

SDD Accelerated HSC Notes 2022

These are not-very-summarised notes for the SDD HSC course.

Contents:

1. [Social and Ethical Issues](#)
2. [Application of Software Development Approaches](#)
3. [Defining and Understanding the Problem](#)
4. [Planning and Designing Software Solutions](#)
5. [Implementation of Software Solutions](#)
6. [Testing and Evaluating Software Solutions](#)
7. [Maintaining Software Solutions](#)
8. [Developing a Solution Package](#)
9. [Option 2: The Interrelationship between Software and Hardware](#)

Some notes:

- These are really extensive sorry
 - Essentially a “summary” of Davis’ textbook which might go more in depth than the syllabus requires
 - Kind of aligned to the syllabus???
- I didn’t really study from these much tbh writing these notes was a lot more useful in terms of studying and understanding content
- I recommend doing past hsc papers (2012+) to become familiar with answering all parts of the question using relevant justification
- Algorithms and related stuff come with practice

Useful BoS post: <https://boredofstudies.org/threads/2021-hsc-sdd-first-in-course-my-thoughts-comments-and-advice-for-future-students-guide.398706/>

1. Social and Ethical Issues

- The Impact of Software
- Rights and Responsibilities of Software Developers
- Software Piracy and Copyright
- Use of Networks
- Software Market
- Legal Implications

The Impact of Software

Software is a part of many aspects of our lives and society, such as communication, electronic products, and web access. When software is inappropriately designed, developed, or used, the impacts can be significant.

Problems can be due to:

- Poorly developed software
- Intentional issues such as malware
- Unforeseen circumstances (Y2K)
- Privacy issues e.g., social networking, cyber safety

Consequences include illegal access of personal information and poor end-user productivity.

The Y2K Problem – Inappropriate data structures

In the 60s and 70s, memory was expensive, so dates were stored as 2 digits (e.g., 1987 was stored as 87) to save memory and money. Near the turn of the century, developers realised that this could lead to incorrect dates in the 21st century (e.g., 00 would be 1900 instead of 2000), and therefore cause major problems. For example, systems could perform incorrect calculations, affecting libraries to airports.

Inappropriate data structures can lead to major errors such as the Y2K problem, incorrect functioning, or slow processing.

Computer Malware

Malware or malicious software includes any software which performs an unwanted and uninvited task. It is commonly attached or embedded within apparently legitimate software or files.

4 main types:

- **Virus:** Has the ability to replicate itself. When the software is executed, the virus quickly duplicates itself and spreads throughout the system to perform its malicious processing.
- **Worms:** A virus which aims to slow down a system, by replicating itself over and over again until it consumes all the RAM, and hence other software will run slowly or not be able to execute at all.
- **Trojan Horse:** Malware masquerading as legitimate software, and when it is executed, it begins unwanted processes such as deleting or corrupting files, creating botnets to repeatedly access a single website and cause it to crash, or spamming messages to all contacts.

- **Spyware:** Aims to obtain your information without your knowledge. For example, through capturing web browsing habits, reading keystrokes for passwords and credit card details, or reading email contacts. The information is then sent back to the criminal who uses it for illegal acts such as identity theft.

Anti-virus, anti-malware, and anti-spyware utilities are installed to detect and destroy malware and advise users of potentially unsafe practices or websites.

Reliance on Software

Software must be very reliable and perform functions correctly, as it runs transport (cars, buses, planes), utilities (electrical grids, gas), everyday products (TVs), and important systems (hospitals, banks). There could otherwise be major problems and consequences.

Social Networking

Useful for communicating with others, viewing feeds and content based on tags, and sharing contacts. Examples: Facebook, LinkedIn, Twitter, Instagram

However, there are disadvantages such as private and personal information being easily shared (leading to identity theft, stalking, and online bullying). And anything posted online can be remained accessible indefinitely (e.g., sketchy comments as a teenager).

Cyber Safety

All internet connected devices have risks of different dangers and security issues. Cyber safety minimises the risk of dangers, particularly for children, to protect yourself and personal information when online.

- **Location based services:** Turn off tracking location unless it is specifically required e.g., Google Maps
- **Unwanted contact:** Report and do not respond to messages that make you uncomfortable
- **Cyberbullying:** Block the cyberbully and report to school/police
- **Online friends:** Some people are not who they claim to be, do not share private information with people you don't know IRL, meet them in a public place w/ adult
- **Digital footprint:** Messages and private information can remain accessible forever and read by future employer/partner/friends
- **Online purchasing:** Only purchase from well-known organisations such as eBay, examine details and conditions
- **Identity theft:** Information obtained by spyware, use malware protection utilities, do not respond to emails requesting passwords

Evaluating Information on the Internet

Information can be uploaded by anyone regardless of their qualifications or experience. Information can be biased, unsupported, unverifiable, misleading, or incorrect. You must verify the accuracy of the information, especially when doing research.

- Look for author and confirm expertise, or check if educational/government site
- See if information is up to date
- Determine who the intended audience is
- Check if information is accurate and unbiased e.g., political/religious bias

- Think about purpose e.g., advertise or educate

Rights and Responsibilities of Software Developers

Software developers have rights in regard to how other people may use and distribute/duplicate their products, and responsibilities to provide high quality solutions that are ergonomic, inclusive, free of malware, well documented and supported, and perform their stated task correctly and efficiently.

Summary

Software Developers	Rights	Responsibilities
<i>Intellectual Property</i>	IP rights of own work acknowledged/protected	Acknowledge others' IP rights
<i>Quality</i>	Avoid external OS/hardware problems	Produce software of high quality
<i>Response to Problems</i>	Not be harassed by trivial solvable problems	Response to problems encountered with solutions
<i>Malware</i>	Protection of their products by users	Distribute malware free software

More responsibilities include:

- Ergonomic (UI), inclusive (culture, disabilities) software
- Ensure user's privacy is not compromised
- Reliability of solution, performing correctly and efficiently
- Adhering to Code of Conduct

The Developer's Code of Conduct formalises the responsibilities.

Intellectual Property

IP is property resulting from mental labour. The creator of a work owns the right to control how it is sold and distributed.

In software, it is often less clear who owns the work, as many people are involved (developers, artists, outsourced code, writers of programming language).

Developers have the right to have their work acknowledged, and the responsibility to acknowledge the works of others.

Publishers of software products have a responsibility to ensure the IP rights of all authors involved in the creation of the final product are acknowledged and compensated.

Quality

Solutions should be high quality, especially when involving important data processing like financial data, because otherwise it will be very inconvenient when errors occur.

Some factors of quality include reliability, accuracy, integrity of data, efficiency.

Quality software is ensured through thorough planning and testing, and QA to ensure standards are adhered to. QA is an ongoing process throughout design and development.

The quality of a product depends on whether customer expectations are met and/or exceeded. It is up to the developer to know the customer's requirements and ensure they are met, for the product to be perceived of a higher quality.

Often compromises have to be made because of financial constraints, as developing high quality applications is time consuming and costly.

External factors affecting quality:

- **Hardware:** Developers have the right to expect that hardware operates reliably. The product should be tested with different hardware configuration and devices, and should have modules which react appropriately to hardware problems.
- **Operating System:** Developers have the right to expect the OS is reliable and robust. The product interface should fit correctly within the OS specifications, and the software should not make changes to the OS settings. The product should be as forward compatible as possible.
- **Other Software:** Other software should not be affected or affect the product.
- **Runtime Errors:** Software must include error checking built into the code, and have a fatal error detection module, that executes a save procedure before it is terminated.

Response to Problems

Responsibility to ensure that any problems users encounter with the product are resolved in a timely, accurate, and efficient manner.

- Mechanism in place to assist in the identification of errors and their resolution
- Error messages should be unambiguous and aim to identify the source of error
- Should have a support department or some way to respond to user problems

Developers should also provide free updates to solve errors and bug fixes as part of quality and maintenance, especially if the software is paid. (could be part of licence agreement)

More critical problems should be solved first, and less important ones such as cosmetics can be solved later.

Code of Conduct

Set of standards by which software developers agree to abide, normally part of professional associations. The standards increase the quality of developed software across the industry. It also formalises software developers' responsibilities.

Points include:

- Acknowledge others' work
- Honour contracts
- Strive to achieve the highest quality
- Respect laws

Malware

Software developers have a responsibility to ensure their products do not generate, feature, or transmit malware. Can be held responsible for distributing malware with their products.

Preventative actions:

- Check new data and software being added for viruses
- Scan all emails and attachments received by employees
- Update anti-virus software regularly
- Scan all software before distribution
- Use firewalls and other security mechanisms

Ergonomic Issues

(software) Ergonomics is the relationship between the human worker and software design.

Acceptable Response Time

Slow response times can be frustrating. If the user does not know that their action is actually registered, because the response time is slow and has no user feedback, then they may give up using the software or think that the system has crashed.

If the response time is going to be slow, indicate to the user with feedback on the screen, e.g., hourglass, progress bar, message saying “please wait” or “loading”. The program can also perform the time-consuming task in the background and allow the user to continue working on other tasks.

User Interface

User Interface is the screen designs and connections between screens that allow the user to communicate with the software.

Software design is all about the UI for the user, so it is the most important aspect for most software products, as functionality may not be fully discovered without a good UI.

Ease of use: User’s experience should be intuitive

- Group similar elements together
- Functions not available should be greyed out instead of removed from view
- Meaning of icons should be obvious

Consistency: Setting standards and sticking to them

- User interface elements correctly e.g., check boxes for their recognised functions
- Colours, icons, fonts used appropriately
- Alignment of data entry elements
- Consistent shortcuts with other applications

Messages to User

Wording/language should be plain, non-threatening, and consistent. Messages should be clear and easy to understand, have an appropriate icon and response options, and be placed consistently.

Usability Testing

A measure of how successfully the software product meets the needs of the users. It is not about performance and functionality, rather an efficient and satisfactory user experience.

Usability testing needs to be an ongoing process throughout the development process.

- Prototypes of each screen distributed to users to comment
- Representative users included in development team
- Randomly selected groups of users to test

Inclusivity Issues

Inclusive software should take into account the different users of the product. Developers must ensure inclusivity of their software and address inclusivity issues.

Software that is inclusive will sell better.

Aspects to consider in inclusive software:

- **Cultural and Social:** Different standards in moral and social issues, different way information is displayed (e.g., dates, names, currency).
 - Should ask for dominant language and include users from a variety of cultures during testing.
- **Economic:** Different economic situations of consumers.
 - Find a balance between quality and cost, view nature of market, manage development process efficiently to reduce costs.
- **Gender:** Men and women should be involved in the development process and take into account gender differences during development.
- **Disabilities:** Visual disabilities, hearing disabilities.
 - Ability to increase font size, easy to read font, text descriptions of images, transcripts/subtitles for videos in product.

Privacy

Protecting an individual's personal information. Some software legitimately requires personal information e.g., bank systems, social media, so you should ensure that individuals' privacy is not compromised in software and that data is stored securely.

Processes that might be performed in software that could cause privacy issues include credit card transactions, linking to friends and accessing their pages, and instant chat.

Methods to address privacy issues:

- Information transmitted via an encrypted connection
- Security settings the user must select
- Terms and conditions which describe privacy risk (privacy policy)
- Must explain why personal information is being collected and how it will be used
- Passwords to protect individual accounts
- Firewall to prevent unauthorised access to server e.g., for banking details

Software Piracy and Copyright

Software piracy is the illegal copying and use of software i.e., when copyright is infringed. The issue can be complex due to differing methods of software distribution and licensing.

Intellectual Property

IP is property resulting from mental labour and belong to the author. IP rights are protected using copyright laws, which give exclusive rights to copy/modify/sell/distribute the program. Licences do not give the purchaser the right of ownership of the IP, just the right to use it.

Deciding who has ownership of the final software IP can be complicated as modules of code can be combined and used by others, as well as art and programming languages.

Plagiarism: Stealing someone else's ideas and using them as your own. Work obtained from elsewhere should be acknowledged.

Reverse/Backwards Engineering: Analysis of a computer program to determine its original design, find out how it works, and create a similar product.

Decompilation: Process of converting machine executable object code to higher-level code.

Impacts of Software Piracy on Developers: Loss of profits for rightful owner of IP, discouraged to further create software solutions or updates.

Copyright Laws

In place to safeguard and legally enforce the IP rights of authors of any original works.

Copyright laws give the author of IP exclusive rights to licence others in regard to copying work, publishing, performing in public, distributing etc.

By protecting authorship with the laws, it provides economic incentives for creative activity.

IP rights are protected using the laws of copyright. Software licence agreements create formal contracts that allow the laws of copyright to be better enforced and define the conditions under which the software can be used.

Comment amendments to copyright laws particular to software:

- One copy may be made for backup purposes only
- When ownership of a licence is transferred all copies must be handed over/destroyed
- Decompilation and reverse engineering are not permitted

Copyright doesn't protect you against another developer creating a similar program.

Software Licence Conditions/Agreements

Software licence agreements are a legal contract between the author of the software and the user, which:

- Defines what a purchaser of the licence may do with the software
- Defines conditions such as whether it may be shared, or if decompilation is allowed
- Protects and enforces the author of the software by ensuring the authorship is appropriately acknowledged and protected by copyright laws
- Is enforceable by law

By agreeing to the licence terms, you are entering a legal contract with the copyright holder. Most software includes a compulsory reading and acceptance before installation.

Ownership vs Licensing: When purchasing software, you are purchasing a licence to use it. The licensing company owns the software, and you only own a right to use it.

Some Licence Types idk what these are

- **Single User:** Customer can install software on one computer, one copy for backup
- **Site Licence:** Organisation can install software on a large number of computers
- **Multi-User:** Site licence + specifies the exact number of users to access at one time
- **Domain Licence:** Allow software to be run on any computer that is part of a specific network domain e.g., organisation, school

Commercial Licence

- Covered by copyright
- One archival copy for backup
- Modifications, distributing, reverse engineering, decompilation not allowed
- E.g., MS Office

Shareware: distributed on free trial basis

- Covered by copyright
- Copies allowed for distribution/backup
- Modifications, reverse engineering, decompilation not allowed
- E.g., WinZip

Freeware: basically free software covered by copyright and no selling/modification

Public Domain: often in form of source code

- Copyright has been relinquished
- Copies allowed
- Modifications/decompilation allowed
- E.g., SQLite

Open Source: Aim to encourage collaboration within software development community

- Covered by copyright, removes many traditional copyrights
- Author must be recognised
- Modified products must be distributed under same open source licence
- E.g., Linux

Relationship between copyright laws and licence agreements

Licence agreements are used to better enforce copyright laws.

They are a legal contract between the author of software and the user and defines the conditions (such as if decompilation is allowed, how many copies can be made) and what the user may do with the software. In doing so, it protects the author of the software, by ensuring that the IP is appropriately acknowledge and protected by copyright laws.

Licence Terminology

Areas that should be included within a software licence agreement:

- **Licence:** formal permission to use a product
- **Agreement:** mutual contract between parties
- **Term:** period of time agreement is in force
- **Warranty:** an assurance of some kind e.g., limited warranty: no bug fixes afterwards
- **Limited Use:** on how many machines can the software be installed
- **Liability:** obligation as consequence e.g., developer will refund if there is an error
- **Program:** computer software including executable files and data files
- **Backup copy:** copy made for archival purposes, should be destroyed when resold

Reasons for need of Software Licence Agreements

It is easy to reproduce and copy, there is a collaborative development history, and the internet is open and easy to distribute copies on it.

Current and Emerging Technologies to Combat Software Piracy

Without some form of protection, software is simple to copy, and it is virtually impossible to determine that a copy has been made (unlike physical copies), especially with the internet.

Various technologies/strategies to minimise piracy:

Non copyable data sheets: Used in 1980s. A user enters codes/answers from a data sheet included with the product to continue using the software.

Disk copy protection: Most software was distributed on floppy disks. To prevent them being copied, data was written on areas of the disk that were not normally able to be read, or placed on specific sectors of the disk, and these features couldn't be replicated by most copying software.

Hardware serial numbers: Some hardware components include imbedded serial numbers that can't be altered. Software can examine these serial numbers and if they don't match, the program won't run.

Site licence installation counter on a network: Some software can be installed from a network server e.g., in larger organisations. A site licence is purchased which specifies the maximum number of machines that may install or simultaneously execute the software. A network uses a counter to do this.

Registration code: A code used to activate the software during installation. Most registration codes are generated and verified over the internet.

Encryption key: Encryption scrambles the data, so it is impossible to make sense of. Decryption reverses the encryption. Single key uses the same key to encrypt/decrypt, two keys use one key (public key) to encrypt and a private key to decrypt. Used for financial transactions over the internet.

Back-to-base authentication: Application contacts the software purchaser's server to verify that a valid software licence is held. Can be done on installation or every time the software runs. Requires internet connection.

Methods of Software Piracy

End User Copying: Individuals make copies of software or lend software to others/friends.

Piracy of OEM Products: Manufacturers of PCs install operating systems and other applications illegally on new machines they sell, without including the licences for them.

Counterfeiting: Organisations copy original software and then sell it as legal copies.

Mischannelling: Software sold with discounted licences being resold as fully licensed products. E.g., academic version being retailed as fully licensed version.

Use of Networks

A network is a collection of computers connected electronically to facilitate the transfer of data and software.

By Developers

- **Access to resources:** The internet has many resources, such as documentation, source code, programming language libraries. Developers can access code libraries, search for solutions for problems or how to use a specific function in a language etc
- **Ease of communication:** Networks and the internet allow developers to communicate and collaborate on projects without having to meet face-to-face. They can communicate with others who may be greatly separated geographically.
- **Productivity:** Communication and access to resources improve productivity.

By Users

- **Response times:** Response time is the time between a user's request and the server's response (loading times). It is important for users; slow response times lead to dissatisfied customers and poor perception of quality.
 - Can be difficult to control by developer as periods with high activity can lead to slow response times
- **Interface design:** When designing an interface, the developer has to be careful about using large files (Flash, high quality videos/images), because if response time is slow, users won't be able to see them.
- **Privacy and security issues:** Often sensitive data is transferred across networks. Precautions must be made to ensure that the data remains private.

The Software Market

Software companies should consider social and ethical issues when creating and marketing a product, to maintain their position in the market.

Product

What do you sell?

- Selling an expectation of the product

- It is unethical for the product to not meet user's expectations
- Product should conform to requirements and expectations of users

Place

Where do you sell it?

- Place it is sold is related to type of product and related audience
- Sold in shops, software retailers, industry specific, direct sales (internet or mail order)
- The place where it is sold can have an effect on the expectations e.g., medical software sold by medical company

Price

What do you sell it for?

- Massively discounted software makes it difficult for smaller distributors to compete
- Expensive software is not inclusive for all economic situations
- Cost-plus pricing: adding profit margin to cost of production
- Consumer-based pricing: Looks at what consumer wants and how much they can pay

Promotion

How do you spread the word?

- Promotional material should help consumers make informed choices
- Advertising should be accurate and not misleading
- 'Word of mouth' is a powerful promotional technique

Effect of Dominant Developers of Software

If one product dominates, it becomes the default purchase. This is the case in OS's (Apple and Windows), word processors (MS Office), and spreadsheets. This means that there is less incentive for innovation and less incentive to respond to customer support. However, it also means that there is more support and larger user support groups.

Despite the dominance of some products, new players can emerge due to inventiveness (Google). Large amounts of developers (app developers) can also impact the industry.

Legal Implications

There are many significant social and ethical issues that need to be considered by those in business creating and distributing software.

Software implemented on systems throughout a country/s can result in significant legal action if the software breaches the law in some way or doesn't comply with the legal contract between the developer and the client (e.g., copyright breaches or not performing as intended, or breaking the country's law).

Legal action can also result from developers using copyrighted code from other sources without paying licensing fees or gaining permission to use the code. The copyright owner then has the option of taking legal action against the developer.

National Examples

RACV vs Unisys

In 1993, RACV (motor vehicle and general insurance company) wanted to convert their current paper-based claims management system to an electronic storage and retrieval system. Unisys Systems won the proposal to supply RACV with their new system, which would be more efficient and reduce costs. During the presentation of the new system, Unisys assured RACV the response times and functionality shown would be present in the live system. However, when the system was delivered, the expected response times and functionality failed. RACV went back to the paper-based system during redesign, but the redesign was also slower. In 1996 RACV terminated their contract, sought damages, and was awarded \$4 million in damages, as the functionality of the system was misrepresented.

NSW Card System

Basically the NSW government wanted to replace the automated ticketing system in 1996. The developers were the ERG group and deadline 2005. But the project was full of delays, lots of deadlines were missed (e.g., full commuter trials). The NSW government in 2007 terminated the contract and sought damages of \$95 million, ERG said the government refused to allow live testing and said that nothing could be installed until the system was 100% perfect, did not receive sufficient cooperation, and besides they had already successfully developed smart card transport systems for Hong Kong and Singapore etc. So uh they counterclaimed for \$200 million for illegal termination of contract lmao.

International Examples

Microsoft vs Apple

Apple released their Macintosh PC with a GUI, and Microsoft soon after released Windows, which looked and felt like the Apple OS. Both were distributed globally. Apple claimed Microsoft stole their GUI designs, after a long court battle, they settled that both could continue distributing their GUI OS's. wow ok

Napster vs RIAA

Napster was a Peer-to-Peer file sharing software on the internet that allowed people to share music without having to purchase an original CD. In 1999 the RIAA filed a lawsuit on behalf of 5 major US record labels alleging Napster violated copyright laws. That's literally it.

2. Application of Software Development Approaches

- Main Software Development Approaches
- CASE Tools
- Methods of Implementation/Installation
- Current Trends in Software Development

Software Development Cycle

1. Defining the Problem
 - Determine the requirements of the system and needs of client
 - Study existing systems and consult with users
 - Feasibility study undertaken
 - Design specifications are made
 - Development plan is constructed
2. Planning and Designing the Solution
 - Design of data structures
 - Algorithms are created
 - Interface design in consultation with client
 - Project is broken down into modules
3. Implementing the Solution
 - Solution is coded in a programming language
 - Each module is tested as it is coded and then tested in combination with others
 - Documentation is made
4. Testing and Evaluating the Solution
 - Testing for errors as well as performance under live conditions
 - Evaluate the system to ensure that all requirements have been met
5. Modifying/Maintaining the Solution
 - Upgrading to correct errors, add new functionality, improve current functionality
 - Each modification uses the steps of the software development cycle

Main Software Development Approaches

Summary

Development Approach	Characteristics	Suited for:
<i>Structured</i>	5 stages, detailed planning, large team, long timeline	Large, complex projects with high budget and time
<i>Agile</i>	Fast, iterative, small team w client, planning as needed	Software that is regularly modified and updated
<i>Prototyping</i>	Client involvement, successive prototypes	UI/user-focused projects, unique needs/small budget
<i>RAD</i>	Fast, cheap, uses existing modules, client involvement	Low budget, small scale projects, quick timeline
<i>End-User</i>	End user developer, cheap, fast, no documentation	Small long-term projects for the end user

And a combination of approaches, such as prototyping for UI and structured for the system.

Factors to consider when selecting an approach:

- Budget and time available
- Size and scope of project (e.g., if 100% reliability is required for large project)
- Skill of development team
- Complexity of project
- Detail of requirements (e.g., if users know their requirements or not)
- Type of project (e.g., user-interface focused might use prototyping)

Using a suitable development approach ensures that the software will better meet the needs of the client and therefore be of a higher quality.

Personnel Involved

System Analysts: Defining, Planning, Modifying

- Determine and define requirements, construct development plan
- Analyse current system, model flow of data, compile list of inputs and outputs
- Collect new requirements for modified solution

Programmers: Planning, Implementing, Testing

- Code the modules of the solution in a programming language
- Create and update documentation
- Test their modules to ensure they perform their task correctly and efficiently

Software Testers: Evaluating/Testing

- Perform testing of final product
- Ensure product achieves requirements determined in defining the problem
- Test for errors and performance of product under real conditions

Users: Defining, Planning, Testing, Modifying

- Design specifications and requirements (surveys, interviews etc.)
- Screen design/UI development
- Testing and feedback

Structured Approach

Has 5 distinct stages, which must be fully completed before the next stage commences. It has a longer development time and is more expensive as there is detailed planning and testing involved. There is a team of developers with varying specialised skills who complete a particular module, and not much collaboration with users during the actual development.

Suited for:

- Large projects where performance and reliability are vital
- Complex projects that require a high-quality solution
- Large budget projects with a large audience (e.g., across country/s)

Pros

- Clear structure
- Skilled team
- Detailed planning and testing, project is reliable

Cons

- Longer development time and expensive, large team/personnel
- Very structured, not a lot of flexibility if requirements change

- Thorough documentation
- Less involvement with client

Example

A system for an airline to manage aircrew rosters.

- Large company with lots of data to handle, and a large budget
- They want good documentation and robust, well-planned system to be used for a long time
- Modules make it easier to modify and adjust – good as they are relying on the system to keep their company running

Agile Approach

Acknowledges that in many cases, requirements are not fully known until development is under way. Involves a small, multi-skilled development team that works closely together with the client. An iterative approach that focuses on quickly testing and releasing a working solution and adding onto it with each iteration. No detailed planning is done in advance but done as needed. New features can be gathered from user or client feedback.

Suited for:

- Software that is regularly modified and evolves overtime (web or mobile app)
- Software where the final product or vision is not well defined at the beginning
- Software with a fast initial delivery timeframe but can be updated later

Pros

- Responds well to changing specifications
- Good collaboration with client
- Suitable for a smaller team, cheaper
- Initial delivery is fast

Cons

- Requires small team of multi-skilled people
- Limited planning, lack of cohesion
- No detailed planning means the time frame is not clear

Example

A small restaurant is making a website for their business.

- Small company, smaller budget, likely to not fully know their requirements
- Start with menu, location, contact details, for online presence
- Can gradually add on new features, like online ordering, booking seats, delivery

Prototyping Development

Involves close interaction with the clients to refine requirements and design of final system. Each prototype is created, tested, and evaluated, and then modified and tested again. A prototype is a model used to get information about how the elements of the system work together. More formal than RAD and more flexible than structured approach.

Concept prototype: Information-gathering tool, to determine the requirements of the users. The users evaluate it and once they are satisfied, the prototype can be used to get information on the system specifications.

Evolutionary prototype: A development approach, the prototype is created, evaluated by the user, modified, and evaluated, until the user is satisfied with the software, and the final prototype is the final product.

Suited for:

- User-centred applications, interactive software, developing a suitable UI
- When client has unique needs but a limited budget
- Projects with evolving requirements or client is unsure of some requirements

Pros

- Cheaper and faster than structured
- Quick user feedback, better quality
- Missed functionality can be identified easily

Cons

- Planning not as clear
- Time-consuming as each prototype must be tested and evaluated
- Requires strong user involvement

Example

A small plant shop owner wants an interactive plant guide.

- Cheap, faster, not a lot of complex functions
- May not know full requirements, so prototyping can help determine them
- Interactive, user and UI focused system

RAD Approach

Rapid Application Development. Main aim is to produce a usable product as quickly as possible, by maximising use of existing code modules and software (CASE tools like VB, spreadsheets), modifying to suit. There are no formal stages, but there is very close collaboration with the client to determine the needs and way the product is produced.

Communication between the clients and developers is vital in RAD, as it speeds up the development process greatly. As the product is developed, new and changing requirements are discovered, so close communication means they are quickly determined and resolved.

Suited for:

- Small scale projects with a small (sometimes one developer) team
- Low budget simpler projects

Pros

- Low cost and time
- Generic code modules have been tested so are reliable
- Developer does not need to have detailed programming knowledge

Cons

- Added processing and size due to use of generic code modules
- Quality compromised for time
- Limits functionality due to reliance on existing modules

Example

A sports club needs a system to manage their members.

- Probably needs it quickly, at a lower cost
- Pre-existing modules for sorting members by different criteria (name, address)
- Small scope, not very complex functions

End-User Development

Very informal, the end-user develops their own software solution, generally using 4th-gen programming environments (spreadsheets, database management system). They often customise software and functions already in the application. It is generally unstructured with minimal planning, over a short timeframe. Development costs are usually none.

Suited for:

- Potential long-term projects for the end-user to use
- Small, easy to develop solutions

Pros

- Cheap, fast
- Easy to modify, flexible structure
- Simple development, use of standard software packages

Cons

- Little to no documentation
- Not suitable for complex projects
- Difficult if end-user has no programming knowledge

Example

A student wants to create a study timetable manager.

- Small scale project that can be done by the student using available applications
- Student probably doesn't want to spend money
- Not very complex features

Combination of Approaches

Certain aspects of a solution may be suited to different approaches, so a combination of approaches can be used. Some modules use one approach, and other modules use another.

For example, prototypes can be used to develop the screens and interface with the user, and the structured approach can be used to reliably develop the product based on the agreed prototype. Or, RAD can be used to quickly develop successive prototypes for user feedback to modify and create another prototype with RAD.

CASE Tools

Computer Aided Software Engineering tools are any software that is used to assist developers. They include the following functions:

- System modelling
- Data dictionary creation
- Production of documentation
- Automatic test data generation
- Software version control
- Production of source code
- Project management

CASE Tools for each stage

Understanding the problem:

- Software to create system models
- Gantt chart generators

Planning and designing:

- Software to build data dictionaries, algorithms, models like DFDs
- Software to design UI and screens

Implementing:

- Versioning software
- Automated source code generation
- Test data generators
- Documentation generators
- The IDE itself with its help and automated correction features

Testing:

- Test data generators
- Automated testing software to simulate multiple users performing multiple tasks at the same time

Maintaining:

- Documentation tools for ongoing access and updating of documentation
- Versioning software as changes are made and released to users

Software Version Control

As software is revised, CASE tools can help keep track of version numbers (1.0, 1.1, 2.1). They can also document what changes were made in each version, and by whom.

Data Dictionary

CASE tools to manage the data dictionary, easily allows developers to have an up-to-date data dictionary. Useful to large projects with complex data. Data dictionaries are vital to keep variable names and identifiers well documented, especially for large teams with separate modules accessing the same data files, and helps programmers make sense of the code. Lists things like variable names, data types, scope, purpose.

Test Data Generation

Generates data to be used for testing. CASE tools (e.g., DataFactory) can produce large amounts of data easily, and can check boundaries (e.g., $x > 2$).

Production of Documentation

CASE tools can be used to easily create documentation at the same time the software is produced. Graphical applications can be used to create DFDs, system flowcharts, and structure charts. (e.g., adobe illustrator, lucidchart). System models like structure charts help developers visualise how their module fits into the entire project and models help clients to see a simple representation of the system to provide feedback.

Production of Code - Oracle Designer

A software that specialises in system modelling. Source code can be generated and help files can be generated from the module definitions.

Methods of Implementation/Installation

The new system must be installed on site. The conversion needs to be smooth.

Factors to consider when selecting a method:

- Size and complexity of system
- How vital the loss of data would be (lives lost)
- If it's feasible to run two systems at the same time (personnel, hardware)
- Time and budget available

Direct Cut-Over

Complete and immediate conversion, where the old system is finished before the new system is started (e.g., cut off on Friday, install over weekend and start on Monday). The new system must be fully functional and error free before installation, data must be converted and imported beforehand, and users must be trained before conversion.

Suitable for:

- Projects, where using both systems at the same time is inconvenient/impossible
- Small projects, where data loss is not detrimental, or uncomplex systems
- Installing completely new systems/apps, or ones that are very well-tested

Pros

- Faster and cheaper
- Less work required as there is no duplication of data and processes, data conversion only occurs once
- Avoids confusion between two systems running at once

Cons

- If new system fails, there is no backup system and data is lost
- Users must be trained beforehand
- New procedures cannot be trialled beforehand
- It is difficult to revert to old system if major problems emerge

Example: A bank is introducing a new ATM to its suburban branches. Direct cut-over is the only method, as it is impractical for customers to use both ATMs at the same time. The software and hardware for the ATM should have been thoroughly tested beforehand.

Parallel Conversion

Both systems are used fully for a period of time. This can also mean the new system is installed in some sites but not others. The new system can be tested fully without loss of data, the two systems can be compared, and the users learn to use the new system.

Suitable for:

- Projects, where loss of data would have dire consequences (e.g., hospitals, lives)
- Places with enough processing power to have 2 systems running at once

Pros

- The old system can be a backup if the new system fails
- Users can get familiar with new system overtime
- Systems can be compared & tested

Cons

- Double the processing and workload
- More expensive (sometimes)
- A longer process??

Example: A hospital is implementing a new patient monitoring system using technology instead of manual observations. However, a nurse is still appointed to manually monitor the patient in the beginning, in case the software system has an error not found during testing.

Phased Conversion

Gradual introduction of the new system over a period of time. New modules are tested, and then introduced. If the new system fails, only one module is affected.

Suitable for:

- When the rest of the system is still under development, completed modules are released as they become available and tested
- When the system has separate modules that can be implemented independently

Pros

- Only one module is affected if it fails, less data lost
- More manageable method
- Modules are tested before added

Cons

- More expensive and slower
- Only suitable for systems that can be implemented as separate, independent modules

Example: A library management system has separate modules for borrowing and returning, which can be implemented separately. Additionally, if the staff is a small team, they can focus on learning one new module at a time, making it less stressful.

Pilot Conversion

New system is installed for a small number of users, while the rest of the users still use the old system. The new system can be evaluated and tested, considered the final testing of a product for the developers and users in an operational environment. Errors can be identified and corrected before release. Eventually, the new system is released for all users.

When getting pilot users, the group should test out all functionality. The level of proficiency of users, their roles, and their attitudes towards the new system should be diverse.

Suitable for:

- When a system needs to be evaluated, tested, & compared, but parallel is not viable
- Where data loss would have some impact

Pros

- First users can be trained, and then help train others
- Old system can be a backup if new system fails, only small data loss
- New system can be tested

Cons

- There will still be data loss if errors occur in new system
- Pilot users may feel lost without support (???)
- Longer process

Example: A school is implementing a new online learning platform for students, starting with year 7 to test it. After a semester, the year 7s and teachers give feedback, and after modifications are made, the system is introduced to the rest of the school.

Issues when implementing on multiple platforms

Support for different functionality needs to be available for each platform.

- Resolution and size of screens are significantly different
- Method of distribution is different for each platform (Appstore, Website)
- Input devices are different – touch screen, keyboard, joysticks, mouse

Current Trends in Software Development

Employment and Work

Outsourcing: Software development is contracted to outside developers. This is cheaper than employing their own IT staff (frees up internal resources for other purposes) and can get higher quality work from specialists with faster development times. Companies can also outsource for particular areas of expertise and for access to new technologies. However, if the company is not satisfied with their work, they might not be able to get rid of them because it is contract work. Additionally, it can be costly if the project fails, and the company has less control over them.

Contract Employment: Outsourcing has led to many IT jobs becoming contract employment. Contracts can last until the end of development or for a fixed-term contract. There are websites to set up contract programmers with customers, often with a ratings system.

Environment in which developers work: Networks of computers allow developers across the globe to collaborate. Managing and combining contributions is done with CASE tools. In the past, programmers worked on assigned tasks in isolation, but now programmers are multi-skilled and work together with other developers and users, and can work from home.

Collaborative environments: Software that allows workers to communicate as if they are alongside each other, that has voice and real time video interaction. Workers can work from home or in different countries, however, video/audio quality can be an issue.

Software Products

Changing nature of applications: Software development used to be just for desktops or servers, but now there is software for mobile devices and websites. Programmers need to constantly upgrade their skills for new software devices and apps, and be multi-skilled. Different devices also have different screen sizes/resolutions, processing power etc.

Learning objects: A collection of digital resources used to teach a specific chunk of knowledge. Learning management and content management systems can monitor student progress, deliver assessment items and tests, and record results.

Widgets: Small apps that reside on the desktops of PCs. Use the internet to update their content. Web widgets are included in a web page, e.g., twitter feeds on news articles.

Apps: Applications, generally on mobile phones. Can be downloaded from App Store for iPhone. Apps are generally smaller and use less processing than other software because mobile phone CPUs are less powerful.

Applets: A small application (e.g., notepad, paint), which has basic functions. Many applets run within a web page or another app (e.g., plugins like QuickTime and Flash Player).

Web 2.0 Tools: Where the users collaborate, share, and create their own content (e.g., Wikipedia). Promotes users creating, sharing, and combining data across a variety of platforms e.g., linking photos to locations on Google Maps.

Cloud computing: Apps run over the internet, commonly with a web browser. The user doesn't know where execution occurs or where data is stored (can be saved locally, but usually on a remote server). Files can be accessed and edited by multiple users, so facilitates collaboration, however, requires internet connection to do so. There are also privacy and security issues involved with storing data remotely. Developers can develop applications that can be stored in the cloud and accessed from multiple locations. Storage is unlimited.

Mobile phone technologies: Phones now include digital cameras, internet connectivity, Bluetooth, GPS, social networking etc. All this has led to more apps on the market.

Examples

- **Collaborative environments:** Zoom, Google Meets, Skype
- **Web-based software:** Facebook, Twitter, Gmail
- **Learning objects:** Google classroom, Moodle, Khan Academy
- **Desktop widgets:** Clocks (time zones), weather, news, calendar, image slideshow
- **Apps:** Minecraft, Instagram, Notes
- **Web 2.0 Tools:** Google Maps, Facebook, Twitter, YouTube, Wikipedia
- **Cloud-Computing:** Google Docs, Google Drive apps, Dropbox

3. Defining and Understanding the Problem

- Defining the Problem
- Issues Relevant to a Proposed Solution
- Design Specifications
- System Modelling
- Communication between Client and Developer
- Quality Assurance

This is the most important step in the development cycle. If there are mistakes in this step, the product might not meet the needs of the client and redoing will be costly. The requirements of the solution must be found, the new solution must be considered feasible to produce, and design specifications must be made.

Defining the Problem

Defining the problem to get a clearer idea of the client's needs and requirements and create the design specifications. It involves:

- Determining the needs of the client (what needs will be met by the product?)
- Determining the functionality requirements
- Evaluating compatibility issues (existing hardware and software)
- And performance issues (internet, graphics loading)
- Determining the boundaries of the new system

Needs of the Client

A solution is required to meet a need (e.g., improving efficiency of a system): something that the client wants it to do. Needs must be articulated clearly to develop a clear picture of the problem to be solved. It is useless to develop a new product without establishing a need for it. Needs can be analysed via surveys, interviews, observations, demonstrations etc.

Objectives are the short term and long term aims and plans.

Functionality Requirements

As opposed to needs, these must be precise, specific, and measurable. They describe what the system will do and should be continuously checked during development to make sure they are fulfilled.

Cannot be subjective, e.g., "fast", because then the requirements are not clear and may not meet the client's needs. They should be measurable, e.g., "within 3 seconds", so they can be used to check if the project meets the requirements or not.

The final evaluation of a project's success is based on how well these requirements are met.

Compatibility Issues

Developers must ensure the product is compatible with a wide range of devices and network conditions, e.g., Android vs iPhone, computer vs phone. To do this the system must be tested using a wide range of environments.

Common compatibility issues include:

- Different devices – screen size and resolution, input devices
- Different versions of the OS
- Different network connection conditions
- Browsers and HTML version support

Performance Issues

Performance requirements are often stated in the documentation (e.g., the app will load in less than 3 seconds). When developing for a broad and unknown audience, it is difficult to know if time sensitive requirements can be met. Regardless, developers should use the most efficient algorithms for each task and try to minimise performance issues.

Performance issues are generally just the software running too slow or slower than expected. This may be due to multi-user software performing slower under real world conditions, large files, or a spike in activity over a network slowing response times.

Visual aids are requirements to show progress of slow processing; user feedback, so they know the program hasn't crashed (response times – ergonomics).

Boundaries of the Problem

Boundaries define the limits of a system. Determining boundaries is finding out what is and is not part of the system. Items from the environment need to be considered if they have an influence on the system.

Inside the boundary: the system

Outside the boundary: the environment

The system interacts with the environment via an interface.

When defining a problem, it is important to define the boundaries, so the customer has realistic expectations of the limit of the system.

Issues Relevant to a Proposed Solution

Is a new solution worth the time, effort, and money? **Or can an existing solution be used?**

Generic stuff

Client points to consider:

- Will the new system meet future needs?
- Are there existing similar systems?
- How crucial is the new system to the total organisation?
- What budget is available?
- How much staff retraining will be required?

Software developer points to consider:

- Expertise needed for development
- Resources required to develop the system
- How will future support be provided?
- Staff for development of the system
- How much time is available?

Modifying an existing solution is sometimes cheaper, and it is already tested/documented.

Feasibility Study

Undertaken by system analysts, determines whether it is possible to solve the problem effectively under the constraints:

- **Economic:** Will the new product be profitable, and is it affordable within the budget?
- **Time:** How long will it take to implement the product and train staff? What is the schedule and time limit, when is the deadline?
- **Technical:** Can existing hardware/software run the system or is new stuff needed?
- **Operational:** Can the organisation/staff/target customers use the software? Will it meet the needs?

If a project is deemed unfeasible, the developer may choose to explore alternative solutions to the problem, choose to discontinue with it, or scale down the scope of it.

Social and Ethical Considerations

Changing nature of work for users: The new system will change a user's job, so their needs and input should be considered. Retraining may be needed.

Effects on level of employment: Computers have replaced the work done by people (often labour intensive and repetitive jobs). Businesses use computer systems to reduce wages and costs, so a new system may result in employees made redundant.

Effect on the public: The system should be inclusive for different ages, cultures, and disabilities. The older generation takes more time to learn new technologies (ATMs involved retraining the entire population).

Also generally: copyright and IP, Inclusivity (existing software may have accessibility features already available), Privacy.

Legal Issues

Copyright stuff.

Security, safeguarding information (privacy).

Laws/legal standards e.g., GST calculation.

Licence considerations – making a program using an existing solution may require paying for licences which would make it more expensive vs custom software.

Customisation of Existing Software Products

This can be a cost-effective solution. It will modify an existing product to suit the needs of a client. The software is already well tested and documented as well.

Open-source software is often customised to add new features. It is also common for tools to be included in many commercial products which allow the user to create customised screens and functionality, such as drag and drop design screens.

Cost Effectiveness

A constraint can be a limited budget.

Areas to consider regarding cost:

- Hardware and software costs (new hardware/software needed to develop product)
- Personnel costs and salaries, outsourcing costs
- Future upgrades and maintenance costs
- Training costs
- Cost savings as a result of the product, predicted sales
- Available money

Selecting an Appropriate Development Approach

If no existing solution can be found, a new solution must be made.

A software development approach is chosen based on the project, considering the budget, time, complexity, development team skill etc.

Design Specifications

The aim of design specs is to interpret the needs, objectives, and boundaries of the client into a set of specifications which developers can use to create the final solution. They need to be developed from both the developer's perspective and user's perspectives.

Evaluation throughout the development process is done with reference to these specs, so they must be non-ambiguous or vague, use diagrams to clarify structure, and have measurable outcomes.

Design Specifications from the Developer's Perspective

Specs create a framework under which the development process will operate, so that the solution will look, feel, and be documented similarly, as different members of the team work on their own individual modules. Documentation and coding standards are developed. Often CASE tools are used.

Consideration of: Data types and structures, algorithms, variables, development approach, quality assurance, modelling the system, documentation

Includes: System flowcharts, DFDs, IPOs, data dictionary system, development approach

Design Specifications from the User's Perspective

Any specifications that affect the user experience should be stated. Standards in regard to the user interface and other ergonomic factors should be considered.

Consideration of: Interface design for continuity, appropriate message/icons, data formats for display, user's computer configuration and environment, ergonomic issues

Interface design for continuity: Use of menus, use of colour, placement of common elements, wording of messages, designs of icons, format of data presented.

Ergonomic issues: Which functions are accessed most often, keyboard shortcuts

User's environment: Use of familiar features e.g., shortcuts from other applications, hardware needed e.g., processor speed, RAM.

Communication and feedback from users are vital when specs are being created.

System Modelling

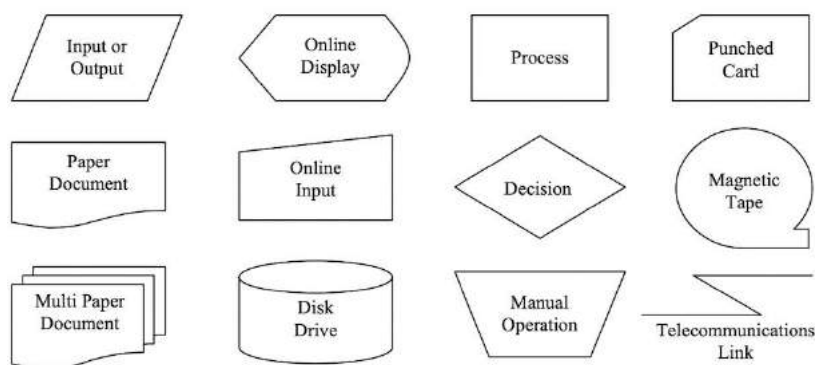
Summary

System Modelling Method	Purpose (what it shows)
<i>System Flowcharts</i>	Logic and movement through a system, nature of inputs, processes, storages + overall view of system.
<i>Context Diagrams and DFDs</i>	The movement and flow of data between external entities, processes, and storages.
<i>Structure Charts/Diagrams</i>	Top-down hierarchy, sequence of processes, data movement between them, decisions/repetitions.
<i>IPO Charts</i>	Data entering process, processes performed, outputs. Steps to transform inputs to outputs.
<i>Data Dictionaries</i>	Carefully document variable names, data types, scope, purpose etc. Useful for large teams.
<i>Screen Designs/Concept Prototypes</i>	The design of the interface between the user and software, clients can evaluate screen design.
<i>Storyboards</i>	The connection and navigation between different screens/user interfaces.

Representing a system using diagrams.

System Flowcharts

Describe the logic and flow of data through a system, showing the interactions between input, processes, storage, and the nature of these components.



Can include manual processes.

System flowcharts are used at a higher level than other modelling techniques to show an overall view of the entire system. (probably not assessed because they're old™)

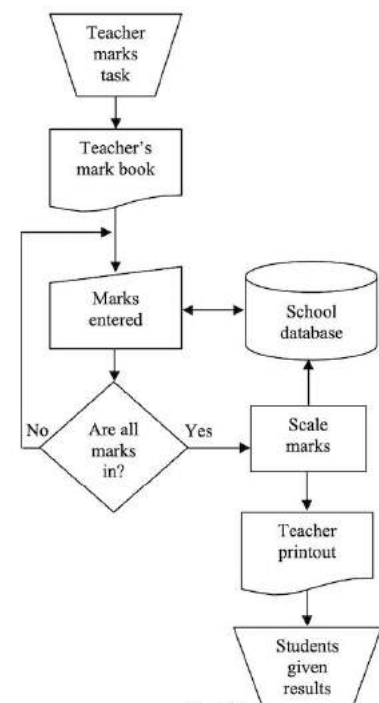
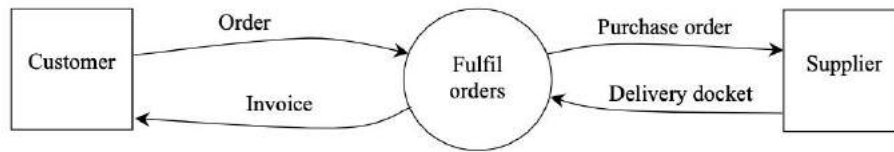


Fig 4.11

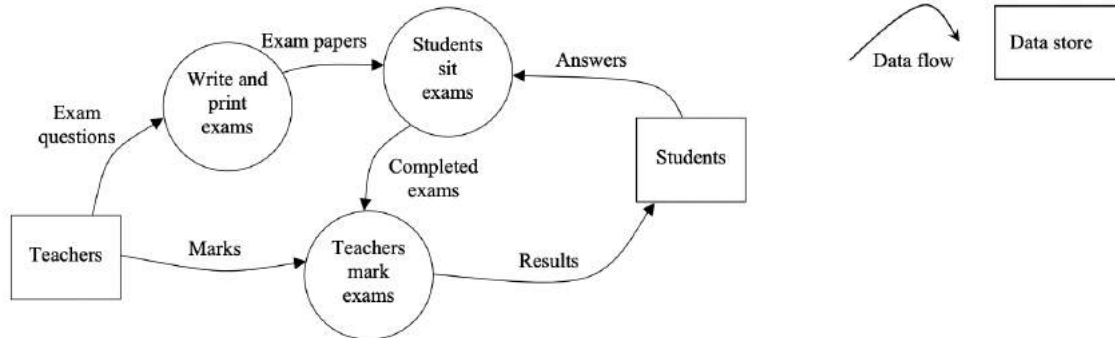
Context Diagrams and DFDs

Context diagrams represent the entire system as a single process, identifying all the data entering and leaving the system, and indicating where the system interfaces with its environment.



DFDs describe the path data takes through a system, from external entities, processes (actions that take place ON DATA within a system), and storages. ONLY DATA FLOW.

There are different levels of DFDs, e.g., process 1 turns into a level 2 DFD, with processes 1.1, 1.2, etc. Context diagram is level 0.

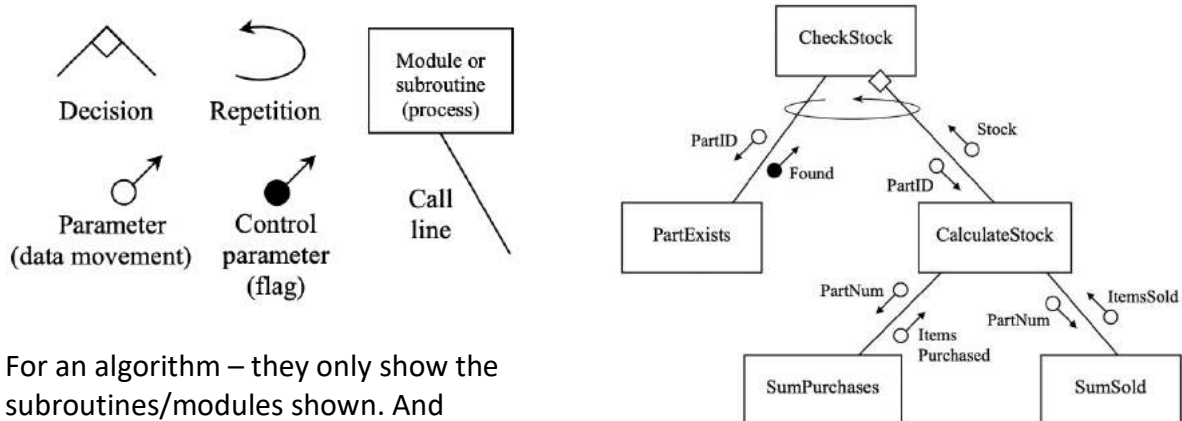


Structure Charts/Diagrams

Show the top-down hierarchy of processes and the sequence in which they take place, along with the data movements between processes, and decisions and repetitions of processes.

It can create a template for the actual code. It is useful for maintaining and upgrading software, because it is easier to identify the relationship between the subroutines.

Control parameters are anything that determines the order of processing, IF/ELSE. Are generally Booleans but could be an integer variable like Choice which determines a CASEWHERE.



For an algorithm – they only show the subroutines/modules shown. And variables shown.

CASE tools can analyse existing source code and automatically generate models similar to structure charts.

IPO Charts

Describe data entering a process, the processing performed, and the resulting information leaving the process(es). (Can be used to expand on lower processes in DFD/structure chart).

It is a table of inputs, processes, and outputs, used to work out what data (input) is needed, what information (output) is needed, and how to transform the data into information. It documents a system, describing the steps that transform inputs to outputs. (Outputs can include error or status messages, inputs/outputs include files.)

Inputs	Processes	Outputs
Student ID	Retrieve time of swipe in	
	Retrieve record of student ID	
	Record time of swipe in to record	Updated student record

Data Dictionaries

When developing a software solution, it is vital that all variable names or identifiers are carefully documented. This is especially important for a large team of programmers that are working on separate modules, accessing the same data files.

Data dictionaries make it easier for current future programmers to make sense of the code, e.g., there will be no confusion from duplicate identifiers, and for future maintenance.

They document all identifiers in a program, including their name, data type, length, scope, purpose, validation (e.g., what is a valid date), example.

Identifier	Data Type	Storage Bytes	Description	Example
ItemFound	Boolean	2	Returns if the item is found	True
Count	Integer	2	Incremented for each item in the array	1
ItemList	Array(Integer)	10 * 2	List of integers	ItemList(2) = 2
TargetItem	Integer	2	Item to find	6

Data dictionary functions are available with CASE tools.

Screen Designs/Concept Prototypes

Screen designs provide the interface between the user and the software.

Effectiveness of screen designs include placement of items, consistency, use of colour etc.

Concept prototypes are used to allow clients to evaluate a screen design.

Adhering to screen design standards make it easier for users to learn and use a program. When companies produce OSs they create standards e.g., X button on top of application to close it, radio buttons are round, check boxes are a square with a tick.

CASE tools have areas where screen design preferences are stored. When code is generated, the screens will adhere to the preferences entered in the CASE tool.

Storyboards

Describe each screen and how they interact with other screens. It is a collection of screen designs as a diagram that describe the links between the screens e.g., where a button leads to. They document the flow of the screens and navigation between them.

It is important to have a logical navigation system between screens, so users don't get lost and confused. Navigations should be in consistent positions on the screens.

Other stuff idk

The logic in modules and subroutines are represented before coding using pseudocode and flowcharts. Pseudocode uses word statements, flowcharts are diagrammatic.

Test Data: Normally in sets. A set of inputs and expected outputs = one set of data. Test data should test every possible route through the algorithm and boundary conditions.

Communication between Client and Developer

User's needs should be considered at all stages of the SDC, especially during defining and understanding the problem.

Communication methods:

- **Meetings:** formal, explain information to a large number of people, used to inform users in regard to progress and address major issues
- **Questionnaires/Surveys:** large groups, quick, limited in detail due to group size
- **Interviews:** allow for more free expression of ideas, time consuming
- **Phone calls and emails:** maintaining communication channels when apart
- **Concept prototypes:** for gauging user thoughts on user interface prototypes

Communication should be a mix of formal and informal communication. Formal communication accomplishes a specific task, while informal communication elicits more useful feedback about the existing system and perceived problems with the new system.

Importance of communication

Many conflicts can be avoided if good communication is carried out between the developers and the users. The users are more familiar with the current system and can provide feedback, and users will feel more confident if they have contributed to the development.

Change: Change is easier to deal with when you are fully informed. Users should be kept informed about the progress of the new system. User involvement gives them confidence about the new product and results in more successful implementation, as they are more likely to accept the final product.

Client Feedback: Lack of communication will diminish the effectiveness of the final product. Users can provide feedback on their requirements and the current development progress, and developers should incorporate the client's feedback and perspective into the software to make it more effective.

Communication Issues: Differing time zones and different countries may make it harder for consistent communication to happen in real time. The client or developer may not have good communication/collaboration skills. In prototyping, the developer may not have consulted the client enough or implemented their feedback, resulting in poor prototypes.

Quality Assurance

QA is an ongoing process throughout development to ensure the quality criteria will be met. The quality criteria is set during defining and understanding and monitored during development. The quality of the product is judged according to the extent to which the quality criteria has been met.

Software Quality: The extent to which desirable features are incorporated into software so it can reliably and efficiently continue to meet the needs of users.

Areas that should be assessed/considered when developing QA criteria:

Efficiency: The software should perform its functions making the best use of the computer's resources. Improved with optimised algorithms, well-structured code, use of appropriate language. Should be judged using a range of configurations, different OS's, RAM, CPU etc.

Integrity: The correctness of data within the system. Ensured through data validation as it enters the system, preventing system data being accessed illegally, checking that time-dependent data (e.g., phone numbers, addresses) is current.

Reliability: The ability of the system to continue to perform its functions over time. It should be able to recover from a failure quickly, and be updated in a timely manner to reflect changes in requirements or changes in the operating environment (e.g., new version of OS).

Usability: The ability of software to be learnt and used by users. Consider the design of the UI (consistency, correct use of screen elements and colour), recovery from errors, warning or undo features when data changes are made, shortcut keys etc.

Accuracy: The software should perform its functions correctly and according to its specifications. This can be ensured through thorough testing, for all possible paths and inputs. Code should be well commented to ensure accuracy of future upgrades.

Maintainability: The ease with which changes can be made to the software (correct errors, improve functionality). Code should be well documented, modular in design, and test data and routines should be retained so testing can be performance after any changes are made.

Testability: The ability to test all aspects of the software. Individual subroutines, modules, and the system should be tested to ensure it performs according to requirements. There should also be a client acceptance test to ensure reqs have been met with a live system.

Reusability: The ability to reuse code in other related systems. A fully tested module can be reused/modified as part of future projects. There are libraries of commonly used modules and subroutines, which have been tested so they are more reliable, and can be free.

4. Planning and Designing Software Solutions

- Standard Algorithms – Searches, Sorts, Strings, Random Numbers
- Data Structures and Files
- Custom Designed Algorithms
- Standard Modules (library routines) used in Software Solutions
- Interface Design in Software Solutions
- Selection of Language and Technology to be Used

This is where data structures are designed and algorithms are created. Often modules or subroutines from previous solutions or from libraries will be incorporated. Documentation such as DFDs and screen designs provide the information necessary to develop subroutines.

Top-Down Design: Breaking down a problem into smaller, more manageable modules. Each module is considered an isolated problem and written separately (abstraction). Once all modules are written, they are combined to form the whole system. Modules are advantageous because they can be reused for other programs, they make it easier to understand a complex solution, coding them is easier, and testing/maintenance is simplified. Structure charts are a good tool to show this, and each component has an algorithm written for it.

Algorithms: A method of solution for a problem. Describe the steps taken to perform a task. Can be pseudocode or flowcharts. 3 control structures: Sequence, selection, iteration.

Functions and Procedures: Each subroutine is implemented as a function or a procedure, called by higher level modules. Procedures return their results via parameters, and procedural calls are statements on their own: average(marks, result). Functions return one result via the function itself and are used as part of an expression: result=average(marks).

Standard Algorithms

Standard algorithms/modules are included in libraries of routines for future use, because many problems use similar techniques as part of their solution. Programmers don't need to reinvent the wheel, so it saves time and money. The modules are well tested and reliable.

Searches

Searching through an array, for a target/s, or to count a target/s, or for max/min.

```
BEGIN LinearSearch
  Set Count to 0
  Get ItemToFind
  WHILE more items in Item array
    IF Item(Count) = ItemToFind THEN
      Display "Found"
    ENDIF
    Add 1 to Count
  ENDWHILE
END LinearSearch
```

Linear Search: Examining each item one at a time from the start to the end.
Does not require data to be sorted.
Must examine entire list to find the number of items or if item is not found.
Can exit loop when item is found if looking for one item.
Is less efficient when the array is longer.

Maximum and Minimum Search: A linear search but a variable maintains the largest or smallest value as it goes through the list, making comparisons. Unless it's sorted. Then it's the first/last item lol.

```
BEGIN FindMaximum
  Set Count to 1
  Set Maximum to 0
  WHILE more items in Item array
    IF Item(Count) > Item(Maximum) THEN
      Set Maximum to Count
    ENDIF
    Add 1 to Count
  ENDWHILE
  Display Item(Maximum)
END FindMaximum
```

```
BEGIN FindMinimum
  Set Count to 1
  Set Minimum to 0
  WHILE more items in list
    IF Item(Count) < Item(Minimum) THEN
      Set Minimum to Count
    ENDIF
    Add 1 to Count
  ENDWHILE
  Display Item(Minimum)
END FindMinimum
```

```
BEGIN BinarySearch
  Set Low to 1
  Set High to number of items in list
  Set Found to False
  Get ItemToFind
  WHILE High >= Low AND Found = False
    Set Middle to INT((Low + High)/2)
    IF ItemToFind < Item(Middle) THEN
      Set High to Middle - 1
    ELSEIF ItemToFind = Item(Middle) THEN
      Set Found to True
    ELSE
      Set Low to Middle + 1
    ENDIF
  ENDWHILE
  IF Found = True THEN
    Display "Found"
  ELSE
    Display "Not found"
  ENDIF
END BinarySearch
```

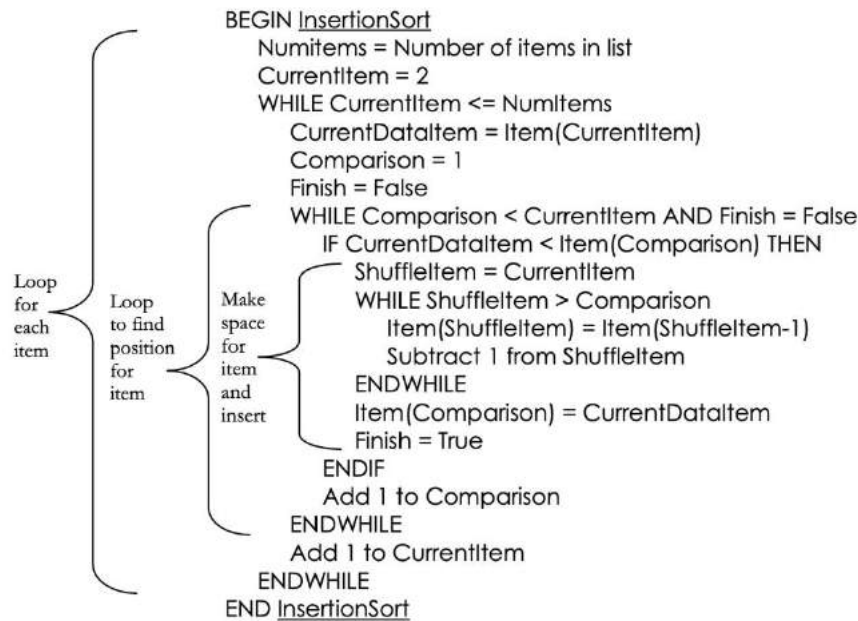
Binary Search: Splitting the list into two parts each time a comparison is made. List must be sorted. Starting from the middle, one half is discarded depending on the comparison with the middle and the target. Middle > target: top half is discarded. Faster than a linear search.

When several searches are to be done on an array of records: If there are multiple different target fields, linear searches would be faster than sorting the array each time a new target is required and then using a binary search. If there are multiple searches of the same target field, sorting and a binary search would be faster. If there are several target fields in one search e.g., field1 = A and field2 = B, a linear search would be faster.

Sorts

Arranging data into order.

Insertion Sort: Two "arrays", sorted and unsorted. The first item starts as sorted, check the next item, find where it fits into the array. Shuffle to make space, insert it in. Next item. Good for when you have a sorted list and you want to add items in it.



Selection Sort: Linear search to find the smallest item in unsorted list. Swap it with the first item: that is now the sorted list. Keep finding smallest item in unsorted list, swapping it with the first item in the unsorted list, until pass count > num items = list is sorted.

```

BEGIN SelectionSort
  Pass = 1
  WHILE Pass < Number of items
    Count = Pass + 1
    Minimum = Pass
    WHILE Count <= Number of Items
      IF Item(Count) < Item(Minimum) THEN
        Minimum = Count
      ENDIF
      Count = Count + 1
    ENDWHILE
    Swap Item(Minimum) with Item(Pass)
    Pass = Pass + 1
  ENDWHILE
END SelectionSort

```

```

BEGIN EnhancedBubbleSort
  Swapped = True
  Pass = 0
  WHILE Swapped = True
    Swapped = False
    Comparison = 1
    WHILE Comparison < Number of items - Pass
      IF Item(Comparison) > Item(Comparison + 1)
        Swap Item(Comparison) with Item(Comparison + 1)
        Swapped = True
      ENDIF
      Add 1 to Comparison
    ENDWHILE
    Add 1 to Pass
  ENDWHILE
END EnhancedBubbleSort

```

Bubble Sort: Check comparison with comparison + 1, if it's bigger then swap, keep incrementing comparison, loop back to beginning until no swaps have been made in one pass, which means it's sorted. Slower than other sorts.

Strings

Strings are data types used to hold characters or text. They are essentially an array of characters.

Concatenate string: "The " + "cat " + "is " + "fab."

Extracting: Making a copy of a section of the original string. (algorithm is inclusive)

```
BEGIN Extract(Start,Finish,ExtractString)
```

```
Temp = Null string
```

```
Position = Start
```

```
WHILE Position <= Finish
```

```
Temp = Temp + ExtractString(Position)
```

```
Add 1 to Position
```

```
ENDWHILE
```

```
RETURN Temp
```

```
END Extract
```

Deleting: Cutting a portion from the string.

```
BEGIN Delete(Start,Finish,DeleteString)
```

```
Temp = Extract(1,Start-1,DeleteString)
```

```
Length = Length of DeleteString
```

```
Temp = Temp + Extract(Finish+1,Length,DeleteString)
```

```
RETURN Temp
```

```
END Delete
```

Fig 4.18

Inserting: Inserting a string into another string. Splitting the initial string, and then concatenating first part + insert + last part.

```
BEGIN Insert(Start,InsertString,MainString)
```

```
Temp = Extract(1,Start-1,MainString)
```

```
Temp = Temp + InsertString
```

```
Length = Length of MainString
```

```
Temp = Temp + Extract(Start,Length,MainString)
```

```
RETURN Temp
```

```
END Insert
```

Random Numbers

Generating Random Numbers: Either integer or floats between 0 and 1. Needs a max/min range. $\text{randomFloat} * (\text{max} - \text{min}) + \text{min}$.

```
BEGIN randArray(min, max, num)
```

```
FOR i = 0 TO max - min
```

```
used(i) = False
```

```
NEXT i
```

```
FOR i = 0 TO num - 1
```

```
REPEAT
```

```
r = random integer from min to max inclusive
```

```
UNTIL used(r) = False
```

```
used(r) = True
```

```
arr(i) = r + min
```

```
NEXT i
```

```
RETURN arr
```

```
END randArray
```

Generating a list of unique random

numbers: well there are 3 but the third one is too confusing lol. In this one, you make an array of all possible numbers, set them to false, generate a random number and checking if that has been used (true). If not, add it to a random numbers array and set it to true.

Data Structures and Files

Efficient and appropriate data structures greatly assist in the development of algorithms.

Arrays of Records

Records have several fields, each one can be a different data type (inc. arrays). Once a record is defined, it becomes a data type, so you can have an array of records.

Contact.Age = 5

Contacts[100 contact records]

Contacts(i).Age = 5

Arrays of records are advantageous because they can be searched and sorted, and records can hold different data types of information for one item in a list.

Index	Surname	CName	Title	Street	Suburb	Postcode	DOBDay	DOBMonth	DOBYear
1	Davis	Sam	Mr	4 Data St	Romany	1234	14	5	1988
2	Hart	Rob	Mr	1 Cap Ln	Billings	9876	29	10	1951
3	Jackson	Jack	Mr	7a Pot Rd	Potfield	8765	13	9	1983
4	Unders	Wilma	Mrs	18 May St	Kingsley	1005	30	1	1977
5	Milson	Margaret	Miss	17 Jax Cr	Watson	5633	23	6	1966

Multidimensional Arrays

An array of two or more dimensions. 2d is like a table or grid. Basically each index in the array holds an array until the final one. Each element has its own unique index.

		Assessment Task Number				
		1	2	3	4	5
Student Number	1	60	65	67	68	69
	2	65	73	70	62	71
	3	72	60	58	71	69
	4	85	86	80	82	66
	5	68	74	85	90	88
	6	58	55	52	59	62
	7	70	75	70	71	69

Marks(StudentNum, TaskNum) Marks(Year, StudentNum, TaskNum)

Marks(School, Year, StudentNum, TaskNum)

Marks(1, 11, 2, 5) = marks of school 1 student 2 in year 11 of task 5

Sequential Files

e.g., text files

Data is stored in a continuous stream. It must be accessed from beginning to end. The data doesn't really have much structure.

BEGIN ReadFile

Open file for input

Read item from file

WHILE item ≠ "ZZZ"

Process item

Read item from file

ENDWHILE

Close file

END ReadFile

Sequential files can be opened for one of three actions:

Input: Read data within the file

Output: Write data to a new file

Append: Write data commencing at the end of the file

Sentinel values can be used to indicate logical breaks in the data or indicate the end of file. The program reads each line and stops processing the file once the sentinel value is reached. e.g., tabs between fields, checking for a comma, a return character, a particular string e.g., "ZZZ".

Priming read: Read the first item before the loop to detect files which only contain the sentinel, to ensure it is not processed as if it were data.

Random/Relative Access Files

The structure of the file is known, which allows the position of the start of individual records to be determined. Since each record is the same length, then the relative position of each record can be used to access individual records. Each record must be of the same length, so padding (blank) characters should be used to ensure that.

Key: Specifies the relative position of a record within the file, to allow direct access to it. For example, a key of 99 would access the 99th record in the file.

A sequential file must be searched linearly, so if there are many details, it might take too long. This is especially important if the details are updated constantly as the file has to be rewritten each time. A relative file is therefore much more appropriate as each record can be quickly retrieved and modified in place within the file. However, the ID or key must be appropriate to denote the relative position of the record in the file. If this cannot be done, then a relative file cannot be used.

Custom Designed Algorithms

Once a model of the system has been created and data structures have been designed, creating algorithms can commence. Many solutions use existing standard modules (searches/sorts etc.), but they also need custom algorithms. These are harder to create as there are many challenges, and several attempts are made before a successful algorithm.

First, the inputs, processes, and outputs must be identified. System models, IPO charts, DFDs, structure charts etc. can help.

Data Structures

Design data structures and assign data types, determine the most appropriate data structures for solving a problem. Determine which ones are global and which ones are local.

Algorithms

Algorithms provide a detailed description of the logic carried out in a program in precise steps. Writing an algorithm helps the programmer understand the logic of the problem, making it easier to convert it into code.

When new modules/functions are created, the structure charts and other relevant documentation should be updated to retain the integrity of the documentation (especially with a larger team of developers). Same as when they are modified during development.

Developing Test Data

Developing suitable test data to ensure the correct operation of algorithms. Test data should test every possible route through the algorithm and each boundary condition. It ensures that each statement is correct and works correctly with every other statement, and that decisions are correct. It also tests for syntax errors and unexpected inputs.

CASE Tools in Planning and Designing a Software Solution

CASE tools make planning and designing much easier for programmers.

System modelling: CASE tools that automate the creation of system models such as structure charts allow programmers to have a clearer understanding of the modules, their interactions, and the data involved. This is good for large teams, as they can easily see how their module fits in the overall system, and the automation of updates removes human error in updating diagrams, and frees up time, allowing them to focus on development instead of updating documentation.

Test data generation: CASE tools that can generate large amounts of test data for an algorithm make it much faster to test them. They generate appropriate test data, testing each boundary condition and route through an algorithm.

Customised Off-the-shelf Packages/Existing Software Solutions

It is generally easier, faster, and cheaper to develop a new program from an existing working program instead of starting from scratch. An existing solution is modified to meet specific user requirements.

Many custom off-the-shelf (COTS) packages allow customisation using a programming language included as part of the COTS package. These languages allow developers to add features not included in the original package. It is also common for tools to be included such as wizards or drag and drop design screens.

E.g., A package developed using Microsoft Access can add extra features using Visual Basic. MSACCESS might not have a feature to return a specific word depending on a number, so this function can be coded instead and included in the package.

Identifying an Appropriate Package: Cost benefit analysis: if modifying a package will take more time and effort than building a new solution from scratch, it is not worth modifying. Relevant products must also be identified – many applications are designed for programmers to modify.

Making Changes: Determine the portions of the program which are relevant to the solution that you are developing. Identify what changes you are going to make and how to make them. CASE tools can be used to assist. Customisation includes macros, customised data entry forms, and screen elements such as buttons.

Standard Modules (library routines) used in Software Solutions

Most programming languages have library routines that can be accessed for use in a program, such as “print”, file handling, random number generation etc.

Reasons for Development and Use of Standard Modules

Standard modules are included in libraries because many problems use similar techniques as part of their solution, so they can be easily reused. This means programmers don't have to write common modules that already exist, so it saves time and money.

The modules are also well tested and therefore their functionality is dependable, and it contains thorough documentation so that developer doesn't need to write that either. Other developers understand standard modules so less maintenance is needed.

Requirements for Generating a Module/Subroutine for Re-Use

Identification of Appropriate Modules/Subroutines

Thorough documentation (including intrinsic), standard control structures utilised, choice of language suits the modules, and the modules perform only the required task for efficiency.

Appropriate Testing using Drivers

Must test the module before becoming a standard module. Drivers are temporary code/programs used to test the execution of a module when the module cannot function individually without a mainline. For example, a driver could be written to quickly call a random number generation module 1000 times and return the amount of each output.

Thorough Documentation of the Routine

Any modules of code should be thoroughly documented. This makes it easier to understand and reuse, and makes future maintenance easier. Programmers must understand the processes of a module if they are to successfully use or modify it.

Documentation must also include the author, date, and purpose of the module, and the order and nature of parameters passed.

Issues Associated with Reusable Modules/Subroutines

Identifying Appropriate Modules/Subroutines

Modules may not be well documented, may be inefficient and/or have unnecessary functioning, or may be written using an inappropriate language.

Considering Local and Global Variables

Local variables can only be used within their own module/subprogram, while global variables can be accessed throughout the whole program. Care must be taken when global variables are in use in either the reusable module or the system in development – global variables cannot have the same name as any local variables. The scope of a variable refers to where it can be used/referenced/modified (local or global).

Appropriate use of Parameters/Arguments

A module often requires parameters to be included in the call. Parameters are data items that can be passed from one part of the program to another, where the values can be used. The parameters used should be correct and appropriate. Some constants shouldn't be hard-coded (e.g., length checking should have minLength and maxLength parameters instead).

Interface Design in Software Solutions

The design of the screens for a software project will be influenced by the nature of the problem. Interactive software needs more user-friendly screens, as the layout and selection of screen elements will have a large effect on the productivity of the user. Software projects where most of the product operates in the background need simple screens as they are used less often and have minimal impact (e.g., virus scanners, drivers).

In event-driven software, the screens are crucial as the user decides the order of processing. In sequential software, the order is determined by the developer, so the user has less input and screens are more specific.

Factors to be considered when developing user interfaces include data fields and screen elements, current popular approaches, design of help screens (online help), the intended audience, consistency in approach, and social and ethical issues.

And the screen size and the devices – there should be different designs for different devices (e.g., phone screen vs laptop screen), and screen resolutions.

Visual Disabilities: Have a text label/description for each image, option to adjust font size, don't rely solely on colours to convey meaning.

Data Fields and Screen Elements

The data included on each screen needs to be identified. Then the most effective screen element to display each data item can be determined.

A consistent screen design will make the user experience and the program easier to use, and will reduce development time as a common module can be called to display the same common information on each screen.

Menus



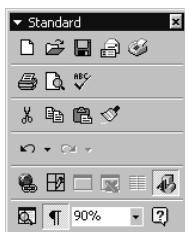
Used to initiate execution of a particular module within a program. Drop down menus are used in most GUI based products.

Command Buttons

Used to select a different path for execution. Often used to confirm a task should take place.



Tool Bars



Used to access commonly used items and functions quickly and easily. Each tool is represented as an icon. Can be horizontally at top or bottom of screen.

Text Boxes

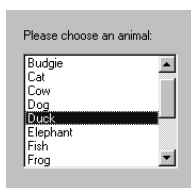
Name:

Initials:

Mailing address:

Used to receive string input. Numeric data is converted to a string. Data validation must be in the code, as it is not self-validating. Labels describe input required. Used when other, self-validating elements aren't suitable.

List Boxes



Force user to select from given options. No unexpected inputs, so it is self-validating. Scroll bars to scroll through available options and save

space. Data can be from an array, and data returned can be an index.

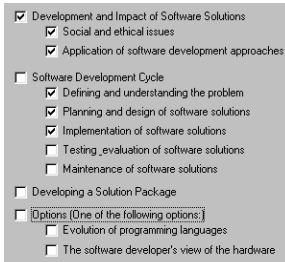
Combination Boxes

Please enter an animal name:

Please enter an animal name:

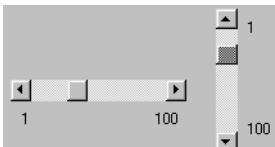
Combine functions of text and list box. Force input of items in list or set so user can enter their own item that can become part of the list, or search for an item in the list.

Check Boxes



Used to obtain Boolean input. Clicking it changes the value from true to false or vis versa. Self-validating. Can select multiple options.

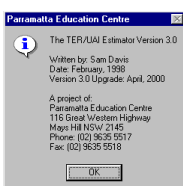
Scroll Bars



Display the position of a numeric data value within a given range. Often used to navigate

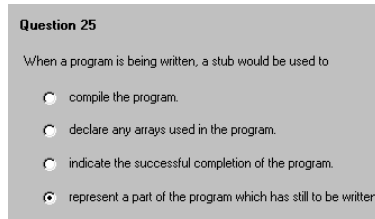
within another screen element. Give user a visual interpretation of their relative position. Current position is returned as a numeric value within a given range.

Labels



An output screen element. Provides information and guidance to the user, often instructions regarding inputs.

Option/Radio Buttons

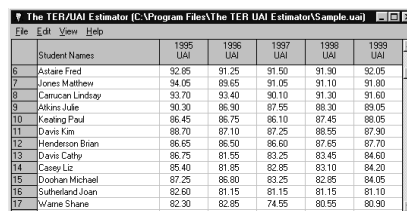


Force the user to select only one available option. Used for multiple choice or selections.

Index of option selected is returned.

Grids

Present data in the form of a table. Each cell is an individual text box – needs validation.



	1995	1996	1997	1998	1999
Student Names	UIAI	UIAI	UIAI	UIAI	UIAI
6 Austen Fred	92.05	91.25	91.50	91.90	92.05
7 Jones Matthew	94.05	93.65	91.05	91.10	91.60
8 Camucan Lindsay	93.70	93.40	90.10	91.30	91.60
9 Atkins Julie	90.30	86.90	87.55	88.30	89.05
10 Keating Paul	86.45	86.75	86.10	87.45	89.05
11 Davis Kim	88.70	87.10	87.25	88.55	87.90
12 Henderson Brian	86.65	86.50	86.60	87.65	87.70
13 Davis Cathy	86.75	81.55	83.25	83.45	84.60
14 Carey Liz	95.40	91.85	92.85	93.10	94.20
15 Doohan Michael	87.25	86.80	83.25	82.85	84.05
16 Sutherland Joan	82.60	81.15	81.15	81.15	81.10
17 Warrne Shane	82.30	82.85	74.95	80.95	80.90

Referenced with the row and column as indexes, like a 2D array.

Pictures/Image Boxes



Used to display graphics. This is an output screen element.

Use whitespace to separate logical parts of the form.

Current Popular Approaches

Different solutions will need different approaches to design the user interface. The design of the screen elements is often dictated by the chosen OS, especially with GUI-based OS's.

- **Internet solutions** will be executed on a large and unknown selection of systems, so they need solutions that are suitable for many operating systems.
- Software for **dedicated devices** such as mobile phones, microwave ovens, video recorders etc will require unique interface designs.
- **High-resolution monitors** give more colour and resolution, so they need more detail on the interface.

Design of Help Screens

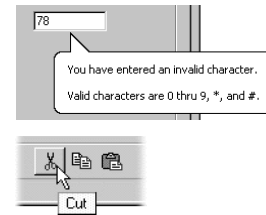
Help screens are used to assist the user with the operation of the software product.

Context Sensitive Help

By examining the screen element the user is currently working with, help can be provided.

If the user specifically asks for help, a new window can be displayed containing information. Normally the help screen does not take up the whole screen; the application is still visible.

Simple tips in a small window can display relating to the current screen element if the user hovers the mouse over the element. After the mouse moves off, the tip disappears. For example, tool tips and balloon text.



Procedural Help

Provide concise instructions on how to do a specific task. Often lists a series of steps to complete the task. Explains 'how' to do a task.

The help screen to access procedural help should contain a contents, index, and search facilities to allow the user to quickly locate required information.

Conceptual Help

Explains 'what' and 'why' rather than 'how' to do a task – the concepts behind a task. For example, why a file should be saved, instead of how it is saved.

Often accessed from a procedural help topic e.g., through a link in the description of steps.

Tours, Tutorials, and Wizards

Tours are used to give new users an overview of the product's purpose and basic features and capabilities. They are often multimedia presentations and separate to the actual software product.

Tutorials provide instructions on how to do tasks. Step-by-step instructions are given with examples such as screenshots, or leading users through the example task with sample data.

Wizards are used to automate complex tasks. A series of dialogues guides the user to supply the required inputs and the wizard completes the task for the user.

Standards and recommendations for the design of UI for these are provided by many leading OS companies. Many CASE tools assist in the creation of these screens.

Intended Audience

The UI needs to be designed to suit the intended audience. A product for schoolchildren will need a different interface compared to one for engineers. Products used by specific people e.g., accountants can use relevant jargon, but products for the general public shouldn't.

Systems intended for large audiences with unknown levels of expertise may include UIs that can be personalised, e.g., choosing experience levels for different message wording.

Feedback from users should be sought and used during interface design before it is finalised. Design specifications provide a framework for the final implementation of the UI.

Consistency in Approach

Consistency allows the software to feel familiar and predictable, and make it easier for the user to learn, use, and transfer existing skills. UIs should be consistent across the application, and similar elements should be similar to other applications.

Names of commands: Common command names are consistent, e.g., cut, copy, and paste.

Icons: Should be the same as other applications, e.g., floppy disk for save.

Placement of screen elements: Changing the placement would reduce efficiency, e.g., close button is always on the top of the screen, save button in toolbar.

Feedback: Users expect a response or feedback after an action. Should be consistent e.g., always using progress bars.

Forgiveness: Warnings and recovery methods should be consistent. The interface should allow users to cancel operations before damaging results happen.

Selection of Language and Technology to be Used

Selecting the Programming Language to be used

Algorithms should be written in a way that could be implemented in a variety of languages.

The most appropriate programming language should be selected after considering criteria:

- Sequential or event-driven
- Does the language provide for all the required features of the solution?
- Hardware and operating system ramifications
- GUI required
- Expertise of developers

Sequential or Event-Driven: Event-driven languages allow the user to choose the order in which processing occurs (e.g., a word processor). Each process is initiated by the user, e.g., by selecting an item in a menu which executes a code module. Sequential languages ensure that software executes in a distinct order, decided by the programmer. The user must follow a route through the software. Many problems involve a combination of approaches, mostly, event-driven software combines a number of smaller sequential modules.

Solution Features: Different languages have different features to solve different problems. The appropriate language will reduce development times and increase product efficiency.

Hardware and OS: Many languages are designed for particular OS's, and OS's are specific to particular hardware. A product developed for one OS environment might need changes to operate in another OS environment. Certain programs may be more suited to different hardware types. The language should have commands to interface with required hardware.

GUI: If the software operates with minimal user interaction, the GUI overhead is not necessary. Some languages are not suitable for developing GUIs.

Developer Expertise: Learning a new language is difficult and time-consuming. If a new language is used, the development times will need to be extended.

Selecting the Technology to be used

Hardware also has an impact on the performance of the software.

Performance Requirements

When comparing the performance of different hardware components, such as CPUs, there are many factors that should be considered and not all are relevant for all systems.

Performance factors include:

- Speed
- Fault tolerance: can it recover from power spikes/failures?
- Can it perform under load? E.g., many users accessing a database at the same time

Specific performance requirements should be established so tests can be developed.

Benchmarking

The process of evaluating the product by comparing it to an established standard. This enables competing products to be fairly compared. Software and computer component standards are developed to assess performance requirements found to be important to many users, e.g., the ability to render 3D graphics in HD at a particular speed (for gamers). Often the benchmarking software is able to record technical details of variables so a fair mathematical and statistical evaluation can be made.

It involves looking at the performance of systems using similar technologies under controlled conditions. This way, developers can look at the relative performance of a range of similar programs/algorithms, and thereby assess their system in terms of speed of their programs/algorithms.

5. Implementation of Software Solutions

- Programming Languages and Language Syntax
- Translation Methods in Software Solutions
- The Role of Machine Code and the CPU in the Execution of a Program
- Techniques Used in Developing Well-Written Code
- Documentation of a Software Solution
- Hardware Environment to Enable Implementation of the Software Solution
- Emerging Technologies

Programming Languages and Language Syntax

Low Level Languages: Machine language and assembler language, machine dependent, difficult for humans to understand. Assembler has mnemonics like ADD.

High Level Languages: 3rd+ generation languages, machine independent (translation), easier for humans to understand, control structures and built-in commands.

Choosing an Appropriate Programming Language


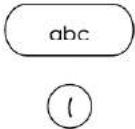

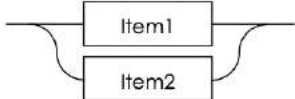
For some projects there will be little choice due to the targeted environment and device. Web-based software commonly has the server-side code, commonly PHP, and client-side code, which often runs within the user's browser, commonly JavaScript.

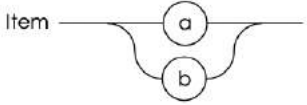
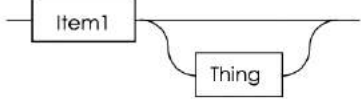
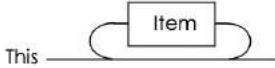
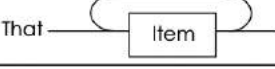
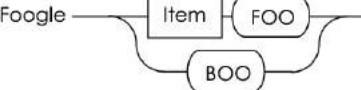
- **Nature of Project:** Different languages for different types of problems, e.g., AI application vs online store vs hardware driver vs platform game.
- **Intended Environment:** Hardware, OS, network use. Servers vs client web browsers, languages that can execute on different OS's, languages for smart phones etc.
- **Experience:** Takes time to learn a new language. Expertise comes with time.
- **Event/Sequential:** Event-driven requires GUI, executes modules as initiated by user.
- **Maintainability:** Does language enforce well-structured modules? Is there a large community of programmers and libraries of code for ongoing support?

Language Syntax Required for Software Solutions

Metalanguage: A language used to describe the rules/syntax of a programming language, to allow developers to understand the structure of the language and how it works.

EBNF and Railroad Diagrams

Interpretation	EBNF example	Railroad Diagram example
Terminal Symbol for a reserved word. BEGIN is a reserved word.	BEGIN	
Terminal symbol for a literal. The characters abc are written as they appear. Quotes are used to enclose symbols used by the metalanguage.	abc "	
Non-terminal symbol. Item is defined elsewhere.	<Item>	
"or" a choice between two alternatives. Either Item1 or Item2.	<Item1> <Item2>	

"is defined as".Item can take the value a or b	Item=a b	
Optional part. Item followed optionally by a Thing.	<Item>[<Thing>]	
Possible repetition. This is an Item repeated zero or more times.	This={<Item>}	
Repetition. That is an Item repeated one or more times.	That=<Item>{<Item>}	
Grouping. A Foogle is an Item followed by the reserved word FOO or it is the reserved word BOO.	Foogle=(<Item>FOO) BOO	

Metalanguage Descriptions of Programming Languages

In the exam, refer to the stimulus (metalanguage desc of a language) provided.

Translation Methods in Software Solutions

Translation is the process of converting high-level or source code into machine executable code. High-level languages cannot be understood directly by processes, so they must be converted to another form.

Source code: Machine independent, used on different processes (with translation).

Machine code: Each processor needs different machine language instructions.

The WWW needs code that can run on an unknown processor. Some languages have two-stage translation, where source code is translated to byte code, which is then translated by a web browser to execute code specific to that computer.

Translation Method	Desc	Advantages	Disadvantages
<i>Interpretation</i>	Each statement is translated and run immediately	Find and correct errors quickly	Slower, requires interpreter installed, less IP right control
<i>Compilation</i>	Whole program compiled into one executable	App runs faster & efficient, hard to determine original source code	Takes more time to compile full program, programs are machine specific

Incremental Compilation: Parts of code that have been modified since the last compile are recompiled. Can reduce time spent in recompiling source code.

Interpretation

Each line/statement of source code is translated into machine code and then immediately executed. If errors are encountered, execution is halted. This allows errors to be found and corrected quickly, so it is useful for debugging.

Interpretation is slower as each statement must be translated one at a time. Users of interpreted programs must have the interpreter installed on their machine for execution, which involves further costs and memory overheads. The actual source code is distributed to users, so the developer has limited control over their IP rights.

Many software development environments have an interpreter which allows the programmer to execute a small section of code quickly while they are coding, to help with development. The final distributed product is produced using a compiler.

Compilation

The source code is translated into executable code (file), which can be later executed without any need for a translator. Errors encountered during the compilation process are reported as messages to the programmer/at the end of an unsuccessful compilation.

A compiler does not produce any actual machine code until the entire source code is known to be correct in terms of syntax.

Compilers are used to produce executable code for most commercial apps. Compiled programs generally run faster and more efficiently than interpreted ones. As it is made of machine code, it is very difficult to determine the original source code (protects IP rights).

Executables created with compilers are always machine specific. To operate on different processors/OS's, the source code needs to be recompiled with different machine code.

Steps in Compilation Process

Creation of object code from original source code statements, then linking this code to existing runtime libraries and dynamic link libraries (DLLs). In larger projects, several translated object code files are linked together into one final executable program.

Lexical Analysis

Analysing and checking that all reserved words, identifiers, constants, and operators are a legitimate part of the high-level language. Order of language elements doesn't matter.

A table of symbols and tokens contains all reserved words and operators that are included elements of the language. As lexical analysis continues, identifiers and constants are added to the table. The source code is read sequentially, and once a string of characters matches an element in the symbol table, it is replaced by the appropriate token. Lexical analysis ignores comments and white space.

Each time a constant appears it is assigned its own token. Thus, it is better to assign the value of frequently used constants to an identifier and use it instead.

Errors involve the use of undeclared identifiers, incorrect naming of identifiers, or incorrect constant syntax. E.g., identifiers encountered during lexical analysis must conform with the language syntax for an identifier.

Syntactical Analysis

Checking the source code statements adhere to the programming language's syntax rules, and that the type of identifiers and constants used in statements are compatible.

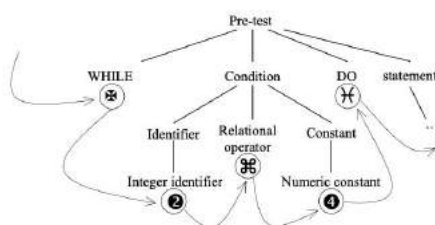
Tokens are delivered for syntactical analysis, and are used to create a parse tree, which diagrammatically describes the component parts of a syntactically correct statement. The parse tree is constructed using the tokens, based on rules for the programming language described in metalanguage. Each statement will have exactly one parse tree, resulting in one precise meaning. Each statement in the source code is checked against the precise description of the syntax using the parse tree and metalanguage.

If a component of the source code cannot be parsed, an error message is generated, always in regard to the incorrected arrangement of tokens.

Once a parse tree has been created, type checking occurs, where the data type of identifiers and constants within each statement are checked for compatibility. E.g., an identifier to store simple integers cannot be assigned a string. Errors found during type checking are reported as type mismatch errors.

Code Generation

Machine code is generated if no errors occur during lexical and syntactical analysis.



The parse tree created during syntactical analysis contains tokens in their correct order. The tree is traversed, and the tokens are collected and converted into machine code instructions.

If a compiler is being used, the machine code instructions are stored in an object file, which contains all the executable code derived from the source code. They generally do not have links to runtime libraries and DLLs – a linker is used to do this. Most compilers have a linker as part of the compilation process.

If an interpreter is being used, once each instruction is generated, it will be executed immediately. The interpreter takes care of access to runtime libraries.

The Role of Machine Code and the CPU in the Execution of a Program

Machine Code

The native language of the CPU, a collection of machine language instructions. Each instruction is in binary.

The CPU contains a series of microcode instructions which are permanent and cannot be modified, and a machine language instruction translates into one or more microcode

instructions. Different CPUs contain different microcode instructions, so they require a different set of machine language instructions.

Instruction Format

Made up of 32 or 64-bit strings of 1s and 0s.

- Opcode: The instruction
- First operand: The memory address of where the data is to be stored
- Second operand: The data

Assembler Code

A way of writing machine code that can be understood by humans.

One assembly language statement corresponds directly to a machine language statement. Assembler programs execute faster than higher level languages.

Assembler code fragment for Intel 8086 processors. Code waits for escape key to be pressed and then exits the program.

```
escwait: mov al,0
         int 16h
         cmp al,1bh
         jnz exit
         jmp escwait
exit:    mov ax,3
         int 10h
         mov ax,4c00h
         int 21h
```

Two procedures/labels: escwait and exit. Labels in assembler are markers in the code that can be returned to using a jump.

In 8086 assembler the accumulator is signified by the letters ax. This code operates on 16-bit processors, so each register can be split into two bytes. In assembler, these parts of the

escwait: mov al,0	Moves the value 0 into the register al. This initialises the al register. (Escwait: is a label, we need to return to this position later).
int 16h	Interrupt 16h is the keyboard. This statement returns the value of the key pressed to the accumulator. The ASCII code of the key is held in al and a code signifying the particular key to ah. For example typing an "a" results in ah=1eh and al=61h=97, 97 is the ASCII code for "a". Typing an "A" results in ah=1eh and al=41h=65, 65 is the ASCII code for "A".
cmp al,1bh	Compares the value in al with the value 1bh. If they are equal then the flag called nz is set to true. 1b in hexadecimal is 27 in decimal, which is the ASCII code for the ESC character generated by pressing escape on the keyboard.
jnz exit	J stands for jump, in this case a jump to the statement following the label exit is performed if the flag nz is true. True is represented by a binary 1 and false by a binary 0. The flag nz is a single binary digit.
jmp escwait	An unconditional jump to the statement following the label escwait is performed.
exit: mov ax,3	The value 3 is moved into the accumulator.
int 10h	The screen is instructed to read the accumulator value 3. This clears the screen. A nice thing to do before we exit!
mov ax,4c00h	Place the value 4c00h into the accumulator.
int 21h	Interrupt 21h is used to communicate with the operating system. The value 4c00h held in the accumulator causes the program to terminate and hands control back to the operating system.

accumulator are named ah (accumulator high) and al (accumulator low). E.g., if ah = 11000110 and al = 00111111 then ax = 1100011000111111.

Statements are executed by the ALU and the result is stored in the accumulator.

So basically just use the instructions and desc given in exam stimulus.

Translating Assembler code to Machine code Instructions

Mostly, an individual assembler code statement is translated into an individual binary machine language instruction. Each CPU design has its own instruction set (list of microcode procedures permanently stored in processor).

The instructions are permanently stored in the processor and accessed using its hexadecimal code. The microcode instructions contained in the CPU explain how to execute the command. Each machine language instruction activates a microcode procedure that takes one or more microcode steps/instructions. Each microcode instruction is executed using the fetch-execute cycle and integrated circuits within the CPU.

CPU

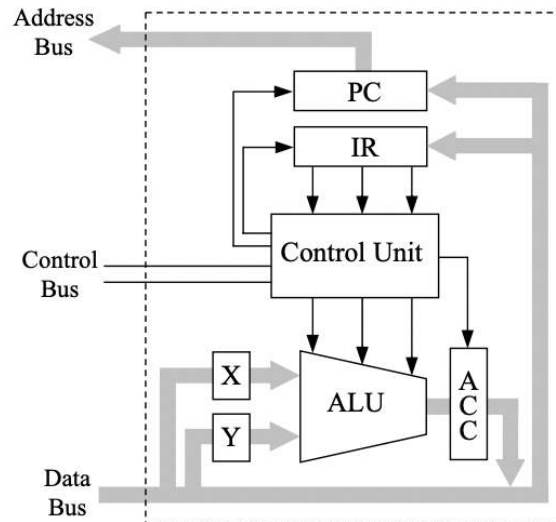
Executes each inbuilt microcode instruction. Three basic units: control unit (CU), arithmetic logic unit (ALU), registers. Each component is made up of circuits. Communication is done via a series of connections called buses.

Registers are temporary storage locations/circuits. The size is the same as the word size of the processor (how many bits it can process at a time).

CU: Directs system to carry out instructions, controls data moving through processor, timing of operations, and instructions sent to the processor.

ALU: Performs calculations and arithmetic & logic operations.

Accumulator: A general purpose register used to accept the result of computations performed by the ALU.



Instruction Register: Holds the instruction currently being executed

Program Counter: Holds the address of the instruction currently being executed. As each instruction gets fetched, the stored value increases by 1.

In the diagram, **X** and **Y** are other registers to store temporary data for quick access.

Interrupts

Communication with outside devices is done with interrupts. An interrupt is a communication channel into and out of the CPU. Each channel is assigned a unique value. E.g., interrupt 16h receives data from the keyboard and places it in the accumulator.

Flags

A flag is a single bit stored in a register, used to indicate an event has occurred in the CPU.

Fetch-Execute Cycle

1: Fetch instruction from primary memory and place in the instruction register.

2: Set the program counter to the next instruction's address. As instructions are stored sequentially, the program counter is incremented by 1 (actually incremented by the instruction size in bytes)

3: Decode the instruction (determine nature of it), performed by the control unit.

4: Load operands from memory (if required) into a register. E.g., an add instruction includes an operand (number) to add. A store instruction includes an operand (memory address).

5: Execute the instruction and store the result, performed by the ALU. Normally the result is stored in the accumulator. Tasks with more than one result use other locations. E.g., an add with carry stores the main result in the accumulator and sets the carry flag appropriately.

6: Reset the fetch-execute counter back to 1, to commence the process again from step 1.

The CU sends a signal to the appropriate component via the control bus to begin each step. E.g., in step 5, the CU sends a signal to the ALU indicating which microcode instruction needs to be executed. The CU sends one signal for each tick of the system clock.

2GHz = 2 billion ticks per second. 800MHz = 800 million ticks per second.

Execution of Called Routines

Commands to be executed are stored with the operation code in the computer's RAM. The stack is a special register that keeps records of where the CPU is up to in the current operation it is carrying out. 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.

Linking

Subprograms include:

- Individually made ones by a programmer to be used in one application e.g., high score subprogram to calculate and store new high score
- Subprograms included within a programming language development package. Part of a library and can be accessed by programmer to perform simple tasks quicker e.g., randomise number program, just call it from the development library
- Subprograms built into the OS. Creates universal features e.g., save, command

Linker: Joins subprograms to the mainline of the program.

Dynamic Link Libraries (DLLs) are files containing object code subroutines that add extra functionality to a high-level language. Used to reduce the need for multiple subprograms. They allow appropriate system modules to be accessed at run-time.

Techniques Used in Developing Well-Written Code

To reduce number of errors in final product and improve maintainability of software.

Good Programming Practices

Planning

Lots of planning beforehand, modelling software systems, creating clear algorithms.

Use of a clear Modular Structure

Top-down design, described in structure charts. The mainline should be clear so it is easy to follow and identify calls to subroutines. Each subroutine can be coded independently and tested by itself before tested in conjunction with other subroutines.

If no global variables are used in the program, then data shared between modules is passed as parameters. This gives developers more flexibility in variable names and reduces conflict arising from global variables sharing local variable names.

One logical Task per Subroutine

A short succinct name to describe the processing in the subroutines. Long algorithms are difficult to understand, debug, test, and maintain. If an error is linked to a particular task, it is easier to isolate the source of error if each subroutines has one logical task.

Restricting subroutines to one logical task enhances the reusability of the code.

Quality of Solution

A quality solution has better performance, design, and maintainability. QA strategies and good problem-solving abilities ensure understandable and efficient quality solutions.

Appropriate use of Control structures and Data structures

Only use recognised statements e.g., pre-test and post-test loops, condition should determine the only exit from the loop etc.

Programmers should consider unusual cases e.g., reading empty file, unexpected values, infinite loops.

Data structures (arrays, records etc.) should be selected and designed to assist the processing. E.g., using one record with 4 related fields instead of 4 arrays of related data.

Writing for subsequent Maintenance

Coding done in a way that future maintenance programmers can easily update the code. Clear documentation (comments, identifiers), using an identifier instead of a constant value.

Version control and regular Backup

Regularly save work to prevent loss of source code and allow programmers to revert to earlier versions if needed.

Version control allows complete applications to be continuously developed with regular versions being released to users. The version control system maintains copies of all modules for a specific version of the application, so any previous version can be compiled.

VCS's can also allow multiple developers to work on the same project. Developers must check out the file they wish to modify, and the system maintains a record of every change made and by which developer it was made by.

If a developer does not regularly back up work, they will still be held liable to produce the work required by their contract, so they will be under more pressure (resources, finances, time) to deliver the contract requirements as they have to re-do lost work.

Recognition of relevant Social and Ethical Issues

Issues that should be considered when writing source code include:

- Respecting IP rights when using code written by others
- Thorough testing to ensure software performs correctly and as expected
- Documentation for future maintainers of the software

Errors Occurring in Software Solutions

Syntax Errors

Detected when the code is translated. Any errors that prevent the translator converting the high-level code into object code. Can occur in both lexical and syntactical analysis.

Lexical analysis: sequences of characters that cannot be represented as tokens. E.g., identifiers that have an exclamation mark or other non-letter/digit characters.

Syntactical analysis: syntax is incorrect. E.g., a missing bracket or reserve word, typos.

Most programming environments check the syntax of statements as they are entered. Once a syntax error has been detected, many interpreters and compilers inform the programmer how to correct the error.

Logic Errors

A result of syntactically correct code that does not complete the required task. E.g., continuous loops, incorrect calculations or conditions. Logic errors can cause the program to halt, generating a runtime error.

Thorough testing of algorithms decreases the number of logic errors during implementation.

Logic errors are more difficult to correct. The development environment cannot detect logic errors. Debugging techniques assist in identifying the source of logic errors.

Runtime Errors

Errors detected by the computer while a program is executing. Can be linked to either hardware or software problems. Can result in the computer crashing.

Possible sources of software runtime errors:

- **BIOS (Basic Input Output System):** The BIOS provides the interface between the OS and the hardware. BIOS settings are stored in CMOS (type of RAM). Incorrect CMOS settings often result in runtime errors.
- **Operating System:** Runtime errors generated by the operating system can result from various sources (BIOS, hardware driver, application software errors).
- **Hardware Drivers:** The driver provides an interface between the OS and any hardware devices installed on the computer (hard disks, printers etc). If the hardware driver generates a runtime error, it may cause the OS to generate a runtime error.
- **Application Software:** Runtime errors generated by apps have a number of causes:
 - ◆ **Arithmetic overflow:** When a value is assigned to a variable that is outside the range allowed for that data type.
 - ◆ **Division by zero:** Often caused by unexpected inputs (e.g., calculating the average of an empty list)

- ◆ **Accessing inappropriate memory locations:** When an attempt is made to assign a value to an identifier that either does not exist or is of a different data type (mismatch), or assigning to an index outside of an array's range.
- ◆ **Logic Errors:** E.g., stack overflow runtime error from recursive calls.

Techniques for Detecting Errors in Code

Stubs

A small subroutine used in place of a yet to be coded subroutine. Allows higher-level subroutines to be tested as they are being coded, as in top-down design, higher-level subroutines are created first. They do not perform any real processing – instead, they simulate the processing in the final subroutine they are replacing.

- Set the value of any variables affecting their calling routines
- May include an output statement to indicate the call to the stub was successful

Drivers

Provide an interface between two components. A driver in this context is a subroutine written to test the operation of other subroutines. Used in bottom-up design, where lower-level subroutines are written first and tested with drivers.

- Set the value of any required variables
- Call the subroutine to be tested
- Output the results of the subroutine call

Flags

Used to indicate that a certain condition has been met. Usually a Boolean variable.

- Check if certain sections of code have been executed (set to true if called)
- Check if certain conditions have been met
- Can determine the flow of control through the program

Debugging Output Statements

Strategic placement of temporary output statements can help isolate the source of error.

- Determine which subroutines have been called and monitor the flow of execution
- Display the contents of variables at certain stages during execution, to monitor changes in variables during execution, and identify the place of error in calculations

Peer Checking

Other team members analyse each other's work to detect and correct errors, offering a different point of view that may notice more errors.

Desk Checking

Working through a piece of code by hand, with a table of variables. The values of the variables are changed as the code is stepped through. Helps to understand code.

Structured Walkthrough

Formal meetings. Developer(s) present their work to a group of interested parties (management, marketing, potential users) to demonstrate the software and formally work through its functionality.

- The developers walk the group step-by-step through each aspect of the program. No corrections are made, only comments are taken, and feedback is given.
- Each person should be given relevant documentation, and comments are only considered after the meeting
- Used to evaluate the design at different levels (algorithms, modules, final product)

Use of Expected Output

Expected outputs for test data should be calculated by hand. The results of the test data inputs in the final subroutine are checked against the expected output. If they match, the program is logically correct.

Test driven development (TDD): thorough tests are written before coding, passing these tests means the source code works. As new subroutines are coded, new tests are added. All tests can be re-run each time new code is written or modified.

Software Tools for Detecting Errors in Code

Software debugging tools to help isolate errors in the source code.

Use of Breakpoints

Temporarily halt the execution of code to examine the current value of each variable.

Resetting Variable Contents

Altering the value of variables during the execution of the code can help determine the nature of the error, or which values are causing errors.

Can be done when code is halted temporarily, or by inserting assignment statements within code, to control the path of execution.

Program Traces

Enables the order of execution of statements to be tracked, or the changing contents of variables to be tracked.

- Display each line of code as it is executed at a slow speed
- Analyse the call stack (list of subroutines that have been called)
- Produce a trace file (log of all transactions that have taken place)

A window in the development environment containing all local variables and their current value and data type. Can be used in conjunction with breakpoints/line stepping.

Single Line Stepping

Process of halting execution after each statement is executed, so the programmer can see the effect of each line separately. Can be modified e.g., step over procedure and function calls, completely execute procedures and functions before execution halts again.

Watch Expression

An expression whose value is updated in the watch window as execution continues. Usually variable names. Combined with single line stepping, it creates an automated desk check.

Documentation of a Software Solution

Documentation should be continually updated during the development process.

Technical Documentation

Designed for use by professionals with expert knowledge. Describes structure and engineering behind the product. Used by personnel when maintaining/upgrading solutions.

Logbooks (or process diaries)

Chronologically records processes and actions undertaken during development. Can be for individual members or a whole team. Logbooks are reflective, so mistakes, delays, or inefficiencies during development can be analysed to more effectively manage projects and ensure that such issues do not continue to arise.

Entries include:

- Date
- Description of progress since last entry
- Issues and possible approaches to manage them
- Reflective comments

Source Code Documentation (internal documentation)

The most important form, well documented code is useful for future maintenance.

Comments summarise and explain what each section of code does, not how it does it. They give information for future programmers and are ignored by the translator.

Intrinsic documentation involves using meaning identifiers for variables, functions etc. They should describe the purpose of the element and should adhere to the conventions in the programming language (e.g., text boxes start with txt).

Formatting (also intrinsic) involves indenting sections of code within control structures to improve readability and visually describe the logic. It also includes leaving blank lines between comments and the code, and using different colours for different elements.

Other Forms

Various forms of documentation are created through the SDC, and should be edited to reflect changes made.

CASE tools assist in ensuring technical documentation reflects the final nature of the product. E.g., system models, data dictionaries, design specs, for future maintenance.

User Documentation

Should be directed at the target users' level of knowledge and expertise.

User Manual

Provide information about the operation and purpose of software. Includes:

- What tasks the software can complete
- Why these tasks are required (conceptual explanation)
- How these tasks are completed using the software (procedural explanation)

- Screen dumps to assist with procedural explanations

Topics can be ordered in terms of difficulty or in order of usage e.g., common tasks first.

The format must be accessible and consistent. Often much of the information is included in help that is directly accessed from within the application.

Reference Manual

Read in a random order to quickly reference a command in the application. Includes:

- Every command within the application in alphabetical order (easier to locate)
- A brief description and example of the command
- Short cut keys for the command

Installation Guide

Provide the user with sufficient information to successfully install the software product on their machine. Includes:

- Hardware and software requirements, minimum and recommended specs
- Step-by-step instructions to install, e.g.:
 - ◆ Download the installation package and let the installation program auto-run
 - ◆ Configuring the system for the particular installation site
 - ◆ Installing hardware/cables for networks
- Notes addressing common installation errors and support contact details

For large systems sometimes an installer from the software company performs or directs the installation process.

Tutorial

Provide instruction in the use of software products using example scenarios. The user is led step-by-step through the processes included in the product, and performs the tasks under the direction of the tutorial. Includes:

- Sample data files to use during tutorials, to familiarise users with the functions without the risk of damaging real data
- Content split into individual sections/lessons focusing on a particular set of skills

Online Help

It is now more common for most user documentation to be online rather than printed.

Hypertext allows the user to efficiently search for specific items. Other forms:

- Help stored locally on user's machine
- Help stored on remote server requiring internet connection, but software company can update help files and all users can access the new content immediately
- Interactive help such as user support forums and chats

Hardware Environment to Enable Implementation of the Software Solution

All software solutions are designed with particular hardware requirements in mind. The installation guide includes specifications regarding minimum requirements.

- **Large custom systems** are usually designed for a specific hardware configuration
- Solutions designed for a **broad market** need to operate on a variety of configurations

Hardware specs must be specified and tested before software solutions are released.

Minimum Hardware Configuration

Will vary for different software solutions. Should allow the software to operate successfully with acceptable performance and response times (testing to ensure it meets expectations).

- Processor type and speed
- RAM size and type
- Secondary storage size and access speed
- Input and output devices
- Network hardware

Possible Additional Hardware

Items supported by the software product but not essential for its operation.

- Extra RAM to operate with larger files
- Sound cards for optional audio feedback for visual disabilities
- Network/internet connection for foreign exchange rates to be updated electronically
 - Or for online gaming, communicating with others etc

Appropriate Drivers or Extensions

Hardware devices require software drivers (aka interfaces, convert signals from one device into those understood by another) to allow them to communicate with the system. Before a hardware device can be used by a software product, its driver must be correctly installed and configured. Most devices come packaged with drivers for popular OS's or can use drivers included with the OS.

Custom built hardware may require a purpose-built driver to be created. The driver is then an extension to the software package.

Emerging Technologies

New advances in hardware and software mean that software developers must remain up to date with current and emerging technologies to remain competitive. They provide new opportunities for software development.

Quantum Computers

Currently a theoretical possibility, but many companies are investing heavily in research and development. They will have incredibly fast processing. Also they will potentially break encryption codes.

Human-Computer Interactions

Aim is for the computer to simulate human interactions (eye movement, facial expressions, touch, voice).

- Eye movements can be followed and analysed to move a cursor, POV in virtual world
- Body movements can direct computers and game consoles e.g., dance movements and shooting directions (where the user is pointing)
- Voice recognition, detecting lip movements

Devices with new methods for human-computer interaction: game consoles, smart phones, touch screens, scanning pens etc.

Internet Access Initiatives

More flexible access to the internet and utilise the internet in innovative ways.

- Social networking software
- Location based applications, real time GPS coordinates
- Wireless connectivity

6. Testing and Evaluating of Software Solutions

- Testing the Software Solution
- Reporting on the Testing Process
- Use of CASE Tools
- Evaluating the Software Solution

Testing and evaluating to ensure the product meets the original objectives and design specifications, and that it functions correctly. It is integral to all stages of the SDC and QA.

Alpha Testing: Testing of the final solution done by personnel within the development company prior to the product's release.

Beta Testing: Testing of the final solution done by a limited number of outside users using real world data and conditions. The users report faults and recommendations.

Black box testing: Functional testing. Inputs and expected outputs are known; processes occurring are unknown. Used to identify if a problem exists.

White box testing: Structural/open box testing. Explicit knowledge of the processes is needed. Used to identify the source and correct the problem.

Prior testing in SDC: (this stage uses mostly black box, previous use mostly white box)

- **Desk checks:** ensure algorithms operate as intended
- **Analyse algorithms:** evaluate their efficiency
- **Pre-designed tests, test sets with input and expected results:** allow for continuous testing/checking of subroutines/modules
- **Debugging tools/techniques during implementation:** test code for errors

Testing the Software Solution

Comparison of the Solution with the Design Specifications

The aim of the project is to implement the full set of design specifications, so they must be tested and compared. Design specifications were developed as a framework for the development process and need to be tested against the final product.

The original specifications and requirements may have been modified during the development of the product.

A checklist of all requirements is made.

- Each requirement is tested
- If faults are encountered, then that aspect is altered until the requirement is met
- The results of the testing process will uncover inconsistencies in approaches that have occurred during development

Generating Relevant Test Data

Operational factors outside the control of the programmer include user/environmental inputs, variations in hardware e.g., CPU, graphics processors, and different OS's etc.

Black box and white box testing can be used. Execute every statement in the module, test every execution of each decision, and test all possible combinations/paths.

Levels of Testing

Each individual subroutine and module within a program are tested as it is created. Each program will be tested to ensure modules work together correctly. (Mostly in implementation stage, greater focus on operation and actual code).

Module Level Testing

Test that each individual module and subroutine function correctly. Involves use of drivers, and black/white box testing. Mostly in the implementation stages.

Individual modules should be well tested before it is incorporated into the larger solution to program and system testing is easier. However, while an individual module may be successfully tested, when incorporated into the larger system, there may be problems.

Program Level Testing

Test that the overall program functions correctly. E.g., internal logic of the program, all subroutines and modules function together correctly. Test data tests for pathways and boundaries. Mostly in the implementation stage.

Interactions between modules may cause errors. Bottom-up testing involves drivers to test each module starting from the bottom. Top-down testing involves stubs to test the main program and then each lower module.

System Level Testing

Aims to ensure the hardware, software, personnel, and procedures work together efficiently, correctly, and in the manner intended, in the various system environments in which it will be installed.

To achieve this, a real test environment and live test data is needed.

- **Hardware:** Tests done on different combinations of hardware, ranging from minimum requirements to combinations with any additional hardware developed. Large systems often use very specific hardware configurations, so it is easier to test. If a broad hardware base is allowed, it is more difficult to test, and may not be possible to cater for all possibilities.
- **Software:** Other software installed can cause problems with the new software (e.g., different versions of software or OS's). The software must be tested with combinations of other software. Software that interfaces with other software and passes data to other software needs thorough testing, which can be done with live data. Internet applications need to be tested across a broad range of OS's and browsers (e.g., browser size and reshaping, network speeds).
- **Data:** Data input into a system is a likely source of errors. The product must be tested with a large quantity and combination of live data to ensure reliability under live conditions.
- **Personnel and Procedures:** User skills and knowledge will vary; software should cater for this. User internal evaluation is very important. Choose a variety of personnel to perform beta testing, which is used to analyse the ability of personnel to efficiently implement any new procedures and modify old ones.

Use of Live Test Data for System Testing

Live test data ensures that the testing environment accurately reflects the expected environment in which the new system will operate. It simulates extreme conditions that could occur to ensure that the software can achieve its objectives under these conditions. CASE tools can help produce live test data and automate the testing.

Different sets of live test data are used to test particular scenarios. They all test how the overall system will perform and function when live.

Large File Sizes

During the development, relatively small data sets are used. Large files should be used during alpha/beta testing to test the system's performance and highlight problems associated with data access (particularly w/ large databases, files, access via networks).

Large data files highlight aspect of the code that are inefficient or require extensive processing. Many large systems postpone these intensive processes until times when system resources are available. E.g., updating bank transactions occurs during night.

Mix of Transaction Types

Need to test that transactions occurring in a random order do not create problems. E.g., transactions in an account at a bank. It must also be tested that when data is altered while it is being processed, it does not cause problems.

Response Times

Response time: Time taken for a process to complete (user activated or internally activated.) They are dependent on all system components, their interactions, and processes occurring.

Any process taking longer than 1 second should give feedback via progress bars etc. Processes such as data validation for data entry forms should preferably be 0.1 seconds to reduce frustration and deterioration of concentration.

Response times should be tested on the minimum requirements/hardware using typical data of different types and any applications that affect or interfere with the software should be operating under normal or heavy load when testing takes place.

Sites that include animations and complex graphics increase response times, due to the time taken to download graphics. By analysing response times, the effectiveness of a product can be improved and there will be better user experience = more money.

Volume Data (Load Testing)

Testing with large amounts of data entered into the system to test the application under extreme load conditions. Multi-user products should be tested with large numbers of users entering and processing data simultaneously. Especially important for large systems that require extensive data input.

CASE tools can enable automatic completion of data entry forms and allow one machine to simulate thousands of users simultaneously entering data. This is very useful as it is

generally not possible to test a product with thousands of actual users, should the new system be used by thousands of people.

Interfaces between Modules and Programs

An interface provides a communication link between system components. The interface between modules in a program is usually provided through parameters that pass data. Testing the interfaces between modules and programs involves creating tests and test data that analyse the interactions between different software components of the system.

Large systems often have a communication link (e.g., from an ATM to a bank) as an integral part of the interface between programs. Tests should examine the accuracy of data being passed, response times, the ability of the interface to cope with large files, different transaction types, and large volumes of data.

Comparison of Actual Results with Expected Results

Test sets have test data and their expected outputs. Test data should be entered into the finished product to ensure actual outputs match the expected outputs, to ensure the final product performs processing correctly.

Benchmarking

Benchmarking: the process of evaluating a product using a series of tests. The test results are compared to recognised standards, to compare products.

- Users can make informed purchasing decisions from the comparisons
- Developers have objective data to compare products with competitors

Many variables can affect performance e.g., a particular hardware configuration may make one product outperform others. Benchmarking should reduce the number of variables and report results objectively.

The product and competing ones are subjects to a series of tests using identical hardware configurations and test data sets. The results highlight the strengths and weaknesses of the product and can then evaluate the performance of the product compared to its competitors, maintaining and improving market performance.

Benchmarking can be done internally by the software development company, or by several competing companies working together to result in mutual gains and an increase in quality and performance across the industry.

Some companies specialise in benchmarking. They provide benchmarking software to test specific aspects of systems. Benchmarking products can test the performance of specific hardware functions e.g., graphics rendering, file input/output, floating-point calculations. The results can optimise software products for particular system configurations.

Quality Assurance

The quality of a product is how well the product meets or exceeds user expectations. QA techniques should be implemented throughout the development process. A number of tests must be done to assess and evaluate the following for a software product:

- **Correctness:** Does it do what it is supposed to do?
- **Reliability:** Does it do it all of the time?
- **Efficiency:** Does it do it the best way possible?
- **Integrity:** Is it secure?
- **Useability:** Is it designed for the end user?
- **Maintainability:** Can it be understood?
- **Flexibility:** Can it be modified?
- **Testability:** Can it be tested?
- **Portability:** Will it work with other hardware?
- **Reusability:** Can parts of it be used in another project?
- **Interoperability:** Will it work with other software?

ISO (International Standards Organisation) develops standards for quality assurance. Companies and products can adopt these. In Australia, QAS (Quality Assurance Services) will provide certification services, product testing, auditing, and educational services. Companies can then use the standards logos on their advertising and product packaging.

Use of CASE Tools when Testing Software Solutions

CASE tools automate the tedious tasks of testing. They can specialise in particular functions such as test data generation, user interface testing, or volume testing.

WinRunner

Tests that user interface-based applications behave as expected. Automates tasks associated with user data entry and navigation. Records keystrokes necessary to complete a particular process, creates a script, inputs and outputs data as parameters (to execute tests w/ multiple test data items), and can have checkpoints/breakpoints inserted in the script to pause and compare expected results with actual results from the automated process.

LoadRunner

Tests volume/load testing. Simulates thousands of users simultaneously using an application. Scripts are recorded and then given parameters to allow the use of different live test data. Virtual users are created, with different characteristics e.g., different time intervals between inputs. Tests involve multiple virtual users entering data into the same field simultaneously. Has real-time monitors to view application's performance during testing. E.g., view average transaction response times, application/database servers.

DataTech

Test data generation tool. Can create millions of test data records. Rules for data creation can be made or determined from existing databases and test data can be written to a large variety of database formats or text files. Data can be randomly generated or extracted from other data sources e.g., address details can be extracted from a particular database.

UseableNet

Website dedicated to testing the useability of other websites. Analyses the HTML code behind a website and then generates a series of useability reports and recommendations. Each page of the website is tested, and potential problems of the site/pages are reported (e.g., inconsistent link colour, too many complex graphics).

Reporting on the Testing Process

Results of testing need to be evaluated and acted upon to correct problems. To do this, they must be well documented and communicated to the development team and users.

Documentation of the Testing Process

Documentation should be developed and maintained during testing.

Test Requirements: What needs to be tested?

Test requirements will reflect the requirements and design specifications for the project. Detail the scope of the testing process: What tests need be done and why? Each component of the program and system should be considered (hardware, software, data, personnel, processes/procedures).

Test Plan: How do we implement these tests?

Includes a schedule and timeline of events occurring during the testing process, and tools used, including CASE tools and hardware.

Test Data and Expected Results: What are the necessary inputs and expected outputs?

The actual test data and expected results used throughout the testing process should be retained, in case it is needed again after modifications and corrections have been made.

Test Results: Do the actual results match the expected results?

Should be kept to justify any recommendations for changes. Needed for later to confirm that changes have been implemented.

Recommendations: What needs to be done now?

Recommendations will be made in regard to bugs and other problems encountered during testing. Should describe each problem, its source, and its severity. Management and client will decide which recommendations to act upon, which ones are acceptable problems, and then the development team will apply the accepted recommendations.

Communication of Test Results

Audience for test results could be the client (projects written specifically for them), the development team (large development companies w/ development team and testing team), or development company (developers outsourcing testing specialists).

Test results will include problems from all aspects of the system tests e.g., bugs, user interface design problems, unacceptable response times, hardware conflicts, ambiguous errors messages etc.

Communication to a client

Use non-technical language. Report any requirements and design specifications that haven't been fulfilled. Reporting should be in terms of client's original requirements.

Rank problems in order of severity e.g., crucial requirement that hasn't been fulfilled vs inconsistent link colour. Problems may be outside the scope of the project, but should still be included. All the capabilities and problems of the product should be known.

It is common to give a demonstration of the system to the users to aid in communicating test results. It can show both positive and negative features, so users can discuss and evaluate the product more realistically.

Communication to a developer

Testing departments or companies outsourced to do testing need to communicate their results back to the development team.

Problems should be highlighted in order of severity, and recommendations should be backed up by technical justifications. Actual test data together with the error results will help developers correct/modify the product.

Testers and developers must cooperate (developers may be defensive about errors). The quality of the final product is more important.

Problem	Source	Description	Severity
Duplicate orders processed.	Orders module	If a client sends an order twice, this is not detected by the system.	High
Address validation slow.	Scheduling module	Client addresses are validated against their phone number. This takes between 2 and 3 seconds.	Low
Discounts calculated incorrectly.	Invoicing module	Discounts are deducted after GST has been calculated.	Extreme

Evaluating the Software Solution

Testing aims to detect and correct errors before the system is installed for use. System level testing assesses the extent to which requirements and design specifications have been met. Design specifications describe what software should do together with how it should be done. Evaluation of design specifications thus ensures a product realises its requirements.

Post Implementation Review

A review of the system once operational is often done prior to the client formally accepting the software as complete.

For smaller projects, this is often a discussion between the client and installers, including a demonstration to confirm and describe how the requirements have been met.

For larger systems, professional independent assessors perform thorough acceptance tests followed by a formal review.

This is done to evaluate the new system based on client feedback after implementation.

Acceptance Testing

Designed to confirm all requirements have been met so the system can be signed off as complete. If the test is successful, then the client makes their final payment and the developer team's job is complete.

Acceptance tests should be done by people who were not involved in the software development process to ensure an unbiased view, which will better evaluate whether or not requirements have been met as well as they could be.

Clients can also perform their own acceptance tests prior to signing off the new system. However, they may be disagreements between the client's view of an acceptable system and the developer's views. It is preferable to agree on the nature of the testing and who performs it early on during the development process.

Once the acceptance tests and post-implementation reviews confirm the requirements have been met, the client will sign off on the project and pay all outstanding costs.

7. Maintaining Software Solutions

- Modifying Code to meet Changed Requirements
- Documenting Changes

Maintenance is an ongoing process of correction and refinement. Software products require maintenance for them to continue to meet the expectations of their users. It is much easier to rewrite aspects of a product than to create an entire new product to meet changing requirements and expectations.

Reasons for maintenance include:

- Correcting errors in the source code (fixing bugs)
- Upgrading to enhance functionality
 - ◆ E.g., changing UI, improved security, improving efficiency
- Meeting new or changing requirements or government requirements
- Changes in the operating environment
 - ◆ E.g., changes in hardware, changes in interfacing programs, changes in the OS

Small bug fixes can be distributed to clients as a patch (piece of executable code inserted into an existing executable program). Large changes require a complete upgrade to be distributed. Documentation will need to be updated.

Features of Maintainable Code:

- Modular in design (easier to modify individual modules as new needs are identified, easier to incorporate pre-written modules if needed, separate modules to handle things that may change or need flexibility e.g., interfaces for different devices)
- Well-documented (intrinsic, comments, documentation)
- Test data and plans retained for each module
- Identifiers for constants (easier to modify)
- Version control to handle incorporation of changes
- Team members writing in defined standards to code is uniform and consistent

Modifying Code to meet Changed Requirements

Identifying Reasons for Change in Source Code

Bug fixes, using new features available in new OS's, meeting requirements, new hardware/software, OS changes etc.

Software with a large customer base may receive thousands of modification requests. They must prioritise these requests and decide which ones to add, sometimes with a survey. Support departments can provide data on the no. of issues regarding a specific function.

Customised software written for a specific client requires continual maintenance as a company grows and requirements change.

When COTS products are upgraded (e.g., macro commands changing), software based on them need to be upgraded to continue operating correctly. The Internet can automate the upgrading of many commonly used applications. Often software will need to be re-compiled using the new version of the COTS product.

Solving Problems in the Product

Actual problems with the software must be corrected before enhancement requests, in order of severity. Problems that cause the entire product to crash must be corrected quickly. Problems that compromise privacy or result in incorrect financial outputs must also be resolved. Problems that can be overcome using other means can be left until time constraints allow for them to be resolved.

Processes for maintenance departments to deal with urgent requests promptly include maintaining a log of requests, assigning each request responsibility to individual personnel, and maintaining regular contact with users.

Locating Section to be Altered

After deciding to modify a particular aspect of the product, the location of the section to be altered must be determined. System models like structure charts and DFDs can assist in locating the module/s. Programmers can analyse algorithms and IPO diagrams to understand the original source code of the identified module.

A thorough understanding of the code is required to alter it.

Determining Changes to be Made

Depending on the nature of the modifications, changes may be made to data structures, files, user interface, as well as the source code.

Changes made to one module may affect others that have access to the altered data (e.g., adding a field to a record). Analysis of the documentation can help programmers see possible consequences of changes.

Implementing and Testing the Solution

Different ways:

- Changes made to the source code, application is recompiled, tested, and distributed.
- Small application or patch to implement the changes on the end-user's systems.
- Custom COTS products changed directly on the site or over an internet link.

The implementation of modifications can affect other aspects of the application. This can be minimised by using a structured modular approach, to easily identify the relationship between an independent module and the other modules.

Often CASE tools and test data/methods used in the original testing are reused. Testing applies to all aspects of the application affected, not just the modified sections.

Documenting Changes

Changes must be tracked and fully documented. Documentation must be modified to reflect changes. Configuration management is the systematic control of the changes to a product, and tracks the versions and changes made.

Small projects may have manual configuration management, but larger projects will need configuration management CASE tools to automate control and documentation of changes.

Source Code Documentation

Includes comments and intrinsic documentation. Comments should describe what each code segment does. Details of who made the change and when should be included in the code. Any original comments that are no longer relevant should be removed or modified.

Comments should be included in the source code to highlight the modification.

Updating Associated Hardcopy Documentation and Online Help

Printed documentation is rare, and it is hard to distributed modified versions of it.

Most user and technical documentation is stored and distributed electronically. Electronic formats are more efficiently edited and distributed, e.g., as a PDF.

Modifying online should reflect any modifications made to the application.

Use of CASE Tools to Monitor Changes and Versions

CASE tools can help automate the process of keeping accurate records of maintenance.

Change Management: Monitors each step of the software development process for each change, beginning with the initial change request to include the modification in the final product. History is maintained of each modification made, when it was made, and by whom.

Parallel Development: Enables multiple developers to work on the same project without fear of duplication or destroying another person's work. Individual files are checked out, locking the file, and preventing others modifying it. When the modification is complete, the file is checked in, unlocking the file, and retaining a record of the changes made.

Version Control: Manage multiple versions of modules. The system tracks which version of each module should be used when the final application is reassembled. A record of prior versions of modules is maintained for reconstruction of previous versions.

8. Developing a Solution Package

Random notes

Gantt chart to help plan project and describe the timing and sequence of events to complete the project. Keeps development team accountable and ensures management don't have unrealistic expectations of what developers could achieve in a period of time. Facilitates scheduling and coordination of tasks, allocation of resources/tools and personnel.

Logbook to describe tasks and decisions completed and notes regarding issues encountered throughout development. A useful part of evaluation and project management. Facilitates communication of progress among team members, allow review and evaluation of project and how well the team works together. Provides detail about daily progress of tasks and any issues encountered, can provide information to assist in current and future development, as future developers can look back on progress to see reasons for actions that were taken.

Project management techniques

- Apply consistently throughout development
- CASE tools, Gantt charts, logbook
- Objectives to which the final product is compared
- Good communication/meets with team members

Consider social and ethical issues

Feedback from users at regular intervals

Communicate effectively with potential users (gather feedback, requirements etc)

Single developer = minimise communication issues (saves time), e.g., communicating specific requirements for design and implementation. But they may not be skilled in all requirements e.g., interface design, security.

9. Option 2: The Interrelationship between Software and Hardware

- Representations of Data within the Computer
- Electronic Circuits to Perform Standard Software Operations
- Programming of Hardware Devices

Representations of Data within the Computer

Character Representation

Characters must be represented using binary code if they are to be stored and processed by computers. There are 2 types of characters: printable and non-printable.

Printable characters: include everything in the keyboard e.g., "A", ";".

Non-printable characters: control characters, used for communications and printing e.g., backspace/delete, carriage returns, line feed.

Char	Dec	Char	Dec	US	31		95
NUL	0	@	64	Space	32	`	96
SOH	1	A	65	!	33	a	97
STX	2	B	66	"	34	b	98
ETX	3	C	67	#	35	c	99
EOT	4	D	68	\$	36	d	100
ENQ	5	E	69	%	37	e	101
ACK	6	F	70	&	38	f	102
BEL	7	G	71	'	39	g	103
BS	8	H	72	(40	h	104
HT	9	I	73)	41	i	105
LF	10	J	74	*	42	j	106
VT	11	K	75	+	43	k	107
FF	12	L	76	,	44	l	108
CR	13	M	77	-	45	m	109
SO	14	N	78	.	46	n	110
SI	15	O	79	/	47	o	111
DLE	16	P	80	0	48	p	112
DC1	17	Q	81	1	49	q	113
DC2	18	R	82	2	50	r	114
DC3	19	S	83	3	51	s	115
DC4	20	T	84	4	52	t	116
NAK	21	U	85	5	53	u	117
SYN	22	V	86	6	54	v	118
ETB	23	W	87	7	55	w	119
CAN	24	X	88	8	56	x	120
EM	25	Y	89	9	57	y	121
SUB	26	Z	90	:	58	z	122
ESC	27	[91	;	59	{	123
FS	28	\	92	<	60		124
GS	29]	93	=	61	}	125
RS	30	^	94	>	62	~	126
				?	63	DEL	127

ASCII

American Standard Code for Information Interchange

Characters are assigned a number which can be converted into binary for computers. The first 32 are control characters.

ASCII is 7-bit, so there are 128 different characters. Some extended versions have 8 bits (256 characters) e.g., the standard ISO Latin 1 system. 8-bit systems use values above 127 to represent foreign language, graphical, and mathematical characters.

ASCII and ISO Latin 1 are incorporated within the modern Unicode system.

Standard ASCII has been developed to simply sorting: code for A is less than B, each capital letter is 32 less than the lower case, the last 5 bits of each letter are the same in upper and lower case. A = 65 = 1000001, a = 97 = 1100001.

Unicode

Unicode specifies a unique number for every character in every language. Currently, Unicode is the global standard for coding characters. New characters and languages are being added to the standard, and once a code is assigned to a particular character it will never change and forever be readable by any Unicode aware software.

Unicode has 17 planes, each being 16 bits long (65536 characters per plane). Plane 0 has code for most modern languages, the rest of the planes are mostly unused apart from ancient languages and special symbols such as card symbols. All the ASCII characters have identical code points in the Unicode planes (A is 65 in both).

Unicode standard recommends that code points be specified using “U+” followed by the hexadecimal value. (A is U+0041). The 0th plane uses 4 hexadecimal digits, the rest have an extra hexadecimal digit at the beginning (U+1FOA1 is plane 1).

There are different Unicode standards for different environments. Webpages use UTF-8 (smaller and faster), and many desktop applications use UTF-16. UTF-32 is not used much.

Unicode can represent all ASCII characters, as well as other languages and more characters. ASCII code takes less space as it is 7 or 8 bit.

Different Number Systems

Remember to put the bases in.

Decimal: base 10

Binary: base 2

Hexadecimal: base 16

Binary system uses 2 digits: 0 and 1. These binary digits/bits can be stored in different ways e.g., lands and pits on a CD-ROM. Basically from right to left, $0 \text{ or } 1 * 2^{\text{position}}$.

$$10101_2 = 1(16) + 0(8) + 1(4) + 0(2) + 1(1) = 21_{10}$$

Binary numbers are generally expressed in 8 bits (1 byte). Most computers store and process data in multiples of 8 bits; the number of bits processed at a time is called the word length of the processor. e.g., 8, 16, 32, 64.

After the decimal point, it is $2^{-\text{position}}$, like 0.1 is 2^{-1} , ($\frac{1}{2}$). $10.101 = 2 + \frac{1}{2} + \frac{1}{8}$.

Hexadecimal system uses 16 digits: 0-9, A-F (10-15). Hexadecimal numbers are simpler for humans to read compared to binary numbers. Same as binary reading except 16 powers.

$$2B04_{16} = 2(4096) + 11(256) + 0(16) + 4(1) = 11,012_{10}$$

Binary to hexadecimal: grouping in 4s (4 bits) from the right side, each byte is one hex digit.

Hexadecimal to binary: each hexadecimal digit is 4 binary digits.

Decimal to binary: Subtract largest possible power of 2 from the number (e.g., subtract 64 from 70). If the result is larger than the power of 2, put 0, otherwise put 1. Then subtract next power of 2 (half the original) etc etc.

Decimal to hexadecimal is the same except powers 16 lol.

Integer Representation

Whole numbers, positive or negative. Negative numbers cannot be stored with ‘-’.

Sign and Modulus

Add an extra bit to the front of each number to represent the sign. 0 = positive, 1 = negative. The modulus is the magnitude/absolute value of the number.

$$12 = 00001100, -12 = 10001100. \quad 0 = 00000000 \text{ or } 10000000.$$

This method is generally not used as computers don't tend to subtract, and the sign bit must be considered separately.

The complements are where the sign of the number is included as an integral part of the representation of the integer.

One's Complement

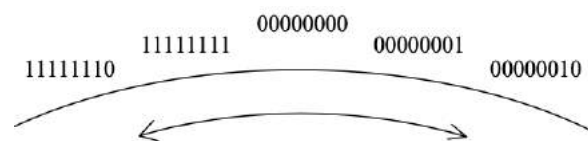
0 at the beginning is positive, 1 at the beginning is negative.

To find the one's complement of a number, reverse the 0s to 1s and 1s to 0s.

Two's Complement

0 at the beginning is positive, 1 at the beginning is negative.

A "number circle", 0 at the top, right from 0 is positive, left from 0 is negative.



E.g., a 3-bit number system, 000 to 111.

- 000, 001, 010, 011 (0, 1, 2, 3) represent positive numbers
- 100, 101, 110, 111 (4, 5, 6, 7) represent negative numbers -4, -3, -2, -1
- Each positive number begins with 0, each negative with 1

To find the two's complement of a number:

- Reverse the 0s and 1s and add 1 to the number (addition)
- OR: From right to left, reverse all the digits after the first 1 you encounter

To convert a negative decimal number to its two's complement, Make the highest power a negative, and add the rest of the powers as positive. E.g., in a 4-bit system:

$$1101 = -8 + 4 + 0 + 1 = -3$$

The two's complement allows for easier binary subtraction, as you simply get the two's complement of the number (resulting in the negative number) and add it.

8 Bits	Smallest Value		Largest Value	
	Binary	Decimal	Binary	Decimal
Sign and Modulus	1111 1111	-127	0111 1111	127
One's Complement	1000 0000	-127	0111 1111	127
Two's Complement	1000 0000	-128	0111 1111	127

Floating Point/Real Representation

Real numbers cannot always be represented precisely using computers because some real numbers can go on forever, and processing/storage ability must be considered. Fractions are stored in a similar fashion to scientific notation, with a sign, exponent, and mantissa.

Fixed-Point

The decimal point is in the same position for all numbers, e.g., 16-bit fixed-point may have 8 bits for whole number part and 8 bits for fractional part. This cannot represent very large

and very small numbers, so it is not included as a standard data type in most high-level languages. However, fixed-point representations are faster for processing.

Floating-Point

The position of the decimal is included in the format, so it can move.

The IEEE has a standard for representing binary numbers using the floating-point system.

Single precision: 32 bits

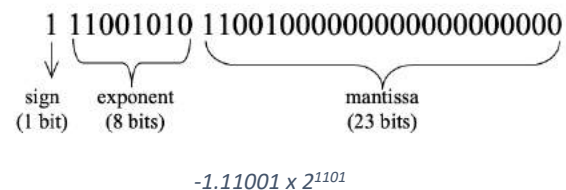
Double precision: 64 bits (more accuracy and memory)

Sign: 0 if positive, 1 if negative

Exponent: the actual number plus 127. First half is negative, and second half is positive.

Exponents from -126 to 127 (128 for infinity and -127 for 0).

Mantissa: the leading one is assumed (before decimal point), so it isn't included in mantissa.



Double precision uses 1-bit sign, 11-bit exponent (actual value plus 1023), 52-bit mantissa.

Converting binary to floating point: 1.101×2^{110}

- Sign is 0
- Exponent is $110 + 01111111 (127) = 10000101$
- Mantissa is 101 (ignore leading 1)
- Floating point: 0100001011010...0

Converting floating point to binary: $1 \ 10011010 \ 0011010...0$

- Sign is negative
- Exponent is $10011010 + 10000001 (-127) = 10011011$
- Mantissa is 1.001101 (add leading 1)
- Binary: $-1.001101 \times 2^{10011011}$

Converting decimal to floating point: $37 \frac{5}{8}$

- Sign is 0
- $37 = 100101$ $\frac{5}{8} = \frac{1}{2} + \frac{1}{8} = 101$ $37 \frac{5}{8} = 100101.101$
- $100101.101 = 1.00101101 \times 2^5 = 101$ (moving decimal point 5 to the left)
- Exponent is $101 + 01111111 = 10000100$
- Floating point: 010000100001011010...0

Binary Arithmetic

Addition

+	0	1
0	0	1
1	1	10

$$1 + 1 + 1 = 11$$

Literally just normal addition, with the rules in the table. Carry the 1s.

When doing addition with the two's complement, you can ignore the final carry (cut off left-most digit if there aren't enough bits). Any calculations resulting in values outside of the range will result in an overflow error.

$$\begin{array}{r} 1111 \\ 01010111 \\ + 00011100 \\ \hline 01110011 \end{array}$$

Subtraction

Instead of subtracting a number, we add the negative of it. $5 - 3 = 5 + -3$. Final carry ignored, overflow errors can occur from the result.

Multiplication

$\begin{array}{r} 1001 \times \\ 101 \\ \hline 1001 \\ 00000 \\ 100100 \\ \hline 101101 \end{array}$	<p>Same as decimal multiplication. Shifting to the left.</p> <p>When multiplying by powers of 2, you can shift the number left by the power. e.g., $110 \times 100 = 11000$ (110 with 2 0s at end)</p> <p>Shift and add: the number is multiplied by the first bit in the multiplier. Then it is shifted left and multiplied by the second bit, and so on. The results of each multiplication are added together.</p>
------------------------------------------------------------------------------------------------------	--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Division

Same as decimal division. Add the two's complement of the number to subtract. (ignore davis's subtraction). With the subtraction: ignore final carry.

$$\begin{array}{r} 1001 \\ 110 \overline{) 111010} \\ \underline{110} \\ 1010 \\ \underline{110} \\ 100 \end{array}$$

remainder 100

Shift and subtract: the divisor is shifted right across the bits and compared to the remainder. If it is smaller, subtract it from the remainder and record 1 in the quotient. If it is larger, shift it right again until it can be subtracted.

Electronic Circuits to Perform Standard Software Operations

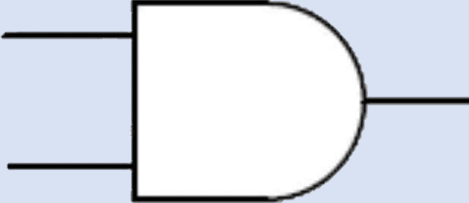
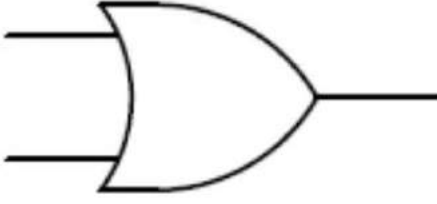
How processes are implemented using electronic circuits.

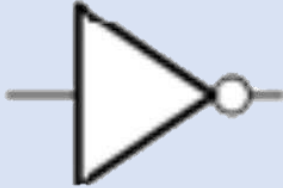
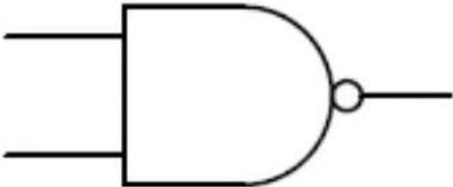
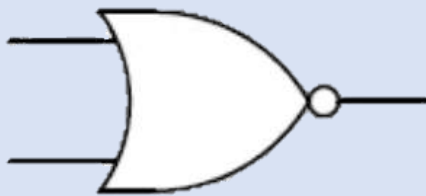
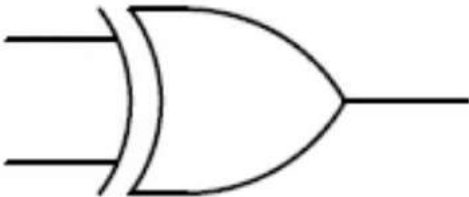
All digital computers are built from simple switches, which are either OFF or ON – states stored as binary digits 0 and 1. Logic gates are the basic building blocks for the creation of microprocessors. In various combinations they can perform all functions carried out by computers.

Logic Gates and Truth Tables

Logic Gates: Electronic circuits, transform inputs into outputs based on strict logical rules.

Truth Table: Summarises all possible inputs and their resulting outputs for a circuit.

Gate	Description	Symbol	Truth Table		
AND	Output is 1 when both inputs are 1.		Input		Output
			A	B	C = AB
			0	0	0
			0	1	0
			1	0	0
OR	Output is 1 when at least one of the inputs is 1.		Input		Output
			A	B	C = A+B
			0	0	0
			0	1	1
			1	0	1
			1	1	1

NOT	Reverses input.		Input		Output
			A		$C = \bar{A}$
			0		1
			1		0
NAND	The opposite of the AND output.		Input		Output
			A	B	$C = \overline{AB}$
			0	0	1
			0	1	1
			1	0	1
			1	1	0
NOR	The opposite of the OR output.		Input		Output
			A	B	$C = \overline{A + B}$
			0	0	1
			0	1	0
			1	0	0
			1	1	0
XOR	Output is 1 when only one of the inputs is 1.		Input		Output
			A	B	$C = A \oplus B$
			0	0	0
			0	1	1
			1	0	1
			1	1	0

XOR gate is also $(A + B)(\overline{AB})$

Boolean Algebra

Describing a circuit and simplifying an existing circuit. Basically leave the answer in the simplest way to draw the circuit.

1a. $A+B=B+A$

1b. $AB=BA$

2a. $(A+B)+C=A+(B+C)$

2b. $(AB)C=A(BC)$

3a. $A(B+C)=AB+AC$

3b. $A+(BC)=(A+B)(A+C)$

4a. $A+A=A$

4b. $AA=A$

5a. $0+A=A$

5b. $0A=0$

6a. $1+A=1$

6b. $1A=A$

7a. $\bar{A} + A = 1$

7b. $\bar{A} A = 0$

8a. $A + \bar{A} B = A + B$

8b. $A(\bar{A} + B) = AB$

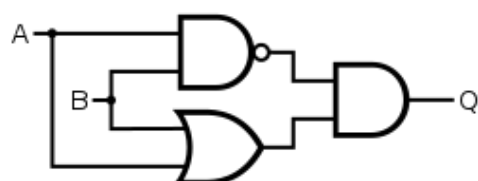
DeMorgan's Theorem

9a. $\overline{(A + B)} = \bar{A} \bar{B}$

9b. $\overline{(AB)} = \bar{A} + \bar{B}$

Double NOTting something to return to its original.

Per 4a and 4b, anything plus itself is itself, and anything times itself is itself.



$$Q = (A+B) \cdot (\bar{A} + \bar{B})$$

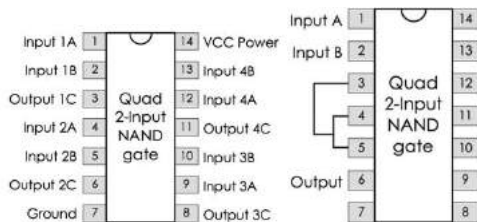
A XOR gate is a NAND gate ANDed with an OR gate.

Circuit Design

Creating each gate using only NAND gates. idk why davis

Integrated circuit with 4 NAND gates.

Basically you draw up a truth table with a lot of columns. E.g., for OR you see that NAND is 1110, while OR is 0111. So if you inverse the inputs before they NAND, it becomes 0111. So you put NOT (A NAND A) for each input first.



However, XOR uses 6 gates (Boolean algebra to simplify to 4 gates).

NOT	$\overline{AA} = \bar{A}$	Input Output
AND	$\overline{\overline{AB}}$	Input A Input B Output
OR	$\overline{\overline{AB}}$	Input A Input B Output
XOR	$(A + B)\overline{AB}$	Input A Input B Output

Circuit Design Steps

Deriving a Boolean equation from a truth table to make a circuit:

- Sum of Products (SoP): Using rows with an output of 1
- Product of Sums (PoS): Using rows with an output of 0
- Use the one with the least number of rows

Sum of Products:

- Make the inputs of each row into a product that creates an output of 1 i.e., NOTing them if needed e.g., $AB = 1, \bar{C}B = 1$ ($A = 1, B = 1, C = 0$)
- Add the products of the inputs in each row e.g., $AB + \bar{C}B$
- Simplify to make a circuit

Product of Sums:

- Make the inputs of each row into a sum that creates an output of 0 i.e., NOTing them if needed e.g., $A + B = 0, \bar{C} + B = 0$ ($A = 0, B = 0, C = 1$)
- Multiply the sums of the inputs in each row e.g., $(A+B)(\bar{C}+B)$
- Simplify to make a circuit

Special Circuits

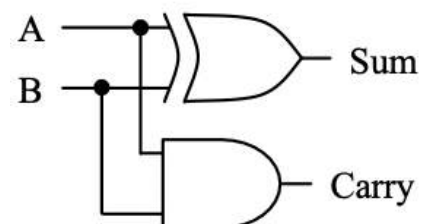
Half Adder

Adds two binary digits together, not accounting for carry ins. Outputs the sum and carry.

Input		Output	
A	B	Carry	Sum
0	0	0	0
0	1	0	1
1	0	0	1
1	1	1	0

Inputs and outputs in binary addition.

Sum is XOR inputs.
Carry is AND inputs.



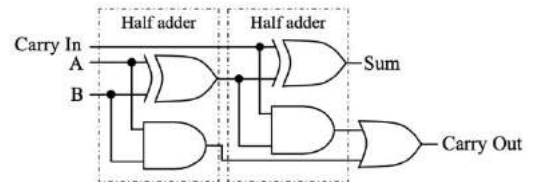
Full Adder

Adds two binary digits together along with a carry in. Outputs the sum and carry.

Input			Output	
Carry In	A	B	Carry Out	Sum
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Two half adders combined. The first adder adds the inputs, and the second adder adds the first adder's output with the carry in. The carry outs from both adders are ORed.

Chaining full adders together for 8-bit addition. Carry out from one adder is carry in for next.

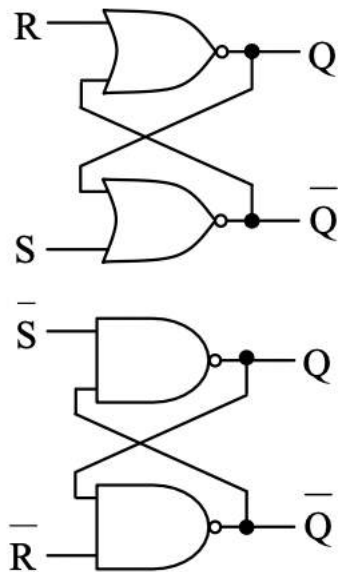


Flip-Flops (storage of a bit in memory)

Bistable memory stores for binary digits (either 0 or 1). Used to store temporary memory in CPU etc.

Three components: latch (storage), clocking (timing), edge trigger (integrity).

Latch: A circuit that stores a binary digit. Latches are the way flip flops store digits, with a feedback loop that maintains a particular state. S = set, R = reset.



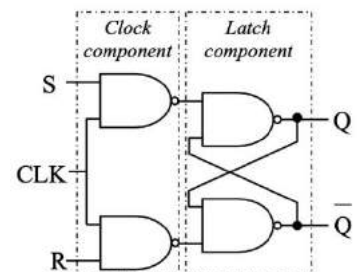
Q and Q' cannot be the same value, so SR cannot be 11.

Input		Output		
S	R	Q	\overline{Q}	
0	0	Last Q	Last \overline{Q}	<i>Hold</i>
0	1	0	1	<i>Reset</i>
1	0	1	0	<i>Set</i>
1	1	X	X	<i>Illegal</i>

- S is the value that the latch stores (Q)
- R is set to 1 to reset the latch to store 0.
- $0 \rightarrow 1$, S to 1 and R to 0
- $1 \rightarrow 0$, S to 0 and R to 1
- Assume the last Q (from previous cycle) is 0 when starting

Clock: Controls when the latches can change or not. Synchronises process of changing value stored in the circuit, ensures that they are all changed at the same time.

- When CLK is 0, changing S or R does not change the latch
- When CLK is 1, changing S or R changes the latch



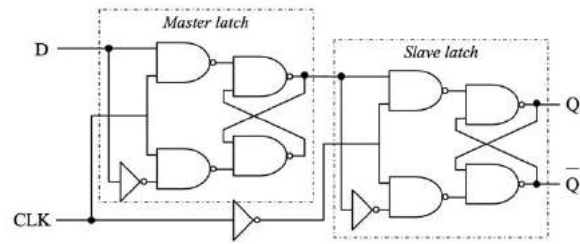
Illegal 11 input: data latch. Only have one input (D) as S, and NOT it for R.

Edge trigger component: Ensures flip-flop only changes once for each tick of the clock input.

Two latch circuits: master and slave.

CLK for M, inverted CLK for S.

- When CLK is 0, changes in D won't affect M output.
- When CLK is 1, changes in D affects M output, M output doesn't affect S value.
- When CLK falls to 0 (negative edge), D doesn't affect M, but M output updates S value. So now M and S both store the same value Q.



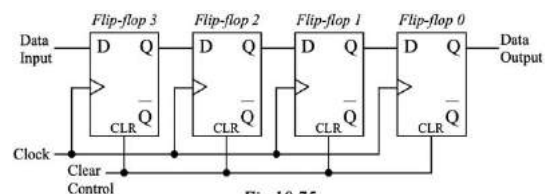
Shift Registers (shifting of a bit in memory)

Used to store binary data, normally in multiples of 4 bits. Allow for data to be moved into or out of the register. There are also designs to shift data left/right (multiplication/division). Linking the output from one flip-flop to the input of another allows bits to be shifted.

Serial in – serial out, parallel in – parallel out, bi-direction, serial in – parallel out etc

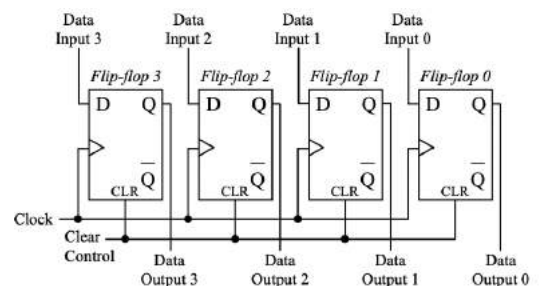
Serial in – serial out shift register:

- Populated one bit at a time
- Loading 4 bits requires 4 clock ticks
- As a bit is inputted from the left, the other bits shift to the right

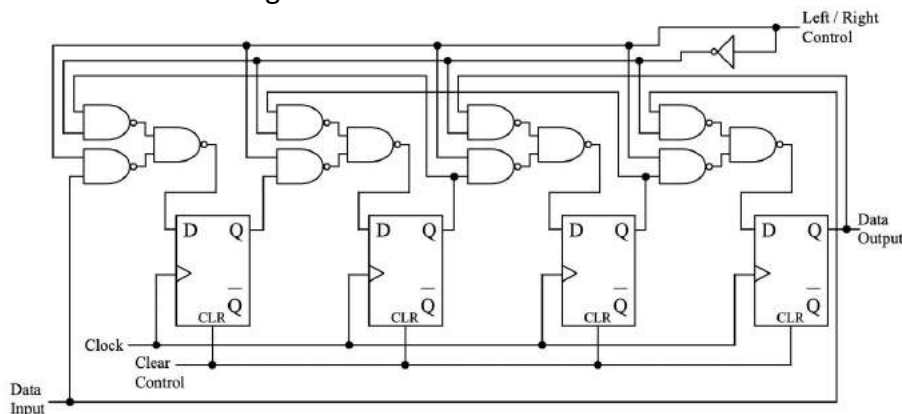


Parallel in – parallel out shift register:

- Receives and outputs data in/out of flip-flops simultaneously
- One tick is required to fill register with new data
- Data cannot be altered, storage register instead of a processing register
- Data can be clocked in and out in a single tick – suitable for fast access of data



Bi-directional shift register



Basically it can do left shift (control set to 0) for multiplication by 2, right shift (control set to 1) for division by 2 is this even in the syllabus

Programming of Hardware Devices

The Data Stream

A data stream is received from input devices and sent to output devices to control hardware devices with software.

3 components:

- Header information
- Data
- Trailer information

The header and trailer bits are used to ensure the data is delivered correctly.

Manufacturers of hardware devices provide specifications for the nature and format of the data streams sent to and from the product.

Header information: A single start bit to signal to the receiving device that a new data packet is commencing.

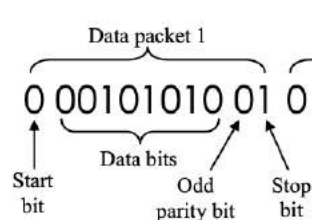
Data block: The actual data being sent, used by the receiving device. Often control characters are used which cause actions predefined by the hardware (e.g., new page), or to indicate which part of the data block is text and which part is text styling etc.

Trailer information: Includes error checking bits and stop bits. Often a simple parity bit is used for error checking. A stop bit is often used to indicate the end of the data packet.

Example: Mouse Data Streams

2 sensors to detect X and Y motion, 3 sensors for the states of each button.

PS2 port receives the data from the mouse.



Mouse sends 3 consecutive data packets of information, each 11 bits long: start bit, 8 data bits, odd parity bit (check for errors), stop bit. The packets contain information about the state of each button and the movement in the X and Y direction since last transmission.

Packet No.	Bit number							
	7	6	5	4	3	2	1	0
1	Yv	Xv	Yd	Xd	1	0	Bl	Br
2	X7	X6	X5	X4	X3	X2	X1	Y0
3	Y7	Y6	Y5	Y4	Y3	Y2	Y1	Y0

Yv Overflow in Y direction.
 Xv Overflow in X direction.
 Yd Direction of movement in Y direction (1 means left, 0 means right).
 Xd Direction of movement in X direction (1 means up, 0 means down).
 Bl Left button state (1 means down, 0 up).
 Br Right button state (1 means down, 0 up).
 X0-X7 Number of bits moved in X direction since last transmission.
 Y0-Y7 Number of bits moved in Y direction since last transmission.

Processing an Input Data Stream

Once a data stream is received from a hardware device, it is processed by the computer. The software used to process must understand the precise format of the data stream. The data characters must be identified from the stream. Control characters must be recognised, isolated, and acted upon. The processing required depends on the particular device.

The protocol supplied by the hardware developer will determine the structure, form, and nature of the data stream, how control characters are recognised and processed, etc.

Header fields may contain information about how much data is being transmitted.

Printer Control Character Sample Table

The Epson FX series of dot matrix printers is controlled by a series of control codes; the table below details some of the available codes. Some of the control codes are single ASCII characters; many use the ASCII escape character (decimal 27) to indicate the information is a command to the printer rather than text to be printed.

ASCII	Decimal	Description	ASCII	Decimal	Description
BEL	7	Cause bell to sound	BS	8	Backspace. Move left one char.
LF	10	Line feed (Move paper up 1 line)	FF	12	Form feed. Load a new page
CR	13	Move print head to left hand side	ESC C n	27 67 n	Set page length in lines (n=1-127)
ESC - 0	27 45 48	Cancel underline	ESC - 1	27 45 49	Select underline
ESC 5	27 53	Cancel italic mode	ESC 4	27 52	Select italic mode
ESC F	27 70	Cancel bold mode	ESC E	27 69	Select bold mode

Example data stream: [FF] [ESC] - 1 My Favourite subject [ESC] - 0 [LF] [LF] [CR] Definitely Software Design and Development. [LF] [CR] [ESC] 4 It's the best! [ESC]

Generating Output to an Appropriate Device

The CPU sends information to the connected devices. Output data packets must specify the device to which the information is being transmitted.

Uh basically just follow whatever information/stimulus is given about the stream.

Determining the structure of the data stream:

- Required **header** information e.g., start bit, sender and receiver, length of message
- **Control** characters to do predefined tasks e.g., line feed
- **Data** characters for information e.g., temperature, ASCII letters, state of a switch
- Required **trailer** information e.g., error checking bits, stop bit

Issues with Interpreting Data Streams

A string of binary digits can have many different meanings (8-bit integer, decimal number, ASCII, floating point, two's complement etc.)

Control Systems

Control systems are made up of sensors that get data from the environment and actuators that perform actions under the direction of a controller (a computer system).

Sensors: provide input by detecting information from the environment (e.g., pressure, light)

Controllers: perform processing (may be a computer or specially designed circuit)

Actuators: perform the output (e.g., mechanical arm, robot)

Open control system: Has no feedback and can't respond to the environment.

Closed control systems: Has feedback, so it can react to the environment. Sensors provide the feedback so controller can control the actuators thus reacting to the environment

Dumb robots: continue to carry out their task regardless of the environment

Intelligent robots: can react to the environment