

# Software Design and Development HSC Course Notes 2011

By MrBrightside

# Chapter 1: Social and Ethical Issues

## Rights and responsibilities of software developers

Issue	Rights	Responsibilities
Authorship	Protection of their products against theft and modification without permission.	Acknowledge the authors and sources used in development.
Reliability	Right to avoid OS, hardware / program problems which may interfere with the program.	Check that the product works with the OS, ensure there are no runtime errors. Runs as it should.
Quality	Codes that follow high standards.	Thorough testing, meet user's needs.
Response to problems	Not to be harassed by basic user-created problems.	Provide troubleshooting methods, online help, user manuals, free bug fixes asap.
Code of conduct	Follow same ethical standards.	Follow the standards set by the organisation to which they belong.
Viruses	Protection of their products with users, by users using updated anti virus solutions.	Do not distribute viruses.

## Software piracy and copyright

Concept	Meaning
Software piracy	Theft of software.
Intellectual property	Personal ownership of the creative ideas that develop from an individual's mind.
Plagiarism	Theft of ideas and expressions of another person.
Shareware	Software distributed for trial use before purchase.
Freeware	Free and copyrighted software that may not be sold or modified.
Public domain software	Free software that can be modified and copied legally. No copyright.
Ownership versus licensing	User only owns the medium, the right to use the software (licence) and any files personally created by the user from the program. The developer still retains all aspects of the program.
Copyright laws	Legal protection of computer programs against illegal copying, modifying or distribution. Developer(s) control the use of their products. 'Fair dealing' when software may be copied and used ONLY for educational / research purposes.
Reverse engineering	Process of reading source code and translating it to an algorithm, which may then be recoded in the same or another programming language.
Decompilation	Object code (machine code) translated into source code (user friendly code). Legal only if the developer owns the program.
Licence conditions	Determine what can be done with software. EULA determines what the user can do and must be accepted by the user before the installation.
Network use	Computers joined together that share files and devices. <u>Centralised software</u> : single copy from a central server, <u>Distributed software</u> : individual copies on local machines. Cheaper licences that cover each machine. Specific number of machines is stated by the licence.

## National perspectives to software piracy and copyright laws

**Warez:** commercial software that have been illegally copied / pirated and made public available.

Universal copyright law did not succeed, as some countries did not want to accept the law, thus in some countries pirating is legally accepted, but not by software developers, which creates conflict. People around the world need to be ethically educated in order to even try to avoid this problem.

## The relationship between copyright laws and software license agreements

Software licence agreements are contracts that are designed to protect the developer's ownership of the software they have created, also know as EULA – end user licence agreement. Copyright laws protect EULAs from abuse. A huge amount of money is spent on security measures to help prevent piracy, often resulting in expensive software prices.

*Types of software licences:*

Licence	Meaning
Single-use	One copy, installed on one computer only.
Multi-use	One copy, installed on specified number of computers.
Network	One copy, installed on a huge number of machines, across several sites. (Cheaper per machine).
Site	One copy, installed on a single site only. E.g. school. (Cheaper per machine).

## The software market

### Maintaining market position

- Customers expect the product to meet their needs and live up to their expectations.
- Smart advertising is also required to promote the product.
- Software developers need to continue to support their products well after they have been released, in order to retain good customer satisfaction and reputation.
- Software developers need to have the following:
- Good Selling price, good customer support, ethical reputation, membership offers, wide distribution and bug-free software.

### The effect on the marketplace

- Competitive products need to emphasise their unique capabilities to attract the market.
- New products need to convince users that they will do the task more efficiently.
- Packaging and retailing of the product is also important.
- Product is then discussed in the public domain through internet chat rooms and news which stimulates further sales.

# Significant social and ethical issues

## National and international legal action resulting from software development

Rulings have developed from laws based on other forms of intellectual property.

### Sega versus Accolade

Accolade won against Sega, because Sega did not declare reverse engineering in their EULA, therefore the court ruled out that Accolade did not steal Sega's creative works and was legally accepted.

If codes looked similar, Accolade would have been in the wrong, but because they coded in a different language, the codes looked different therefore it was the property of Accolade.

### Apple versus Microsoft

Apple wanted to sue Microsoft for using a similar GUI. The court ruled out that apple's GUI components were not original and that windows, icons, menus and pointers are universal and could not be copyrighted. Xerox and IBM had invented the original ideas. Court only ruled out that, only precise designs can be copyrighted, not broad designs. Apple lost.

## Public Issues relating to software development

Essential utilities, such as water, electricity and telephones have software as their basic controllers. Just about all machinery is controlled by software.

### Year 2000 problem (Y2K):

Ram was expensive with early computers. Programmers reduced the storage of the year format, 2 digits instead of 4 digits. Apple stayed with 4 digits. Governments and large businesses had to upgrade to 4 digit year format (costly). This was done, no problems were created.

Two reasons why it happened apart from memory:

- Programmers assumed the 2 digit year would be adjusted, as software was advancing rapidly.
- Other programmers did not even see the problem as it was too distant into the future.
- The lesson: programmers have to consider all aspects of software they develop, especially its viability into the future.

### Computer viruses

- **Virus:** A program that alters the functioning of software without the permission of the user.
- Viruses can be annoying interruptions such as pop ups to potentially destructive codes that can destroy all data across a network.
- Software developers and users must install and update anti virus programs.
- Virus definitions can never be 100% up to date, although anti virus companies are always on the look out for new emerging viruses.

# Chapter 2: Application of Software Development Approaches

## Approaches used in commercial systems

**Off-the-shelf:** software bought 'off the shelf' which are standard retail packages, such as Microsoft office. The user can customise the software to an extent. Cheaper selling price because more people will buy it.

**Custom:** software written specifically to suit the needs of a particular organisation or individual. Much more expensive and time consuming than off-the-self.

### Major approaches to commercial software development

Approach	Advantages	Disadvantages
<b>Structured</b>	Thoroughly tested Meets exact needs Requires experts	Expensive Time consuming Requires wide range of different skills
<b>Prototyping</b>	Fast development Models a larger project	Difficult to implement as a final product
<b>Rapid application development (RAD)</b>	Fast development Cheap Uses reusable code	May not meet exact needs May involve copyright and intellectual property conflicts
<b>End user development (EUD)</b>	Meet exact user requirements Quick and cheap	Limited to simple projects and application programs

## The Structured Approach

- Follows the system development cycle (series of steps followed by a development team), i.e. defining, planning, implementing, testing and maintaining. May be necessary to revisit previous steps or start again if problems arise.
- Lengthy process, suitable for complex problems that need to be solved.
- People involved: System analyst, software engineers, programmers, graphic designers, consultants, managers, trainers and users.
- A system analyst starts the entire process; they see the problem as a whole and divide it into several parts.

### Phase 1: Defining the problem

- Thoroughly understand and define the problem.
- Little time and cost will be required to fix any problems at this phase.
- One third of the development time.
- Report may need to be repeatedly modified until all personnel are convinced that it will solve the problem.

### Phase 2: Planning the solution

- Further understanding of user needs and methods to solve a problem are undertaken
- Data will need to be collected from written documents, questionnaires, surveys, interviews and observation to provide a basis for a decision.
- Plan what data and structures are to be used (dataflow diagrams, IPO charts, desk check- manual check)
- Design algorithms (a finite number of steps that solve a problem)
- Plan user interface (screen designs and storyboards)

- Plan scheduling (Gantt chart)
- Choose a programming language (system analyst and team decide based on, the program, skills and user requirements).
- Initial programming is undertaken as testing.
- Documentation is finished up and past by management.

### **Phase 3: Building the solution**

- Requires full involvement of the development team.
- **Stepwise refinement** is used, i.e. overall problem is divided into smaller, more easily managed modules.
- Modules should be consistent.
- Greater efficiency as all members can design and test different modules separately, all at the same time.
- Modules can be reused and modified in the same or different project.
- The program is coded according to the algorithms.
- **Runtime testing:** Testing that is undertaken throughout code development to ensure there are no errors.
- Cost of fixing the solution is still low at this phase.

### **Phase 4: Checking the solution**

- Continuous phase
- Testing the program using real data with acceptance and beta testing.
- **Acceptance testing:** users that test custom software.
- **Beta testing:** potential users of the product test the program.
- Program is passed by the system analyst when it is free of error and meets the requirements set.
- Management approval is then needed for the program to be implemented.

### **Phase 5: Modifying the solution**

- Modifications will generally be required after a program has been implemented.
- Updates can lead to greater efficiency, solve bugs caused by external factors such as new hardware and other software changes or even introduce new features to cover new tasks.
- Very expensive to make changes at this phase.
- Only undertaken if changes are small and crucial to the operation of the program.
- If a major upgrade is needed, the system analyst may draw to a conclusion that better results could be achieved by beginning the software development cycle again.

### **Advantages of using the structured approach**

- Easy to write. Programmers can reuse modules.
- Easy to understand. The use of comments and meaningful names allows modules to be tracked easily.
- Easy to test and debug.
- Easy to modify.
- 

### **Characteristics of the structured approach**

Used when:

- The project is large scale and complex
- The budget is large
- The time available is considerable
- A program that is easier to debug, understand and modify is required
- Project development team is involved

## The Prototyping Approach

- Design by creating and refining a prototype through user feedback.
  - Higher amount of interaction with user.
  - Used mainly in smaller scale, low budget projects.
  - Rapid development time.
  - Improved communication with the end user.
  - Documentation is crucial for future maintenance.
  - Prototypes are more about getting the GUI right and will not deal with security or error recovery.
  - Reduced documentation adds a faster completion time but may lead to a system that is hard to maintain.
  - Tools: Fourth generation languages are usually used and libraries of reusable code with CASE tools to generate test data.
- 
- **Information-gathering prototypes:** developed to gather information that can be used in another program. Concentrates on input and out with minimal processing. Never intended to be a full working program.
  - **Evolutionary prototype:** become the full working program.

Used when:

- A small team is needed
- Further understanding is required to develop a larger program
- Less time is available
- Small budget/scale

## Rapid Applications Development Approach

- Quickly generates a program for the user
- Uses existing modules, CASE tools, templates and reuses code
- Lacks formal stages
- Programmer is directly involved with the user
- Sometimes implements the modification of COTS (Customised off-the-shelf packages).

Used when:

- A very small team is involved
- Low budget
- Minimal time
- Small-scale projects
- Small range of coding languages is used.

## End User Development (EUD)

- When the person who is going to be operating the software develops it themselves.
- Useful as the user solves their own problem quickly.
- Lack of formal stages.
- Small scale and low budget.
- Often involves the modification of COTS (Customised off-the-shelf packages).
- Fourth generation language, such as a spreadsheet, database management system, macro
- Slight disadvantage is that end users may be repeating the same process in an organisation, this can be overcome by making a program for each machine.

Used when:

- Existing software solutions are utilised
- Very little cost / time
- Small problems E.g. Changing the colour of font, changing the wallpaper, creating SQL statements, Utilising predefined functions in Excel spreadsheets such as SUM.

### Combinations of any of the four major approaches

- Some aspects of a solution may lend themselves to one development approach over the other.
- This may lead to a combination of development approaches being used to create a solution.
- Useful for when developers find that another approach may be more efficient over the chosen approach.

## Methods of Implementation

- **Direct cut over:** The new program immediately replaces the old program.
  - All data in the old system will need to be converted to work with the new system.
  - If the new system fails, the old system is not available as a backup.
  - Cheapest and quickest method of implementation.
  - Less pressure on users / participants.
- **Parallel:** The old and new program work simultaneously for a period of time.
  - All operations of the new system duplicate similar operations in the old system.
  - If the new system fails, the old one can still be used with minimal loss.
  - Creates additional workloads and costs more to run two systems at once.
  - May cause confusion, example: two different file formats.
- **Phased:** One or more tasks of the new program are gradually implemented until the new program takes over all tasks of the old program.
  - Allows each new module to be tested individually.
  - If a module fails, the whole system is not affected.
- **Pilot:** One section of the organisation uses the new program and all other sections continue with the old program until a decision is made to put the new program into place across the whole system.
  - Once the system is proved, another implementation method (usually direct cut over) is used to implement the rest of the system.
  - Usually requires a set of keen participants to trial the new system

### Factors that are involved in choosing a method:

- Size, scope and nature of the organisation
- Complexity of the program
- Purpose of the program
- Amount of time available for conversion
- Skills of participants / users.



# Current trends in software development

**Outsourcing:** Contracting work to outside developers, often specialists in their field.

- Reasons why companies outsource:
  - Better response to change.
  - Reducing control costs.
  - Higher quality results.
  - Free internal resources.
  - Resources not available internally.
  - Access to new technology.
  - Faster development times.
- Disadvantages of outsourcing include:
  - Increased dependence on third parties.
  - Lack of control over outsourced departments (may be a breach in contract)
- **Popular approaches**
  - Moved from being process oriented to data oriented and object oriented.
  - Data oriented involves the use of logical and abstract data types.
  - Object oriented allows data abstraction, encapsulation and inheritance as well as easily accommodates code reuse.
  - End-user development
  - Visual Studio (Visual basic) – with its large library of reusable codes.
  - Increased usage of CASE tools to aid development.
  - Hypermedia – the use of text, sound, video, animation and graphics (multimedia) with multiple hyperlinks.
  - Authoring Languages such as XML (extensible markup language) guide the developer through steps.
- **Popular languages**
  - Majority of code is still written in 3rd generation languages.
  - Growing use of 4th generation languages.
  - Languages like Visual Basic, Visual C++ and Delphi which increase programmer productivity.
  - Most of the popular languages have the same set of rules, called the syntax.
- **Employment trends**
  - Greater number of jobs going to programmers with a knowledge of object oriented languages.
  - Move towards contract based work and outsourcing rather than full time employment.
  - Programmers may spend 6 months with one company, then go to a new company to start a new project
- **Networked software**
  - Networks of computers, especially the Internet have created a new and growing environment for software development.
  - Software applications can be shared amongst users in the following ways:
  - Mainframe computers where slices of time are allocated to each terminal to execute programs on the mainframe's processors.
  - Computers linked together to enable sharing of software, data, and peripheral devices.
  - Client/server applications where many clients access and manipulate a central database.
  - Involves the full structured approach in developing network operating systems.
  - Security becomes very important and the use of product keys to ensure piracy does not prevail and that the public respect developer's copyrights and conditions of use. The ability to share so easily across millions of machines world wide becomes an immense task to handle and control.
  - Software developers must ensure that participant and user personal data remain private and secured, especially over internet banking sites and even over local intranet networks.

- **Customised off-the-shelf (COTS) packages**

- Software that can be purchased and modified to suit particular needs.
- Examples include Microsoft Office for desktop publishing and MYOB for accounting.
- Increasing in popularity in programming industry as they are usually:
  - Cost effective
  - High quality and reliable
  - Upgradeable

## **The use of CASE tools and their application in large systems development**

### CASE tools are designed for:

- Problem tracking, example: automatic error detection and correction tools in Microsoft Visual Studio 2010.
- Data modelling for processing modules.
- Document generation for manuals and troubleshooting guides.
- Programs to incorporate reusable library modules (code for a basic function/task).
- Analysis of design and scripts.
- Reverse engineering, from compiled program to algorithms, so that a program can be reprogrammed in a new language.
- Simulation tools for prototype models.
- Support for design specifications (knowledge based requirements assistant -KBRA)
- Support for software design, a consistent user interface.
- Support for implementation, such as language specific editors.
- Generation of code from documentation (algorithms and data flow diagrams)

### Limitations of CASE tools:

- May not allow the developer to work with what they prefer with, example: graphical or text mediums.
- Not suitable for rapidly application development, as double the work has to be done to cater for CASE tools.
- If the wrong CASE tools are chosen, the project may be rendered incapable of completion.
- CASE tools are slower than the human programmer and far less in flexibility.

- **Software versions**

- Allows teams of developers to collaborate and track the changes made to a program.
- Old versions can be easily restored if necessary.
- Major versions contain whole numbers and usually add a new GUI, more useful features and speed up the program. Example: Windows 7: Version 7.00
- Minor changes occur when a bug may need to be fixed or compatibility issues arise with new hardware and need to be fixed quickly with minimal disturbance. Example: Windows 7: Version 7.004.

- **Data dictionary**

- Used to identify and fully define data, data characteristics and data relationships.
- CASE tools combine graphics tools and the data dictionary to provide design & development tools for programmers.

- **Test data**

- CASE tools can be used to generate test data for testing a coded solution.
- Used to test many/all of the possible data paths.
- CASE tools are available which automatically acquire data to be used, analyse the source code and simulate hardware. They will then generate a report and list the tested and untested data paths and their success.

- **Production of documentation**

- Many CASE tools can be used for the creation of both internal and external documentation.
- CASE tools are available for the creation of diagrams in the planning & designing phase.
- CASE tools are also available to generate both online and offline user manuals.

## **Chapter 3: Defining and Understanding the Problem**

### **Software development cycle:**

1. Define
2. Plan
3. Build
4. Check
5. Modify

### **Defining the problem**

- **Identifying the problem**

- **Needs:**

- Need: A necessity, example: the solution MUST display to two decimal places.
- Preference: A want, example: a new colour for the GUI.
- Programmers focus on the needs first and then provide a range of preferences.
- Without defining the need it is difficult to develop a clear picture of the problem to be solved. The user must be contacted.

- Ways to determine and analyse customer needs:

- Observation (simple observation of current systems and its use can help programmers understand about how a new system can be implemented).
- Participant observation (first hand observation of how the system is run through participation in day to day operations).
- Surveys (surveying managers, IT personnel, users and clients).
- Interviews (allows participants to more freely express their opinions, however can be quite time consuming).
- Focus groups (similar to interview, but saves time and therefore money).
- Time management studies (determining how much time is actually spent on each specific function).
- Business analysis (examining different aspects of business activities and finding ways to improve them).
- Reading documentation (read the documentation for the current system to learn about its operation).
- Industrial trends (assess trends in technology usage and equipment design).

➤ Screen design

- User interface and interaction are very important to consider.
- Users will judge a program by its interface.
- Users will have very precise needs in the screen design, which include:
- Appropriate messages and icons.
- Messages must be CONCISE, state the problem/reason for being displayed CLEARLY in NON-TECHNICAL terms and provide positive feedback or help to the user.
- Messages should appear at the appropriate times in a NON-threatening manner.
- Messages should not blame the user.
- Icons should contain meaningful graphics and have immediate response times.
- Icons carry out a task or act as short cuts to open files.
- Icons are located in consistent locations and are NON-intrusive.
- Excellent icon management can make training much easier and faster for users.

- **Objectives:**

- Objective: An aim to specially achieve a task.
- Should be continually examined to ensure project work is meeting the objectives.
- Some objectives should include:
  - Meeting social and ethical standards
  - Using appropriate display methods
  - Providing clear input and output methods
  - Deciding on relevant data formats
  - Clear and understandable methods for navigating.

- **Boundaries:**

- Boundaries: limits / borders of a system.
- The software developer must take into account all possible boundaries such as:
  - The user's environment (the specifications of the user's system).
  - Computer configuration.
  - Peripheral devices (external hardware such as printers and digital cameras).
  - Other programs that may run along side the program that's being developed.
  - The operating systems capabilities.
- Carefully documenting the boundaries, allows the project team to have a realistic approach on what will work and what won't work.
- The users should also be aware of the capabilities of the program by providing easy to understand documentation.

## Determining the feasibility of the solution

### ▪ Is it worth solving?

- **Feasibility** determines whether the problem is worth solving.
- A preliminary investigation into whether a problem can be solved and whether it will be worth the time, effort and money involved to the organisation.
- Two perspectives:
- Can the problem be solved?
- If it can, is it worth proceeding with the solution?
- Unless these two questions are positive, it is useless continuing.
- Some points to consider when evaluating a system's worth:
- Would the existing system be able to perform the required tasks by aid of a quick fix or patch?
- Would the proposed system meet future needs?
- Have existing similar solutions been examined?
- How crucial is the new system to the organisation?

### ▪ Constraints

- Constraints: any restrictions or limitations placed on the development of the solution.
- Some constraints are listed below
- Systems analysts are responsible for researching the constraints of a new solution.

### ▪ Budgetary feasibility

- Whether the solution to the problem is affordable.
- Whether the benefit from the solution is worth the cost (cost-benefit analysis).
- Factor in any recurring costs of the final solution to maintain.

### ▪ Operational feasibility

- Whether a solution will be useable by the participants and users.
- Users must be able to effectively operate the system.
- Costs of training users must be factored into the cost/benefit study.

### ▪ Technical

- What hardware and software is currently being used?
- Can this hardware be used with the software to build a workable solution?
- How much will it cost to upgrade hardware?
- Is the hardware needed acquirable?

### ▪ Scheduling

- The time frame in which a solution must be developed.
- Is the solution achievable within the set time frame?

### ▪ Possible alternatives

- It may not be worth developing a new solution, if one already exists on the market.
- It could be more efficient to just buy the rights of an alternate program that does similar tasks and then just modify it for the users' needs.
- Other solutions will need to be investigated and recommended by a good analyst.

## Social and Ethical Considerations

- Peoples' lifestyles can change due to software. (People now pay bills, shop, bank and email others all online).
- This can have a bad effect on people without computer skills, however can aid people with disabilities (to fit in with society).
- The nature of work is also changed:
- Employees are cut from their jobs and replaced by machines. (Example: banks with ATMS).
- However new jobs are created, such as technicians, programmers and systems analysts.
- Misuse of software must also be considered and how it can be counter acted.

## Design Specifications

**White-box perspective:** The developers' view of what happens and how it happens (the process).

**Black-Box perspective:** what the users are concerns with, that is the input and expected output.

The **developer's perspective** includes looking at data types, algorithms, variables and the software design approach to be used.

### Data types

- A way of describing particular data terms.
- **Simple data types** – available in nearly all programming languages:
  - Integer (Positive or negative whole numbers only example: 5000 (stored as 2 bytes)).
  - Real/floating point (positive/negative numbers with decimal places, example: 36.777 – uses up more memory with more decimal places).
  - Character (any alphanumeric character, command, punctual or symbol-represented as one byte (8-bits) example: a, \$, / \* , ).
  - Boolean variable with one of two possible outputs (0/1, true/false, yes/no, etc.).
- **Structured data types** – combining simple data types into related groups:
  - String (an array of characters used to form words).
  - One-dimensional array (data of the same type accessed by index, example: Name(7) stores a list of string data between 0 -7, that is 8 items).
  - Multi-dimensional array (data of the same type accessed by co-ordinate- example: temperature (2,1) = 16).
  - Simple record (data of different types in a set of related fields).
  - Array of records (data of different types accessed by index/co-ordinate, example: Databases).
  - Files
    - **Sequential** (stored one after the other, useful for data read and written by programs, disadvantage: needs to read a large amount of files to find one location-takes longer time).
    - **Random access files:** Are significantly faster but use more memory and are harder to program.
- **Algorithms**
  - A series of procedural / finite steps that will result in the solution to a specific problem.
  - Can be described in a number of different ways – most commonly by pseudocode and flowcharts.
  - Three basic control structures govern the logic flow in an algorithm or program, they are:



Sequence (logical and sequential order of steps) – A -> B

Selection (a selection is needed between two or more choices) – A -> (B OR C)

Repetition (a loop is required) – A -> B -> A

- **Loop** (iteration): a set of instructions to be repeated.
- **Count:** a variable that can be used to store the number of marks calculated.
- **Pre-test repetition:** ALWAYS returns to the start of the loop after the body has been executed. (Loop is not executed if the criterion is met at the start, requires two input statements, just before the loop and in the body).
- Example: checking if your parachute is on BEFORE you jump out of a plane, if yes then jump out. If not, then run the loop until you have the parachute equipped.
- **Post-test repetition:** returns to the start of the loop ONLY if the criteria are met at the end of the loop. (Criteria is tested after the loop is executed, loop is always executed at least once, requires one input).
- Example: Keep cooking BBQ until meat tastes cooked. Does meat taste cooked? No. Keep cooking. Does meat taste cooked? Yes, stop cooking and serve food.

## ▪ Variables



Memory locations that are used by a program to store changing data.

Must be declared as a specific data type with a unique variable name.

## ▪ The user's perspective



The user is focussed on solving a particular task or problem and is not overly concerned about how the program goes about doing it.



The user knows what the program does, but does not know how – this is called the “black box” concept.



Appearance: program needs to have a user-friendly GUI (consistent appearance, icons, drop-down menus, short cuts, hot keys, simple navigation, and easy help wizards).



Functionality: programs need to carry out the required tasks and to minimise the user's actions as much as possible.



Scope: The design specifications of a program should clearly state the range and boundaries of the program. (Example: system requirements to run the program).

# Modelling

## Representing a system using diagrams

- **System:** a collecting of interacting parts working together to achieve a particular purpose.
- **Modelled:** A way a system can be represented in some way to show its structure.

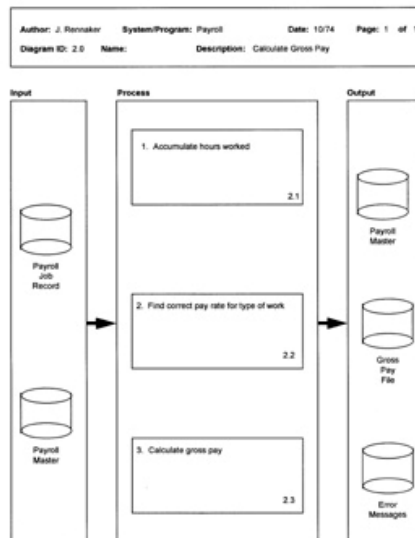
## Input Process Output (IPO) diagrams

- Used to describe the inputs, processes and outputs of data / information in a system.
- **IPO charts:** A table that outlines the inputs, processes and outputs.
- **IPO Diagrams:** graphically represents the inputs processes and outputs of a program AND shows the EXTERNAL CONNECTIONS of the software program to OTHER SYSTEMS that provide input and accept output from this program.
- **String concatenation (link):** the processing of joining / linking strings.

## IPO chart:

Input	Process	Output
2*2	2*2 = 4	4

## IPO diagram:



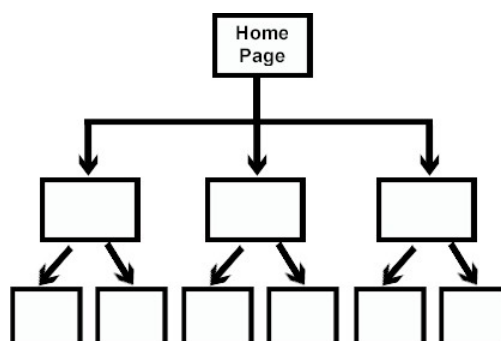
## Storyboards

- Linear and non-linear Used to show the screens in a project and the navigation between them.
- Storyboards give a general overview of an information system. They are used for designing the user interface for the system.
- They are a good method for organising content for web pages.
- Factors that ensure good user-friendly design:
  - The users' experience levels and needs
  - Consistency- allows user to apply their existing skills with confidence.
  - Legibility – The logical placement of words, the font and style chosen, the use of white space and careful use of colour and appropriate graphics.

## Storyboard structures:

**Hierarchical storyboard:** This design uses a top –down design and is useful for connecting the home page to all other pages. Also used when data fall into well-defined categories. This is the most frequently used storyboard format for web pages.

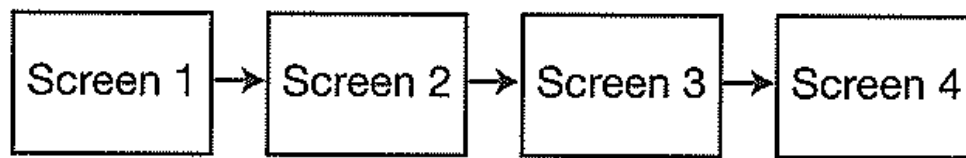
Example:



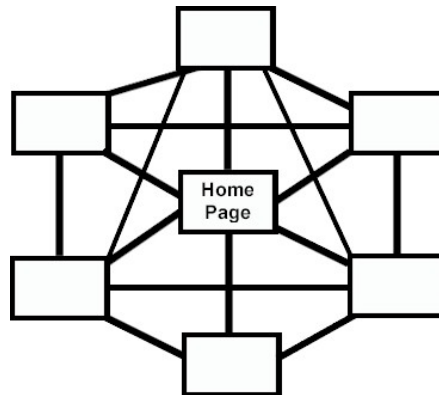


**Linear storyboard:** A simple sequential design that moves users through a predetermined path of web pages.

Example:

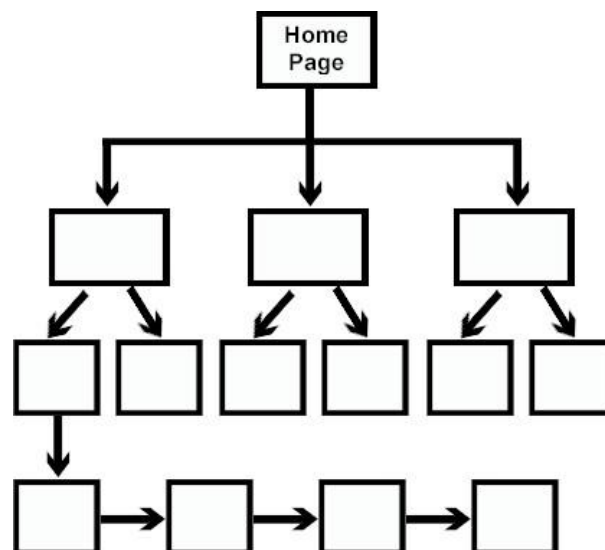


**Non-linear / Branch:** Allows users to connect from the homepage to a series of linked pages which then further connect to a number of other pages. Example:



**Hybrid / combination storyboard:** combines aspects of two or more common types of structures. For example, a predominately hierarchical structure could include a linear component.

Example:



## Data flow diagrams

- Describes the path data flows throughout a system, the data source and its destination.
- No attempt is made to indicate the timing of events.
- Used for both system analysis and system design.
- Many DFDs may have to be integrated/overlayed to see the full picture of the system.




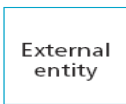
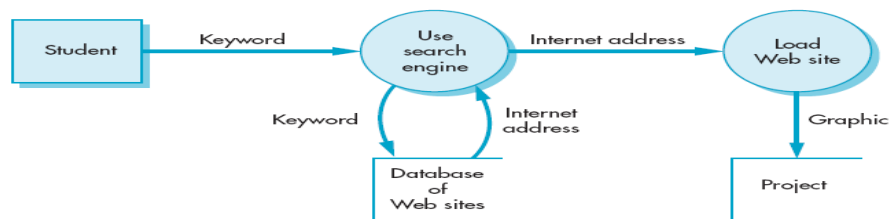
Symbol	Meaning
	Process: A circle is used to represent the processes or actions that transform inputs to outputs.
	Data flow: An arrow is used to represent the flow of data between the process, external entity and data store.
	Data store: An open rectangle represents the location where data is stored. It could be a filing cabinet, hard disk or DVD.
	External entity: A square or rectangle represents any person or organisation that sends data to or receives data from the system.

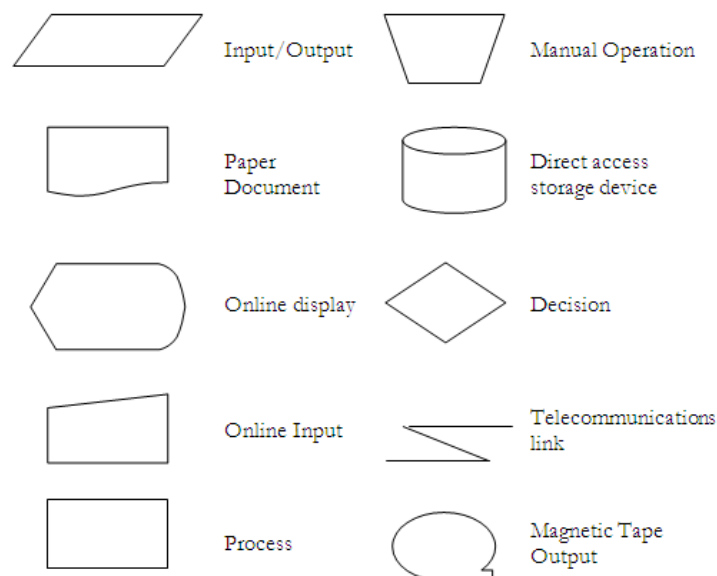
Table 1.3 Symbols used in a data flow diagram.

Example:



## System flowcharts

- A diagrammatic way of representing the logic AND flow of data.



(Note: Some symbols are old due to advances in technology- see book for more symbols-pg75)

### Screen designs

- Made so as to consider item placement, consistent design, order, grouping and nature screen elements, use colour, graphics, fonts.
- Give control to the user
- Allow the user to access other parts of the program
- Easy, simple (not too basic) and as user-friendly as possible.

### Consideration of use of a limited prototype

- **Prototyping:** the design of working models so that program solutions can be quickly tested and evaluated.
- Allows the user to give feedback about the program's design before it is actually made.
- Gives the user an idea of how the final program will look and function.
- Clarifies the users' requirements and needs with the software developer (better/refined understanding).
- CASE tools are used to rapidly generate a prototype.

## Communication Issues

### The need to empower the user

- It is important to keep the user involved in the software development cycle
- Listening is an important skill for developers
- Best for the developer to provide a range of possible alternatives with good functional features than for the user to come up with their own design.
- The end result must satisfy the users' needs. This will be decided by the users.

### The need to acknowledge the user's perspective

- The user is not interested in how the task is achieved, but more on the finished product.
- User doesn't need to know the technical details
- If a user comes up with an idea that will make the tasks more comfortable and productive then it is worth considering.
- The user needs all tasks that they specified to run as smooth as possible and most efficiently.

### Enabling and accepting feedback

- It is important for the software developer to continually use verbal and written contact with the user to clarify needs and the purpose of the program.
- Far cheaper to solve problems at this stage.
- Surveys, questionnaires, interviews, observation and prototypes are all ways of gathering feedback to help the developer clarify needs.

# Chapter 4: Planning and Design of Software Solutions

## Standard algorithms for searching and sorting

### Standard logic used in software solutions:

- **Finding maximum and minimum values in arrays**
  - **Parameters:** is a variable that is passed from the mainline to the sub program or vice versa.
  - Three parameters required:
    - An array of items
    - A start index
    - A stop index
  - In the program the actual location of the item is returned not the value itself. (Another form of coding displays the value of that location in the array).

### Finding maximum and minimum values Algorithm:

```
BEGIN FindMinMax
  Max = Array[0]
  Min = Array[0]
  Index = 1
  WHILE Index < Array.Length
    IF Array[Index] > Max THEN
      Max = Array[Index]
    ENDIF
    IF Array[Index] < Min THEN
      Min = Array[Index]
    ENDIF
    Increment Index
  ENDWHILE
  Print "The highest number is:" Max
  Print "The lowest number is:" Min
END FindMinMax
```

## Processing Strings

- **Strings:** are sequential series of characters that form a sentence. Strings can be:
  - Concatenated (joined together)
  - Extracted
  - Inserted
  - Deleted
- Before any process can be carried out on a string its length must be determined:

```
BEGIN StringLength (String)
  Index = 0
  WHILE String[Index] contains a character
    Increment Index
  ENDWHILE
  Length = Index
  RETURN Length
END StringLength
```

- String Concatenation:

```
BEGIN Concatenate(String1, String2)
  Index = 1
  String1Length = StringLength(String1)
  String2Length = StringLength(String2)
  NewString is an empty array
  WHILE Index <= String1Length
    NewString[Index] = String1[Index]
    Increment Index
  ENDWHILE
  Counter = 0
  WHILE Index <= String1Length + String2Length
    NewString[Index] = String2[Counter]
    Increment Index
    Increment Counter
  ENDWHILE
  RETURN NewString
END Concatenate
```

- Extracting:

- Equivalent to the **copy** command in a word processor
- Characters to be extracted are copied into temporary memory while leaving the original characters intact.
- The original string is not altered in this process, instead a new string is created using the original data of the first string.
- Note: parameters in brackets on the first line of an algorithm indicate that they are READ into the algorithm from the MAINPROGRAM.
- Algorithm:

```
BEGIN ExtractString(StartPosition, NoOfCharactersToExtract)
  Index = StartPosition
  Counter = 0
  Set NewString() to an empty array of characters
  WHILE Counter <= NoOfCharactersToExtract
    NewString[Counter] = OriginalString[Index]
    Increment Index
    Increment Counter
  ENDWHILE
  RETURN NewString
END ExtractString
```

- **Inserting:**

- Equivalent to the **paste** command
- A **new string** is **inserted into a specific place** in the **original string**.
- Characters to be inserted are placed into position in the existing string
- Algorithm:

```

BEGIN InsertString(NewString, PositionToInsert)
    {Calculate Length of original and new strings}
    OriginalStringLength = StringLength(OriginalString)
    NewStringLength = StringLength(NewString)
    TempStringLength = 0
    Index = 0
    ExtractOriginal(PositionToInsert, OriginalStringLength)
    InsertNewString(PositionToInsert, NewStringLength)
    ReplaceOriginal(Index, TempStringLength)
END InsertString

{Create a copy of the end of the original string}
BEGIN ExtractOriginal(PositionToInsert, OriginalStringLength)
    Index = PositionToInsert
    Counter = 0
    Set TempString() to an empty array of characters
    WHILE Index <= OriginalStringLength
        TempString[Counter] = OriginalString[Index]
        Increment Index
        Increment Counter
    ENDWHILE
    TempStringLength = Index
    RETURN TempStringLength
END ExtractOriginal

{Insert new string}
BEGIN InsertNewString(PositionToInsert, NewStringLength)
    Index = PositionToInsert
    Counter = 0
    WHILE Counter <= NewStringLength
        OriginalString[Index] = NewString[Counter]
        Increment Index
        Increment Counter
    ENDWHILE
    RETURN Index
END InsertNewString

{Attach end of original string back onto the string}
BEGIN ReplaceOriginal(Index, TempStringLength)
    Counter = 0
    WHILE Counter <= TempStringLength
        OriginalString[Index] = TempString[Counter]
        Increment Index
        Increment Counter
    ENDWHILE
END ReplaceOriginal

```

- **Deleting:**

- Equivalent to the **cut** command
- Characters to be removed are **taken out of the string completely**
- **Removing a section** of a string.
- Algorithm

```
BEGIN DeleteString(StartPosition, NoOfCharactersToDelete)
  Index = StartPosition
  OriginalStringLength = StringLength(OriginalString)
  WHILE Index <= OriginalStringLength - NoOfCharactersToDelete
    {Overwrites elements to be deleted}
    OriginalString[Index] = OriginalString[Index + NoOfCharactersToDelete]
    Increment Index
  ENDWHILE
  WHILE Index <= OriginalStringLength
    OriginalString[Index] = ""
    {Clear contents of these elements}
    Increment Index
  ENDWHILE
  {The original string will now be NoOfCharacters Shorter}
END DeleteString
```

### **File processing (including sentinel values)**

- Files are a collection of data stored on a secondary storage device.
- Files/ loops are often terminated by special symbols called sentinel values, often “eof” (end of file).
- Example: A database file is used to provide data for a program.
- Algorithm to READ a file:

```
BEGIN ReadFile
  Open File
  LineNo = 0
  While NOT END OF FILE
    ReadLineNo
    WriteLineNo
  ENDWHILE
  Close File
END ReadFile
```

## Linear search

- Simplest searching algorithm.
- Highly inefficient, especially with large arrays.
- Array can be sorted or unsorted.
- Compare search item with every other item in the array until either the item is found or the end of the list is reached.
- Data does NOT have to be sorted

## Linear Search Algorithm

```
BEGIN LinearSearch
  Set Upper to Array.Length
  Set Current to First Position
  Set FoundIt to FALSE
  READ TargetItem
  WHILE (Current <= Upper) AND (FoundIt = FALSE)
    IF Array[Current] = TargetItem THEN
      Set FoundIt to TRUE
      Set TargetItemPosition to Current
    ENDIF
    Increment Current
  ENDWHILE
  IF FoundIt = TRUE THEN
    Print "Target Item was found at:" TargetItemPosition
  ELSE
    Print "Target Item was not found"
  ENDIF
END LinearSearch
```

## Binary search

- Very efficient search even with large arrays as the array is already sorted.
- Reduces the number of searches by half each time a question is asked.
- Continually cuts the array in half until the item is found or it is determined the item is not in the list.
- Data (array) MUST be ORDERED into ascending or descending order.
- Halves numbers to be searched in each pass of the array.

```
BEGIN BinarySearch
  Set Upper to Array.Length
  Set Lower to First Position
  Set FoundIt to FALSE
  READ TargetItem
  WHILE (Lower < Upper) AND (FoundIt = FALSE)
    Set Mid to (Lower + Upper) / 2
    IF TargetItem = Array[Mid] THEN
      Set FoundIt to TRUE
      Set TargetItemPosition to Mid
    ELSE
      IF TargetItem < Array[Mid] THEN
        Set Upper to Mid - 1
      ELSE
        Set Lower to Mid + 1
      ENDIF
    ENDIF
  ENDWHILE
  IF FoundIt = TRUE THEN
    Print "Target Item was found at:" TargetItemPosition
  ELSE
    Print "Target Item was not found"
  ENDIF
END BinarySearch
```

- Bubble sort



- Each element is compared with the element next to it and if they are out of order they are swapped.
- Elements end up “bubbling” towards their correct location in the array.
- May be sort on an ascending or descending condition.
- A first pass involves comparing all data items to the next one, until the end of the array is reached.
- A second pass is executed to clarify all elements are in the correct order.
- If more swaps are needed, then another pass will take place and so on until all elements are sorted.
- Requires more passes than other types of sorts
- A flag is used to indicate a swap has taken place
- A temporary storage location is required to hold a data element every time a swap occurs, otherwise each data element would overwrite the other and both locations will end up holding the same data.
- Utilises the SWAP PROCESS

```

BEGIN BubbleSort
  Set End to Array.Length
  WHILE End > First Position
    Set Current to First Position
    WHILE Current < End
      IF Array[Current] > Array[Current + 1] THEN
        Swap(Array[], Current, Current + 1)
      ENDIF
      Increment Current
    ENDWHILE
    Decrement End
  ENDWHILE
END BubbleSort

BEGIN Swap(Array[], Position1, Position2)
  Temp = Array[Position1]
  Array[Position1] = Array[Position2]
  Array[Position2] = Temp
END Swap

```

- **Insertion sort**

- Scanning forward until an out of place item is found and then scan backward until you find its correct location and inserted it there.
- Firstly, One Item in the array is declared as SORTED
- Every other item is then compared to the SORTED section of the array and INSERTED (or left in the same location if it fulfils the condition) into its correct location.
- Can be sorted in an ascending or descending order
- Array(0) is sorted, Array (1) is the first item to be compared to Array(0)
- Count is a temporary storage location used to indicate the amount of spaces every other item in the array must move in case a data item may have to be moved in between two values of the sorted section (creates space for the new sorted item, by shifting every other item a specific amount of spaces).
- Utilises the INSERT and COUNT PROCESS

```

BEGIN InsertionSort
  Set First to First Position
  Set Last to Array.Length
  Set NextPosition to Last - 1
  WHILE NextPosition >= First
    NextItem = Array[NextPosition]
    Current = NextPosition
    WHILE (Current < Last) AND (NextItem > Array[current + 1])
      {Shuffle sorted part along}
      Increment Current
      Array[Current-1] = Array[Current]
    ENDWHILE
    Array[Current] = NextItem
    Decrement NextPosition
  ENDWHILE
END InsertionSort

```

- **Selection sort**

- Scanning through the array to find the largest unsorted item and then selecting it and swapping it with the item in its correct position.
- Firstly, the LARGEST value in the array is searched for and SELECTED. This LARGEST value now becomes declared as SORTED and placed in the last place
- The UNSORTED part of the array is then searched AGAIN to find the LARGEST value and is placed in the second last place
- A descending sort would be done by finding the smallest value instead of the largest value and placing it as the last location in the array.
- Can be used to order data in ascending and descending
- Array(0) is the LARGEST value declared as SORTED
- Array(1) is used to start the linear search to find the LARGEST value.
- Utilises the SWAP PROCESS

```

BEGIN SelectionSort
  Set EndUnsorted to Array.Length
  WHILE EndUnsorted > First Position
    Set Current to First Position
    Set Largest to Array[Current]
    Set PositionOfLargest to Current
    WHILE Current < EndUnsorted
      Increment Current
      IF Array[Current] > Largest THEN
        Largest = Array[Current]
        PositionOfLargest = Current
      ENDIF
    ENDWHILE
    Swap(Array[], PositionOfLargest, EndUnsorted)
    Decrement EndUnsorted
  ENDWHILE
END SelectionSort

```

## Custom-designed logic used in software solutions

- requirements to generate these, including:

Introduction:

- Custom design – design specifically addressing the particular solutions require by the current problem
- Structured/modular programming is used and very effective today for complex programs
- This structure of programming relies on breaking up a problem into a series of smaller problems and solving each 'little' problem individually to make it easier and more efficient in solving the overall problem. This is known as stepwise refinement or top-down design.

- identification of inputs, processes and outputs

A software developer must ask the following questions:

Input Data

- What type and source of data?
- What method to input data and computer instructions for data input handling?

Process Data

- Type of processing required
- What steps will be taken to process data?
- How will the computer handle the processing

Output Data

- Type of data output
- Method of data output
- How will the computer handle the data output?

- All these processes can be summaries in an IPO diagram

- representation as an algorithm

- Algorithms are a finite number of clear steps to solve a problem.
- Writing one helps understand the logic of a problem.
- Can be pseudocode (Keywords) or flowchart (diagram ) forms
- Control structures are used to show the following:
- Sequence
- Selection
- Binary selection
- Multiway selection
- Pretest iteration (guarded loop)
- Post-test iteration (unguarded loop)
- Fixed iteration
- Indentation helps display nested control structures algorithms clearly.
- Two types of subprograms:
- Procedures: miniature programs that carry out a small simple task such as swapping data between memory locations
- Functions: specialised subprograms that produce one single output
  - Abstraction: tool used whereby a large complex program is built around separate logical tasks known as subprograms, which are called from the mainline.

- definition of required data structures
  - Along with algorithms, programmers must also determine the most appropriate data structures for solving a problem.
  - A data dictionary may be used to determine all the information that is to be stored and determine a suitable data type.
  - Basic data structures include: characters, strings, Boolean, integer, floating point, literal constants
- use of data structures, including multi-dimensional arrays, arrays of records, files (sequential and relative/random)

#### *Arrays*

- Arrays are a method of handling data of the same data type.
- Linear (one-dimensional array): is a collection of data items of the same type treated as a single entity.
- Multidimensional arrays are a collection of data entities which are related to each other. (useful for storing data in a grid)
- Array of records is a collection of fields. Each field can contain different data types, each row is a record. E.g. a database.
- Files: blocks of data treated as single units, usually stored on a secondary storage medium.

#### *Access to data structures*

- *Sequential*: is when each data location in an array has to be read one after the other until the required data item can be found. Slow process and is used on magnetic tape storage mediums.
- *Random / direct*: is when a data location is accessed straight away. Only works if the data is in order.
- *Hashing*: is a process used for direct access, whereby the process sets the range of the storage locations. This now enables direct access to find the data location. E.g. the guess number game. Say the number is 20. Is number 25? No, its < 25, is the number 20. Yes, therefore correct. Much faster way to find data that is order.

- use of random numbers
  - A simple syntax allows for random number generation. E.g. Randomize in vb.
  - Random numbers are needed in simulations and games where each outcome will be different. E.g. a 6 sided dice has 6 different outcomes, the integers would be 0 – 5, with 1 added to each integer.
  - Random numbers are chosen from a range numbers
- thorough testing
  - Testing must be completed at each stage of the development cycle.
  - If a mistake is carried through the software development cycle, it becomes increasingly expensive to fix.
  - Structured walkthroughs (bench test), desk checking (dry test) and peer checking are used to test solutions.
    - **Structured walkthrough**: step-by step analysis of each procedure, checking for changed conditions.

- **Desk checking:** A manual way of checking the logic and correct outputs of a program with pen and paper. This requires test data to cover a range of values and critical values.
- **Peer checking:** A process whereby peers test the solution to find any errors that may have been missed by the initial programmer. Different alternatives may be suggested.

### **Design of test data**

When design test data it must take into account the:

- **Legal and expect values:** i.e. critical values (values which a condition is based on) and boundary values.
- **Legal but unexpected values:** i.e. data in an incorrect format (e.g. decimal) but accepted by the program's guidelines.
- **Illegal but expect values:** i.e. illegal data due to ignorance, typing errors or poor instructions. These errors should be trapped and not halt the execution of the program.

Once all these values have been considered a test data table can be drawn up and then a desk check can be carried out.

### *Types of testing*

- **Black-box testing:** each module is checked for inputs/outputs. (what the user is concerned with)
- **White-box testing:** each module is checked for correct processing. (more of what the programmer is involved with)

### **Standard modules (library routines) used in software solutions**

- Requirements for generating or subsequent use include:

– Identification of appropriate modules

Factors that include the appropriate use of modules include:

- Thorough documentation including intrinsic naming of variables and objects
- Standard control structures are utilised
- Choice of language suits the module
- Social and ethical issues related to the module
- Modules that contain standard AND redundant tasks will only waste memory

– consideration of local and global variables

- Local variables can only be used within their own module/subprogram.
- Global variables can be accessed throughout the whole program.
- Scope: refers to the amount of the program in which the variable can be used. (There are global and local variables).

– appropriate use of parameters (arguments)

- Parameters is another term for variable
- Parameters can be passed from the mainline to a subprogram and vice versa.

– appropriate testing using drivers

- Drivers are temporary code used to test the execution of a module when the module cannot function individually without a mainline.
- Drivers pass values to and from the subprogram.

– thorough documentation

- Documentation MUST be included with any project and aspect of a program.
- This leads to successful teamwork and new members can be hired and understand what's going on (esp. on long projects)
- Every programmer needs to understand the processes of a module if they are to successfully use or modify a module.

### **Customisation of existing software solutions**

A significant amount of time and money can be saved by customising a program rather than starting a new program from scratch. Provided that intellectual and property rights are correctly followed.

- identification of relevant products

- Applications: Many applications are designed for programmers to modify..
  - Customisation can create:
    - Macros-special commands that carry out a particular task.
    - Create conditions such as If .. Then ..End if.
    - Customised error messages
    - Use objects within an application such as buttons and check boxes
    - Customise forms for easy data entry.

- customisation

- The process of modifying an existing solution to meet specific user requirements.
- Processes include:
- Obtaining permission from original author, having a thorough knowledge of the application and testing the modified solution
- CASE tools are used to assist developers with the customisation of applications, such as rule checking, methodology and diagrams.

- cost effective

- Cost-benefit analysis needs to be carried out.
- If a package will take more time and effort to modify rather than build a new application from scratch it is not worth modifying.

## Documentation of the overall software solution

### • tools for representing a complex software solution include:

#### o **Algorithm descriptions**

- o Provide a detailed description of the logic carried out in a program in precise steps.
- o Do not indicate data manipulation.

#### o **System flowcharts**

- o Graphical means of representing the logic of a computer system.
- o Also demonstrate the source and destination of data which is used by the system.

#### o **Structure diagrams**

- o Method of representing the elements of a system in a hierarchical form.
- o Modules represented by boxes with data and controls past between modules.
- o Closed (black) circles are Boolean flags (true/ false)
- o Open (white) circles are data

#### o **Data flow diagrams**

- o Useful for tracking the movement of data through a system.
- o Graphically show the processing which occurs in a system and indicate where data is stored.

#### o **Data dictionary**

- o Helps the programmer understand the data that will be used in a particular program.
- o Includes the names, data types and descriptions of the data items used.

## Selection of language to be used

- **Sequential programming language:** screens follow one after the other and minimum user input is required. E.g. a find and replace function (user inputs keyword, program searches for that keyword and then displays the output of where that keyword is). Data items are accessed from outside the program.
- **Event-driven programming language:** data items are accessed from within the program and the user controls the order of processing, creates an interactive and dynamic pattern to follow. (Non-linear pattern). E.g. multimedia CD-ROM.

– driven by the user

– program logic

- Event-driven software requires user's actions to trigger an event / module
- Features menus, buttons icons etc
- Order of module execution is defined by the user.
- Polling: The process of continuously checking the status of events
- Event parsing: executes events that the user has instigated
- Used in computer games and graphical user interfaces.

• sequential approach

– defined by the programmer

- Sequential programming follows a set of steps to solve a given problem
- Utilises standard control structures such as Begin.. End, Do.. Until etc.
- Order of module execution is defined by the programmer.
- Used in data handling programs such as databases and word processes

• relevant language features

• hardware ramifications

• Graphical User Interface (GUI)

○ The type of programming language chosen will be decided on the following factors:

- The type of problem solution to be designed (complexity of the problem)
- Type of hardware (memory and processing power) that will execute the program
- Peripheral devices used (mouse and keyboards, printers etc)
- The existence of library routines available
- The designated target users (are they professionally qualified or just general everyday users)



# Chapter 5 – Implementation of Software Solutions

## Interface Design in Software Solutions

· The design of individual screens, including:

### o Identification of data required

- ❖ Individual screens must display data.
- ❖ Good design will enhance the ability of the user to view or enter data and reduce user errors.
- ❖ Such screen layout techniques include:
  - Appropriate use of graphics
  - Positioning of text on the screen
  - Alignment and justification
  - Appropriate use of upper and lower case
  - Colour of text and background – don't go overboard with colour and have excellent contrast such as black and white.
- o Appropriate use of White space
  - ❖ Event driven usually uses data within program
  - ❖ Sequential obtains data from outside the program.

GUIs must be:

- o Consistent: i.e. similar icons and colours should be used right throughout the GUI and should perform the same function no matter what version.
- o Familiarity: Design should cater for target users. Design should stick to previous conventions.
- o Forgiveness: Allows users to easily recover from their mistakes without crashing the program.
- o Robustness: Unusual procedures should be handled without problems
- o Feedback: Information should be provided back to the user, whenever necessary e.g. loading times.
- o Guidance: Should provide some help to users. E.g. a popup – grey box when the mouse hovers over an object to explain what the object does.

Appropriate messages

- o Should be used to prevent the user from making destructive decisions, such as “Do you want to save before quitting?” the default button chosen should be “Yes” this will prevent minor user mistakes from losing information.

The GUI should also retrace user actions

- o The “Undo” command can be used to retrace common mistakes done by the user.
- o Multiple levels of undo should be in place as users are usually not aware of mistakes straight away.

### o Current popular approaches

- ❖ Early computer programs relied heavily on command line interfaces.
- ❖ Graphic User Interfaces (GUIs) that use the following elements are now usually used:
  - o WIMP (Windows, icons, menus, pointer)
  - o Menus
  - o Windows
  - o Icons
  - o Scroll bars
  - o Radio Buttons
  - o Dialogue boxes

### o Design of help screens

- ❖ Help screens should be simple, non-threatening and guide the user in a positive manner.
- ❖ Context sensitive, procedural and conceptual help as well as tours and tutorials can be used.
  - o Context help screens – small assistants that become active when the user comes across a problem
  - o Browser help screens – using the WWW to obtain a huge amount of information about a specific problem.
  - o Procedural help screens – screens that sequentially show the steps the user needs to take in order to solve a problem
  - o Tutorials – teaching help, that show the user how to use different features of a program in more detail.
  - o Error messages – non-intrusive messages that provide help to overcome a particular problem at a particular point.

### o Audience identification

- ❖ Language should be clear and unambiguous.
- ❖ The program must use language appropriate for its target group. Language level (jargon)
- ❖ Age
- ❖ Level of expertise

### o Consistency in approach

- It is important to keep fonts, colours and placement consistent throughout the program so that it is easier to use to for the user.

## Language syntax required for software solutions

- use of BNF, EBNF and railroad diagrams to describe the syntax of new statements in the chosen language
  - o Sentence: a program instruction
  - o Syntax: the way a sentence can be constructed in a programming language.
  - o **Metalinguage**: a means of specifying the syntax of a programming language usually through text based BNF/EBNF or railroad diagrams. (basically information about the syntax of a language)

### BNF and EBNF

- **BNF: Backus Naur Form** (uses the “:=”(defined as)” to assign the meaning of a non-terminal symbol)
- **EBNF: Extended Backus Naur Form** (includes looping structures – parentheses “() – grouped elements, []- optional elements, {} – loop may not be carried out. and uses “=”(defined as))
  - o Terminals: symbols which represent themselves e.g. <output>display<type> (display is the terminal)
  - o Non-terminals: symbols that represent another structured part of the language. E.g.< output>
  - o Rules: set the patterns of symbols that may be used in a particular language.
  - o Refer to book – pg 143 for symbols

Clear and unambiguous description of the syntax of a programming language.

- o Examples:  
BNF

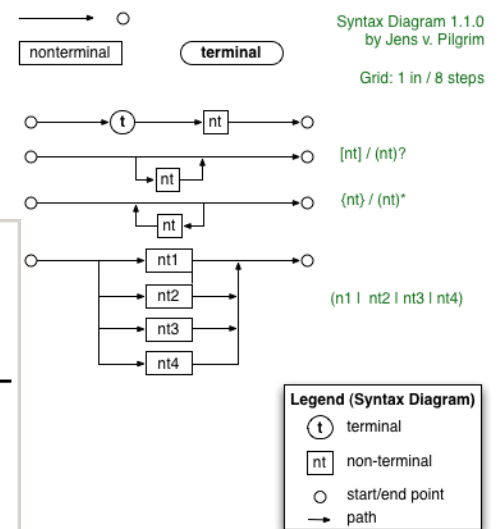
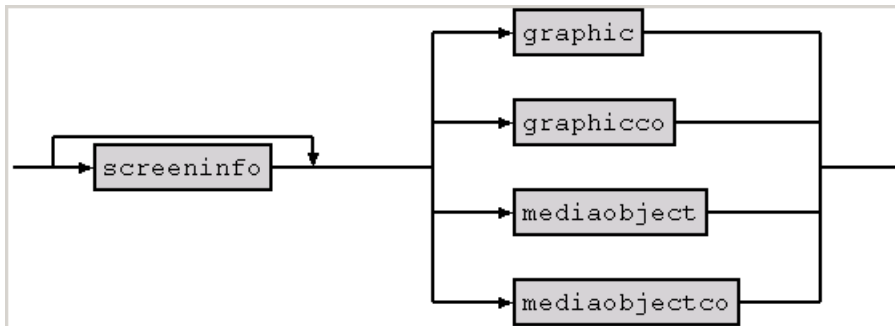
• **word** ::= <letter><word> | <letter>  
• **letter** ::= a | b | c | d | ... | y | z

EBNF

• **word** = <letter> {<letter>}  
• **letter** = a | b | c | d | ... | y | z

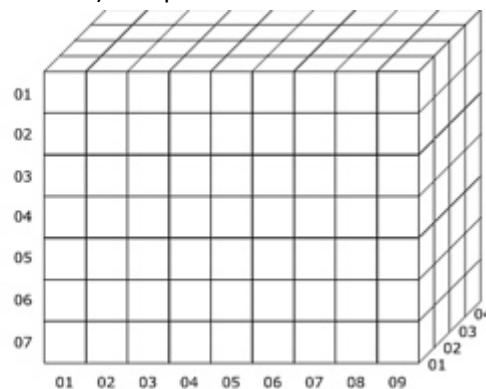
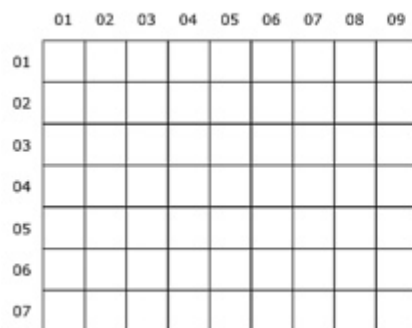
## Railroad diagrams

- o Rectangles are predefined elements
- o Circles are fixed elements (does not change)
- o Pathways go from left to right
  - o Examples:



## Commands using Arrays

- o Multidimensional arrays:
  - ❖ Table of values that unite different entities.
  - ❖ Adding dimensions to arrays requires more storage.
  - ❖ E.g. a chess board's coordinates (two-dimensional) or a prism like structure for a third-dimensional array.



### Example: Visual basic

'4 school, 12 subject

Dim AsstMarkArray(3, 1 to 4, 1 to 12)

'Assign average assessment mark to each school

AsstMarkArray (1,1) = 67

AsstMarkArray (2,1) = 72

AsstMarkArray (3,1) = 81

AsstMarkArray (4,1) = 71

### **Array of records**

- An array of records is a Grid of different data types stored as a collection of records. E.g. database.

	id_number	name	club	run_time
1 →	42	"Joe Bloggs "	"Belconnen "	168.5
2 →	86	"Wendy Brown "	"Woden "	149.0
3 →	23	"John Smith "	"Tuggeranong "	151.5
4 →	01	"Sally Black "	"Dickson "	178.6

### **Files**

- A file is a set of related data elements made up of many different data types. All these are stores as a block of data on the hard disk drive.
  - ❖ Sequential files: read and processes the data elements in a file in the order of which they are stored.
    - Programs must know the data structure inside that file to access the file
    - A sentinel value is used to let the program know that it is the end of the file and to stop reading at that point.
    - Visual basic: Input#filename, varlist
  - ❖ Random or relative file: files that have been stored in any order on a storage medium, wherever space is available.
    - The files have an organised structure so that the data within a file can be located directly.
    - Index becomes the address of a record in a file
    - Reclength – calculates the length of each record
    - FileNum – get the next available file number

## Random number generators

- Function creates a set of random numbers required by a program (visual basic code: randomize)
- Used to create a mix of numbers in any order.
- E.g. `GetNumbers = Int((6 *Rnd) + 1)` gets a random number between 1 and 7.

Visual Basic EBNF

- `RandomNumber = RND["("<number>")"]`

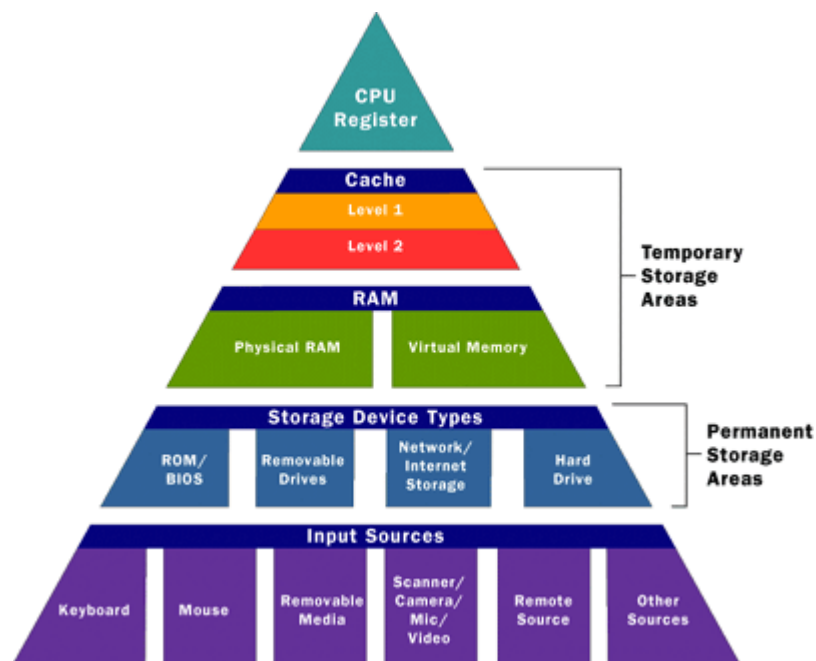
## The role of the CPU in the operation of software

### – Instruction format

- Instructions are made up of 32bit strings or 64bit strings of 1s and 0s (machine code)
- Made up of 3 parts
- **Opcode:** (1 byte) The **instruction**, telling the processor what to do. E.g. tells the CPU that a byte must be copied
- **First operand:** (2 bytes) is the **memory address** of where the data is to be stored.
- **Second operand:** (1 byte) is the **data**.
- E.g. the whole instruction could be written as: Copy the memory location specified by the byte of data represented by 10000001.

### – Use of registers and accumulators

- **Registers** are temporary storage holding areas located ON THE CPU.
- They **store address locations for instructions, results** of calculations and flags which show the result of a comparison.
- The **accumulator** is a special register that **holds the data items CURRENTLY being processed such as results**.



## – Use of program counter and fetch-execute cycle

- The program counter is a register that STORES the ADDRESS of the NEXT executable instruction.
- It is automatically incremented when an instruction is executed.

### FETCH-execute cycle

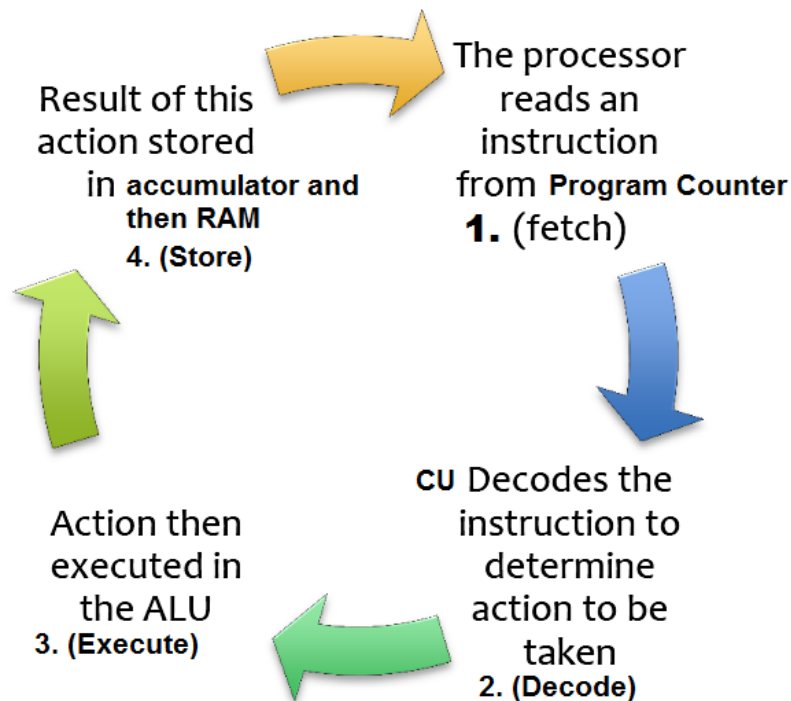
1. **FETCH:** the **control unit** fetches the instruction from **main memory (RAM)**. The control unit does this by using the **program counter (register)** to store the memory address for the NEXT instruction to be carried out. The program counter's job is to obtain and store the NEXT instruction for the CPU.
2. **DECODE:** The **opcode** is separated from the data addresses (operands) by the **control unit (CU)**. The control unit uses this opcode to figure out what type of instruction has to be carried out. The opcode is then loaded into the instruction register, and the data addresses copied into the address register.
3. **EXECUTE:** the opcode in the instruction register is sent to and then executed by the **Arithmetic logic unit (ALU)**
4. **STORE:** While the instruction is being executed, the results of any data being processed are stored in the **accumulator**. The results from the accumulator are then stored in registers or main memory (RAM). The CPU is now ready for the next instruction.

### What happens next?

- The fetch-execute cycle then starts again. The length of the first instruction is now known. The CPU then fetches the next instruction of the same length in sequential order, the program counter increments and the Fetch-execute cycle continues.

### Speed factors

- The fetch-execute cycle is regulated by the **clock speed** of the CPU. E.g. 3GHz means the clock generates 3000 million electrical pulses per second. The '**computer word**' size refers to the number of bits that can be processed at one time and is another factor affecting the speed of a computer. E.g. 64 bits means 64 bits of data are processed in one cycle as compared with a 32 bit machine.



## – Addresses of called routines

- Commands to be executed are stored with the operation code in the computer's RAM.
- Each memory address is accessed sequentially unless the instruction is to jump to another part of the program.
- The **stack** is a special register that keeps record of where the CPU is up to in the current operation it is carrying out.
- The stack is important as it keeps track/a list of what process the CPU is up to. If the CPU needs to re-check the memory address of the current item being processed, it can look into the stack and retrieve the current address, a program counter would not do this, as it has already dumped the current address and loaded the NEXT instruction address.

## – linking, including use of DLL's

- *Subprogram*: special functions called from the mainline to perform ONE specific task.

### Three types of subprograms

1. Individually made by a programmer to be used within one application such as a game. A high score subprogram is used to calculate and store a new high score.
  2. Subprograms included within a programming language development package. These subprograms form part of a library and can be accessed by programmers to perform simple tasks quicker. The programmer does not need to re-write a subprogram, instead the programmer just calls it from the development library. E.g. to randomise numbers, a function can be called.
  3. Subprograms built in to the operating system. The programmer is able to use subprograms from the operating system. This creates universal features such as the save and print commands which we see in almost every word processing application. Even the user interface is using re-used subprograms to create the same interface such as the minimise, maximise and close buttons on every application window.
- A **linker** is the process by which subprograms are joined to the mainline of the program. A Linker handles the call and return processes. It calls the subprogram, gives it control then returns control to the mainline after it has executed.
  - **DLLs (dynamic link libraries)** are common subprograms used to reduce the need for multiple subprograms. DLLs can be called from many different applications and used by the application.
  - If a new program is installed with a newer version of the DLL then it overwrites the old version. All other programs then have to use this newer version of the DLL.

## Translation methods in software solutions

Translation is the process whereby a translator converts source code (human readable code) into machine/object code (binary operational code).

Different methods include:

### – Compilation

- All code is translated before the program is executed. E.g. a compiled .exe
- ✓ More quickly executed as the computer has stored the object code and has a direct understand of the code
- ✓ Compiled object code smaller than it's original source code
- × The code is harder to modify
- × Errors during runtime only appear when the program has been compiled
- × Any change of code to the program requires the entire program to be re-compiled.
- × Testing can be more complex as each change requires recompiling and it is harder to follow the original algorithm.

### - Incremental compilation

- Program modules are compiled and stored as they are written and then the whole code is executed.
  - ✓ Commonly executed modules are compiled and stored
  - ✓ The mainline of the program is interpreted and the modules run as compiled code (easier to check errors within the mainline.
  - ✓ Only the code that has changed needs to be recompiled.
  - × An interpreter is used for the mainline, which slows down runtime.
  - × Interpreted sections of code can still be easily stolen by others.

### - Interpretation

- Source code is translated line by line immediately.
- ✓ Errors can be quickly identified and corrected
- ✓ The program can be tested for syntax and runtime errors as each line is executed and the line with an error ONLY gets re-translated not the entire program.
- ✓ Easier to use flags and debugging output statements
- × Program execution is slower than compiled code as the code needs to be translated each time the program is executed
- × Much easier for code to be stolen and used by others
- × Interpreted code takes up more space than compiled code.
- × An interpreter must be installed on the user's machine.



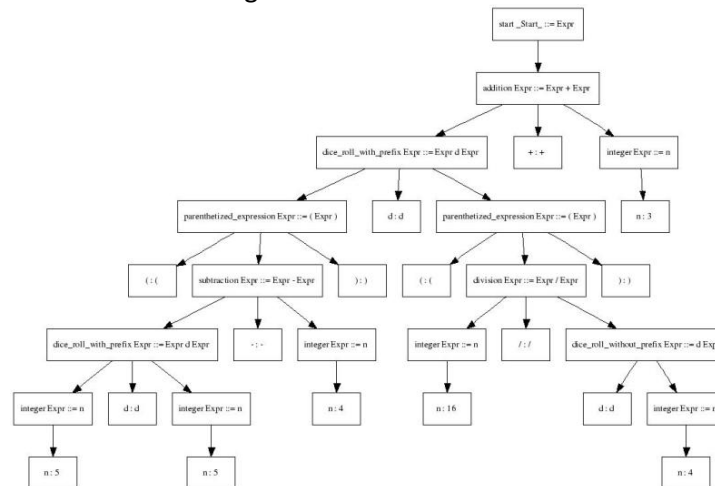
## The translation process

**Lexical analysis** is the process of reading source code one character at a time.

- Useless characters such as spaces, indentation and comments are discarded.
- Other characters are assigned a token according to the type of character they are.
- This is determined from the syntax elements, constants and operators.
- Programmer-created characters such as variables are represented by tokens.
- A symbol table or token dictionary is used to store tokens and their attributes.
- This becomes a look up table that can be accessed by the interpreter or compiler when a program is executed.

**Syntactical analysis** is a method of arranging the tokens in a way that can be understood by the processor

- Parse tree: the process of analysing a section of code is carried out. It creates a hierarchy so that code is always interpreted in exactly the same way.
- Token: Characters that are assigned.
- A parse tree shows what each token is assigned to.



- **Type checking** is a process whereby data types are identified and the information passed to the translator and decisions are made as to whether the processes involved are allowable operations with those data types.
- E.g. a mathematical operation would need to have its variables and operators checked by consulting a symbol table.

## Code creation

- Once a line of code has been found to be legal the tokens are converted into object code by the process of **traversing**.

## Translation Tools

### Linker

- The **linker**: Links code to other processes. E.g. calls to the operating system or links other modules to more modules to form a single object file.
- It will also link subprograms to the mainline of a program
- The linker places a link in the correct position in the object code, once the link is read, it then passes control to the OS or module, after the subprogram has finished, the object code resumes where the link left off.

## Loader

- The **loader** is the component of the translator that calls a load module (a small program) to set the memory locations to run from main memory.
- The loader allocates memory locations to hold the code while it is being executed so that the memory addresses do not interfere with other operations taking place within the computer system.

## Optimiser

- An optimiser is used to reduce redundant code in a program and improve runtime
- An interpreter cannot remove unnecessary code as it executes code line by line.
- However, a compiler holds all machine code for a program in main memory and can use an optimiser to reorganise code and remove redundant code.
- This speeds up the execution of a program and may reduce the time spent compiling the program.
- Some compilers only optimise code that is likely to be repeated such as functions and loops.

## Program development techniques in software solutions

- structured approach to a complex solution, including:
  - one logical task per subroutine
    - Each task...
    - Can be easily tested before joining to the mainline of the program.
    - Isolated and reused as a library routine in another program.
    - Replaced when the task needs to be updated
    - Easily understood by the maintenance programmer.
  - The components of the subroutine do not rely on other external components to work carry out the task. This is done using local variables.
  - E.g. reading an array, sorting it and printing it can all be done with 3 separate subroutine.
- stubs
  - Stubs: are dummy modules written to substitute for those sections of the program that require output to be returned to the mainline.
  - They are used to test and debug a program.
  - They allow the programmer to determine that the **procedure was called at the correct time**.
  - E.g. a stub may include expected values to display on a screen when called. This will be much easier than actually programming a fully functional subroutine just for a test.
  - A stub is **always simpler** than a procedure it represents.
  - Allows the main program to be **debugged and tested before modules are added**.
  - Allows **more control of the conditions being tested** rather than full programming code for all modules.
- flags
  - Boolean variables used to record whether certain events have taken place in a program.
  - Used to gather information within a loop that can be used after the loop has finished or as a means of terminating the loop.
  - Start value usually set to false, then when condition is met, it is set to true.
  - Example: EOF (End of file): Where a program retrieves data from a file using a loop. The EOF flag indicates when the end of the file is reached.

- isolation of errors
  - Always best to initially avoid errors by following excellent design specifications in the Algorithms.
  - Errors can be reduced by:
    - Avoiding goto statements, they are often unreliable and difficult to trace.
    - Floating point numbers can be imprecise and never ending, this could cause runtime errors or large amounts of memory and processor usage. The mantissa can take up 2 bytes of data – 32768 to +32767. The exponent is the integer; the mantissa is the fractional part. It is best using fixed point numbers set to a specific amount of decimal places.
    - Recursion: Process of having nested loops that call other loops from inside each level of the looping structure. Having deep nested loop structures is not recommended as it will be difficult to find and solve errors.
    - **Dynamic memory allocation** refers to the use of open arrays storing data at runtime rather than allocating memory addresses when the code is compiled. This can also take up an undesirable amount of memory as each array slot is progressively filled and often lead to a crash when memory runs out.
- debugging output statements
  - **Debugging output statement** is a statement inserted as a line of code to **display the value of variables at a particular point** in the execution of a program.
  - Some high-level programming languages do this automatically, others don't and need to be manually coded to include these statements as lines of code.
  - These statements are really good for testing a program where test data is not possible.
  - E.g. a program design to read large amounts of data files sequentially and to insert data before from various lines of a the file into specific sections of the program. Debugging output statements can be used to check that each piece of data is being loaded correctly from the file.
  - When the correctness of a program has been achieved, debugging output statements are removed as they are no longer necessary and will disrupt the program's execution.
- elegance of solution
  - Elegance is also referred to as software build quality.
  - It means a solution should successfully implement the specifications in a maintainable manner (programmed in sub modules, arrays and loops).
  - The program's resources and code need to be as compact/simple as possible while delivering all of its outcomes to the user.
  - This allows existing programs to be easily modified to allow for updated and new functions.
- writing for subsequent maintenance
  - Maintenance is expensive and inevitable
  - Code needs to be written with future maintenance in mind to reduce future costs and makes for a good quality built program.
  - **Modularisation, good internal documentation** (comments and intrinsic names) and **process / technical documentation** that cover the program's design and testing will ensure future maintenance will be easy and cost effective.

- the process of detecting and correcting errors, including:
  - Detecting errors is done by **testing**
  - Correcting errors is done by **debugging**
- syntax errors
  - **Syntax errors** are errors in the written code, they break the rules of the programming language in some way.
  - They consist of typing/spelling mistakes and incorrect punctuation.
  - Most third or later generations will provide CASE tools to automatically pick up on syntax errors and provide explanations so they can be corrected by the programmer.
- logic errors
  - Logic errors are errors in the design or the code implementation of a program that result in **incorrect output**.
  - E.g.  $\text{Average} = \text{Integer1} + \text{integer2} / 2$  would result in integer1 being added to interger2 divided by 2. The correct logic will be produced using  $\text{Average} = (\text{Integer1} + \text{integer2}) / 2$
  - Logic errors are the **most difficult errors** to detect as they are **syntactically correct** and **do not cause a system crash** but instead produce incorrect outputs.
  - It is essential that a programmer checks all of their design algorithms to avoid these errors before coding. The programmer also must make sure they are coding the right logic, as simple logical mistakes can occur.
  - Peer checking and desk checking are often resulted to remove logic errors and improve the elegance of a program.
- peer checking
  - Peer checking is having somebody else of comparable standard to the programmer check the code or algorithm for errors in logic.
  - It is important that the people checking the program have the same or higher skill level in that given programming language as the original programmer.
  - Another person will often pick up logical errors overlooked by the original programmer who may be 'too familiar' with the content and structure.
- desk checking
  - Desk checking is a manual check usually by pen and paper of the solution's expected and actual outputs using a dynamic range of test data.
  - The purpose is to find and correct any logical errors.
  - It takes place at the algorithm stage before coding.
  - Involves tracing the values of variables and writing down their output after 'mentally executing' each line of the algorithm.
  - Test data should contain the expected and legal data, unexpected but legal data and the expected but illegal data and check all possible paths data will follow.
  - The same tests are often used during the implementation stage to check the coded solution.

- use of expected output
  - The expected output is what the program is expected to show after it has been executed.
  - A range of test data is used as input data to test all expected outputs
  - **Expected** outputs are then checked against their respective **actual** output. If they do not meet their actual output, the program needs to be checked for logic errors.
  
- run time errors, including:
  - Runtime errors **occur while the program is running**, and **will lead to a crash**.
  - Sometimes they are caused by wrong syntax, other times they are caused by the inability of the computer to perform the intended task.
  - Most high-level language source code editors in integrated development environments (IDE) will pause the execution of the program and set it into a break mode. The editors will indicate the line in which the error occurred.
  
- arithmetic overflow
  - Caused by calculations which result in extensively long outputs which are too **complex for the arithmetic process** or **are too large to store for the current allocated memory**
  - These can happen when lots of calculations are made on the same set of data or an incorrect data type was used.
  - Two main types:
  - **Range errors:** occur when the number of bits allocated to store a number is not enough for the size of the number.
  - **Floating point errors:** caused when the number of decimals is too large for the memory locations allocated which causes the digits to be shortened leading to calculation errors and a crash.
  
- division by zero
  - Occurs when a number is divided by zero.
  - E.g. Total = 1 / 0
  - The processor is unable to perform this task and will result in a runtime crash.
  
- accessing inappropriate memory locations
  - **'File not found' error:** This will occur when a program is requested to read an external file, and that file is not at the specified location. This will result in a run-time error.
  - Run-time errors will also occur when a range value is set for an array and the loaded array is larger or smaller than the value.
  - E.g. Array(59) and the value is set to 60. This will produce a run-time error. Or the range value is set, 0 to 58, the 59<sup>th</sup> array item will never be initialised which will cause run-time errors.
  - Some programming language editors include error-trapping routines to **identify the error and provide feedback** to the programmer.
  - Programmers usually include code to handle some types of errors, such as division by 0 and overflow errors. This is known as **exception handling** and deals with unexpected problems.

- the use of software debugging tools, including:
  - use of breakpoints
    - **Breakpoints:** certain points in lines of code that can be selected by the programmer to break (stop) during the execution, to assist with removing errors.
    - The programmer can then make a decision on what steps to take next or study what has occurred at that particular point in the program.
    - Breakpoints are good for testing variables at given points of a program to ensure they are doing what is expected.
  - resetting variable contents
    - Programmers may decide to reset a variable contents to see it's effect of such changes on the program, to find logic and run-time errors.
  - program traces
    - **Tracing:** is the process of checking the **flow of control** in a program.
    - Editors are used to display which line of code is being executed and provides a history of the variable values. The programmer is then left to determine what is happening.
  - single line stepping
    - **Stepping:** the process of executing a program one statement at a time.
    - After each step, the values of variables or expressions can be displayed and minor changes made.
    - Subprograms can be executed as a whole or by each line.
    - This is usually done sequentially and may be started by a breakpoint. The control of the program can be traced as the programmer sees when the control is passed to and from subprograms.

## Documentation of a software solution

- forms of documentation, including:

- process diary
  - A log kept by the development team of all the steps taken in the development process.
  - It is dated and annotated regularly to avoid loss of important steps and tests.
- user documentation
  - This is either paper or online external documentation that is user-friendly so an everyday user can understand the information presented, and is provided to the user for instructions on how to use the software effectively.
  - The manual needs to be designed with accordance to the level of expertise the target user has and needs to cater for the experience and inexperienced.
  - Should state:
    - How to get started
    - How users may use common functions
    - How to fix mistakes
    - How to recover work that may have been lost
    - How to fix common problems also known as the troubleshooting guide.
  - A program installation guide is also included within user documentation and should include:
    - Details on how to install the program and descriptions of the mediums supplied and manuals
    - Minimum hardware specifications
    - A list of files included on the installation media
    - Details of any known conflicts with any other software
  - It is very useful to use **screen dumps** (screen shots) in the user manual as the users can effectively compare what stage they are up to and confirm that they are on the right track.
  - There are functional screen shots which show the controls of a keyboard / mouse.
  - There are also screen capture screen shots to show the GUI.
  - Online help: Provides updated solutions to various problems, providing the user with information from the internet. – FAQs - frequently asked questions are also provided to aid the user.
  - Help browsers: Provides a database of useful terms and explanations and provides help based on the user's current area of the program.
  - Help wizards: step-by-step guides to performing a task. They take the user through steps on the screen.
  - Tutorials: guides the user through the steps of working with features of a program
- self-documentation of the code
  - Self-documentation: internal documentation included within the coded solution to help programmers refer to what each segment of code does. There are two types:
  - **Intrinsic naming**: is naming variables and objects with meaningful names so that it is easy to identify and instantly understand what they are required for. E.g. Answer will hold the output.
  - **Comments** in the code to assist with the understanding of what each code segment does. They make very clear what the commented code will do so that other programmers and maintainers can alter and update the code as time goes on.

- technical documentation, including source code, algorithms, data dictionary and systems documentation
  - **Technical documentation** is used by programmers and analyst to allow for future maintenance and assist with error correction in the coded solution. There are four components:
    - **Source code:** is the original code in a programming language. It can only be interpreted by programmers experienced in its given programming language. A hard copy of the source coded is included in the technical documentation as it is often easier to read than as a coded solution inside the IDE.
    - **Algorithms:** Are design structures usually in pseudocode or flowcharts which outline the entire program structure. They give programmers guidelines to build a solution to solve a problem so they understand clearly what they have to code.
    - **Data dictionaries:** Provide information about all data that will be used within the program. They are design in a table format and list all variables with their name, data type, description, item length and their subprogram referrals.
    - **System documentation:** documentation designed in the defining stage that provides system flowcharts and dataflow diagrams to describe the flow of data. Also a specifications report is created to state the testing of the program such as using different OSes, clock speeds of CPUs, memory and running common other programs in the background.
- documentation for subsequent maintenance of the code
  - The more thorough the documentation provided with a program, the simpler any future modification and maintenance will be.
- use of application software to assist in the documentation process
- use of CASE tools
  - CASE tool: computer-aided software engineering is any computer-based tool that helps support aspects of the software development cycle.
  - They are used to help plan projects and manage parts of the software cycle such as creating Gantt charts for time scheduling or drawing dataflow diagrams or flowcharts for technical documentation.
  - They take a great amount of training time to use effectively.
  - It is important that the programmer selects the right tools for a particular project based on the data input, function and expected output.
  - They are used to improve efficiency and accuracy of the software development cycle.

### Hardware environment to enable implementation of the software solution

- hardware requirements
  - minimum configuration
    - Minimum hardware requirements are the hardware needed for software to run efficiently.
    - Software requirements may also be needed such as the correct OS and updates.
    - Software may run on lower requirements, but the tasks will not perform within reasonable time limits, resulting in user frustration.
    - It is up to the developer to state reasonable minimum specifications such as enough RAM, and processing speeds for their program to run efficiently without problems.
    - The user must comply with these minimum hardware specifications to use the software effectively, or else upgrade their current system to use the software.



- possible additional hardware
  - Software developers should never assume that user's will have more than the basic hardware requirements.
  - E.g. Standard home PCs will most likely be less powerful than the computers the software developer use. These must be taken into consideration to deliver a reliable and well built program.
  - A software developer must make sure their user documentation states any additional hardware required to take full advantage of a particular software product, such as a printer for printing documents off a word editor.
  
- appropriate drivers or extensions
  - **Drivers:** refers to additional small programs required to run or execute tasks carried out by peripheral devices.
  - **Extensions:** files that usually reside with the OS and assist in the execution of some programs. Dynamic link libraries (DLLs) are extensions.
  - These add-ons are necessary for a program to run successfully and provide all of its features.
  - If a program is required to have any of these add-ons, they should be provided by the software developer to the user or have instructions on how to download and install them.

## Emerging technologies

- Hardware
  - Hardware is always advancing, becoming smaller with faster processing speeds and huge amounts of memory.
  - We now have as of 2011, Quad core CPUs such as Intel's core i7s with dual threading capabilities. We now have around 4 GB – 8 GB DDR3 RAM as the average and dedicated graphic cards with DDR3/5 1 GB memory are becoming much more common. USB 3.0.
  - Many digital devices have replace analog equipment and will continue to do so, such as digital T.V, this allows faster processing rates as signals do not have to be converted.
  - 25 GB / 50 GB DL Blu ray, is slowly replacing 9 GB DL DVDs due to its higher storage capacity needed for large amounts of high definition content. 700 MB CDs are still used for audio storage as they are cheap and audio doesn't take as much space.
  - New generation of software requires touch screens and audio recognition devices such as microphones to do their tasks.
  - *Quantum Computers* – Will vastly increase the speed at which computers can operate.
  - *Nanotechnology* – Will allow for the sizes of computers to decrease so they can be embedded in more places.
  - *Holographic Storage* – Expand storage devices to a three dimensional level to increase the amount of data that can be stored greatly.
  
- Software
  - Software is becoming more graphical, taking up more and more space and requiring more processing power for graphics, animation and videos.

- Software needs to be regularly updated to keep up with the latest OSs to ensure user's will buy and use your software.
  - Some users may see software that is overloaded with unnecessary features as bloat ware when a more elegant solution is available.
  - Software needs to be designed with elegance in mind and future maintenance.
  - Programmers need to build a program so that it can become **backward compatible** with previous OSs and older hardware, as well as being able to run on the latest technology. An alternative is to build two separate versions. Such as 32 bit and 64 bit versions where a single solution is not possible.
  - *Java* – Allows for the development of cross-platform applications. Becoming increasingly popular as a programming language.
- their effect on:
    - human environment
      - With the rapid advances in hardware, a lot of **hardware is never used to its full potential** and is often thrown away. This causes much of the plastic components not to be recycled and **toxic chemicals** such as lead and mercury can seep into the earth at the landfill.
      - Sand is wasted as we mine it, to make silicon chips.
      - **Equity problems** such as the **wealthier people** being able to stay up to date on the latest hardware where as **poorer people** are stuck with older hardware and some with no technology at all.
      - **Technology aids human society** in everyday life, such as communicating news, directing air traffic and providing medical diagnosis. Technology can only become as good as the humans who make and use them. Tasks can be automated and simplified. This can be good and bad, as jobs can be lost. Sensitive data is also more freely prone to theft due its digital environment.
    - development process
      - Development process is always moving rapidly to produce new hardware to keep up with software demands. Sometimes there is too much powerful hardware and software needs to still catch up with the latest hardware trends. E.g. developers are still trying to understand how to maximise their programs to run efficiently on more than one CPU at the same time.
      - Software is becoming **increasingly cross-platform**, e.g. from phones, to PCs, GPS, gaming consoles.

## Chapter 6 – Testing and Evaluation of Software Solutions

### Testing the software solution

- comparison of the solution with the original design specifications
  - A solution must meet all of its design specifications and be free from errors. Testing is a continuous process.
  - **Design specifications:** specific lists of all criteria that must be met in the end product.
  - The interface, input and output, procedures must all meet the design.
  - **Live testing:** using real data to test the program in an environment in which it will be used.
  - These will ensure the software is **relevant, reliable and of good quality**.
  - Testing does not prove the absence of bugs, only their presence.
  - Testing finds bugs, good software engineering prevents them.
  - Errors and solutions should be well documented.
  - Testing is done incrementally on large programs, that is, each module is tested individually before it is implemented with the mainline. Smaller programs can be tested as one whole.
  - Fixes to any bugs found should be done before release, or updated if found out after release.
- generating relevant test data for complex solutions
  - There are often way too many variables to test, they would take too much time and money to test all of them. So a range of techniques are used to cover a wide range of conditions.
  - **Black-Box or functional testing:** checks the inputs of each module against the expected and the actual outputs. The **user is more concerned** with black box testing as it involves what they see, that is, the input and outputs. Identifies logical and run-time overflow errors. Types of Black-Box testing:
    - **Range checking:** The upper and lower values are checked
    - **Critical value checking:** the conditional values are checked
    - **Equivalence partitioning:** A range of different data types are checked
  - **White-box or structural testing:** checks the procedures or processes within each module to determine their correctness. It is what the **programmer is more concerned** with as it identifies bugs in the coded solution. Types of white-box testing:
    - **Statement coverage testing:** test data is chosen to test each statement in a module.
    - **Decision condition testing:** Test data is selected to test each decision within a module.

- levels of testing
  - Testing is done progressively; it starts at the lowest level, i.e. unit testing, than works towards program testing and finally system testing where it is testing in different environments. Errors can be removed from the bottom up.
  - unit or module testing
    - Tests each module separately and makes sure that each module is performing their task successfully.
    - A **driver program** may be needed to provide inputs and outputs to and from a module, as the mainline may not be available at this time.
    - Within complex programs, module testing will involve integrating related modules and testing them as a subsystem.
    - Black and white box testing is used.
  - program testing
    - Ensures that the modules work together and that the mainline of the program performs correctly.
    - Concentrates on the interfaces and the relationship of each module to the mainline.
    - Uses white and black box testing to ensure the program performs the **overall task(s)** set in the design specifications.
    - Can be done in two different ways:
      - **Bottom-up testing:** test and corrects the lowest level modules first and works up to the higher level modules. Relies extensively on driver programs to test modules. Easier to detect problems at an earlier stage. The main program is tested last once all modules are in place.
      - **Top-down testing:** starts with testing the main program first and uses stubs for some incomplete modules as it links each module to the main line. Any problems are most likely to be caused by the most recent addition, therefore problems can be easily located if top-down testing is constantly used.
    - How to choose: Matters on preference and commonsense. E.g. A library of routines may already be checked and free from errors, so it would be more reasonable to use a top-down approach. A bottom-up approach would be more useful if a program and all of its modules are built from scratch.
  - system testing
    - Uses black box testing to check that the program can run outside the integrated development environment in a range of other environments.
    - E.g. a program may run excellently on one set of hardware and software, yet if you try it on a different set of hardware and software configurations, problems that were never present on the development machine are now an issue on several machines and must be fixed.
    - System testing is often done by testers outside of the development team.
    - For custom software, the program is tested in the environment in which it will be used.
    - **Acceptance testing (for custom software):** using potential users of a program to test custom software that is only designed to run for one particular set of hardware and software specifications. This allows a program to optimise for one particular system. E.g. PlayStation 3 console is only one

system and exclusive game developers can run acceptance testing with users to optimise their game for the PS3 system only.

- **Alpha testing (more general/commercial software):** is the first phase of testing done under controlled conditions with selected participants. Once all initial alpha bugs found have been fixed beta testing takes place.
- **Beta testing (more general/commercial software):** is the second phase of testing and involves volunteers to test a program on a wide range of specified hardware and software systems. E.g. Windows 7 Beta, could be downloaded freely and tested by a wide range of user's systems. These provide an enormous amount of feedback very quickly to the developers and final bugs of the beta stage can be fixed up before the release candidate is released.
- the use of live test data to test the complete solution:
  - **Live / real test data:** The program can now be tested in the real world under live conditions. The program at this stage should have most of its errors solved. This type of testing will push the program to its limits to see what it can achieve in the real world. This will determine whether the program meets the expectation of the customer. 4 types of live testing:
  - **Stress testing:** placing higher requirements than would occur under normal use to see if it has errors under extreme use.
  - **Security testing:** Forced attempts to break the security of the program to determine if the program can be corrupted in any way.
  - **Recovery testing:** Forced failure of the program to determine if the program can be recovered with no data loss or minimal data loss.
  - **Performance testing:** tests the load limits of the program, speed of execution and response time for user request.
- larger file sizes
  - Large files must be tested through the program to ensure it can handle a large files. Not only should it cover the bare minimum of the real world specifications, but it should cater for 10% more than the stated specifications to ensure one off scenarios that may exceed the minimum will cope well.
  - Limits of the program must be also be understood by the users, so they know what the stress limits of the program are.
  - The test may reveal very slow processing times or slow down the network due to reading large amounts of files. E.g. This would suggest that the program be run out of school hours or in different year groups.
- mix of transaction types
  - The real world will provide greater possibilities to the testing of the program.
  - E.g. A new school reporting program may be installed, different users will enter data at different times and enter data in different orders. The program must accept all varieties of data input within reason.
  - The program can then been seen if it needs added features to support further real time processes.

- response times
  - Response time: the times required for processes to complete.
  - Heavy and light loads are used to test response times using live data amongst system components, interactions and concurrent users (same time users) to test response times.
  - Delayed response times may mean feedback messages must be used to indicate the user to wait or else modules may have to be redesigned to be more elegant.
- volume data
  - Volume data: refers to the amount of data to be processed.
  - The program needs to be stress tested with large amounts of data to see if it can handle more than the minimum specified volume data.
- interfaces between modules
  - The meeting point between the modules and the main program.
  - The link is connected via a communication link.
  - Each module must be tested to ensure the variables from the module are sent to the main program and back without conflict. That is, timing must be checked as well as the processes fulfilling their logical outputs.
- comparison with program test data
  - Test data must always be checked logically to sustain the actual output every time.
  - Live test data allows for a variety of test data that is not initially checked, therefore live test data has been checked to ensure they fulfil the actual output before carry out a second test of this same set of live data to ensure expected outputs are met.
- benchmarking
  - **Benchmarking:** is the comparison of a program against a set of established standards.
  - These standards are either set by **programs recognised for their quality and performance** or **in-house** standards developed by the software development team for a goal to be met.
  - This then allows the program to be marketed as having a standard or above-standard level of performance.
- quality assurance
  - **Quality assurance:** continually assures customers that a program will meet its requirements as an efficient and elegant solution.
  - A great program contains the following factors:
  - **consistent** every time it is run (GUI and processes are consistent)
  - **intensively documented** (internal, external and user manual documentation)
  - **independent** on hardware it was designed for
  - **Integrity:** have good security measures included, even when power outages occur.
  - **Interaction with other software:** program operates with other common software without conflict
  - **Maintenance coverage:** program is easily updated and errors can be easily fixed when needed.
  - **Memory requirements:** program uses a reasonable amount of memory for its functionality.
  - **Recovery:** ability for a program to recover itself and potentially lost data.

- **Reliability:** ability to perform consistently without failure.
- **Stress resistance:** ability of the program to handle larger than the expected processing demands.
- **User response:** program is easy to learn and answers the user's problems.
- **Validity:** meets user specification.

## Reporting on the testing process

- documentation of the test data and output produced
  - Tests should be ongoing throughout the define, plan, implement, test and maintenance stages.
  - Report and review of tests is needed.
- use of CASE tools
  - CASE tools are used to automated and assist developers with documentation. There are two types:
  - **General application CASE tools:** suitable for creating written reports and for basic analysis of some results. E.g. word processors and spreadsheets.
  - **Specialised CASE tools:** provide **structure assistance**, **sometimes automated** processes and the **development of test data**.
  - **Examples of specialised CASE tools:**
  - Oracle: program used to generate what the expected results will be
  - Test data generator: develops a variety of large amounts of test data.
  - Data dictionary creators: allows data to be tested against the data criteria.
  - File comparisons ensures input files follow set rules and output files are correct.
  - Test management: tracks test data and results. Conducts tests of multiple modules.
  - Volume tester: used to test a program under network conditions with high amounts of users.
  - Functional tester: Provides user interaction with a program to test all possible pathways by inputting test data to cover all pathways.
  - Dynamic analyser: allows statements to be counted every time they are executed.
  - Simulator: creates a real time environment in which the program can be tested in.
- communication with those for whom the solution has been developed, including:
  - Communicating with clients or users must be done in an honest, direct and non-technical way.
  - This will ensure the program meets the specifications, if not, the program can be easily adjusted to meet them.
- ❖ test results
  - Test results should document both the positive and negative experiences of a program.
  - **Problems should be identified** such as bugs, length response times, inability to test a module due to lack of data input.
  - **Limitations of the program:** scope or size of the data type handling capacity. Data input restrictions and processing limits are placed here.
  - **Assets:** user interface, results from live data and results that show user needs being met are documented here.
  - **Recommendations should be made** such as that certain bugs will be fixed in feature development, modules will be rewritten or a program is open to new features.

- ❖ comparison with the original design specification.
  - Summary of the program and its tasks, which compare to the design specifications, are communicated to the client base.
  - Any specifications that have not been met should be alerted to the client and a reason why it wasn't met. If possible these problems should be fixed.

## Chapter 7 – Maintenance of Software Solutions

### Modification of code to meet changed requirements

- **Software must be built to be maintained.** That is, provided with constant documentation and built elegantly so changes can be easily made in the shortest time frame.
- It must be built to allow other developers other than the original developers to maintain it.
- Maintenance can take up to half of the software development costs.
- It is an ongoing process and never stops.
  
- **Corrective maintenance:** where reported errors about bugs are fixed. This could be crashes, illogical outputs or slow performance issues. This can be as **simple and cheap as changing a coding error** or as **complex and expensive as changing the design** of a program.
- **Adaptive maintenance:** involves changing the program to meet some other external change, such as a new operating system is released or new hardware. Or new laws could be released. Such as the introduction of GSTs accounting software had to be updated to include this function.
- **Improvement maintenance:** adds new features or requirements to meet updated user needs. This may include, increasing the processing speed of a program or providing more user documentation.
  
- **Maintenance begins with assessing whether it is economical to continue to maintain a program.**
- The program may be too old to abide by today's standard structures. It may be better off abandoned and a new program created. Else if a lot of money has been invested in the program, it may be viable to maintain an older program, e.g. the bank applications are around 60 years old but still maintained to today's standards.
  
- **Code:** high-level language written in an editor, low level languages are harder to understand.
- **Macros:** pre-recorded tasks by the user included within an application and accessible by hot-keys.
- **Scripts:** commands carried out without user interaction and they are usually written in a programming language that works with another programming language. E.g. JavaScript and Visual Basic can be used to write applications for HyperText Markup language (html)..
- Scripts can be easily altered to provide altered or new functions to an existing program.
- **Patches:** Smaller bug fixes that are easily resolved, while maintaining client support.
  
- **Modification** refers to updating program code to fix logical, run-time or syntactical errors. Modification can also add new features to a program to meet user needs as time progresses.
- It follows the same steps as the SDC.



- identification of the reasons for change in code, macros and scripts
  - Custom software is less likely to be frequently changed in terms of features as it continues to meet a specific organisation's predefined needs. It may only need an updated version 1 every 10 years. However it is still maintained for bug fixes and future developments (less often).
  - Macros are frequently modified by users because of their simplicity and hastiness to modify.
  - Scripts are frequently modified as they are used in keeping data up to date.
  - **Reasons for changing code:**
  - Currency of code: Bring the code up to date such as from assembly to event-driven.
  - Bug fixes: fix problems identified by users.
  - Improved functionality: Expand the programs functionality. E.g. add a module to add more game levels.
  - Meet user requirements: allow for new data input requirements, e.g. included OCR as well as mouse and keyboard. Or add a new user interface such as an enhanced GUI.
  - Improved maintainability: clean up code to make it easier to read and understand. E.g. separate individual tasks into individual subprograms.
  - Upgrade to meet hardware or software changes: allow for new use of software and hardware. E.g. allow software to write on Blu ray optical media.
  - Change processing format: Change batch processing to real-time processing to meet user demands.
- location of section to be altered
  - Two steps must be taken to decide what needs to be changed:
  - **Communication** between the user and the modification team
    - ❖ Communication will help establish the exact nature of the problem or need that has to be met.
    - ❖ The maintenance team must find out as much as possible about the problem or upgrade needed through:
    - ❖ **Interviews:** personal open ended talks with users to allow problems to be located. Although they are time consuming, expensive, not always consistent and results may be difficult to analyse.
    - ❖ **Observation:** overall picture of how the program works and the need or problem or need required can be seen, time consuming, expensive, and may be disruptive to users.
    - ❖ **Questionnaires and surveys:** Cheaper methods to allow consistent results to be obtained and easier to analyse. Impersonal, do not allow for open responses to provide a wider picture. Need really good closed, open and mixed questions.
    - ❖ **Research:** Documentation of the original product should provide a good understanding of how the program works and its limitations. Although the documentation may be poorly done or out of date.
  - 📌 **Feedback** is the crucial information given back to the maintenance team identifying the problems or needs about the program. Feedback can be communicated through formal meetings or written reports or letters.
  - **Thorough understanding of the original code** through study of the original documentation
    - ❖ **The maintenance team can easily analyse:**
    - ❖ Design specifications
    - ❖ Good quality structure diagrams, dataflow diagrams and system flowcharts
    - ❖ Quality internal documentation such as comments within the coded solution and intrinsic naming of modules and variables.

- ❖ Quality technical documentation with data dictionaries, algorithms and source code listings
- ❖ Modular independence: single logical tasks handled by separate modules, this is known as modularisation.
- ❖ Program testing and validation should be carried out to verify that the documentation is still valid with modern day settings.
- ❖ An understanding of the programming languages used for the program is essential for maintenance.
- ❖ Documentation should be continually updated and managed, to ensure future updates are always easy and cost effective to work on.
- ❖ Age of the problem: There's not much value modifying a program that has been already modified extensively. Its old programming language may not provide the standard structures for today's ever changing world.
- ❖ Hardware stability: The hardware on which the program is run on should be documented very clearly. Many hardware changes can happen very quickly, thus maintenance must ensure all hardware requirements are met for the software to run smoothly.

- determining changes to be made

- Users, management or customers determine when a problem occurs and a change is required. 4 issues will determine this change happening:
- **Priority: Is the fault urgent?** E.g. custom software that needs urgent maintenance must be done as there is no other alternative a company can use. They have designed that software for one specialised task only.
- **Extent: How many changes have to be made and how big are those changes?** Simple changes can be made quickly, however complex changes will take time, money and parts of the SDC will have to be used
- **Personnel: Who will do the changes?** Will changes be carried out in-house or will they be outsourced to another company.
- **Skills and expertise: What skills are needed to complete the changes?** Are such skills available when maintenance is needed?
- Strong analysis of the original internal and external documentation should provide sufficient help to the maintenance team for working towards a suitable modification.
- Compatibility problems of mixing new code with original code should be identified before implementation takes place.

- implementing and testing solution

- Implementing modifications are often easier than implementing a whole new program
- They still require testing procedures to be just as rigorous and detailed to ensure that changes interact correctly and effectively with the original code.
- Unit testing, program testing and system testing must all be done again.
- Modifications may cause conflicts with other existing functions of the program. This will cause another round of maintenance to take place.
- **Downtime:** is the time lost while a new version of a program is installed or modifications are carried out. This may also involve training users to master new features.
- It is important to minimise the downtime of a large program.

- To minimise change, it is best to implement modifications in the process of evolution, that is, implement smaller, less disruptive modifications first, and then as time goes on and when the time is right, release big updates when users are ready.
- This requires careful planning to minimise the impact of change.

## Documentation of changes

- **Any modification** to the original source code needs to be documented just as detailed as the original program was documented.
- **Every** modification needs to be documented, no matter how small. E.g. changing a constant value of a variable. This could lead to an unusual output error later in future, and every possibility must be checked by using good quality documentation.
- It would be very hard to find this problem in a very big complex program without good documentation.
- source code, macro and script documentation
  - Source code, macros and script documentation will involve internal documentation such as comments and intrinsic naming within the coded solution.
- modification of associated hard copy documentation and online help
  - External documentation and the user manual will need to be modified where necessary to **show where changes were made**. Such as in the algorithms and data dictionaries.
  - User **manual needs to be updated** to allow users to understand and use what benefits were implemented in the latest modifications. This can be in the form of a hard copy, but is usually in the form of a 'change log / fixes' online.
- use of CASE tools to monitor changes and versions
  - CASE tools cost effective tools used to track and monitor changes of different versions for **large programs**.
  - **They can help keep track of:**
    - Documentation and different versions of user manuals and developer reports.
    - Changes to data files and data structures using data dictionary tools
    - Modules that have been modified and modules that have been reused from a library.
    - Testing procedures and results including generating new sets of test data.
    - Conflicts and resolution of procedures. During implementation and testing stages.
  - **Version numbering:** Hierarchical process of keeping track of the changes to a program.
  - **Major changes** usually result in a new version, such as version 1 becomes version 2.0.0.0. These major changes usually include a new graphical user interface and contain significant new features.
  - **Minor changes** result in an adaptation of the version number such as version 2 becomes version 2.0.0.1. A minor change could be that a simple bug was fixed or performance issues were solved.
  - **It is also possible** to go from a previous version rather than the latest version to a new version. E.g. going from version 2.0.0.0 to version 3.0.0.0 rather than from version 2.5.0.4.
  - There is no real standard format for version numbers. It is up to the developer to state how they are going to version track their software. E.g. some software developers may want to choose a 2 number based version number, such as 1.7 or 3 based numbers, 1.3.1. The fourth number is used to

state the build number of a program, e.g. 1.0.0.120, means the program has had 120 builds or modifications since release. The format should remain consistent for easy organisation.

- Some companies choose to alert their users with product names rather than advertise technical version numbers. E.g. Windows 95, 98, 2000, ME, XP, Vista, 7.
- Some companies choose to skip version numbers for different system platforms. E.g. Windows MS Word went from version 2 to version 6, while the Mac MS word went 1, 2, 3, 4, 5, 6. Since they are build/ported versions off the previous build, some argue it should go up sequentially on all systems.

# **Chapter 8 – Developing a Solution Package**

**Defining the problem and its solution including:**

**Defining the problem:**

- Identification of the problem
- Idea generation
- Communication with others involved in the proposed system

**Understanding**

- Interface design
- Communication with others involved in the proposed system
- Representing the system using diagrams
- Selection of appropriate data structures
- Applying project management techniques
- Consideration of all social and ethical issues

**Planning and designing**

- Interface design
- Selection of software environment
- Identification of appropriate hardware
- Selection of appropriate data structures
- Production of a data dictionary
- Definition of required validation processes
- Definition of files – record layout and creation
- Algorithm design
- Inclusion of standard or common routines
- Use of software to document design
- Identification of appropriate test data
- Enabling and incorporating feedback from users at regular intervals
- Consideration of all social and ethical issues
- Applying project management techniques

## **Systems Implementation**

**Implementing the software solution by:**

**Implementation**

- Production and maintenance of data dictionary
- Inclusion of standard or common routines
- Use of software to document design
- Translating the solution into code
- Creating online help
- Program testing
- Reporting on the status of the system at regular intervals
- Applying project management techniques
- Enabling and incorporating feedback from users at regular intervals
- Completing all user documentation for the project
- Consideration of all social and ethical issues
- Completing full program and systems testing

**Maintenance**

- Modifying the project to ensure an improved solution

# Option 1: Chapter 9 – Evolution of Programming Languages

## Historical reasons for the development of the different paradigms

- **Paradigm:** is a model or pattern followed in the design of a programming language.
- a need for greater productivity
  - Society has demanded software to cover the everyday needs of a business or just simple email checking. The enhancements in greater and cheaper memory has allowed new programming languages to be created to cater for more demanding needs such as C++ for video game development.

### Low-level languages:

- **First generation:** Initially computers were programmed in **machine code (binary)**, such as they were fast to execute as they were coded directly to the CPU.
- They are slow and tedious to code
- Processor dependant, meaning that they could not run on a range of other CPUs.
- It was hard to understand what 101010101101 did or even in hexadecimal 133B still gave no clue to the programmer about what the code did.
- Hard to understand code and the time consuming process which lead to low productivity with this programming language.
- **Second generation:** Programming languages came to adopt meaningful names towards computer instructions. This is known as **assembly code**. E.g. they used commands such as LOAD, STORE and ADD to give the programmer an idea of what's happening.
- They needed an **assembler to translate** the assembly code (source code) into object code / machine code. They may have been a debugger included to identify errors when testing.
- There are a variety of assembly languages.
- They are still processor dependent, and are quickly executed.
- Still time consuming
- Uses hexadecimal figures.
- They used symbolic words, **mnemonics**: to help programmers remember what the commands were. E.g. using LD to stand for LOAD.
- It was still Hard to understand code and was still a time consuming process which lead to low productivity with this programming language. But easier to learn than binary code.

### High level languages:

- More 'human readable' as a translator or editor handles all the memory and file management code, and the programmer only has to worry about the code to deal with a certain task.
- The code is written in a language very similar to English.
- Processor independent meaning they can run on a wide range of CPUs
- They lose execution speed due to their source code being translated back into machine code by a translator so the machine can understand the code.
- E.g. Binary code
- **Third generation:** These programming languages were very English like, which allowed more people to learn programming and shortened the time required for programmers to program.
- Multiple actions could be completed with the same line of code.
- A translator is required to convert the source code into machine code.
- BASIC (Beginners All Purpose Symbolic Instruction Code).
- Comments can be put into the code

- E.g. Pascal, C, Algol, Fortran, Cobol, Ada and Basic.
- **Fourth generation:** Express the programs in the terms of *what* has to be done rather than *how* it has to be done.
- Fourth generation programming languages moved away from sequence processing or procedural processing as required in 1<sup>st</sup>, 2<sup>nd</sup>, and 3<sup>rd</sup> generations. However many of this structures are still retained within 4<sup>th</sup> generations.
- Non-procedural language, meaning that statements are not executed one after the other.
- They simplify the end-user approach to software development; the user is able to code functions without having an advanced understanding of the language.
- They are useful to program database functions in Visual Basic, such as SQL statements to read a database.
- E.g. Visual Basic, Hypertalk and dBase.
- **Fifth generation:** They use logic programming and are called declarative languages.
- They concentrate on the desired outcome and leave the processor to determine how to achieve this within a set of specifications given.
- The programmer will code a number of facts and then state queries to produce solutions to scenarios faced for a program.
- They are used for artificial intelligence (A.I) applications due to their ability to produce relationships with the facts of data and queries to real world situations.
- Examples: Prolog and LISP

Overall, high-level languages are easier to write than low-level and the programmer can produce greater quality programs with high-level, producing higher levels of productivity. Low-level are good for programs that require critical execution speeds.

- recognition of repetitive standard programming tasks
  - **Keywords and symbols** have developed in high-level languages to represent standard tasks.
  - E.g. BEGIN, +, \*, /, =, END
  - Structures have become standard amongst high-level programming languages, such as pre-test loops and Post test-loops, IF -THEN- ENDIF statements etc
  - Programming languages were developed which could specially solve these sequential types of problems.

Programming paradigm	Examples
Imperative	Pascal, BASIC, C, FORTRAN, COBOL
Functional	LISP, APL, Scheme, ML
Logic	Prolog, SQL
Object-oriented	Smalltalk, C++, Java, VB

- a desire to solve different types of problems (e.g. AI)
  - Low-level are good for writing programs for the BIOS (basic input output system) which can execute quickly with minimal resources.
  - High-level are good for writing applications and graphical operating systems.
  - High-level languages can be used to write- A.I applications and develop human like reasoning “fuzzy logic”
  - Non-procedural languages are being developed to solve new types of problems, often ones where the input and output data are not as clear cut as is currently the case when using computers. Aka, event driven applications..
- the recognition of a range of different basic building blocks
  - High-level languages have evolved from low-level languages to accommodate **compatibility needs** of working with different hardware and software , especially peripheral devices.
- emerging technologies
  - While existing paradigm base their models of sequential data processing which all modern day processors currently do.
  - Current computers are based on, sequence, selection and iteration.
  - Emerging technologies will assist the development of new paradigm models. Such as:
    - **DNA technology:** uses the basic building blocks of the human body and simulates the complex tasks they perform in order to develop technologies that are faster and smaller than anything that is commonly used today.
    - It is possible to use a base-8 number system rather than base-2, this multiplies the performance by 4 times.
    - Uses molecules as storage blocks, for extraordinary levels of storage levels
    - **Quantum processing:** uses atomic theory at its base. Uses electrons and protons as representations of binary data. Since each particle can represent more than one state, e.g. solid, liquid, gas and plasma can allow for far greater processing power. This is mostly theoretical.
    - **Parallel processing:** is being used now. It still uses sequence but it employs multiple processors to handle the data, thus speeding up the processing time required.
    - Parallel processing is still bottlenecked to some extent as every sub processor still has to meet up to a single main processor to deliver and accurate output. Some applications include:
      - Real-time simulations: Multiple processors can be used to calculate volcanic activity around the globe and concurrently analyse their affect on the world such as flight paths.
      - Neural networks: Can be used to simulate the processes of the human mind. Each processor can handle different sets of data, to speed up the process of understanding A.I, speech patterns and how learning occurs.

### Visual programming languages

- These are object-oriented programming languages where by the programmer uses a visual interface and drag and drop techniques to create a screen design layout. It has opened the programming industry to **novice programmers** who are able to generate programs that carry out simple tasks without having to have a great understanding of computers or programming.



## Basic building blocks

Before the basic building blocks of the programming paradigms can be discussed, the below table summarises the features of each of the paradigms.

Paradigm	Examples	Format	Feature
Imperative	Pascal BASIC COBOL FORTRAN	Uses a sequence of operations to produce a result. A procedural sequential language	Specifies a list of operations. Has a defined start and end. Program is a sequence of actions in a fixed order (often defined by line numbers of labels). Uses variables to store data. Programs are procedural
Logical (declarative)	Prolog SQL	Specifies outputs in terms of facts and goals	A program consists of declarations of facts, rules and goals. Non-procedural, as the solution is given but the process is not
Functional	LISP APL	Basic construct is the definition of functions	Does not use variables or assignment statements. A program consists of function definitions and applications of these functions. May use recursion
Object-orientated	Smalltalk C++ Java	Consists of interacting objects, each with its own programming. An object's internal structure is hidden	An object is a list of attributes and procedures or methods. Objects communicate by message passing. A class is a collection of objects which share common attributes and methods. Data and methods are encapsulated within objects

- variables and control structures (**imperative paradigm**)
  - Imperative languages use a sequential flow of control
  - Uses simple data types and is designed off low-level programming languages.
  - **Variables:** used to store input or output data in temporary memory locations
  - **Identifiers:** descriptive names for a location in the computer's memory. Variables are the actual locations used to store data.
  - **Assignment:** Statement used to transfer data from one memory location to another.
  - Example: **Temp = Array[Position1]**
  - Flow of control can be altered by three control structures:
    - **Sequence** – series of instructions carried out by one by one.
    - **Selection** – a control used for a choice between two or more actions
    - **Repetition / iteration** – a control used to execute a set of instructions repeatedly.
      - Subprograms – control structures used to group similar instructions into a single logical group.

- functions (**functional paradigm**)

- Instructions are provided as a series of subsequent statements
- Can handle any type of data and can return a single character called an atom or a list.
- Modelled on mathematical functions
- Example: Fibonacci numbers uses the previous statement to calculate the next.  
 $10 + 5 = 15$   
 $5 + 15 = 20$
- **Recursion:** A form of repetition that calls on itself.
- Run slowly and require more memory, good for concurrent processing (single processor performing several tasks at once)
- More efficient on multi-processor machines than imperative languages as functional languages are non-sequential.

- facts and rules (**logical paradigm**)

- Logical / declarative languages use a knowledge base with a set of inputted facts and rules to prove whether statements are true or false and come to a logical conclusion.
- Do not use variables or assignments
- Used in the creation of expert systems and neural networks (A.I).
- Example:  
If student's SDD mark is  $\geq 90$  Then  
Band = 6  
End If

- objects, with data and methods or operations (**object oriented paradigm**)

- Object-oriented programming (OOP) is based on the concept of programming objects to undertake tasks.
- **Objects** function independently of other code that may reference them e.g. picture boxes and labels.
- **Properties:** the qualities or attributes of an object such as size, colour, font appearance.
- **Methods:** are their behaviours such as the way an object appears onto a screen, how the user interacts with it, and how it removes itself from the users view when not needed. E.g. a drop down menu.
- **Abstraction:** the idea of an object 'invisibly' inheriting behaviours or property code from another object without affecting any objects. No object depends on another, all of their functions are private.
- Each object has its own encapsulated code that always stays with it.
- Message passing: Each object can 'pass messages' or interact with other objects to change their behaviours as events occur.
- OOPs are event driven programming languages, that is, the user picks the order of execution based on certain button clicks or events that happen within a program's runtime.
- Examples include: Games, simulation, GUI and animation applications and GUI OSs.

## Effect on programmers' productivity

The programmer's productivity will be determined by the programming language chosen and testing steps taken.

- speed of code generation
  - High-level programming languages have a shorter development time than a program written in a low-level language, given the programs were of same complexity.
  - This is due to high-level languages ability to be easily understood by humans and the ability for the programmer to ignore details such as memory locations and storage of variables. The programmer can concentrate on the steps to solve the problem.
  - Objects and reusable functions lead solutions to be developed in a rapid application development environment, thus speeding up project development time significantly and much more efficiently than the limited low-level languages.
- approach to testing
  - Testing can be an exhaustive, expensive and complex process, however with the correct testing procedures, most bugs can be eliminated as they are created.
  - However, testing is a very long process and can delay projects for a significant time , some developers may not choose to test extensively due to market pressures and time constraints. These programs will often end up of poor quality, full of bugs.
  - Three approaches to testing:
    - **Structured testing:** each algorithm, unit / module, program and system testing is continuously undertaken to significantly reduce the amount of program and system errors.
    - But it results in delays of the final product.
    - This is the most professional method of testing.
  - **Program testing:** leaves testing until the product is completed, can save time and costs but it results in poor-quality programs with lots of bugs and errors.
  - It is not recommended as bugs can be ignored as it may mean a program was designed incorrectly and a bug in early development may cause the whole solution to be expensively re-developed.
  - **Automated testing:** uses CASE tools to automatically test at unit, program and system levels.
  - It is expensive but it can prevent bugs from be carried on into deeper levels of the program where they are harder to detect.
  - They free up time for the programmer compared to structure testing. But by all means automated testing can be used in structured testing.

### Summary:

Some programming languages are easier to test due to:

- Ability to reuse modules that have been already tested.
- Some languages are **more syntactically compact** and thus require smaller amount of simpler code, reducing the chance of syntax errors. Examples: functional and logic languages.
- Imperative languages are easier to test with CASE tools as they are basically sequential and their control structures are standardised (Sequential, Sequence and Iteration)

- effect on maintenance
  - The fewer the errors, the easier maintenance will be
  - Modular programming reduces maintenance time such as OOPs
  - Object orientated programming languages use in-built modules for each object, the module found with a bug is the only module that needs to be fixed. (easier to locate)
  - Best way to reduce maintenance time, is to ensure all design specifications are correctly understood by the programmer from the client.
  
- efficiency of solution once coded
  - Low-level languages are the fastest to execute, but take the longest time to develop.
  - High-level languages are the slowest to execute, but take the shortest time to develop.
  
  - Imperative languages are often more efficient for basic sequential flow applications, as they work closely with the hardware using variables, assignments and identifiers and follow sequential structures. They are slower to develop than other paradigms but run best on single processors of the Von Neumann cycle.
  
  - Functional languages are more efficient for parallel processors as they involve many mathematical operations with can broken into separate concurrent calculations.
  
  - Logical languages can often result is excessive / inefficient processing requirements due the computer's ability to find its own solution based on facts and rules, the processing path is not specified by the programmer. Quantum processing would help speed up this excessive processing by processing more items concurrently using 1, 0s and the one of the four states of matter.
  
  - Object-orientated languages are more efficient for the development of user productivity but are less efficient in their speed of execution.
  
- learning curve (training required)
  - Different programming languages require more expertise than others.
  - Learning a programming language will take time and practical practice, just as if a person was learning Italian or French.
  - Some languages are easier to learn than others but the skills of learning one language can carry over when learning another language, with the adjustments of syntax to a specific language.
  
  - Imperative and object-orientated are seen as easier to learn as they are considered more popular and are often introduced to students as variables and standard basic constructs are easy to understand for a beginner.
  - Despite OOPs being 'visually' easier to learn, they still require training to fully understand their full capabilities
  - Training may involve formal face-to-face training, sample programs or guided self paced walk guides.
  
  - Logic and functional programming languages are often thought out to be harder and specialised, although these are often easier to teach to children as they don't require as much code or constructs to be coded. They rely on basic mathematical principles with rules and facts, so beginners can often relate these principles to the coding of these rules. Programming of basic structures such as loops and variables can be ignored. The computer comes to a logical conclusion.

## Paradigm specific concepts

### Logic paradigm

- (e.g. Prolog, expert system shells)
  - o Predicate defines the item that follows e.g. ?-is
  - o **Atoms** are items that use lower case in brackets e.g. (**a**, **parent**)
- Heuristics
  - o **Heuristics** are 'rules of thumb' that work most of the time.
  - o They are based on previous experience to come to logical conclusions when there is no YES or NO answer (i.e. between 0 and 1)
  - o But often the knowledge about a problem is not complete thus, a probability of chance is used. E.g. cause of car being broken: is 0.7 of a flat battery, 0.3 of a broken engine.
  - o The language will present what it thinks are the best possible alternative options.
  - o This closely mimics human reasoning or fuzzy logic and is useful in artificial intelligence applications, such as robot control.
- Goal
  - o **Goals** are the output required from the program, given the set of facts and rules.
  - o Facts are values which are known to be true. E.g. Alex is a male.
  - o Rules are the combination of facts. E.g. If person is a male AND Name starts with A, THEN Name could possibly be Alex.
- inference engine
  - o Inference engines apply the rules with the supplied facts located inside the knowledge base and database of facts using logical elimination processes to reason an output.
  - o Inference engines use either backward or forward chaining.
  - o Expert systems use inference engines to make decisions as if it was a real human expert.
  - o Expert system shells are pre-built 'empty' expert systems ready to be filled with knowledge by a human expert and coded by a knowledge engineer. They contain an empty knowledge base, no defined inference engine and an explanation output mechanism.
  - o Neural networks are expert systems that learn by themselves creating new facts and conditions as they process new problems.
  - o Knowledge base refers to a database of many supplied knowledge from a human expert in the form of facts, rules and goals.
  - o Explanation mechanism displays to the user why the inference engine has chosen that solution by showing the user valid rules and facts.
- backward/forward chaining
  - o **Backward chaining** starts with the goal / solution and works backwards to find the reasons why the event occurred. E.g. a plane crashed, backward chaining will derive a valid reason from the facts provided by investigation teams. E.g. backward inference suggests plane crashed by flying at low altitude in a foggy atmosphere with mountains.
  - o **Forward chaining** begins with the known facts and makes assumptions base of these facts and its knowledge base of facts and rules to determine a possible goal / solution. E.g. IF X flies AND is made out of aluminium AND has jet engines THEN X is a plane.

## Object oriented programming

- (e.g. C++, Delphi, Java)

### - Methods

- An object is any item which can be describe by specifying attributes and the procedures that can be performed on these attributes. E.g. picture box, report, drop down menu.
- Objects contain methods which can be accessed by passing messages to them from the mainline of a program's coded solution.
- **Method / operations / services:** the instructions contained within an object.
- A method is performed when a message is sent to that object. This is how different objects interact with each other.

### - Classes

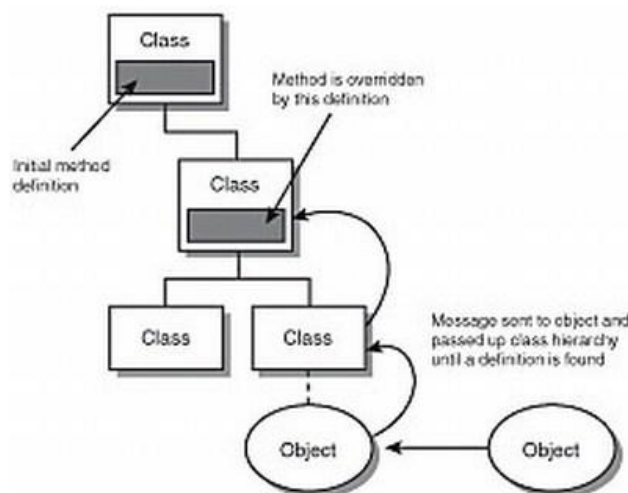
- **Classes** are groups of objects with similar attributes and behaviours.
- Classes form abstract (summary) data types as they share similar characteristics and processes. E.g. radio button, check box and push buttons can all be classified as one class.
- Instances: structure and behaviour of the objects in the class.
- Classes can inherit features from other classes to create superclasses

### - Inheritance

- **Inheritance** is the ability to create new objects from another parent object which includes the same attributes and behaviours of an existing (parent or ancestor) object.
- Allows one class to inherit methods and variables from another superclass (class with all common features to other classes). This can be demonstrated in a hierarchy structure (shown below).

Why is it important?

- Inheritance allows a programmer to efficiently reuse attributes, variables, methods and behaviours on various objects and classes by only coding these ONCE throughout the entirely solution
- Greater functionality can then easily be added by inheriting these existing features on new classes and objects while adding or coding new features into different objects and classes to quickly enhance functionality.



## - Polymorphism

- **Polymorphism** allows DIFFERENT types of objects to use the same segment of code, behaviour or method. E.g. a button can use the same colour as a picture box
- The routines are encapsulated within a class of objects, so these routines can be used by any object within that class.

## - Encapsulation

- **Encapsulation** is the ability to contain and hide the source code behind an object, such as internal data structures and variables.
- The user does not see this code. E.g. when the user clicks a button, they see a little button click animation, they do not need to know or see the coded solution.
- The **scripts are self-sufficient** which means **they stay hidden from the user and other objects** and they always travel with their related object.
- This means that their **existing methods and attributes can be updated** without affecting other classes or objects.

## - Abstraction

- **Abstraction** means to 'take away'. With OOPs, abstraction means hiding the irrelevant characteristic code of objects from the programmer to let the programmer focus on the coded a solution to the problem, not every little detail of an object.
- An example would be visual studio hiding the designer code in a separate tab for each object, rather than included it with the programmer's source code, which could confuse the programmer.
- Abstraction involves representing a complex task in a simple or symbolic way
- All subprograms are process abstractions, they allow a large complex program to be viewed in a more simpler manner defined as many sub sections. This makes it easier for a programmer to read and modify the program's code.
- Data abstraction allows similar data and its operations to be syntactically grouped as one line or region.

Functional (e.g. LISP (list processing), APL (A programming language))

## – Functions

- Functional languages allow a variety of data types to be returned from a function, unlike imperative which strictly only accepts what's defined.
- A function in LISP may return a single character (**atom**) or an entire list (**array**)
- Functional languages are not given a particular path to follow, so they can follow a multiple range of possibilities to arrive at the correct output or goal.
- While this requires powerful parallel processors to work efficiently due to the slow execution rate of these fifth generation A.I languages. They are worth it, as they can provide a huge range of endless possibilities compared to other 'not so expansive' programming paradigms
- Functions are built into modules meaning they can be reused together or used independently in other projects.
- Use recursion (repeating an application until the correct solution is reached) rather than iteration (repeating loops to get the same outputs).
- Modern functional languages allow the uses of variables and a range of data structures

- APL is used to simplify data arrays or matrixes (multi-dimensional arrays), they include an extensively list of operators. APL provides a high level of abstraction, however it is not widely used due to its difficulty to read and use, despite is powerful array handling capabilities.

*By MrBrightside, Good luck with your HSC year!*