine using the Boolean flag would not access the original data variable, and the calling function could pass different values of the flag under different conditions.

This approach reduces the interactions in the code and leads to a more flexible structure.

4.8.3.   Sequence Dependence

Sequence dependence relates to the issue of calling functions in a particular order.

Code that is heavily dependant on the order of execution can be more difficult to develop and maintain than code that includes subroutines and general functions that can be called in any order.

For example, a graphics engine may include functions for initialising the engine, defining lights, loading objects, defining surface textures, defining backgrounds and displaying the image.

In a sequence dependant structure, each of these functions would need to be performed in a particular order.

Using a sequence-independent approach, the same functions could be called in any order. Valid results would be produced regardless of the particular order that the subroutines were called in.

A certain order may still be required for operations that are logically related, rather than related simply due to the code implementation.

For example, the texture of an object could not be defined until the object itself had been defined.

However, the lighting could be defined before the objects were loaded, or the objects could be loaded before the lighting was defined, as these are independent elements.

Sequence-independence increases the flexibility of functions, as a wider range of results may be produced using difference combinations of calls to the subroutines.

Also, less information needs to be taken into account when developing a section of code, which may simplify systems development.

Sequence dependence is reduced when functions are directly mapped. This results in the same results or process being performed, regardless of previous actions. Subroutines that operate in this way can be called in any order, and would produce the same result for the specified input.

Specifically, the use of static and global variables in subroutines can introduce sequence dependence.

This arises due to the fact that the result of the subroutine may be related to external conditions, or to previous calls to the subroutine.

This may result in a different order of execution of the same subroutine calls producing a different result.

Sequence independence can be implemented by separating the definition of object and attributes from the processing steps involved, or by checking that any pre-requisite functions have already been run.

4.8.3.1. Just-In-Time Processing

Sequence independence can be implemented using a "just-in-time" approach to processing and calculation.

Rather than performing each processing step as it is called, a set of attributes could be defined after each function call.

When the final function call is performed, such as the subroutine to display the image, the entire processing is performed in order using the attributes that were specified.

This approach separates the internal processing order from the order of external calls to the subroutines.

The model used in this approach is an attribute and data object model, rather than a process and transformation model.

In this model, each call to the subroutines defines a set of attributes, rather than executing a process.

For example, the generation of a graphics image could involve loading wireframe structures, mapping textures to objects, applying lighting and generating the image.

One approach would be to call a subroutine for each stage in the process. However, these operations may need to be performed in order.

Another approach would be to implement similar subroutine calls that simply defined the objects, textures and lighting, with all the processing being performed when the final "generate image" subroutine was called. This second approach would allow the calls to be performed in any order.

4.8.3.2. Pre-requisite Checking

Another approach involves each subroutine checking that any pre-requisite subroutines had been run.

When each process is performed, a flag could be set to indicate that the data had been updated.

When a subroutine commences it could check the flags that were related to any pre-requisite functions, and call the relevant subroutines before performing its own processing.

This approach avoids making assumptions about previous events, and would allow the subroutine to be used in a wider range of external circumstances.

When all subroutines were implemented this may, the sequence of calls in the main routine could be replaced with a single call to the final routine, which would lead to a backward-chaining series of subroutine calls, followed by a forward chain of execution.

4.8.4.   Direct Mapping

A subroutine or interface function is directly mapped if the operations that are performed are completely specified by the input parameters.

This occurs when the results of the subroutine are not affected by previous calls to the subroutine or by data in other parts of the system.

Direct mapping can be implemented by passing all the data that a subroutine uses as parameters to the subroutine.

Directly mapped subroutines are sequence independent, as their results are not affected by previous operations.

This enables directly mapped subroutines to be called in any combination and in any order.

Indirect mapping is created by storing internal static data that retains its value between calls to the subroutine, or by accessing data outside the subroutine.

For example, the subroutine "list_next_entry()" is not directly mapped, as this subroutine returns the next item in the list after the previous call to the subroutine.

In contrast, the subroutine "list_next_entry(current_entry)" is directly mapped, and would return the same result for any given value of "current_entry", regardless of previous operations.

Direct mapping can increase functionality by enabling calls to the subroutine from different code sections to overlap without conflicting, as well as introducing sequence independence.

For example, the first version of the subroutine allows only one pass through the list at a particular time. However, using the directly mapped version, multiple independent overlapping scans of the list could be

performed by different sections of the code, without conflicting.

A reduction in execution speed could occur when a search is required to locate the current position, such as the "current_entry" variable in the previous example.

This problem can be reduced by recording an internal static cache value of the previous call to the subroutine, and using that value if the parameter specifies the previous call as the current position. This does not introduce sequence dependence or indirect mapping, as the static variable is only used to increase execution speed and does not affect the actual result.

Another example concerns a subroutine "get_current_screen_field()", which returns the text value that is held in the current screen field.

In this case the indirect mapping is not due to static data relating to previous calls to the subroutine, but is due to accessing a variable outside the subroutine.

This code could be re-written as "get_screen_field(field)", in which case the function is direct-mapped and could be used to retrieve the value of any field on the screen.

Subroutines written in a directly mapped way may be more flexible and easier to use that subroutines that include indirect mapping.

Interactions between different sections of the code are also reduced when subroutines are directly mapped.

4.8.5.   Defensive Programming

Defensive programming is a coding style that attempts to increase the robustness of code by checking input data,

and by making the minimum number of assumptions about conditions outside the subroutine.

Robust code is code that responds to invalid data by generating appropriate error conditions, rather than continuing processing and generating invalid results, or preforming an operation that leads to a system crash, such as attempting to divide a number by zero.

Defensively coded routines may also check their own output before completing. For example, a process that produces figures may check that the figures balance to other figures or are consistent in some other way.

Checking output results does not affect the robustness of the individual routine against external problems however this approach may increase robustness throughout the system.

4.8.5.1. Input Data

Checking input parameters may involve two issues. Values can be checked to ensure that incorrect operations are not performed, such as attempting to access a null pointer or processing a loop that would never terminate.

Also, checks can be performed to detect errors, such as negative numbers in a list that should not contain negative numbers, or weights that do not sum to 1.

When lists are scanned and calculations are performed, checks of existing figures can also be made while other processing is being performed.

4.8.5.2. Assumptions about Previous Events

Checking previous operations and conditions outside the subroutine can be done by checking flags that are set by other processes.

When a pre-requisite process has not been performed, the subroutine can run the relevant function before continuing.

4.8.6.   Segmentation of Functionality

4.8.6.1. Functional Applications

In many systems the code can be grouped into several major areas.

This may include the user interface code, calculation code, graphics and image generation, database updating etc.

In functional applications such as software tools, separating the code into major areas may reduce the number of interactions between different sections of the code.

This may lead to a system that is more flexible and easier to maintain, as each section performs independent functions and can be modified without the need to refer to other parts of the system.

4.8.6.2. Process-Driven Applications

In process-driven systems such as data processing applications, separating the code by function instead of process may result in an internal structure that is completely different from the structure of the application itself.

For example, the application may consist of several functional modules, such as an accounting module, an administration module, a product reporting module etc.

Using a process-driven code structure, each application module would contain the code relating to that application module, including the user interface code, calculations, database updating etc.

However, if these functions were grouped internally into separate sections, this could result in a significant reduction in the total code size and reduce inconsistencies.

This process would also result in a system that would provide facilities to enable new application modules to be developed with a minimum amount of code. In contrast, the development of new process-orientated modules is not affected by the development of previous modules.

4.8.6.3. Implementation

Separating functions can be done by creating major sections within the code that contain related functions. These could be accessed through a defined set of subroutine calls and operations.

These modules could be extended to create processing engines. For example, a calculation engine may perform a range of general calculation and numeric processing operations. The engine would be accessed through a clearly defined interface of data and functions, and may be called directly or may execute independently of the main program.

Each section only remains independent while it does not contain processing that relates to another section of the system.

For example, a calculation section would not contain input/output processing, such as screen, file or printing operations.

This approach allows the code section to be used as an independent unit.

Portability between different systems is also improved with the approach.

Maintenance and debugging of a code sections may also be simpler when the section contains only one type of related operation.

Additionally, modules such as calculation engines and graphics facilities developed in this way could also be used with other projects.

4.8.7.   Subroutines

4.8.7.1. Selection

In general, code is simpler and clearer when a collection of small subroutines is used, rather than a number of larger subroutines.

Where a loop contains several statements, these may be split into a separate subroutine. This results in producing two subroutines. One subroutine contains looping but no processing, while the other contains processing and no looping.

When a large subroutine contains separate blocks of code that are not related, each major block could be split into a separate subroutine.

When code that performs a similar logical operation
exists in several parts of a system, this could be grouped
into a single subroutine

For example, different sections of code may scan and
update parts of the same data structure, or perform a
similar calculation.

The variations in the process could be handled by
passing flags to the subroutine, or by storing flags with
the data structure itself, so that the subroutine could
automatically determine the processing that was
required.

Code that was similar by coincidence but performed
logically independent operations, such as calculation
routines for two independent products, should not
generally be combined into a single subroutine.

4.8.7.2. Flags

Subroutines are more general if they are passed flags and
individual options, rather than being passed data and then
testing conditions directly.

For example, a subroutine may be passed a Boolean flag
to specify an option, such as sorting a list in ascending or
descending order.

Other options could also be passed. For example, the key
of a data set could be passed, rather than an entire
structure type that contained the key as one data item.

Using flags and options allows the subroutine to be
called in different ways from different parts of the code.

Also, using flags allows the subroutine to perform
different combinations of processing, in contrast to

checking the data directly, which may result in a fixed result dependant on the data.

Interactions with different parts of the code may also be reduced.

4.8.8.   Automatic Re-Generation

In some cases a set of input data and calculated results may be stored in a data structure.

If the input data is changed, the calculated results may need to be re-generated.

Unnecessary re-calculation can be avoided by including a status flag within the data structure.

When the input figures are changed, the status flag is set to indicate that the calculated results are now invalid.

When a request is made for a calculated figure, the status flag is checked. If the flag is set, then a recalculation is performed and the flag is cleared.

This approach enables multiple changes to the input data to be made without generating repeated re-calculations, and allows multiple requests for calculated results to be made without requiring multiple re-calculations.

This approach can only be used when the calculated results are accessed via subroutines, rather than being accessed directly as data variables.

4.8.9.   Code Design Issues Summary

Design Issue	Description
Interactions	Interactions between distant parts of a program, through global variables, static variables or chains of subroutine calls, can make programs more difficult to debug and modify.
Localisation	Grouping processing relating to a single data item or structure into a small section of code may make code easier to read and modify.
Sequence Dependence	Subroutines that can be called in any order, and which produce the same result regardless of previous events, may be more flexible than alternative approaches.
Pre-requisite checking	Checking within a subroutine that previous functions have been performed, and executing them if necessary, may increase the flexibility and generality of a subroutine.
Direct Mapping	Subroutines that produce a result that is completely specified by their parameters can make program development easier and may reduce bugs caused by

unexpected interactions.

Defensive Programming	Subroutines may be more robust when they check the parameter data that is passed, check external data that is used, and make a minimum number of assumptions about previous events and current conditions.
Segmentation of functionality	Separating major components of a system into separate sections may lead to easier program development and more reliable systems, as each component can be developed independently. Clear interfaces may be useful in designing general functional components.
Subroutine Selection	Systems composed of a large number of small subroutines may be easier to interpret, debug and modify that systems composed of several large and complex subroutines. Several small subroutines may be clearer than a single complex subroutine.
Subroutine Parameters	Subroutines may be more general, and interactions may be

reduced, when subroutines are passed flags and options, rather than data used for internal conditions. Data used in internal conditions causes interactions with distant code and results in a fixed outcome for an individual data condition.

4.9. Coding

Coding is the process of writing the program code.

4.9.1.   Development of Code

Although a general design may be planned, the direction that various sections of the code take may not become apparent until the coding is actually in progress.

Some functions turn out to be more complex and difficult to implement than was originally expected, while others may turn out to be considerably simpler.

An initial development of the code could result in creating a set of medium-sized subroutines to perform the various functions.

After these are written, the code could be consolidated by extracting common sections into smaller subroutines.

This process leads to a larger number of smaller subroutines, and generally clearer and simpler code.

Attempting to predict the useful subroutines in advance can lead to writing code that is not used. Also, the structure of the code becomes artificially constrained, rather than changing in a fluid way as different sections are re-written to simplify the processing.

This approach applies particularly to function-driven code, which uses a range of general-purpose functions and data structures. This approach is less applicable to process-driven code

4.9.2.   Robust Code

The reliability of code is determined when it is written, not during the testing phase.

As code is written, the ideas and structures that are being implemented are used to form complete sections of code.

This is the time at which the interrelations between variables, concepts and operations are mentally combined into a single working model.

Various issues are resolved during this process, such as the handling of all input data combinations and conditions, implementing the functions that the code section supports, and resolving problems with interfaces to other sections of code.

If coding commences on a different section of the system while issues remain unresolved, this may lead to problems occurring at future times.

Robust code would generally respond to invalid data by generating appropriate error conditions, rather than continuing program operation and producing invalid output or attempting an operation that results in a system crash, such as attempting to divide a number by zero.

4.9.3.   Data Attributes Vs. Data Structures

Data in related but different structures can be stored in separate database tables and structures, or combined into a single structure and identified separately using a data variable.

Data structures, code and functions are generally simpler and more flexible when a data variable is used to identify separate types of data, rather than creating separate structures.

For example, corporate clients and individual clients could be stored as separate database tables and program structures.

However, as these two objects record similar fundamental data, they could be combined into a single table.

If a large number of fields apply to only one type, this can be split into a separate table with a one-to-one database link. However, when the number of fields is not excessive, a single table may be the simpler solution.

As another example, creating a several separate tables for different products would lead to a more complex system than using a single table, with data fields to identify product options and calculations.

4.9.4. Layout

The visual layout of code is important. Writing, debugging and maintain code is significantly easier when the code is laid out in a clear way.

Clear layout can be achieved with the ample use of whitespace, such as tabs, blank lines and spaces.

Code written in languages that use control-flow structures, such as "if" statements, can be indented from the start of the line by an additional level for each nested code block.

For example, statements within an "if" statement may be written with an additional 2, 4 or 8 spaces at the start of the line, to identify the nesting of the code within the "if" statement.

Consistency in a layout style throughout a large section
of code leads to code that is easier to read.

For example, the following code uses few spaces and
does not indent the code to reflect the changes in control
flow.

```
for(i=0;i<max_trees;i++)
{       if(!tree_table[i].
            active){
            tree_table[i].usage_count=1;
        tree_table[i].head_node=new node;

        tree_table[i].active=TRUE;}
        queue=queue_head;
        while(queue!=NULL){if(queue->tree==i)
            queue->active=TRUE;
        queue=queue->next;
}
}
```

The following code is identical to the previous example,
however it includes additional spaces, uses a consistent
layout style throughout the code, and is indented to
reflect the nesting structure of the code blocks.

```
for (i=0; i < max_trees; i++)
{
        if (! tree_table[i].active)
        {
                tree_table[i].usage_count= 1;
                tree_table[i].head_node = new
node;
                tree_table[i].active = TRUE;
        }

        queue = queue_head;

        while (queue != NULL)
        {
                if (queue->tree == i)
                        queue->active = TRUE;

                queue = queue->next;
        }
}
```

4.9.5. Comments

A comment is text that is included in the program code for the benefit of a human reader, and is ignored by the compiler.

Comments are useful for adding additional information into the code. This may include facts that are relevant to the section of code, and general descriptions of the processes being performed.

When a subroutine contains a large number of individual statements, comments can be used to separate the statements into related sections.

The following example contains two comments that add information that may not be readily determined from reading the code

```
    ' A year is a leap year if it is divisible by four, is not
    ' divisible by 100, or is divisible by 400


if (year mod 4 = 0 and year mod 100 <> 0) or year mod 400 = 0) then
    is_leap_year = true
else
    is_leap_year = false
end


    ' reduce value by 1 day's interest for February 29th

if is_leap_year then
    value = value / (1 + rate/365)
end
```

The following code illustrates the use of comments to separate groups of related statements

```
subroutine process_model()

        ' intialse graphics engine and load
model

    graphics.initialise
    graphics.allocate_data
    read_model graphics, model

        ' update graphics image

    graphics.define_lights light_table
    grphics.define_viewpoint config_date
    grphics.display

        ' generate report

    process_report_data
    print_report

end
```

4.9.6.   Variable Names

The choice of variable names can have a significant impact on the readability of program code.

Variable names that describe the data item, such as "curr_day" and "total_weights" add more information into the code than variable names that are general terms such as "count" or "amount".

Single letter variable names are sometimes used as loop variables, as this simplifies the appearance of the code

214

and allows the reading to focus on the operations that are being performed.

For example, the letters "i" and "j" are sometimes used as loop variables. This usage follows the convention used in mathematics and early versions of Fortran, where the letters "i", "j" and "k" are used for array indexes.

This particularly applies to scanning arrays that are used to store lists of data.

In other cases, the use of a full name for a loop variable, such as "curr_day" or "mesh" may lead to clearer code. This may apply when the items in an array are accessed directly and the index value itself represents a particular data item.

Some coding conventions add extra letters into the variable name, or change the case, to describe the data type of a variable or its scope.

For example, "iDay" may be an integer variable, and the uppercase letter in "Print_device" may indicate that the variable is a global variable.

Constants may be identified with a code or with a case change, such as the use of uppercase letters for a constant name.

Using different variable names that differ only by case or punctuation, such as "ListHead", "listhead" and "list_head" can make reading and interpreting code more difficult.

Ambiguous names can also result in code that can be difficult to interpret.

For example, the word "output" can be both a noun and a verb. The name "report_output" could mean "output (print) the report", or "this is the report output".

In some cases, variables at different levels of scope may have the same name. For example, a local variable may be defined with the same name as a global variable.

In these cases, the name is taken to refer to the data item with the tightest level of scope.

Reading the code may be more difficult in these cases as an individual name may have different meanings in different sections of the code.

4.9.7.   Boolean Variable Names

The names of Boolean variables may be easier to read if the word "not" is not included within the variable name, and if the value is stated in the positive form rather than the negative form.

This avoids the use of double-negatives in expressions, which involves reversing the description of the variable to determine the actual condition being tested.

For example, the flag "valid" could also be named "not_valid", "invalid" or "not_invalid". Depending on the condition, the other forms may be less clear that the simple positive form.

4.9.8.   Variable Usage

Code may be easier to read when there is a one-to-one correspondence between a data concept and a data variable.

In general, a single data variable should not be used for two different purposes at different points within the code.

Also, two different data variables should not be used to represent the same fundamental data item at different points in the code.

This issue relates to usage within a common level of scope, such as local variables within a subroutine, or module-level variables within a module.

For example, the items in a list may be counted using one variable, and this value may then be assigned to a different variable for further calculations.

In another case, a variable may be used to store a value for a calculation in an early part of the subroutine. The same variable may then be used to store an unrelated value in a later part of the code.

In these cases, the code may be misinterpreted when future changes are made, leading to an increased chance of future bugs.

4.9.9.   Constants

In most languages, a fixed number or string within the code can be defined as a "constant".

A constant is a name that is similar to a variable name, but has a fixed value.

The use of constants allows all the fixed values to be grouped together in a single place within the code. Also, when a constant is used in several places, this ensures that the same value is used in each place, and avoids possible bugs where one value is updated but another is not.

Storing constants within the program code is known as hard-coding.

This reduces the flexibility of the code, as a program change is required whenever a constant value is changed.

The flexibility of a system may be increased if constants are stored in database records or configuration files.

Alternatively, in some cases the use of a constant value can be replaced by variable values or flags.

For example, a constant value that specified a particular record key could be replaced by a flag on the record that indicated that a particular process should occur.

As another example, a constant number, such as a processing value, could be replaced by a standard database field containing a variable value.

Constants are useful for values that are fixed or change rarely, such as the days of the week, scientific constants, or a product name.

Constants are also by used for fixed values that are used in programming constructions, such as end-of-list markers, codes used in data transmission, intermediate code etc.

The name of the constant refers to the meaning of the data, not its value. Two values that were the same by coincidence, such as the size of two different array types, should be defined as two separate constant values.

Generally a hard-coded value should not be included in a program to represent data that could be independently changed, such as the key of a database record. Including constants such as this would decrease the flexibility of the program, and would lead to incorrect processing if the data value was changed.

When processing applied to a certain record, this could be specified by including an option flag on the record itself.

4.9.10. Subroutine Names

Subroutine names may consist of several words. When the subroutine name contains a noun and a verb, this can be listed in either order.

For example, the following code defines variables in the verb-noun order.

```
define_report
format_report
print_report
```

This usage follows the usual sequence of a sentence. The opposite form, in the noun-verb order, can also be used

```
report_define
report_format
report_print
```

This approach groups related functions together, and highlights the data object rather than the function.

Code written in this way has a clearer separation between the individual objects and functions. This approach is used in object orientated programming.

4.9.11. Error Handling

Errors occur when an invalid condition is detected within a section of program code.

This may be due to a program bug, or it may be the result of the data that has been used in the calculation or process.

Errors at the coding level include problems that lead to system crashes or hanging, and problems with data and data structures.

System problems include attempts to divide a number by zero, loops that never terminate, and attempting to access an invalid pointer or an object that has not been initialised.

Data problems include lists that are empty when they should contain data, data within a structure that contains too many or too few entries, or an incorrect set of values for the process being performed.

At the process level, errors may include input data that is invalid, missing data, and input data that is valid but produces results that are outside expected ranges.

4.9.11.1. Error Detection

Errors can be detected by using error checking code.

These statements do not alter the results of the process, but are included to detect errors at an early stage.

This is used to assist in the debugging process, and to prevent incorrect data being stored and incorrect results from being produced.

Error detection can involve checking data values at points in the process for values that are negative, zero, an empty string, or outside expected ranges.

At the process level, checks could include performing reverse calculations, scanning output data structures for

invalid values and combinations, and checking the relationship between various variables to ensure that the output values are consistent with each other and with the input values.

Error detection code may add complexity and volume to the code and slow execution.

However, this code may also simplify debugging and detect errors that would otherwise lead to incorrect output.

Adding permanent error checks that are simple and execute quickly can be used to increase the robustness of the code.

4.9.11.2. Response to Errors

When an error condition is detected, various actions can be taken.

Errors within general subroutines and functions may be handled by returning an error status condition to the calling routine.

Alternatively, in some languages and situations an exception is generated, and this exception results in program termination unless it is trapped by an error-handling routine. The error handling routine is automatically called when the exception condition arises.

When the error occurs within a main processing function, an error message may be displayed for an interactive process, or logged to an error log file for a batch process.

Following the error, the process may continue operation and produce a partial result, or the current process may terminate.

4.9.11.3. Error Messages

Error messages may include information that is used in the debugging process.

Information that can be included in an error message includes the following points:

- An error number.
- A description of the error.
- The place in the program where the error occurred.
- The data that caused the error.
- Related major data keys that can be used to locate the item.
- A description of the action that will be taken following the error.
- Possible causes of the error.
- Recommended actions.
- Actions to avoid the error if the data is actually correct.
- A list or description of alternative data that would be valid.

For example

> "ERROR P2345: A negative policy contribution of $-12.34 is recorded for policy number 0076700044 on 12/03/85. Module PrintStatement, Subroutine CalcBalance"

> "ERROR 934: Model "ShellProfile" has no mesh type. This error may occur if the conversion from version 3.72 has not been completed. If this model is a continuous-surface model, the option "continuous surface" in the

model parameters screen can be selected to enable correct calculation. Module RenderMesh, Subroutine MeshCount"

In some cases, help information is included with an application that provides addition information regarding the error condition.

For the programming perspective, the key elements of an error message are a way to identify the place in the code where the error occurred, and information that would enable the data that was being processed to be identified.

This particularly applies to general system errors such as divide-by-zero errors.

4.9.11.4. Error Recovery

In some processes a significant number of errors may be expected.

This occurs, for example, in compilers while compiling program code, and in data processing while processing large batches of data.

In these cases, ending the process when the first error is detected could result in repeated re-running of the process as each error is corrected.

Error recovery involves taking action to continue operation following an error condition, to generate other results or generate partial output, and to detect the full list of errors.

During program compilation, a compiler will generally attempt to process the entire program and produce a list of all compilation errors within the code.

When an error is detected, an error message may be logged to a file. Depending on the situation, error recovery may involve ignoring the error and continuing operation, terminating part of a process and continuing other parts, using a default value in place of an invalid one, or artificially creating data to substitute for missing information.

4.9.11.5. Self-Compensating Systems

A self-compensating system is a system that automatically adjusts the system to correct existing errors.

This can be done by calculating the expected current position, the actual current position, and creating records to adjust for the difference.

This is in contrast to a process that simply performs a fixed process and ignores other data.

For example, a monthly fee process may create monthly fee transactions. If a previous transaction is incorrect, this will be ignored by the current month process.

However, if the monthly process calculated the year-to-date amount due, and the year-to-date amount paid, and then created a transaction the represent the difference, this approach would adjust for the existing incorrect transaction.

Separate values could also be created for the expected amount and the unexpected adjustment.

4.9.12.   Portability

Portable code is written to avoid using features that are specific to a particular compiler, language implementation, or operating environment.

Code that is written in a portable way is easier to convert to different development environments.

Also, some portability issues affect the clarity and simplicity of the code, and a section of code written in a portable way may be easier to read and maintain than an alternative section that uses implementation-specific features.

4.9.12.1. Language Operations and Extensions

Many compilers support extensions to the standard language, through the implementation of additional keywords and operations.

Some language operations and parameters are not specified as part of the language definition. This may include the size of different numeric data types, the result of certain operations, and the result of some input/output operations.

Code that is dependant on unspecified features of the language, or that uses compiler-specific extensions to the language, may be less portable than code that avoids these features.

Some languages do not have a single definition, and significant differences in the language structure may occur between implementations. In these cases a subset of common features across major implementations could be used to reduce portability problems.

4.9.12.2. External Interfaces

User interface structures, file and database structures and printing operations may all vary from operating system to operating system.

In general, conversion to alternative environments may be simpler when related processes are grouped together. For example, all the user interface code, including calls to operating system functions, could be grouped into a user interface module.

4.9.12.3. Segmentation of Functionality

In many systems the code can be grouped into several major sections.

One section may be the user interface code, which displays screens, menus, and handles input from the user.

Other major sections may include a set of calculation and processing functions, and formatting and printing code.

This would increase portability by enabling one section to be modified for an alternative environment, while the other sections remain unchanged.

However, this approach would not apply to an event-driven system that used a complex user interface design, with small processing steps attached to various functions.

4.9.13.   Language Subsets

Using a subset of a language involves using a set of simple fundamental operations. This would involve avoiding alternative operations, and complex or little-used language features.

Code written using a subset of the full language may be simpler and easier to read than code that uses functions from previous versions of the language and little-used language features.

For example, a language may support several looping statements that provide similar functionality. Writing code in a language subset may involve using one of the loop constructs throughout the code.

4.9.14. Default Values

A default value is a value that is used when a data item is unknown or is not specified.

Default values can be used to avoid specifying multiple options for a particular process. For example, the default values for each option may be specified in a database record or configuration file. These default values would be used unless a specific option was overridden by another value.

Default overrides may occur at multiple levels. For example, a default value could be specified for an option across an entire system, which could be overridden by a separate default value for a particular group of records, which in turn could be overridden by a specific value in an individual record.

The defaults at each level could be independently changed.

When a default override is removed, the actual value used returns to the default value of the previous level.

The term default value is also used in the context of an initial value. This is an initial value placed in a data record, data structure or screen entry field.

However, when initial values are changed the previous
value is lost, unlike default values which can be restored
by removing the override value.

4.9.15.   Complex Expressions

In some cases the clarity of a complex expression can be
improved by breaking the expression into sub-parts and
storing each part in a separate variable.

This allows the variable names to add more information
into the code, and identifies the structure of the
expression.

For example,

```
total = (1 + y)^(e/d)*(c*(1-(1+y)^-n)/y +
fv/((1 + y)^n))
```

This code could be split into several components to
clarify the structure of the expression.

```
    part_period_discount = (1 + y) ^ (e/d)

    annuity = (1 - (1 + y)^(-n))/y

    face_value_pv = fv / (1 + y)^n

    total = part_period * (c * annuity +
face_value_pv)
```

Adding additional brackets into a complex expression
may also clarify the intended operation.

4.9.16. Duplicated Expressions

When a particular expression occurs more than once within a program, it can be placed in a separate subroutine.

This may avoid problems when one version of the expression is changed but the other version is not.

These types of bugs can remain undetected for long periods of time, because the system may continue to operate and appear to be working correctly.

For example

```
    if sample_type = "X" and result_value <
0.35 then
        print_data
    end
    …
    …
    …
    if sample_type = "X" and result_value <
0.33 then
        include_in_averages
    end
```

In this example, the second expression is further through the code. The first expression prints records with a value of less than 0.35, while the second expression calculates the average of all records with a value of less than 0.33.

The intention may have been for this to represent the same expression, and the printed average to be the average of the printed records.

This bug would not affect system operation, and would not be detected unless the printed figures were re-calculated manually to check the "average" figure.

Writing a subroutine to replace these expressions and implement the expression in a single place would avoid this problem.

Also, the fixed number could be replaced with a constant such as RESULT_TOLERANCE, which adds information into the code and would allow this constant to be used in other places within the code.

Where two expressions were the same by coincidence, but they had a different meaning, they should not be combined into a common subroutine.

4.9.17.   Nested "IF" statements

Nested "if" statements can be difficult to interpret.

In general, a series of separate "if" statements may be easier to read and interpret than a set of nested "if" statements.

For example, the following code determines whether a year is a leap year, using nested "if" statements.

```
if year mod 4 = 0 then
    if year mod 100 = 0 then
        if year mod 400 = 0 then
            is_leap_year = true
        else
            is_leap_year = false
        end
    else
        is_leap_year = true
    end
else
    is_leap_year = false
end
```

This code can be re-written as a series of separate "if"
statements. In this case the control flow may be clearer
than the code that uses a nested structure.

```
is_leap_year = false

if year mod 4 = 0 then
    is_leap_year = true
end

if year mod 100 = 0 then
    is_leap_year = false
end

if year mod 400 = 0 then
    is_leap_year = true
end
```

This particular example could also be re-written as a
single condition, as shown in the following code

```
if (year mod 4 = 0 and year mod 100 <> 0)
or
                            year mod 400
= 0 then
    is_leap_year = true
else
    is_leap_year = false
end
```

4.9.18. "Else" conditions

In most cases, the "else" part of an "if" statement is
intended to apply to "if the condition is not true".
However, in other cases the intention is for the code to
apply "for the other value".

For example, a particular numeric variable may generally have a value of 1 or 2. The following code is an example.

```
if process_type = 1 then
    run_process1
else
    run_process2
end
```

In this example, if the value of "process_type" is not 1, then it has been assumed to be 2. However, due to a bug in the system or a misunderstanding of meaning and values of variable "process_type", it may have a different value. This can be checked, as in the following example.

```
if process_type = 1 then
    run_process1
else
if process_type = 2 then
    run_process2
else
    display_message "Invalid value " &
                        process_type & "for
                        variable
                        process_type in
                        run_process"
end
```

This code may detect existing problems, or detect problems that arise following future changes.

4.9.19. Boolean Expressions

4.9.19.1. Conditions

If the variable "sort_ascending" is a Boolean variable, i.e. it can only have the values True or False, then the following two lines of code are equivalent

```
if sort_ascending = true then


if sort_ascending then
```

The second form is clear when a Boolean variable is used, and the "= True" is not necessary.

However, in some languages the second form of the expression can also be used with numeric variables.

In this case the code may be clearer if the actual condition is included.

For example,

```
if record_count then


if record_count <> 0 then
```

In this case the variable "record_count" records a number, not a Boolean value. In languages where "true" was defined as non-zero, the two expressions may be equivalent.

However, this simply relies on the method that has been used to implement Boolean expressions, and the second form may be clearer as it specifies the condition that is actually being tested.

In languages that use strict type checking, the first form may generate an error, as the expression has a numeric type but the "if" statement uses Boolean expressions.

4.9.19.2. Integer Implementations

In some languages, Boolean variables are implemented as integers within the code, rather than as a separate type. The True and False values would be numeric constants.

Zero is generally used for False, while True can be 1, -1, or any non-zero number depending on the language.

In languages where Boolean values are implemented as numbers, problems can arise if numeric values other than True and False are used with variables that are intended to be Boolean.

For example, if False is zero and True is 1, but any non-zero value is accepted as true, the following anomalies can occur.

Value	Condition	Outcome	Reason
1	is false	no	1 is not equal to 0
1	= True	yes	1 is equal to 1
1	is true	yes	1 is non-zero
2	is false	no	2 is not equal to 0
2	= True	no	2 is not equal to 1
2	is true	yes	2 is non-zero

In this case, "2" would match the condition "is true", as is it non-zero, but would not match the condition "= True", as it is not equal to 1.

These problems can be reduced if numeric and Boolean variables and expressions are clearly separated.

4.9.20. Consistency

Consistency in the use of language features may reduce the chance of bugs within the code.

For example, arrays are commonly indexed with the first element having an index of zero or one. In some languages this convention is fixed, while in others it can be defined on a system-wide basis or when an individual array is defined.

If some arrays are indexed from zero and some are indexed from one, this may increase the change of bugs in the existing system, or bugs being introduced through future changes.

For example, this code uses the convention of storing the values starting with element zero.

```
for i = 0 to num_active - 1
    total = total + current_value[i]
end
```

The following code uses the convention of storing values starting with element one.

```
for i = 1 to num_active
    total = total + current_value[i]
end
```

If the wrong loop type was used for the array, one valid element would be missed, and would be replaced with another value that could be a random, initial or previous value.

This may lead to subtle problems in processing, or create general instability and occasional random problems.

When maintenance changes are made to an existing system, the chance of errors occurring may be reduced if the language features are used in a way that is consistent with the existing code.

4.9.21.   Undefined Language Operations

In many languages, there are combinations of statements that form valid sections of code, but where the result of the operation is not clearly defined.

For example, in many languages the value that a loop variable has after the loop has terminated is not defined.

The following code searches an array for a word and adds the word into the array at the first empty space if it is not found.

```
    subroutine add_word( word as string,
                             substring_found as
boolean )

        define i as integer
        define found as boolean

        found = false

        for i = 0 to num_words - 1
            if word_table[i] = word then
                found = true
            end

            if string_search( word_table[i],
word ) then
                substring_found = true
            end
        end

        if not found then
            word_table[i] = word
            num_words = num_words + 1
        end
    end
```

While the value of the variable "i" increases from 0 to "num_words – 1" on each iteration through the loop, in many languages the value of "i" is not clearly defined after the loop has terminated. It may be equal to the last

236

value through the loop, the last value plus one, or some other value.

Most compilers would not detect this problem, as technically this code is a valid set of statements within the language.

This problem could be avoided by using the variable "num_words" after the loop instead of the loop variable itself.

Calling subroutines from within expressions can also result in undefined operations. For example, in some cases the order in which an expression is evaluated is not defined, and when an expression contains two subroutine calls, these calls could occur in either order.

In the case of Boolean expressions, if the first part of an AND expression is false, then the second part does not need to be evaluated, as the result of the AND expression will be false, regardless of the result of the second expression.

In these cases, languages may define that the second part will never be executed, that the second part will always be executed, or the result may be undefined.

4.10. Testing

Testing involves creating data and test scenarios, calling subroutines or using the system, and checking the results.

Testing is done at the level of individual subroutines, complete modules, and an entire system.

Testing cannot repair a system that is poorly designed or coded. The reliability of a system is determined during the design and coding stages, not during the testing stage.

4.10.1. Execution Path Complexity

The numbers of possible paths that program execution can follow is extremely large.

For example, the following code scans an array of 1000 items, and prints all the items that have a value of less than one.

```
for i = 0 to 999
    if table[i] < 1 then
        print table[i]
    end
end
```

The number of possible combinations of output is 2^{1000}, which is approximately 107 followed by 299 zeros.

In large systems, only a small fraction of the possible program flow paths will ever be executed in the entire life of the system operation.

4.10.2. Testing Methods

4.10.2.1. Function Testing

Testing individual functions involves testing individual subroutines and modules as each function is developed.

Debugging is easier and more reliable if testing is done as each small stage is developed, rather than writing a large volume of code and then conducting testing.

Function testing is conducted by writing a test program or test harness.

This may be a separate program, or a section of code within an existing system. The testing routine generates data, calls the function being tested, and either logs the results to a file or checks the figures using an alternative method.

4.10.2.2. System Testing

System testing involves testing an entire system. This is done by creating test data, running processes and functions within the system, and checking the results.

System testing generally results in the production of a list of known problems. The actual error or outcome in each situation is described, along with the data and steps required to re-produce the problem.

When a range of problems have been fixed, a new test version of the system can be released for further testing.

4.10.2.3. Automated testing

Automated testing involves the use of testing tools to conduct testing.

Testing tools are programs that simulate keystrokes entered by the user, check the appearance of screen displays to detect error messages, and log results.

This process is particularly useful when changes are made to an existing system. A set of results can be generated before the change is made, and then automatically re-generated after the change. This method can be used to check that other processes have not been altered by the systems changes.

Automated testing can also be conducted by altering the system to log each keystoke to a file, and then re-reading the file and replaying the keystokes automatically.

4.10.2.4. Stress Testing

Stress testing involves testing a system with large volumes of data and high input transaction rates. This is done to ensure that the system will continue to operate normally with large data volumes, and to ensure that the response times would be within acceptable limits.

Stress testing can be done by generating a large volume of random or sequential data, populating the database tables, and running the system processes.

Stress testing can also be done with internal data structures. For example, a testing routine for testing a general set of binary tree functions, or a sorting module, could create a large volume of data, insert the data into the structure, and call the functions that operate on the data structure.

4.10.2.5. Black Box Testing

Black box testing involves testing a system or module against a definition of the function, while ignoring the internal structure of the process.

This can be a useful initial test, as the testing is done without preconceived ideas about the processing.

However, due to the large number of possible paths in even small sections of code, it is not possible to check that the output results would be correct in all circumstances.

Test cases that take into account the structure and edge cases in the internal code are likely to detect more problems that purely black box testing.

4.10.2.6. Parallel Testing

In some cases, an alternative system is available that can perform similar functions to the system being tested.

This may be an existing system in the case of a system redevelopment, a temporary system, or an alternative system such as a commercial system that provides related functions.

Parallel testing can be done by loading the same set of data into both systems and comparing the results.

The results should also be checked independently, to avoid carrying over bugs from a previous system into a new system.

4.10.3. Test Cases

4.10.3.1. Actual Data

Actual data is derived from previous systems, manual keying, uploading from external systems, and the existing system itself when changes are being made.

Tests conducted on actual data can be used to generate a set of known results. These results can then be re-generated and compared to the previous data after changes have been made. This process can be used to check whether existing functions have been altered by changes made to the system.

4.10.3.2. Typical and Atypical Data

Typical data is a set of created data that follows usual patterns.

Also, a range of unusual data combinations can also be created to test the full functionality of the system.

Typical data is useful in testing processing, as a full set of standard results will generally be generated. This may not occur with unusual cases that are valid system inputs, but do not result in a full set of processing being performed.

Testing generally covers the full range of possible data and functions, and focuses on unusual and atypical cases.

Problems with common cases may appear relatively quickly, while reducing long-term problems involves checking scenarios that may not appear until some time into the future.

4.10.3.3. Edge Cases

Edge cases involve checking the significant points within code or data.

This may include values that result in a change in calculation output, or cases that are different from the next and previous value, such as the last record in a file.

For example, in a calculation the altered after one year, testing could be done using periods of 364, 365, 366 and 367 days.

As another example, in a system for recording details of learner drivers with a minimum age of 17, testing could be done with test cases using ages of 16, 17 and 18.

4.10.3.4. Random Data

Random data generation may produce test cases that are unusual and unexpected, even though they are valid system inputs.

Testing with random data can be useful in checking the full range of functionality of a system or module. Randomly generated data and function sequences may test scenarios that were not contemplated when the system was designed and written.

4.10.4. Checking Results

4.10.4.1. System Operation

In some cases direct problems occur when processes or functions are run. This may include problems such as a program error message being displayed, a system crash with an internal error message, printed output not appearing or being incorrectly arranged, or the program entering an infinite loop and never terminating.

4.10.4.2. Alternative Algorithms

In some cases an alternative method can be used to check the results that have been generated. This may involve writing a testing routine to calculate the same result, using a method that may be simple, but may execute slowly and may only apply to that particular situation.

When this is the case, the test program can generate a large number of test cases, call the module being tested, and check the results that are produced.

4.10.4.3. Known Solutions

When testing is conducted on an existing system, a set of standard results can be produced. This may include a set of data that is updated within a database, or a set of calculated figures that are stored in a file for future comparison.

This process allows a test program to re-generate the data or figures, and check the results against the known correct output.

4.10.4.4. Manual Calculations

The inputs to calculations and the calculation results can be logged to a file for manual checking. This may include writing a simple program to scan the file and check the figures, or it may involve using a program such as a spreadsheet to check the data and re-calculate the figures.

Checks can be run against the input data that was supplied to the system, and also against calculations within the set of output data, such as verifying totals.

4.10.4.5. Consistency of Results

In some cases, the output from a process should have a certain form, and individual outputs should be related in certain ways.

In these situations, a test program can be used to check that the individual results are consistent with each other. These tests could also be built into the routine itself, and used to check output as it is produced.

When these tests are fast and simple they can remain in the code permanently, otherwise they can be activated for testing and debugging.

For example, a set of output weights should sum to 1, a sorted output list should be in a sorted order, and the output of a process that decomposes a value into parts should equal the original value.

As another example, checking that a sort routine has successfully sorted a list is a simple process, and involves scanning the list once and checking that each item is not smaller than the previous item.

4.10.4.6. Reverse Calculations

Some calculations and processes can be performed in two directions, with one direction being difficult and the other straightforward.

For example, the value of "y" in the equation "$y = x^2 + x$" is determined by calculating the result from the value of "x".

However, if the value of "y" is already known but the value of "x" is required, then the equation cannot be directly solved, and an iterative method for determining an approximate solution must be used.

In this case, determining the value of "y" may be a relatively complex process. However, once the result is known, it can be easy checked by preforming the reverse calculation and ensuring that the result matches the original figure.

4.10.5.   Declarative Code

Testing declarative patterns involves checking that the pattern matches the expected set of strings using the expected set of sub-patterns.

4.10.5.1. Testing of the program

In the case of fact-based systems that involve problem solving and goal seeking, testing can be conducted to ensure that facts are not conflicting, and that there are no unintended gaps in the information that would lead to undefined results.

A test program can be written to read the pattern or fact descriptions into an internal data structure for scanning.

A scan of the structure can be performed to check for loops, multiple paths, and conditions that have undefined results.

Where a loop occurs, in some cases this may be the result of a deliberate recursive definition, while in other cases this may signify an error.

In some cases, two points within a pattern may be connected by multiple paths.

In these cases, an input string may be matched by a pattern in two different ways. In some applications this may not be significant, while in other cases this may indicate that the pattern could not be used in processing.

In some applications, algorithms exist to check particular attributes of a pattern structure.

For example, an algorithm could be used to check whether a language grammar was or was not suitable for top-down parsing.

4.10.5.2. Testing Program Operation

Testing can be performed with input data to ensure that the correct result is produced.

This may involve using the program with test cases and checking output.

Traces could be produced of the logic path that was followed to arrive at the result, or the sub-patterns that were matched.

4.10.6. Testing Summary

Testing Methods

Function testing	Testing individual subroutines, modules and functions using test programs.
System Testing	Testing entire systems by running test cases and checking results.
Automated Testing	Using automated testing tools to simulate keying, run processes, and detect screen and data output.
Stress Testing	Testing applications with large data volumes and high transaction rates to check operation and response times.
Black box testing	Testing a function against a specification, ignoring the function's internal structure.
Parallel Testing	Testing a system by running an alternative system using the same data.

Test Data

Actual Data	Data sourced from previous systems, data uploads, the existing system itself or manual keying.
Typical and Atypical Data	Created data using common data combinations and also unusual and unexpected combinations that are valid system inputs.
Edge Cases	Test data that is slightly less, slightly more, and equal to

values that result in changes in calculations, or internal program limits.

Random Data	Data created with random values and combinations, used to generate large data volumes and to test the full range and combinations of system inputs.

Detecting Errors

System Operation	Problems due to program crashes, error message, and incorrect output.
Alternative Algorithms	Generating the same result within a test program using an alternative algorithm, and comparing results.
Known Solutions	Comparing results with results generated previously, and defining a set of input data with known results generated by a separate test program.
Manual Calculations	Checking results using test programs and software tools to re-calculate output results and check figures within outputs such as totals.
Consistent Results	Checking outputs for consistency, such as individual outputs that should balance to zero or equal another output, checking that sorted data is actually sorted etc.

Reverse Calculations	Performing a calculation in reverse to determine the original figure, and comparing this result with the actual input value to verify that the generated result is correct.

Declarative Code

Testing Output	Declarative code can be run against actual test data, and the output results checked.
Checking program traces	Traces of logic and patterns matched during the operation with the input data may highlight problems with the declarative structure.
Structure Checks	Declarative structures can be read by test programs, and a scan of the structure can be performed to check for loops, multiple paths, and combinations of events that would lead to undefined results.

4.11. Debugging

Debugging is the process of locating bugs within a program and correcting the code. A bug is an error in the code, or a design flaw, that leads to an incorrect result being produced.

4.11.1. Procedural Code

Before debugging can be performed, a set of steps that leads to the same problem occurring each time the program is run must be determined. This step is the foundation of the debugging process.

Debugging can be done using some of the following methods

- Reading the code and attempting to follow the execution path of the particular set of steps.
- Running the program with trace code added to log the value of variables to a data file, and to identify which conditions were triggered.
- Stepping through the execution using a debugger, and pausing at various points in the execution to examine the value of data variables.
- Adding error checking code into the system to detect errors at an earlier stage, or to trigger a message when invalid data or results are produced.

Some examples of bugs include:

- An error in entering an expression, such as placing brackets in the wrong place.
- A logic error, where a certain combination of conditions is not handled correctly.

- A mistake in the understanding of a variable; how it is calculated, and when it is updated.
- A sequence problem, when calling a set of functions in a particular order leads to an incorrect result.
- A corruption of memory, due to an update to an incorrect memory location such as an incorrect array reference.
- A loop that terminates under incorrect conditions.
- An incorrect assumption about the concepts and processes that the system uses.

4.11.1.1. Viewing Data Structures

The contents of entire data structures, such as arrays and tables, can be written to a text file and then viewed in a program such as a spreadsheet application. This can be done by writing subroutines to write the contents of the data structure to a text file.

This method may highlight obvious problems, such as a column full of zeros or a large list when there should be a small list.

Data columns from different data structures can be compared, and manual calculations can be done to check that the figures are correct.

4.11.1.2. Data Traces

A log file can be written of the value of variables as the program runs.

This can be used to view the sequence of changes to various data variables.

Traces may also be used to record which sections of the code were executed, and which conditions were triggered.

A Timestamp, containing an update date and time, can be included on data records. This can be used to identify the records that were updated during the process.

4.11.1.3. Cleaning & Rewriting Code

In cases where a difficult bug cannot be found, the code can be reviewed and changes such as adding comments, minor re-writing of code, changing variable names etc. can be made.

In situations where the code is in very poor condition, following a large number of changes, an entire section of code may be re-written.

This may occur when the control flow within a section of code is extremely complex, and it is likely that other problems are also present in the code, as well as the particular problem being checked.

In this case, the original bug should be located before the re-writing if possible, as it may represent a design flaw that is also present in other parts of the system.

4.11.1.4. Internal Checks

Internal checks are code that is added into the system to determine a point in the process when the results become incorrect.

This can be used to determine the approximate location of the problem. This method is particularly useful for

bugs that randomly occur, where a sequence of steps to reproduce the problem cannot be determined.

In some cases, invalid data may occasionally appear in a database, however re-running the process generates the correct results.

Including internal checks may trap the error as it occurs, enabling the problem to be traced.

4.11.1.5. Reverse Calculations

Some calculations can be performed in reverse, to check that the result is correct. For example, an iterative method could be used to determine the value of the variable "x" in the equation "$y = x^2 + x$", when the value of "y" is known.

In this example, when the value of "x" has been calculated, the equation can be used to calculate the value of "y" and ensure that it matched the original value.

4.11.1.6. Alternative Algorithms

In some cases an alternative method can be used to calculate a result and check that it is correct. For example, this may be a simple method that is slower that the method being checked, and only applies in certain circumstances.

4.11.1.7. Checking Results

Some processes produce several results that can be checked against each other. For example, in process that

254

decomposes a value into component parts, the sum of the parts should match the original value.

As another example, sorting is a slow process, however checking that a list is actually sorted can be done with a single pass of the list.

During testing a sort routine could check the output being produced to ensure that the data had been correctly sorted.

4.11.1.8. Data Checking

Data values can be checked at various points in the program to detect problems. This may include checking individual variables, and scanning data structures such as arrays.

Checks can include checking for zero, negative numbers, numbers outside expected ranges, empty strings etc. Lists can be scanned to check that totals match individual entries, and that values are consistent, such as percentage weights summing to 100%.

4.11.1.9. Memory Corruptions

Memory corruptions can occur in some environments where a section of memory containing data items can be overwritten by other data due to a program bug. In some environments, for example, memory may be overwritten by using an array index that is larger than the size of the array.

Some memory corruptions can be detected by using a fixed number code within a data variable in a structure or array. When the structure is accessed, the number is checked against the expected value, and if the value is

different then this indicates that the memory has been overwritten.

A checksum can also be used, which is a sum of the individual values in the structure. This figure is updated each time that a data item is modified. When the checksum is recalculated, a difference from the previous figure would indicate a memory corruption.

4.11.2. Declarative Code

Debugging declarative code may be difficult, as in many cases the definitions are recursive, and allow an infinite number of possible patterns.

For example, the following definition defines a language of simple calculation expressions.

```
expression:   number
              number * expression
              number + expression
              number - expression
              number / expression
              ( expression )
```

This is a recursive definition, as an expression can be a number, a number multiplied by another expression, or a set of brackets containing another expression.

Testing and debugging declarative code may involve reading the pattern or facts into an internal data structure, and scanning for loops, multiple paths and combinations that would lead to undefined outputs.

In some cases, a process could be used to expand a pattern into a tree-like structure which may be easier to interpret visually. In the case of recursive definitions that could be infinite, a tree diagram could extend to several levels to indicate the general pattern of expansion.

The pattern can also be used to generate random data that has a matching pattern based on the definition.

This may highlight some of the cases that are included within the pattern unintentionally, and patterns that were intended to be included but are not being generated.

Debugging can also be performed by executing a process that uses the pattern or lists of facts, and generating a trace of the logic path that was used during processing.

This may include a list of the individual sub-patterns that were matched at each stage, or the facts that were checked and used to determine the result.

4.11.3. Summary of debugging approaches

Approach	Description
Reading code	Reading code and following the execution path that results in the incorrect output.
Program traces	Logging the value of data variables, and details of which sections of code were executed and which conditions were triggered, to a data file as the program runs.
Debuggers	Using a debugger to halt the program during execution, view the value of data variables, and step through sections of code.
Viewing data structures	Writing entire data structures to a file and viewing the contents of the structure, to identify obvious problems and to re-calculate figures and compare data with other structures.
Cleaning code	Minor re-writing of code to add comments, correct inconsistent variable names and statements, and clarify the logic flow within the code.

Detecting errors

Reverse calculations	Performing a calculation in reverse and recalculating the input value from the generated output, to check that the input values match.
Alternative algorithms	Using an alternative algorithm to generate the same output, and check that the output data matches.
Consistent output	Checking output results for consistency, such as checking that output weights sum to 1, that sorted data is actually sorted etc.
Data checking	Checking data items at various stages in a process, including scanning data structures, to check for negative items, zero values, empty lists, weights that do not sum to 1 etc.
Memory corruptions	Using checksums and magic numbers to detect pointers to incorrect types and memory structures that have been overwritten with other data.

Declarative code

Traces of logic flow | Writing a trace of the logic path used and the patterns that were matched to a file, so that problems with the declarative structure can be identified.

Scanning structures | Scanning structures using a test routine to detect loops, multiple paths, and combinations of input conditions that would result in an undefined output.

4.12. Documentation

4.12.1. System Documentation

During the development of a system, several documents
may be produced. This typically includes a "Functional
Specification", which defines in detail the functions and
calculations that the system should perform.

Other documents, such as design documents may also be
produced.

System documentation can be used during maintenance,
and also during enhancements to a system.

4.12.2. User Guide

User documentation may include a user guide. This
document is a guide to using the system. It may include a
description of the process and functions that the system
performs, as well as a reference guide to calculations and
conditions. In some user guides a set-by-step tutorial of
various functions is included.

4.12.3. Procedure Manual

A procedure manual is usually produced by the end users
of the system, rather than the developers of the system.

The procedure manual defines how a system is used in a
particular installation. This may include a description of
the sources of various data items that are maintained and
a list of the timing and order of regular processes.

5. About the author

Mark McIlroy has an undergraduate degree in Computer Science and Applied Mathematics.

He also holds Masters degrees in Applied Finance and Financial Planning.

Mark has extensive experience in IT systems development in the Financial Services sector in Australia.

Following a long career in IT systems development Mark has now made a career change into Financial Planning.

Mark lives in Melbourne, Australia with his wife.

More resources are available on the author's personal website at www.markmcilroy.com

6. Appendix A - Summary of operators

Brackets	(a)	Bracketed Expression	a
Arithmetic	a + b	Addition	a added to b
	a – b	Subtraction	b subtracted from a
	a * b	Multiplication	a multiplied by b
	a / b	Division	a divided by b
	a MOD b	Modulus	a – b * int(a / b)
	- a	Unary Minus	The negative of a
	a ^ b	Exponentiation	a^b a raised to the power of b
String	a & b	Concatenation	b appended to a
Relational	a < b	Less Than	True if a is less than b, otherwise false
	a <= b	Less Than or Equal To	True if a is less than or equal to b, otherwise false
	a > b	Greater Than	True if a is greater than b, otherwise false
	a >= b	Greater Than or Equal To	True if a is greater than or equal to b, otherwise false
	a = b	Equality	True if a and b have the same value, otherwise false
	a <> b	Inequality	True if a and b have different values, otherwise false
Logical Boolean	a OR b	Logical Inclusive OR	True if either a or b is true, otherwise false
	a XOR b	Logical Exclusive OR	True if a is true and b is false, or a is false and b is true, otherwise false.
	a AND b	Logical AND	True if a and b are both true, otherwise false
	NOT a	Logical NOT	True if a is false, false if a is true
Bitwise Boolean	a OR b	Bitwise Inclusive OR	1 is either a or b is 1, otherwise 0
	a XOR b	Bitwise Exclusive OR	1 if a is 1 and b is 0, or a is 0 and b is 1, otherwise 0.
	a AND b	Bitwise AND	1 if a is 1 and b is 1, otherwise 0
	NOT a	Bitwise NOT	1 if a is 0, 0 if a is 1
Addresses	ref a	Reference	The address of the variable a
	deref a	Dereferece	The data value referred to by pointer a
Array Reference	a[b]		The element b within the array a
Subroutine Call	a(b)		A call of subroutine a, passing parameter b
Element Reference	a.b		Item b within structure a
Assignment	a = b	Assignment	Set the value of variable a to equal the value of expression b

e-GO! Library

THE BEST BOOKS IN YOUR POCKET!

Jam-pack your Kindle, iPad, Android, Nook, or any device, full of amazing books with your own e-GO! Library. At your leisure pick your favorite titles from a wide array of categories for all tastes. Total speed, convenience & mobility are yours. Your e-GO! Library is an innovative way "to have & hold" a fresh and plentiful supply of great titles. It's seemingly endless entertainment without the need for the internet or downloading! All in a preloaded, USB memory flash drive which puts a thousand of today's best books in your pocket!

Enjoy Anywhere Anytime!

TODAY'S TOP

1,000 BOOKS

+250 BONUS CLASSICS

e-GO! Library

✓ Total Mobility & Convenience
✓ Pre-Loaded. Over 32 Categories
✓ No Internet / Wifi Needed
✓ Perfect for Travel
✓ Read Anywhere, Anytime

EXTRA BONUS!
+25 AUDIO BOOKS
at
eGoLibrary.com/bonus

 Apple   Windows android nook kindle
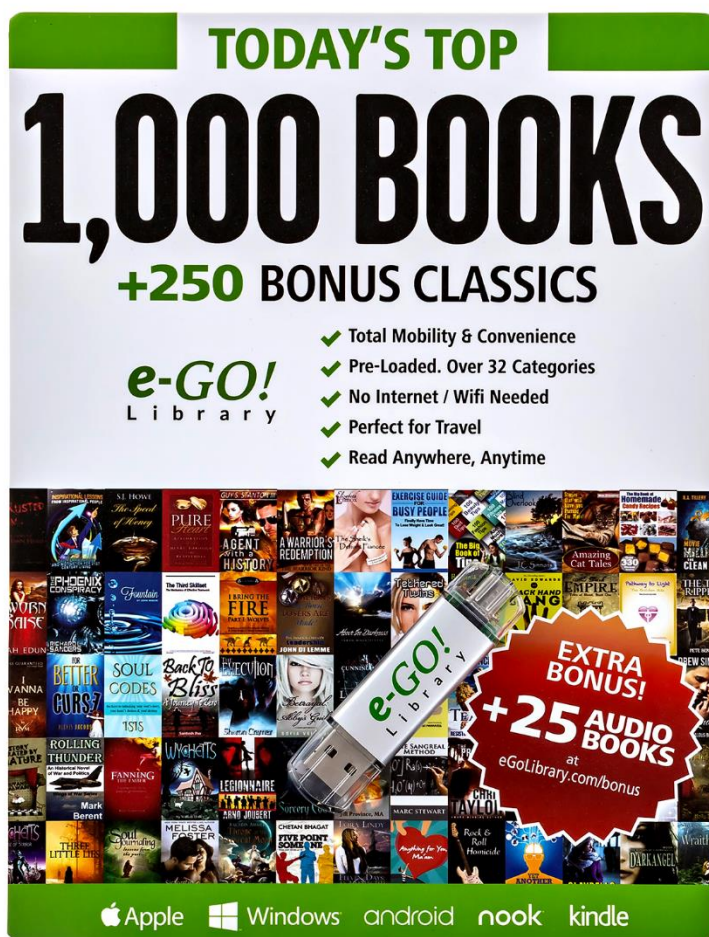
✓ **CONVENIENCE** of Preloaded USB flash drive

✓ **FORMATS** for ALL eReaders & Mobile Devices

✓ **NO** need for Wi-Fi or Internet

✓ **AVOID** Downloading

✓ **SAVE** Tons of Time & Effort

✓ **PERFECT** for Travel

✓ **FUN** for the Whole Family

✓ **MAKES** a Great Gift

CHECK IT OUT HERE