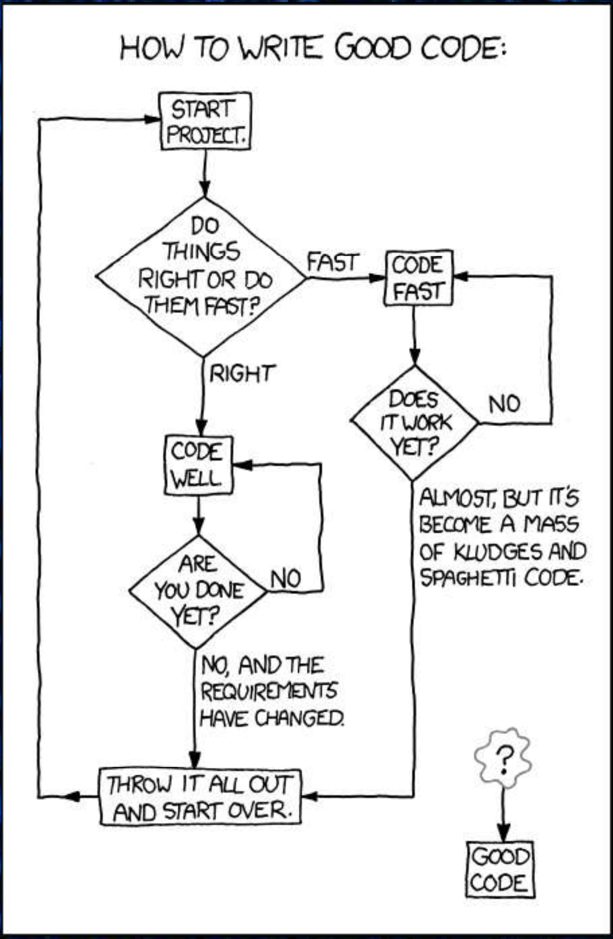
A text book for the VCE Applied Computing Study Design 2020 - 2023
&
Microsoft Visual Basic

2nd EDITION

2021 Edition

Software Development Units 3 & 4

Vic Farrell


HOW TO WRITE GOOD CODE:
START PROJECT.
DO THINGS RIGHT OR DO THEM FAST?
FAST
CODE FAST
DOES IT WORK YET?
NO
ALMOST, BUT IT'S BECOME A MASS OF KLUDGES AND SPAGHETTI CODE.
RIGHT
CODE WELL
ARE YOU DONE YET?
NO
NO, AND THE REQUIREMENTS HAVE CHANGED.
THROW IT ALL OUT AND START OVER.
?
GOOD CODE

ISBN 978-0-6486708-1-0
9 780648 670810

Table of Contents

The 2021 edition of this textbook was written and published after the VCAA released all relevant support documentation. This version contains updates and extra support material. For any further assistance please contact Vic Farrell via the website: vicfarrell.com.au.

THANKS

To Adrian Janson.
To Elisa Brock for being an amazing proofreader right next door.

Self published using free templates from http://copiesandink.com/
Icons made by Freepik from flaticon.com
Many thanks to Randall Munroe for the use of his excellent comics www.xkcd.com

This publication is designed to meet the teaching and learning needs of teachers and students of the Study Design Applied Computing: Units 3 & 4 Software Development 2020 - 2023 provided by the Victorian Curriculum and Assessment Authority.
© Victorian Curriculum and Assessment Authority. For current versions and related content visit www.vcaa.vic.edu.au. Used with permission 2020.

Microsoft Visual Studio screen grabs used with permission 2020.

Thanks to Daniel Viglietti for his Year 12 SAC code for the cover art.

© Vic Farrell 2021 Published on Amazon.com

ISBN: 978-0-6486708-3-4 (e-Book)
ISBN: 978-0-6486708-4-1 (Paperback)

Publisher: Amazon Kindle
Victoria, Australia
2020

For more resources: www.vicfarrell.com.au

Unit 3
Area of Study 1

Programming

In this area of study students examine the features and purposes of different design tools to accurately interpret the requirements and designs for developing working software modules. Students use a programming language and undertake the problem-solving activities of manipulation (coding), validation, testing and documentation in the development stage.
The working modules do not have to be complete solutions and can focus on limited features of the programming language; however, students are expected to fully develop the working modules in accordance with the given designs. This will prepare students for creating a complete solution in Unit 4, Area of Study 1. Validation and testing techniques are applied to ensure modules operate as intended and internal documentation is written to explain the function of the modules. Students justify the use of the selected processing features and algorithms in the development of their working modules.

Key Knowledge

Data and information
- characteristics of data types
- types of data structures, including associative arrays (or dictionaries or hash tables), one-dimensional arrays
- (single data type, integer index) and records (varying data types, field index)

Approaches to problem-solving
- methods for documenting a problem, need or opportunity
- methods for determining solution requirements, constraints and scope
- methods of representing designs, including data dictionaries, mock-ups, object descriptions and pseudocode
- formatting and structural characteristics of files, including delimited (CSV), plain text (TXT) and XML file formats
- a programming language as a method for developing working modules that meet specified needs
- naming conventions for solution elements
- processing features of a programming language, including classes, control structures, functions, instructions and methods
- algorithms for sorting, including selection sort and quick sort
- algorithms for binary and linear searching
- validation techniques, including existence checking, range checking and type checking
- techniques for checking that modules meet design specifications, including trace tables and construction of test data
- purposes and characteristics of internal documentation, including meaningful comments and syntax.

Key skills

- interpret solution requirements and designs to develop working modules
- use a range of data types and data structures
- use and justify appropriate processing features of a programming language to develop working modules
- develop and apply suitable validation, testing and debugging techniques using appropriate test data
- document the functioning of modules and the use of processing features through internal documentation.

Data and Information

Information Systems are any organisational system that sorts, finds, retrieves and displays information. Information is processed data. An example might be your mobile phone as an information system for phone contacts. You input the data - given names, family names and mobile numbers and the system places the data into the memory which can then be sorted, searched and used. Data is organised, sorted and formatted into information such as spreadsheets, magazine layouts, websites, or reports. In digital systems we mainly focus on text, numbers and images in our databases, software development and web development. Data comes in five digital types: text, numbers, images, sound and video. All of these data types are composed of fundamentally the same thing: Numbers! Digitising any data is the process of converting it into numbers.

Binary

Digital machines are built on circuitry that allows for the flow of electricity. A digital machine can detect if a circuit is ON (where there is electricity flowing through the circuit) or OFF (no electricity). This is the only way these machines can "understand" the data that is read in. So we devised a simple code to bridge the divide between humans and machines. If the circuit is OFF we will call it a zero. Computer scientists use Ø to distinguish it from the letter O. If the circuit is ON we will call it 1. This is how the connection between machines and humanity is made.

Humans have a number system based on 10 symbols (Ø, 1, 2, 3, 4, 5, 6, 7, 8, 9) which is called the decimal system. All values can be created with these ten symbols. It is suggested the reason we chose ten symbols was due to having ten fingers. Digital machines do not have the capacity to understand anything beyond ON and OFF, so we have to construct a number system around the two "fingers" they have: Ø and 1. Since there are only two symbols, we call this number system Binary (bi means 2).

Computers use 1s and Øs to represent all the values and all the other symbols we use in text. We call each Ø and each 1 a bit and we have to combine them to create unique identifiers for each symbol. We group eight bits together to represent all the ASCII symbols (all the letters, numbers and symbols input by a keyboard). This group of eight bits is called a byte.

Memory

Bytes are too small to measure memory. We usually store hundreds of characters in a basic text file. Binary is based on the number 2. We set a kilobyte to 2^{10} (1024) bytes. In fig 1.1 you can see the relationship between a bit and a byte and how each character on the keyboard is represented as a byte. Common units of storage size are based on 2.

Fig 1.1

BINARY	MEMORY
One Bit = Ø or 1	1024 byte = 1 KB
One Byte = 8 Bits	1024 Kb = 1 MB
a = Ø11ØØØØ1	1024 Mb = 1 GB
b= Ø11ØØØ1Ø	1024 Gb = 1 TB
c = Ø11ØØØ11	1024 Tb = 1 PB
! = ØØ1ØØØØ1	

Data Types

It is important to be aware of common data types used in software development and their size and range. The key data types you need to know are:

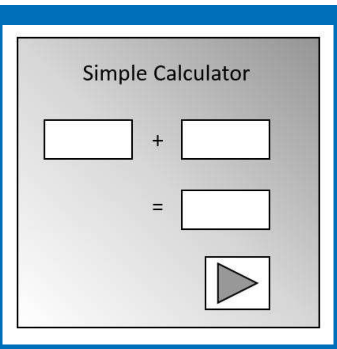
- Boolean - true or false.
- Character - a single letter, number or symbol
- Floating Point - decimal numbers
- Integer - whole numbers
- String - text of any symbols

Variables

Variables are temporary data holding-spaces used in software. These are used to read in data, process it and display the output. Here is a basic example: A simple calculator that reads in two numbers, adds them and displays the answer. A variable holds the value of each of the two numbers typed in by the user and then another variable would hold the answer. Below in fig. 1.2 is an algorithm (a plan for a solution) for the calculator using variables.

Fig 1.2

```
START
    Number1 ← Integer
    Number2 ← Integer
    Answer ← Integer
    Read in Number1
    Read in Number2
    Answer ← Number1 + Number2
    Display Answer
END
```

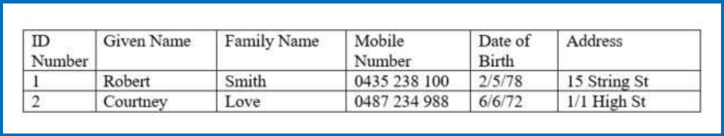


Records

Records are made up of fields. The fields in the example below are: ID Number, Given name, Family Name, Mobile Number, Date of Birth and Address. The records can be sorted and searched by fields so that similar records can be identified. For example all records with Date of Birth data lower than 1/1/2002 will return all records of people who are over the age of 18 years. Records are most easily managed with unique identifiers, such as ID Number. This ensures all records are unique and we can easily distinguish the possible multiple Robert Smiths recorded in the file. In figure 1.3 you can see how a tabled record can be recreated in an algorithm.

Fig 1.3

```
START
    CustomerContacts.IDNumber ← 3 (Data Type = Integer)
    CustomerContacts.GivenName ← Lady (Data Type = String)
    CustomerContacts.familyName ← Gaga (Data Type = String)
    CustomerContacts.MobilePhone ← 0488 239 765 (Data Type = String)
    CustomerContacts.DateofBirth  ← 28/3/89 (Data Type = Date/Time)
    CustomerContacts.Address ← 18 Hollywood Boulevard (Data Type = String)
END
```



ID Number	Given Name	Family Name	Mobile Number	Date of Birth	Address
1	Robert	Smith	0435 238 100	2/5/78	15 String St
2	Courtney	Love	0487 234 988	6/6/72	1/1 High St

Arrays

An array is a very useful data structure for large amounts of data that needs to be sorted or searched. Arrays can only hold one type of data. Each element has an indexed address and this allows arrays to be easily manipulated. In the example below you can see a series of integers. The top row identifies each element's address from zero to nine. This array has 10 elements but is identified as Array[9]. Remember computers start counting at zero! We can edit and access the data elements out of the array using their location. We will use the example array in figure 1.4

Fig 1.4

ARRAY

0	1	2	3	4	5	6	7	8	9
16	34	12	6	28	59	72	13	43	71

Each item can be identified as:

Array[0] = 16
Array[1] = 34
Array[2] = 12
Array[3] = 6
Array[4] = 28
Array[5] = 59
Array[6] = 72
Array[7] = 13
Array[8] = 43
Array[9] = 71

A One-Dimentional Array illustrated below is a list of product data with associated indexes. Product [4] is an array with 5 string elements in it. Each element has an index number associated with it. The index allows the array to be easily searched and manipulated.

Product[0] = "eggs"
Product[1] = "milk"
Product[2] = "bread"
Product[3] = "cheese"
Product[4] = "tomatoes"

A Multi-Dimentional Array as illustrated below has more than one list. The other name for this type of array is an 2D Array. Below is a set of values and their associated soccer team positions. Team[10,1] data type is String and holds a table of two columns [0 and 1] and eleven rows [0 - 10].

If we wanted to add names to our array we could create a third column calling the array Team[10,2] and adding our names in Team[i, 2] where i is a variable that denotes each row. See figure 1.5

Fig 1.5

SOCCER TEAM ARRAY

Team(0, 0) = " 1 "
Team(0, 1) = " Goal Keeper "
Team(1, 0) = "2 "
Team(1, 1) = " Right Full Back "
Team(2, 0) = "3 "
Team(2, 1) = " Left Full Back "
Team(3, 0) = "4 "
Team(3, 1) = " Centre Half Back "
Team(4, 0) = "5 "
Team(4, 1) = " Centre Half Back "
Team(5, 0) = "6 "
Team(5, 1) = " Defensive Midfielder "
Team(6, 0) = "7 "
Team(6, 1) = " Right Winger "
Team(7, 0) = "8 "
Team(7, 1) = " Central Mid Fielder "
Team(8, 0) = "9 "
Team(8, 1) = " Striker "
Team(9, 0) = "10 "
Team(9, 1) = " Attacking Midfielder "
Team(10, 0) = "11 "
Team(10, 1) = " Left Winger "

So all the data in row one could be:

Team[0,0]=1, Team[0,1]=”Goal Keeper” and Team[0,2]= “Boubacar Barry”.

Dictionaries

A Dictionary is a data structure which has many built-in functions that can add, remove and access the elements using a unique key. Compared to alternatives, a Dictionary is easy to use and effective. It has many functions (like ContainsKey and TryGetValue) that process lookups of tabulated data. They can hold many data types while arrays can only hold one. In the example below the key terms identify the team players positions.

```
Team As New Dictionary(Of String, Integer)
        Team.Add(“Goal Keeper”, 1)
        Team.Add(“Striker”, 2)
        Team.Add(“Full Back”, 3)
        Team.Add(“MidField”, 4)
```

Hash Tables

A Hash Table is a data structure which implements all of the dictionary operations but also allows insertion, search and deletion of elements providing the associated keys for each element. Hash Tables are complex solutions that use a calculation with a prime number to find a unique location using a key, to make it easier to find when you have a large file of unsorted data. In a basic address book, you might have:

Bill, Surpreet, Jane, Nqube, Quentin

The problem with locating these names in an address book alphabetically is that will create unused spaces in the table between Bill and Quentin leaving empty wasted storage space. Also a linear search would still be required under each alphabetical section. There are two types of search methods we examine here: Linear Search and Binary Search. They require many operations to find data in a large file or database. Linear search requires every item from the beginning of the file to be checked. For example if we were looking for Quentin, it would take 5 operations to find his data. What if we had millions of items to search? A hash table can use the data itself to calculate a unique location for each item of data.

Hash Tables provide a unique identifier based on the content of the data. If we use a basic conversion of a=1, b=2, c=3 etc we could convert “Jane” to:

(J)11 + (a)1 + (n)15 + (e)5 = 32

Unfortunately if “Neaj” is added to our address book then his converted number would also be 32. So we use a Hash Function that allocates values depending on the location of each character in the string. We use a prime number which helps us create unique values. In figure 1.6 we will use 7.

Hash(Current Letter) = (Prime Number(7) * Hash from previous value) + value of current letter.

Fig 1.6

Example: JANE	Example: NEAJ
Hash = 0	Hash = 0
Hash (J) = (7 * 0) + 10 = 10	Hash (N = (7 * 0) + 14 = 14
Hash (A) = (7 * 10) + 1= 71	Hash (E) = (7 * 14) + 5= 103
Hash (N) = (7 * 71) + 14 = 511	Hash (A) = (7 * 103) + 1 = 722
Hash (E) = (7 * 511) + 5 = 3,582	Hash (J) = (7 * 722) + 10 = 5,064
Hash (JANE) = 3,582	Hash (NEAJ) = 5,064

This provides locations for each item of data based on the data content. So now if you are searching for JANE the search engine need only conduct one operation to calculate the location and go directly to the data in the file.

A common method is to use remainders to consolidate the data into multiple arrays. To calculate a remainder a value is chosen (usually a prime number) to divide the value of the data. A remainder is then output which limits the number of arrays in the matrix. In the example below, the data we have to store in our hash table is:

23, 56, 47, 29, 92, 55, 11.

So we might use 5 to divide each value to find out the remainder on each. In VB we call this a MOD function.

Mod (23/5) = 3 (4 * 5 = 20 leaving a remainder of 3)
Mod (56/5) = 1
Mod (47/5) = 2
Mod (29/5) = 4
Mod (92/5) = 2
Mod (55/5) = 0
Mod (11/5) = 1

[0]	[1]	[2]	[3]	[4]
55	56	47	23	29
	11	92		

Now we have 5 locations to store our data. We can consolidate the locations where our data could be stored. If we used this mod function on our hash table of names we would use less space in our storage files.

Storage

When developing a solution, it is important to consider what data will be input into the system. The system will only be effective if the data input is valid and correct.

When formatting and storing data it is important to consider the following issues:

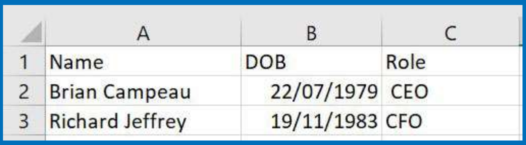
- How soon do I need the data back if lost?
- How fast do I need to access the data?
- How long do I need to retain data?
- How secure does it need to be?
- What regulatory requirements need to be adhered to?

Structuring Data

A lot of hard work can be avoided by organising the data into files or data structures that best suit the purpose of the project. This is why we use databases. Databases are essentially tables of data that hold records made up of fields. Often these databases are complex stand-alone systems so software developers need to find a way of storing data in more fundamental ways such as basic text files.

A text file of unorganised values is not going to be easy to access, sort or process so there are a number of ways data can be structured. CSV files are Comma Separated Value format. Each value data point is separated from the others with a comma character. CSV is a delimited data format that has fields or columns separated by the comma character and records or rows terminated by new lines. Below in figure 1.7 an example of a table containing two records with three fields is illustrated.

Fig 1.7



	A	B	C
1	Name	DOB	Role
2	Brian Campeau	22/07/1979	CEO
3	Richard Jeffrey	19/11/1983	CFO

CSV

```
Name, DOB, Role,
Brian Campeau, 22/07/1979,CEO,
Richard Jeffrey,19/11/1983,CFO,
```

Extensible Markup Language (XML) is a markup language that defines a set of rules for encoding documents in a format that is both human-readable and machine-readable. It uses tags in the same way that HTML tags format a web page. XML tags format records and fields. These structures enable access to database-formatted data with minimum impact on the amount of storage required. Figure 1.8 is an example of XML containing two records with four fields. Each Field is called an element within the record.

Fig 1.8

```
<breakfast_menu>
   <food>
       <name>Belgian Waffles</name>
        <price>$5.95</price>
        <description>Two of our famous Belgian Waffles with plenty of real maple syrup</description>
        <calories>650</calories>
   </food>

   <food>
       <name>Strawberry Belgian Waffles</name>
       <price>$7.95</price>
       <description>Light Belgian waffles covered with strawberries and whipped cream</description>
      <calories>900</calories>
    </food>
</breakfast menu>
```

breakfast_menu			
food			
name	price	description	calories
Belgian Waffles	$5.95	Two of our famous Belgian Waffles with plenty of real maple syrup	650
Strawberry Belgian Waffles	$7.95	Light Belgian waffles covered with strawberries and whipped cream	900

Storage Media

Data storage is the recording and storing of information in a storage medium. Recording data is accomplished by virtually any form of energy. Electronic data storage requires electrical power to store and retrieve data. Data storage in a digital, machine-readable medium is digital data. Barcodes and magnetic ink character recognition (MICR) are two ways of recording machine-readable data on paper. Electronic storage of data can be grouped into Primary, Secondary and Tertiary.

Primary Storage includes the RAM and ROM that directly support the CPU. It is volatile memory, which means all data is lost after the device is powered down.

Secondary Storage differs from primary storage in that it is not directly accessible by the CPU. The computer usually uses its input/output channels to access secondary storage and transfers the desired data using intermediate area in primary storage. Secondary storage does not lose the data when the device is powered down. It is non-volatile memory. Examples of Secondary Devices include; Hard Drives (these can be in-built to a computer system or be stand-alone external drives), CD/ROM, DVD, flash memory (e.g. USB flash drives or keys), magnetic tape, standalone RAM disks and Zip drives.

Tertiary Storage typically involves an automatic mechanism that will attach removable mass storage media to a storage device when required. Data is often copied to secondary storage before use. It is primarily used for archiving rarely accessed information since it is much slower than secondary storage. This is primarily useful for extraordinarily large data stores, accessed without human operators. Typical examples include tape libraries and full back-ups.

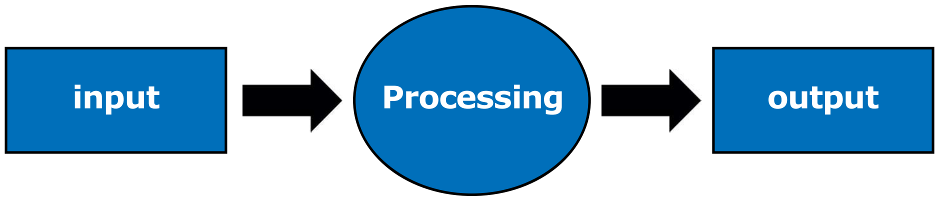
Representing Data

IPO Chart

We need to use tools to plan our software solutions. The most basic tool is the IPO chart. This chart identifies the input and output variables and the processes in between. For the Simple Calculator example see figure 1.9 below. This is a great place to start when designing a complex system solution. Identifying the data that goes in and what comes out is very helpful!

Fig 1.9

INPUT = Number1 and Number2
OUTPUT = Answer
PROCESS = Number1 + Number2 = Answer



Data Dictionary

Once we know what data needs to be handled, we need to design the variables to handle it. Creating a Data Dictionary is an important aspect of the Design Stage in developing a solution. It includes the names of the variables, the data types of the variables, the size of the data held and a description of the data. Returning to our Simple Calculator again here is a Data Dictionary in figure 1.10.

Fig 1.10

Variable Name	Data Type	Size	Description
IntNumber1	Integer	3	This is the first of two numbers to be added
IntNumber2	Integer	3	This is the second of two numbers to be added
IntAnswer	Integer	4	This is the sum of the two numbers entered

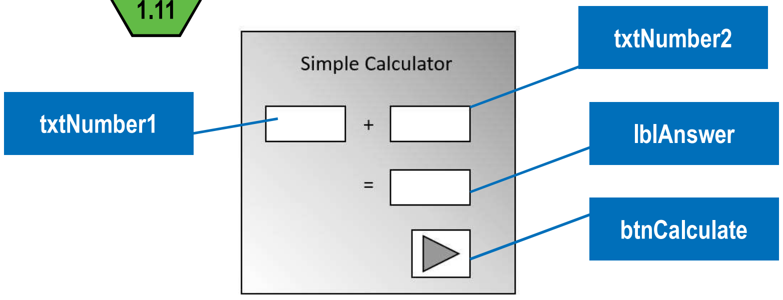
The Data Dictionary above shows the correctly named variables, their data types and descriptions. The size “3” indicates the number of bytes the variable can hold, in this case, the largest number that can be entered is 999. If we changed the size to “2” the largest number we could enter would be 99. It indicates the number of characters. We needed to put “4” as the size for IntAnswer because the largest possible value it may encounter is 999 + 999 = 1998. It is also possible to identify the scope of the variable. If it is limited to a subroutine, then the variable would be “LOCAL” or if it accessible from anywhere in the program is “GLOBAL”.

A data dictionary assists the software developer to plan their program. When naming variables and setting data types it is important then to identify which Graphical User Interface (GUI) objects will handle each of the variables. Setting a consistent approach to naming variables will assist the programmer in naming and organising the objects and modules required in the solution.

Object Description Table

When designing your interface, especially in Visual Basic, you need to identify objects that will handle your variables, and in turn, your data. It is important to use Camel Case and Hungarian Notation in the naming of objects as well. You can see by the Graphical User Interface (GUI) in figure 1.11 that the two input objects are text boxes (txtNumber1 and txtNumber2). The OUTPUT will be displayed in a label (lblAnswer). The entire program will be executed when the button object (btnCalculate) is clicked.

Fig 1.11



Now that we have identified our objects, we can create an Object Description Table, see figure 1.12. This is a way of formally identifying the name, type, purpose and properties of each object. You can see in the table below that there is a column titled: Method/Event. This identifies how the objects are affected by things that happen (events) during the execution of the program or if the properties of the object are changed (methods). For example: lblAnswer is a label that will have its "text" property changed in the execution of the program. This is a method. The button btnCalculate triggers the program execution when the event_Click occurs. This is an event.

Fig 1.12

Object Name	Type	Method/Event	Properties	Description
txtNumber1	textbox	Method	Set Text: Null	Reads in IntNumber1
txtNumber1	textbox	Method	Set Text: Null	Reads in IntNumber2
lblAnswer	label	Method	Set Text: Null	Displays lblAnswer
btnCalculate	button	Event	Text: "Calculate"	When this is clicked it will trigger the execution of the calculation

A good rule of thumb when planning a project is to create your design tools in this order:

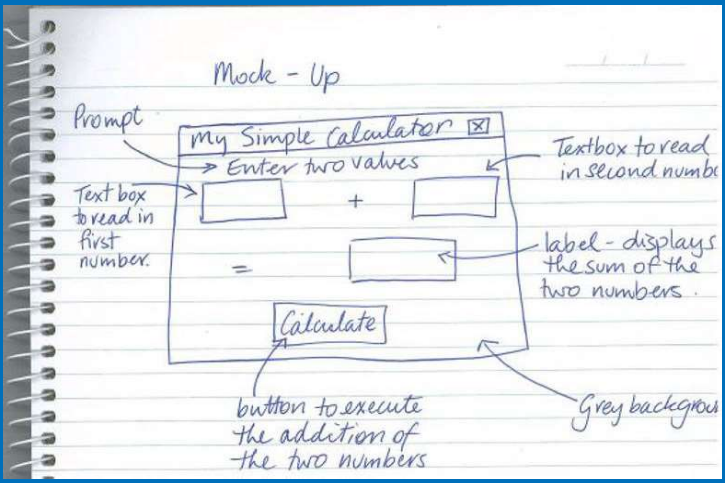
1. Interface design as a Mock Up or Storyboard
2. From the interface design, label all the objects in the interface.
3. Create the Object description table from your labeled interface.
4. List the variables that each object handles.
5. Create the Data Dictionary from the variable list.
6. Work through the DFD in stages to organise your algorithm.
7. Write your pseudocode from the algorithm notes.

Mock Up - Design Layout

A Mock Up or Design Layout is a plan that identifies what the GUI will look like. The illustration below in figure 1.13 shows a rough hand-drawn design for the interface of the simple Calculator. It identifies each of the objects and their purpose as well as design elements. The more details you can include, the better prepared you will be once it is time to develop your solution. Perhaps even add the colour and font choices.

Fig 1.13

Mock Up


Mock - Up
Prompt
My Simple Calculator
Enter two values
Textbox to read in second numb
Text box to read in first number.
+
=
label - displays the sum of the two numbers.
Calculate
button to execute the addition of the two numbers
Grey backgrou

Pseudocode

Once you have decided on the GUI and the objects and variables required to solve your programming problem, it is time to define the events that take place when executed. Rather than going directly to Visual Studio and start typing up your code, it is essential that you plan your structure in an algorithm using pseudocode. Pseudocode is natural language that is easy to read but follows the structure of a programming language.

The Features of Pseudocode

- Every algorithm must begin and end with START and STOP or BEGIN and END.
- When assigning data to a variable use ← . Example: IntNumber ←5
- Show each calculation required. Example: IntAnswer ←IntNumber1 + IntNumber2
- Displaying data as output. Example: lblAnswer.Text ←IntAnswer or Display IntAnswer
- Decision Control Structures. Example: IF condition THEN Action1 ELSE Action2 ENDIF
- Case Control Structures. Example: Case where condition, CASE1 Action1 CASE2 Action2 END CASE
- Repetition Counted Loop Control Structures. Example: FOR counter ← first value to last value END FOR
- Repetition Post-Test Loop Control Structures. Example: REPEAT actions UNTIL condition
- Repetition Pre-Test Loop Control Structures. Example: WHILE condition DO actions END WHILE
- Use indentation and spaces to illustrate how control structures are incorporated
- Conditions: ==(equal), <> (not equal), > (greater than) and < (less than).

Below is an Algorithm written in Pseudocode that allows the user three attempts at getting their username and password correct. The program checks the password is correct at each attempt and prompts the user with the number of remaining attempts. Can you name six algorithm features in the example below?

```
START
    strUsername ← BillBurr123
    strPassword  ← Summer1965

    FOR counter = 1 to 3 DO
        Read in strUsernameEntered
        Read in strPasswordEntered
                IF (strUsername == strUsernameEntered AND strPassword == strPasswordEntered) THEN
                        Display Prompt "Access Granted."
                ELSEIF
                        Display Prompt " Incorrect credentials. You have " (3 - Counter) " attempts left"
                ENDIF
    END FOR
END
```

Formatting and Structural Characteristics of Files

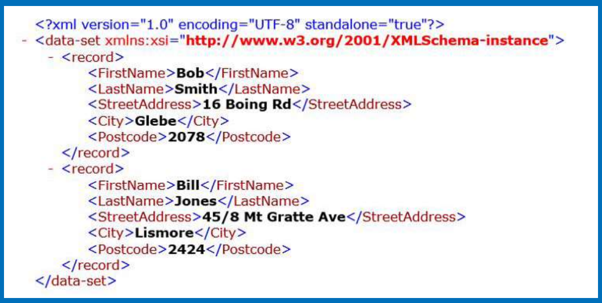
So far we have only discussed software solutions where the user enters the data via a keyboard and mouse. This is not always the most efficient method especially when we have large amounts of complex data. Sometimes data is collected from CCTV capture, such as car registration plates moving through an intersection. Collecting data out in the field may require locations to be input which could be collected via a GPS device. When dealing with large amounts of data it is important that the data is structured in such a way that it can be accessed, sorted, searched, saved and retrieved. Software solutions such as our Simple Calculator do not save data for use after the program has been turned off. To create effective, robust software solutions these programs need to save data permanently so it can be accessed again after the software has been shut down. This requires files that can hold the data.

A fundamental approach is to use a file such as a text file that you can edit in a basic text editor. This is a great approach if you are making a simple application where the client needs to update prices. Examples of how Visual Basic can access and manipulate data stored in a text file can be found in the programming Chapter 5.

If you need a more structured approach such as the use of a spreadsheet you might need to upgrade from a basic text file to an XML file. When you have data structures such as records, it is best to be working with fields. XML uses tags in much the same way HTML is structured. The figure 1.14 shows a table of five records. It is described and controlled by the XML tags. An XML file is a basic text file that is "self-descriptive'. It can easily be read by software solutions and by humans, see figure 1.14. Excel can easily produce an XML file from your data. Excel can also produce Comma Separated Value files (CSV) that can store records in much the same way. CSV files store tabulated data in a basic text file that can be imported into a spreadsheet or database.

Fig 1.14

Records in an XML File



```
  <?xml version="1.0" encoding="UTF-8" standalone="true"?>
- <data-set xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
  - <record>
        <FirstName>Bob</FirstName>
        <LastName>Smith</LastName>
        <StreetAddress>16 Boing Rd</StreetAddress>
        <City>Glebe</City>
        <Postcode>2078</Postcode>
    </record>
  - <record>
        <FirstName>Bill</FirstName>
        <LastName>Jones</LastName>
        <StreetAddress>45/8 Mt Gratte Ave</StreetAddress>
        <City>Lismore</City>
        <Postcode>2424</Postcode>
    </record>
  </data-set>
```

Validation

The interface between machines and humans is fraught with opportunities for data loss, mistakes and possible nefarious intentions. Validation is the process of checking the input of data. Manual validation includes spell checking, proof-reading and fact checking. These are processes the user can conduct before entering data into a system. As a software engineer, it is important to anticipate human error when reading in data to your program by including validation in the code.

Common validation techniques include:
- Existence Checking
- Type Checking
- Range Checking

Existence Checking is a validation method that checks if data is present in a variable or object. This can be an important aspect of your interface design. If the user has not included an important data element, existence checking will detect it and prompt the user to enter the missing data. For example: in a pizza ordering app, if the user has not selected a pizza type, it is impossible to process an order, so a message window would appear to prompt the user to make a pizza type choice.

Type Checking is a validation method that checks the data type of the data entered by the user. This is vital to detect typing errors. If a "K" is typed into the input object for "Quantity", type checking will detect that the data type is not numerical and return a prompt to the user to enter a value to indicate how many pizzas they want to order.

Range Checking is a validation method that checks if a value sits between two limits. A common use of range checking would be on the input of "date of birth". To ensure those logging in to an app are over 16, the range check could test for dates after 1/1/2004. Similarly, dates before 1920 would also be excluded due to the unlikelihood that users would be over the age of 100.

Programming Languages

Programming Languages are coded instructions that both a human and a machine can understand. There are two main types of programming languages: interpreted and compiled. An interpreted language requires software to interpret and run the code. These languages rely on the browser software to interpret and run the code in their window. A common example of interpreted languages are those that run inside browsers on websites such as:

- PHP
- Excel
- XML
- HTML
- JavaScript
- Perl

Compiled Languages are converted into machine code and can run independently of other software. The code can be compiled in an Integrated Development Environment (IDE) or through a text editor. They create a stand-alone software application. Languages include:

- Python
- Visual Basic
- C++
- C#
- Java

Visual Basic is a compiled Object Oriented Programming (OOP) language. It enables the user to design a Graphical User Interface (GUI) easily with a point-and-click process in the IDE. The GUI is designed by arranging input, output and processing objects on a window or form. Each object is named and its properties edited according to its association with a data structure.

Procedures, Subroutines, Events and Modules

In Visual Basic(VB) there is a hierarchy of coding segmentation. A complete Visual Basic application is called a Project. The default name is WindowsApplication1. Within a project there may be many code files called Modules. These maybe called something like Form1.vb.

Once a module has been created we write the procedures as code. In VB there are two types of procedures; functions and subroutines. Functions perform tasks that return a value. Subroutines perform tasks but may not return a value. Below is a module that adds two numbers together. It is a Private Subroutine that reads in IntNumber1 and IntNumber2, adds them and displays the answer. It is a self contained program.

```
MODULE modAddition
    Private Sub AddNumbers (ByVal IntNumber1 As Integer, ByVal IntNumber2 As Integer)
    Dim IntAnswer As Integer
    IntAnswer ← IntNumber1 + IntNumber2
    MsgBox ← IntAnswer
    End Sub
END MODULE
```

If multiple subroutines were required and the intAnswer value shared between them, the subroutines would need to be public. Below is an example of a subroutine that executes when the “Calculate” button is clicked. It reads data from GUI objects but does not return a value. Notice how the subroutine is Public.

```
Public Sub Calculate_Click (sender As Object, e As eventsArgs) _ Handles Button1.Click
        Dim IntNumber1 as Integer
        Dim IntNumber2 as Integer
        IntNumber1 ← txtNumber1.Text
        IntNumber2 ← txtNumber2.Text
End Sub
```

A function procedure is a series of mathematical procedures enclosed by “Function” and “End Function” and returns a value to the code that called up the function. Below are two functions that return the result of a mathematical process. These functions can be called up from anywhere in the program.

```
Public Function AddNumbers (ByVal IntNumber1 As Integer, ByVal IntNumber2 As Integer)
        Dim IntAnswer As Integer
        IntAnswer ← IntNumber1 + IntNumber2
END Function

Public Function DivideByThree (ByVal IntAnswer as Integer)
        Dim DblThird As Double
        DblThird ← IntAnswer / 3
        Return DblThird
END Function
```

It is now possible to create a module that will use the subroutine and the two functions. The example on the next page shows how a Module called “CalculatorApp” utilises the functions “AddNumbers” and “DivideByThree” to process the input from subroutine “Calculate”.

```
Public Class CalculatorApp

    Public Sub Calculate_Click (sender As Object, e As eventsArgs) _ Handles Button1.Click
        Dim IntNumber1 as Integer
        Dim IntNumber2 as Integer
        IntNumber1 ← txtNumber1.Text
        IntNumber2 ← txtNumber2.Text

        Call AddNumbers(IntNumber1, IntNumber2)
        MsgBox(IntAnswer)

        Call DivideByThree(IntAnswer)
        MsgBox(DblThird)
    End Sub

    Public Function AddNumbers (ByVal IntNumber1 As Integer, ByVal IntNumber2 As Integer)
        Dim IntAnswer As Integer
        IntAnswer ← IntNumber1 + IntNumber2
        Return IntAnswer
    END Function

    Public Function DivideByThree (ByVal IntAnswer as Integer)
        Dim DblThird As Double
        DblThird ← IntAnswer / 3
        Return DblThird
    END Function
End Class
```

Calls up the functions inside the subroutine so they can be used multiple times throughout the module.

Control Structures

To control the order in which your functions and procedures are executed, some structure needs to be put in place. There are three main control structures:

- Sequence
- Selection
- Iteration

Sequence is simply the structure and order of instructions in the correct sequence. The example below is an algorithm that reads in two values adds them together then displays the answer.

```
START
    Read In Number1                 (Instruction 1)
    Read In Number2                 (Instruction 2)
    Answer ← Number1 + Number 2     (Instruction 3)
    Display Answer                  (Instruction 4)
END
```

Decision control structures determine the direction of the program based on condition statements. The example below reads in two values, adds them together and tests if that answer is odd or even using the Mod of 2. If there is a remainder after Answer is divided by 2 then it is odd.

```
START
    Read In Number1
    Read In Number2
    Answer ← Number1 + Number 2
    IF Answer Mod 2 > 0 THEN            (IF condition is TRUE it will do Procedure 1)
        Display "The Answer is EVEN!"   (Procedure 1)
    ELSE                                (IF condition is FALSE it will do Procedure 2)
        Display "The Answer is ODD!"    (Procedure 2)
    END IF
END
```

Iteration is a control structure that repeats a set of procedures in a loop. These loops can be controlled by counting the number of times the loop completes or by testing conditions. The first example is a Counted Loop. This algorithm will read in three guesses via an input box.

```
START
    Writeline: Guess the Number! You get three guesses!
    FOR Counter 1 to 3 DO
        Guess ← Input Box( Take a guess )
    END FOR
END
```

The second example is a Pre-Test Loop. Here a condition must be met before the loop is executed. This algorithm shows the loop commences if the condition (the guess is wrong) is true.

```
START
    Writeline: Guess the Number! Keep guessing till you get it right!
    Guess ← Input Box( Take a guess )
    WHILE Guess <> Answer
        Guess ← Input Box( Take a guess )
    End WHILE
END
```

The last example is a Post-Test Loop. This algorithm tests the outcome of the first loop and keeps looping until the guess is correct.

```
START
    Writeline: Guess the Number! Keep guessing till you get it right!
    REPEAT
        Guess ← Input Box( Take a guess )
    UNTIL Guess = Answer
END
```

Searching

Linear Search

If we were looking for "Bunmi" manually in our list of names below, we might start at the top and run our finger down the list to check each element. Linear Search does exactly that. It begins at the start of the array and uses a loop to check each element until it is found. Below is an algorithm to search for Bunmi in the list. EOF means "end of file".

```
START
    Counter ← 0
    Found ← False
    WHILE Found = False AND Not EOF DO
        IF Names (Counter) <> Bunmi THEN
            Counter ← Counter + 1
        ELSE
            Found ← TRUE
        END IF
    END WHILE
END
```

0	Robert
1	Manpreet
2	Brian
3	Julia
4	Sudha
5	Bunmi
6	Jackson
7	Maxine

This type of searching approach is fine for small lists, but if you have a telephone directory of data to search, this is very inefficient.

Binary Search

Clearly we need a better method than a Linear Search for larger files, and that is where Binary Search is useful. Unfortunately the list must be SORTED first to conduct a binary search.

You can see our array Names(7) below has been put into alphabetical order making it easier to search.

```
Names (0) = Brian
Names (1) = Bunmi
Names (2) = Jackson
Names (3) = Julia
Names (4) = Manpreet
Names (5) = Maxine
Names (6) = Robert
Names (7) = Sudha
```

First we define the lowest, highest and middle elements of the list.

```
LOWEST element ← Names (0) = Brian
MIDDLE element ← Names (4) = Manpreet
HIGHEST element ← Names (7) = Sudha
```

Now we check the name we are searching for "Bunmi" against the MIDDLE element.
Is Bunmi before or after Manpreet in the alphabet? Bunmi is before(<) Manpreet. So we now redefine our lowest, middle and highest. Because Bunmi is before Manpreet we make the old MIDDLE the new HIGHEST and then find a new MIDDLE.

```
LOWEST element ← Names (0) = Brian
MIDDLE element ← Names (2) = Jackson
HIGHEST element ← Names (4) = Manpreet
```

Again we test to see if Bunmi is > or < our new MIDDLE element Jackson. Since Bunmi < Jackson, we set the MIDDLE to be the new HIGHEST once again.

```
LOWEST element ← Names (0) = Brian
MIDDLE element ←Names (1) = Bunmi
HIGHEST element ← Names (2) = Jackson
```

Now once again we check our MIDDLE element and we have found our name "Bunmi".

Comparison of Linear and Binary Searches

A Linear Search needs to potentially conduct as many operations as there are elements in the list being searched. Bunmi was the 6th element in the unsorted list so at least 6 operations were required to find that name. However, with the Binary search only 3 operations found the same element. How many operations would take to find Robert?

Operation 1

```
LOWEST element ← Names (0) = Brian
MIDDLE element ← Names (4) = Manpreet
HIGHEST element ← Names (7) = Sudha
Maxine > Manpreet so MIDDLE is the new LOWEST
```

Operation 2

```
LOWEST element ← Names (4) = Manpreet
MIDDLE element ← Names (6) = Robert
HIGHEST element ← Names (7) = Sudha
```

Binary search is more complex to code, but it is more efficient and the most suitable solution for a large index of data. Linear is easier to code and most suitable for short files. We often identify the efficiency of a method by its worst case scenario. If the item you are looking for is the last one in a list of n number of items, the item will be found in O_n number of operations. If the list is 100 items long, it will take 100 operations. For Binary search the efficiency is increased logarithmically. An item in a list of n items will be found in O_{logn} which is about 30 operations for worst case scenario of a 100 item list.

Sorting

Despite many languages containing a 'sort' function, in major software solutions it is important to manage the sorting of data with control structures. We will look at two key sorting methods: Selection Sort and Quick Sort.

Selection Sort

The simplest sort is the Selection Sort which functions the way you would naturally sort items. It looks through the whole list looking for the smallest item and places it at the beginning of the list. This is called a PASS. The sort then passes the rest of the unsorted list for the next smallest item and adds it to the new sorted section. It COMPARES the item to the last item in the sorted section and SWAPS them so they are in order. This process repeats until the list is completely sorted. This is where arrays become very useful. We can use the index of each data item to reposition the data into a new order. We use nested loops. This allows each data item in the array to be compared with every other item in the array on each PASS. This means it can be an inefficient solution for long lists. An array(n) can require up to n^2 number of operations to sort.

A Selection Sort Algorithm to sort 100 data items in an array ArrayList(99) follows. A PassCounter repeats each PASS 97 times. For each PASS the smallest item is searched for and then put at the start of the unsorted section of the array.

```
BEGIN

  FOR PassCounter: 1 to (ArrayLength - 2) DO

     Smallest ← ArrayList(PassCounter - 1)
     SmallestIndex ← PassCounter

        FOR ItemCounter: (PassCounter - 1) to (ArrayLength - 2) DO

           IF ArrayList(ItemCounter + 1) < Smallest THEN
              Smallest ← ArrayList(ItemCounter + 1)
              SmallestIndex ← ItemCounter + 1
           END IF

        END FOR

        Temp ← ArrayList(PassCounter - 1)
        ArrayList(PassCounter - 1) ← ArrayList(SmallestIndex)
        ArrayList(SmallestIndex) ←Temp

  END FOR

   Display ArrayList(99)

END
```

Unsorted Array

k	e	a	m	s	q	b	j

First Pass and Swap

a	e	k	m	s	q	b	j

Second Pass and Swap

a	b	k	m	s	q	e	j

Third Pass and Swap

a	b	e	m	s	q	k	j

Fourth Pass and Swap

a	b	e	j	s	q	k	m

Fifth Pass and Swap

a	b	e	j	k	q	s	m

Sixth Pass and Swap

a	b	e	j	k	m	s	q

Final Pass to Swap - Sorted Array

a	b	e	j	k	m	q	s

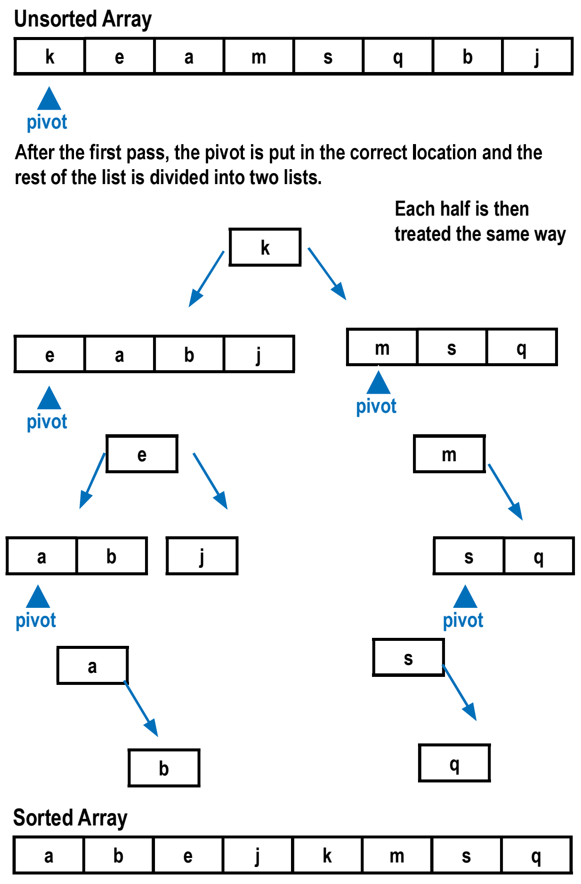
Quick Sort

Quick Sort is more sophisticated sorting technique using Divide and Conquer around a Pivot. This is a more complex sorting solution that uses recursion. Recursion is where a procedure or function calls itself. The algorithm sets StartIndex and EndIndex to the beginning and end of the array. In this situation we set the pivot to the first item. The function checks the StartIndex and the EndIndex data against the Pivot. Depending on the data it will add 1 to the StartIndex or minus 1 from the EndIndex. When data is found to be lower than the ArrayList(StartIndex) it is swapped.

You can see at the bottom of the algorithm that the Function QuickSort calls itself up to run (0 to EndIndex) then again for (StartIndex to ArrayLength). The nature of Quick Sort makes it more efficient and can sort a large list in only $O_{n \log n}$ number of operations.

The bulk of the algorithm runs a pass. Each pass places the pivot in its correct location with all the values lower than it on one side, and all the values higher on it on the other. Each of these sides are now treated as separate lists that will run the pass through. For each half the process continues to divide and conquer until each item has become a pivot and is located in its correct place.

```
START FUNCTION: QuickSort
BEGIN
    IF ArrayLength >1 THEN
        Pivot ← ArrayList(0)
        StartIndex ← 0
        EndIndex ← ArrayLength

        WHILE StartIndex <= EndIndex DO

           WHILE ArrayList(StartIndex) < Pivot DO
                StartIndex ← StartIndex + 1
           END WHILE

           WHILE ArrayList(EndIndex) > Pivot DO
                EndIndex ←EndList - 1
           END WHILE

           IF StartIndex < = EndIndex THEN
                Temp ← AryList(StartIndex)
                ArrayList(StartIndex) ← ArrayList(EndIndex)
                ArrayList(EndList) ← Temp
                StartIndex ← StartIndex + 1
                EndIndex ←  EndIndex - 1
           End IF

        End While

    Function QuickSort (0 to EndIndex)
    Function QuickSort(StartIndex to ArrayLength)

    END IF

END
```


Unsorted Array
k e a m s q b j
pivot
After the first pass, the pivot is put in the correct location and the rest of the list is divided into two lists.
Each half is then treated the same way
k
e a b j
m s q
pivot
pivot
e
m
a b
j
s q
pivot
pivot
a
s
b
q
Sorted Array
a b e j k m s q