

Software Design & Development Notes

Hey guys,

Just a little information about these notes. Being of the class of 2015, I wrote these notes for myself a few years back and never got around to publishing them. I found myself on BOS a few days ago and noticed that some people we're asking for notes and decided I may as well publish mine.

Keep in mind I never intended on publishing these onto the internet. I can't guarantee that everything I said was correct and I'm not even sure if they're complete. That's up to you guys to figure out! If anything seems wrong or inconsistent with more reliable sources of information, do a little research and determine which would be more correct.

You can always use these to complement your own notes, help you understand many of the concepts in the course, to use while listening to your teacher that wasn't qualified to teach SDD, or when you decide to study the entire course within 24 hours of your exam. It can be used pretty much all situations. It works pretty much anywhere.

If you have any questions, I would have recommended to PM me although I most likely wouldn't see it, soz.

Best of luck with your exams.

DEVELOPMENT AND IMPACT OF SOFTWARE SOLUTIONS

1. Social and Ethical Issues

The impact of software

Software developers have the responsibility to ensure their products function in a socially and ethically responsible manner as many large services rely on software such as the electrical grid and airline systems.

Y2K problem

Due to the price of storage during the 60s and 70s, developers would attempt to use the least amount of storage space possible. This would include writing the date using 2 digits e.g. 60 instead of 1960. When the year 2000 would come along, people anticipated that computers would think the year was 1900 not 2000. This could potentially have caused major problems in all computer systems from small computers to large airline systems.

Malware

Malware is any software which performs an unwanted or uninvited task. This includes viruses, Trojan horses, worms and spyware although the syllabus only mentions "malware such as viruses". Viruses are unique in the way that they replicate themselves. They quickly spread throughout a computer and to other computers to expand their reach and complete their desired task.

Syllabus doesn't mention the following forms of malware although I think a basic understanding is still necessary and/or useful.

1. **Worms** aim to slow down a system by replicating itself until the entire disk (and RAM) is full and overwhelmed (sometimes RAM can be so full that the system can't execute any tasks).
2. **Trojan horses** are malicious software which disguises itself as legit software. For example you may download a game but once you run the program the malicious software will run instead (usually in the background without you knowing...while you play your game)
3. **Spyware** is solely built to obtain **your** information. They may attempt to access your browsing habits, emails, keystrokes etc. and may be used for identity theft.

Reliance on software

- Software must be very reliable as it runs planes, electrical grids, dams and many other critical services (as well as everyday products).

Social networking

A social network is an online service which allows people to communicate with each other. Examples include Facebook, Twitter, LinkedIn etc. Although there are many advantages to these sites there are also disadvantages such as identity theft, stalking and online bullying as well as the fact that whatever you put up online, stays online and can be detrimental for some people e.g. what a young person puts up online could be the reason they don't get a job 5 years down the track.

Cyber safety

Cyber safety is about protecting yourself online and minimising the risk of online dangers. Precautions should be taken when meeting people online, sharing information, speaking to known and unknown people (could be cyberbullies or even stalkers) and you should watch your digital footprint. (cyber safety includes identity theft and online purchasing).

Evaluating (legitimacy of) information available through the internet

Anybody can post information on the internet without qualifications or expertise. This can lead to misleading, incorrect and biased information being posted. Steps should be taken to reduce this risk by verifying the author, checking references/citations, checking if info is up to date, checking for bias etc.

Rights and responsibilities of software developers

Intellectual property

Intellectual property is one resulting from mental labour (which could be an invention, trademark, design etc.). Authors (anybody involved in the creation of software e.g. developers, graphic designers, programmers etc.) own the right to control how their work is sold and distributed. Intellectual property is protected by copyright laws.

From internet: Intangible property resulting from creativity.

Quality

Customers perception of quality are influenced by their expectations. If their expectations are exceeded they will perceive the product to be one of high quality and vice versa. Quality Assurance aims to ensure high quality is achieved throughout the development process. It is the developers responsibility that this is achieved. Financial constraints and development expertise are the main factors affecting quality. Hardware, OS, and interference from other running software are all external factors affecting quality.

Response to problems

It is the developers responsibility to have systems in place to deal with problems in their software accurately, efficiently and timely.

Code of conduct

This is a set of standards which the developers agree to abide by. Its aim is to enhance their (and the industry's) reputation and standards. Some points may be: to maintain professionalism, to honour contracts, respect laws etc.

Malware

Developers have the responsibility to ensure their products don't include malware. They should make sure any software they include also doesn't contain malware. It is the users right to purchase software without malware included.

Ergonomic issues

Definition: ergonomics is the study of the relationship between human workers and their work environment (if just mentioning the definition, make sure to link software somehow). Ergonomic strategies in UI design include consistency in design, explanation/manual or walkthrough of usage, colours/fonts/alignment/sizes should be chosen appropriately. According to Davis text., usability testing is about the efficiency and satisfaction users experience as they use the software.

Inclusivity issues

This means not excluding people based on culture, gender, economic background, disability etc. (use examples, such as those in the Davis text. p21-25)

Privacy

Privacy is about protecting personal information (information which may identify someone).

Users have the right to know how their information is being handled. Developers have the responsibility to handle their users information according to law, and only using their information for what they (developer and user) have agreed upon.

Software piracy and copyright

Definition: Software piracy is the illegal copying and use of software

Intellectual property

Discussed previously.

Plagiarism

Plagiarism is taking somebody else's work and passing them off as your own.

Copyright laws

Legal protection given to the author of the original work from illegal copying, modifying or distribution. They safeguard intellectual property and aim to foster the creation of creative work as authors know their work will not be illegally copied, modified or distributed.

The following 4 headings are classifications of software.

Commercial software

Most software purchased from a retailer (or equivalent) is commercial software. When you purchase a copy, you are actually purchasing a licence which means the licensing company owns the product and its copyright. Generally, commercial software is covered by copyright, one archival copy can be made, modifications aren't allowed, decompilation isn't allowed and the software cant be used in other projects.

Shareware

Shareware is acquiring a licence which allows you to use the software for a limited time (free trials basically). It is protected by the same copyright laws as commercial software with the exception that the software can be distributed.

Open Source

Open Source software allows anybody to freely modify and redistribute software as long as the author is recognised and that the modified product is released using the same open source licence. This encourages collaboration and the sharing of ideas (think Linux).

Public Domain

This is when the copyright holder gives up all rights to the software. This means that basically no copyright laws apply so the software can be modified, redistributed, decompiled, used in other projects etc.

Ownership versus licensing

Generally, when you purchase software you are purchasing a licence to use the software which means that the author still owns the software and its copyright.

Collaboratively developed software

*Note: Davis text. didn't mention this title, however, it's fairly self-explanatory.

Decompilation

Decompilation is the opposite of compilation. It involves translating executable machine code back into higher-level code (usually assembly code, not source code). This allows the programs design to be more easily understood.

Reverse engineering

Reverse engineering is analysing a product and its parts to understand how it works and to recreate its original design, usually with the purpose of creating a similar product based on its design. (In the half yearly's the correct answer -after finding the answers online- was 'analysing a product to determine its original design'). (Also, reverse engineering usually involves decompilation)

Current and emerging technologies to combat piracy

1. **Non-copyable datasheet** is a really old technology which works as follows. Basically if you purchased software (e.g. a game) it would have been accompanied by a datasheet (which could be a piece of paper or a book or something similar) and in order to use the software or to continue using it, it would ask you questions (or for a code or something similar) which the answers could only be found in the datasheet.
2. **Disk copy protection** refers to technology used to prevent the duplication of CDs by writing to sectors of the disk which are not usually able to be written to.
3. **Hardware serial numbers:** is software which examines a machine's serial number, and if the serial number doesn't match then the program won't execute. This means that the software is tied to that specific machine and is usually preinstalled or sold together.
4. **Site Licence Installation Counter on a Network** gives organisations a maximum number of installs to their computers (connected to their server). The licence may either limit the maximum installations or limit the amount of simultaneous users. Once the limit has been reached no more installs can be made (or no more simultaneous users can be added).
5. A **registration code** is used to activate software during its installation. You receive a specified amount of registration codes (or uses of a single code) which is usually verified over the internet.
6. An **encryption key** is used to encrypt and decrypt data. The two most common systems are 'single key' and 'two key' systems. Single key uses the same key for both encryption and decryption. Two key produces a public and private key pair (usually specific for that session) where the public key is used for encryption and can be sent publicly, and the private key is local and can be used for decryption.
7. **Back-to-base authentication** means the application checks the publisher's server to see if your application holds a valid licence. If yes, you will be granted access, if not you will be denied access. This is either done during installation or each time the software is run. This is really only possible if an internet connection is available.

Use of networks

Use of networks by developers

1. **Access to resources:** the internet provides many resources including everything from documentation to source code, all available in an instant, some free and some for a fee.
2. **Ease of communication:** networks and the internet allow developers to communicate and collaborate on projects without having to meet face-to-face, they may even live in different countries and speak different languages.
3. **Productivity:** the ease of communication and access to resources both improve productivity.

Use of networks by users

1. **Response times:** response time is the time between a user's request and the server's response, basically loading times. Response time is important for users, slow response

times lead to dissatisfied customers and a poor perception of quality. This can be difficult to control by the developer as periods with high activity/high server load lead to slow response times.

2. **Interface design:** when designing an interface, the developer has to be careful about using large files such as Flash and high quality video and images. These files may produce a well designed UI although if response times are too slow, users won't even be able to see this UI.
3. **Privacy and security issues:** Davis text. didn't mention this.

The software market

Maintaining market position

Company's should always consider social and ethical issues when creating and marketing a software product. Davis splits this section up into the 4ps:

1. **Product:** you are not so much selling a product as you are the expectation of a product. The product should attempt to exceed customers expectations to maintain or gain a market position.
2. **Place:** depending on the type of product, software may be sold through different methods e.g. software retailers OR industry specific software distribution such as MYBOS OR direct sales such as over the internet, direct mail.
3. **Price:** software pricing should be tailored according to the quality of the product, desired profit margin and most importantly what your intended market will pay.
4. **Promotion:** software should be promoted in an ethical manner and consumer needs should also be the centre of the promotion.

The effect of dominant developers of software

If a developer/company dominates the market, it becomes the default purchase for consumers, making it difficult for other to access a piece of the market. An example would be Windows - Windows comes preinstalled in basically every computer so if another OS wanted to come into the market, it would be extremely difficult to gain traction.

The impact of new developers and software products

Despite market domination, new developers are still able to break into the market. Small ideas can lead to large products and companies such as Google and Facebook. New developers can potentially grow so large that they become a threat to other company's who previously dominated the market. As well as this, large amounts of developers (such as app developers) can collectively create a very large impact on the industry.

Legal implications

Software implemented throughout a country can lead to significant legal action if the software breaks the law in some way e.g. breaching a contract, breaching copyright, breaking the country's law etc.

National legal action

Refer to textbook.

International legal action

Refer to textbook.

2. Application of Software Development Approaches

Approaches used in commercial systems

When developing a software product, the company needs to decide on the most appropriate development approach(s), method of installation and what resources are required.

Below is the system development cycle (SDC):

1. **Defining and understanding the problem:** the requirements are thoroughly understood including determining whether the existing system is viable or a new system is required.
2. **Planning and design:** plans about the systems design and function are created here as well as data structures and algorithms. System modelling tools will aid with these tasks. By the completion of this stage, the software should be ready for implementation.
3. **Implementation:** source code is written based on the plans from the previous stage. User manuals and other helpful info are documented here.
4. **Testing and evaluating:** testing of the implemented solution takes place here. Testing is either done internally or released to the public for open testing.
5. **Maintaining:** software has to be continually corrected and refined. Maintenance may include bug fixes (which may be caused by anything including an updated OS version) and software upgrades to meet new requirements.

Structured approach

- Follows the SDC strictly.
- Each step must be completed before moving to the next (which is why this approach is also called the 'waterfall method')
- The 'defining and understanding the problem' stage is vital. Although errors created during this stage are easy to fix before moving on, they become increasingly time consuming and [very] expensive as you work through the remaining stages.

Agile approach

- Focused on the team (generally up to 6 people) developing the project instead of following predetermined requirements.
- Well suited for web development and mobile app development.
- Quick to develop initial release.
- Continual improvements with added features are regularly made and delivered.
- Developed closely with users/clients.
- Can be difficult to outsource because detailed requirements aren't made. The solution is usually to set a fixed budget and time, and once these are exhausted, then the software is released. This requires a lot of trust between developer and customer.



Prototyping approach

- Was created for the same reason as the agile approach, to adapt to changing requirements, unlike the structured approach.
- The problem is defined, prototype is planned, then implemented, then tested, then shown to the user who will define new requirements, this will repeat until the software is considered ready which the prototype will then be made the final solution (evolutionary prototype) or using the prototype, a new system will be replicated then released.
- There are two types of prototypes:
 1. **Concept prototypes** is where the prototype is never meant to be the final product. Its made to aid in the definition and refinement of requirements.
 2. **Evolutionary prototypes** are prototypes which are intended to become the final product. The prototype will be continually refined until it is considered suitable for final implementation.

Rapid Application Development (RAD)

- Focused on quickly implementing an operational product. Quality usually suffers.
- There are no formal stages.
- Usually close communication between developer and client.
- Advantages:
 - Saves time
 - Potentially saves money (if the modules are cheaper than development from scratch)
- Disadvantages:
 - RADs reliance on premade modules and CASE tools limit its functionality.
 - Other software may be required to be installed on users machines (e.g. Adobe Air).
 - May contain many bugs (due to limited time for testing and lack of planning).
 - Performance may suffer.

End user development

- End user development is where the person or business create their own software using wizards and other automatic code generation tools. (e.g. Wix, online app builders.)
- Low cost
- A compromise between requirement and functionality will have to be made.

Combinations of approaches

Usually, a combination of approaches will be most suitable as opposed to strictly following one approach.

CASE tools in large system development

Computer Aided Software Engineering (CASE) tools assist and coordinate activities during software development. Anything which aids in development is classified as a CASE tool e.g. Microsoft Word aids in the development of documentation. But many times specific tools are needed.

Oracle Designer

Oracle designer assists in the creation of system modelling diagrams and source code generation (production of code). Help files for the application can also be automatically generated (production of documentation). Whenever an object is modified Oracle Designer

stores a record of the modification coupled with the user who made the modification and then increments the version number of the object (version control).

DataFactory

DataFactory is a test data generation tool. It allows the testing of software with large amounts of data without actually having to release the product to the masses. They can simulate many real world situations such as testing odd dates e.g. leap years.

Methods of installation

When a new product is produced it must be installed and implemented on site. There are four common methods of installation (visual representation of these methods are extremely useful for memorisation):

Direct cut-over

This is where the old system is completely dropped, and the new system is put into full function. To do this, you must make sure the new system is fully functional and meets all its requirements. This method is usually chosen when it isn't feasible to run two systems at once.

Parallel

Parallel involves running both systems at the same time for a specific amount of time. Once the time is up, the old system will be terminated usually using the direct cut-over method. This method allows any critical problems encountered with the new system to be fixed while the old system is still fully operational (sort of as a backup).

Phased

Phased installation involves the gradual introduction of the new system whilst gradually removing the old system. This is usually done by replacing the old modules with the new ones. This method is often used when the system as a whole isn't complete.

Pilot

The pilot method involves introducing the new system for a small amount of users (think of beta testers). This allows a small amount of users to test the product and when it is finally introduced, they can teach the rest of the team how to use the new software.

*Note: examples, benefits and disadvantages of each method will most likely be required if they are asked in the HSC exam.

Employment trends in software development

Previously, employment in software development was largely based on experience. Today, tertiary-qualified graduates are generally required. On top of this, the IT industry is rapidly changing which means employees have to continue to update and refresh their skills. There are two recent employment trends:

Outsourcing

Businesses nowadays seek the services of a specialist software developer instead of writing their own in-house software. It is more feasible to outsource instead of creating in-house software especially when a business doesn't possess the resources to do so. There are many benefits to outsourcing including:

- Better response to change
- Saves time
- Generally higher quality results
- Reduces costs

Contract programmers

Many job positions are opening for short term contracts. They are usually either specified for a specific time (e.g. 6 months) or until a project is finished (think of seek.com.au contract jobs). They can also be freelance jobs.

Trends in software development

Changing nature of work environment

- This is similar to the 'use of networks' under social and ethical issues, "networks and the internet allow developers to communicate and collaborate on projects without having to meet face-to-face, they may even live in different countries and speak different languages."
- Another example is that when working within a company's development team, you generally must be multi-skilled and work together to collaborate on projects. This never used to be the case, programmers used to be given a specific task which they would then develop in isolation.

Changing nature of applications

Due to the rapid rise of the Internet and mobile devices, the nature of applications have drastically changed (and still is changing). No longer is most software designed solely for desktops or servers. Here are some examples of the current trends in applications:

1. **Web-based software:** today, most websites generate dynamic content using many interactive and intuitive tools (e.g. use of HTML 5 or third party embedded languages).
2. **Learning objects** are collections of resources used for teaching. They are usually delivered over the internet or from an institutions intranet (e.g. Treehouse, Coursera, Udemy).
3. **Widgets** have multiple meanings: they can mean any graphical element e.g. textboxes, they can mean desktop widgets e.g. android widgets, and they could mean an embedded widget into a website e.g. a little scrolling Facebook widget. Whichever meaning, they are becoming increasingly popular.
4. **Apps:** the word 'apps' have evolved to mean mobile applications. Due to limited RAM and storage, apps are generally small apps compared to their desktop counterparts.
5. **Applets** are just small applications. An example would be notepad which includes basic text processing but isn't a fully featured word processor. Applets also include plugins such as Flash Player.
6. **Web 2.0 tools** aren't really new technologies, it is more about the nature of these tools. They allow enhanced collaboration and user participation.
7. **Cloud computing** refers to applications which run over the internet. Some parts of the application may run locally but most run from the server. The application is usually accessed by a web browser. This allows files to be accessed and edited by multiple users (e.g. google docs) upon many other features.
8. **Mobile phone technologies** are progressively becoming more powerful. They include GPS, Cameras, Wifi, Bluetooth, NFC, 4G and the list keeps growing. This allows great potential which developers can harness using their apps.
9. **Collaborative environments** are continually emerging which aim to improve online collaboration. They allows friends, family and workers to communicate as if they were together. This may include audio, video, social networking, collaborative works (e.g. google docs) and much more.

SOFTWARE DEVELOPMENT CYCLE

Below is the system development cycle (SDC):

1. **Defining and understanding the problem:** the requirements are thoroughly understood including determining whether the existing system is viable or a new system is required.
2. **Planning and design:** plans about the systems design and function are created here as well as data structures and algorithms. System modelling tools will aid with these tasks. By the completion of this stage, the software should be ready for implementation.
3. **Implementation:** source code is written based on the plans from the previous stage. User manuals and other helpful info are documented here.
4. **Testing and evaluating:** testing of the implemented solution takes place here. Testing is either done internally or released to the public for open testing.
5. **Maintaining:** software has to be continually corrected and refined. Maintenance may include bug fixes (which may be caused by anything including an updated OS version) and software upgrades to meet new requirements.

3. Defining and Understanding the Problem

From page 2. (before SDC) "the requirements are thoroughly understood including determining whether the existing system is viable or a new system is required. "

Defining the problem

To define the problem the following questions need to be asked (these are from syllabus and textbook):

- What are the needs of the client?
- Possible compatibility with existing hardware and software
- Possible performance issues (especially for internet and media systems)?
- What are the boundaries?

Needs of the client

In order to properly define the problem, the needs of the client must be examined. Needs are often expressed in non-specific, emotive terms such as "we want to improve our turnaround times" or general statements such as "we need to be able to monitor sales made at each of our stores". In order for these needs to be properly understood an array of techniques may be implemented such as:

- Surveys - useful for gathering information from all across the business (large amounts of people). Limited in the way of detail and explanations that participants are able to provide.
- Interviews - useful because participants can freely express (with detailed explanations of) their needs. Interviews are obviously labour intensive and therefore expensive.

The following will also need to be considered as part of understanding client needs.

1. **Functionality requirements** describe the aim of the system. If the aim is achieved, the needs are met. Requirements must be verifiable which means it must easily be checked to see if needs are met and are usually mathematical or scientific. For example 'the system will be able to redraw the screen at least 12 times per second'. This is a result which can be measured and verified.
2. **Compatibility issues:** Requirements must be specified so that compatibility issues are at a minimum. Questions such as 'what software/OS will the program run on?' or 'what hardware will the program support?' must all be considered.
3. **Performance issues:** when defining the problem, performance should always be considered and developers should attempt to create efficient algorithms and software. It is sometimes hard to gauge the consistent performance of a system before it is built. Depending on the complexity, size, networking and many other factors of the system, its performance may vary.

Boundaries of the problem

When designing a system, its boundaries and environment must be thoroughly understood. It is possible that an item in the program's environment will affect the program, so it must be able to cater for that even though the system cannot alter its environment.

Issues relevant to a proposed solution

Sometimes it is feasible to continue using the existing system. The cost, time and effort of developing a new system may outweigh the intended benefits. As well as this, there may be other ways to go about the business' problem. These may be completely unrelated to software and may also solve the problem in a more cost effective and efficient way. For

example, generally a solution to motorway traffic congestion may be to add another lane, however, other methods such as upgrading the public transport system could potentially be the more feasible option.

Determining if an existing solution can be used

An existing system may be the system the business is already using, or it may be a system which is already created somewhere else (premade software which can be purchased e.g. MYOB).

The following underlined headings should be considered when comparing the use of new software with the existing software.

Social and ethical considerations

- **Changing nature of work:** Implementing a new system may affect the employees nature of work. They may have to be retrained and contract workers may have to re-negotiate contracts. Input from all users of the system is very important for these reasons.
- **Effects on level on employment:** computer systems and software are generally developed to reduce costs and some of the highest costs a business incurs are salaries and wages. Developers and clients must take this into account for the reasons of cost saving as well as employee redundancy.
- **Effects on the public:** large software systems can have a tremendous effect on the general public. In the case of ATMs, literally the entire population were retrained in order to use them. Some people - particularly older people- were reluctant to use the technology. Online banking and billing has had the same effect. The effect of software on the public should always be considered.

Legal issues including licensing considerations

not sure what to write here...doesn't seem like Davis does either

The following is a complete guess of something to write:

- Maybe there is a contract with the existing system which is about to terminate and must be renewed to continue using the system and to avoid possible legal issues. This would be a consideration when deciding whether to keep the system I think.

Customisation of existing software products

Many times, customising an existing product is much more cost effective than developing a new one. This is usually done to add new functionality and is either done by updating the existing product, or creating a new one and adding it to the existing one (an add-on).

Cost effectiveness

Costs and budgets are obviously a major constraint on development. Constant monitoring of costs should take place to ensure development won't go over budget. When creating a budget, some of the areas that should be considered are:

- Hardware costs - will new hardware have to be purchased/leased?
- Software costs - will CASE tools, DBMSs, graphic creation software have to be purchased?
- Personnel costs - salaries of all staff (dev. team, support, trainers, management etc.)
- Outsourcing costs - what will have to be outsourced? What's its cost?

Selecting an appropriate development approach

It will need to be decided whether to purchase/upgrade an existing system or to develop a new system. *this may also include development approaches e.g. structured, agile, prototyping etc. Davis seems unsure.*

Design specifications

Design specifications are used as the basis for planning and designing the solution. It aims to interpret the needs, requirements and boundaries of the system and form them into a usable and workable form. They should include specs from both the developer and user's sides.

Developers perspective

The developers specifications will not directly affect the product from the users perspective but they will give the developers a framework for all of them to work by. These may include:

- Specifying what system modelling tools will be used (e.g. flowcharts, DFDs. IPOs etc.)
- **Descriptions of algorithms**
- **Setting up naming conventions for data types and structures**
- Setting up a system to maintain a data dictionary

CASE tools will aid in making sure developers comply with the specifications. Once the specifications are developed, a system model can be developed and specific tasks for each member can be allocated.

User's perspective

These specifications should include any design specs that may affect the user's experience such as:

- **Interface design:** consistency should be used when designing the screen. It should also be user friendly and appealing to the eye.
- **Social and ethical issues:** ergonomics should also be considered. e.g. what text size, font, colour? Keyboard shortcuts? What data entry methods should be used?
- **User's environment and computer configuration:** the software should utilise OS settings such as fonts, font sizes, colours and printer setting (as the user may have changed the settings to meet their needs). Also, a consistency with common usage should try to be achieved such as common keyboard shortcuts and design elements (think google design guidelines which streamline usage across all apps e.g. 3-dot menu button, swipe over navigation drawer etc.) as this allows a transfer of skill sets and minimal retraining.

... (skipped a large portion of the textbook and syllabus. Refer to other resources for this section such as MrBrightside's notes and other online resources as well as the textbook/s)

Quality Assurance

Quality assurance ensures standards and requirements are met throughout the development of software such that the final products are of high quality. This is important as peoples perception of the products quality impacts the overall success of the product.

The areas that should be assessed to determine quality include:

- **Efficiency** - how efficiently does the software utilise the systems/computers resources e.g. RAM and CPU?
- **Integrity** - this is the correctness of data. This improves when data validation is utilised e.g. checking for valid phone numbers, addresses and email addresses.
- **Reliability** - will the software work without failures? If the product does fail (or encounter some kind of error) then how long will it take to fix?
- **Usability** - user friendly? Easy to learn and use? Ergonomic?
- **Accuracy** - does the software perform its functions correctly according to its specifications? The code should be well documented to maintain accuracy during

future upgrades.

- **Maintainability** - how easy can changes be made to the software? Well documented code is much easier to maintain.
- **Testability** - the ability to test all aspects of the software. Testing should occur on individual modules and the system as a whole.
- **Reusability** - the ability to reuse code in other systems. Once modules are created there is no need to create them again. This saves time, effort, money, and generally modules are thoroughly tested .

4. Planning and Designing Software Solutions

...

Small note: you plan and design the software using pseudocode, flowcharts, storyboards, IPO charts etc.

*Note: there are large parts of this chapter which I haven't written notes for. Also, Davis has structured the textbook quite differently to the syllabus so headings may be different and this chapter's notes may be (and probably is) incomplete. Just follow the syllabus closely and make sure to cover everything, even though they may be out of order.

Custom designed logic used in software solutions

Before creating an algorithm, it is necessary to know the inputs, processes and outputs required. This info is obtained through system modelling tools.

To create a custom algorithm (this isn't a syllabus list, although its still helpful):

1. Create and obtain data from system models.
2. Figure the inputs, processes and outputs from the previous step
3. Design data structures and assign data types
4. Document previous step's information in a data dictionary.
5. Commence development of algorithm.

Data structures and files

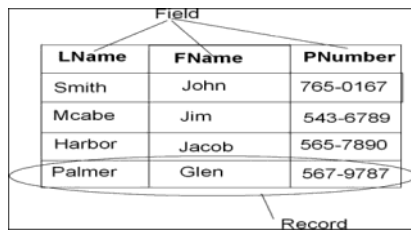
Definition: A data structure is a specialized format for organizing and storing data e.g. array.

Efficient data structures assist in the development of algorithms. Here are some common data structures from the syllabus:

1. **One-dimensional arrays** are a collection of data items of the same data type.
2. **Multi-dimensional arrays** are array with multiple indexes/dimensions. Often the dimensions/indexes are called subscripts.
3. Two-dimensional arrays and other multi-dimensional arrays have
 - a. **Two-dimensional arrays** can be visualised as a table.
 - b. **Three-dimensional arrays** can be visualised as a cube.
 - c. **Four-dimensional arrays** can't be visualised but have 4 indexes.
 - d. An example of a multi-dimensional array would be: AssessResults(YearLevel, ClassNum, StudentNum, TaskNum). So AssessResults(12, 4, 8, 3) would retrieve the results of the 3rd task of a year 12 student in class number 4, with a student number of 8.
4. Files
5. Records
6. Arrays of records

Arrays of records

Records are data structures which are a collection of fields. Arrays of records are just a collection of records. Once a record is declared, it becomes a data type and can be accessed the same as any other data type e.g. you can enter data into a record called MyDetails by typing MyDetails.Surname = "Peters".



Definition: Files are a collection of data that is stored in a logical manner.

Sequential files

- Sequential data must be accessed from beginning to end (think of cassette tapes). You cannot access a part of the file if you haven't first accessed what's preceding it.
- Because of this, if you want to write to the file, you can only append. (you can't change any data, you can only add data).
- You can open a sequential file in three ways - **input**, **output** and **append**. **Input** is used to read data, **output** is to write data to a new file and **append** is used to write data to the end of an existing file.
- A **sentinel value** is a dummy value used to indicate the end of the file (or logical breaks in data).
- When reading a file, a **priming read** is used before the main processing to check if the file only contains a sentinel value.

Relative (or random access) files

- **Definition:** a random-access file enables you to read or write information anywhere in the file.
- To do this, each record must be the exact same length. So if each record was 10 bytes and you wanted to access the 10th record, you could easily jump to the 10th record because it's the 100th byte. In essence, the position in each record in the file is used as a **key** to allow direct access to other files.
- Because each record is the same size, it is possible to modify any data anywhere in the file, as opposed to sequential files where you can only append data.
- Because each record has to be the same length, sometimes blank characters (ASCII code 32) may have to be inserted to fill up the extra space.
- Also known as 'direct access' files.

...

Developing test data

- Test data should test every possible route through the algorithm as well as test each boundary condition. (p207 Davis Text.) Where 'testing possible routes' ensure each statements works correctly with each other statement and 'testing the boundaries' ensure that each decision is made correctly e.g. an error in boundary checking would be Result < 10 instead of Result <= 10.
- For example, looking at the algorithm below, we need to test:
 - Each boundary condition: according to p207, we will need to test 25, 80, and a less than 25.
 - Every possible route: we will need to test a number between 25 and 80, a number greater than 80, and a number less than 25.

5 The following algorithm needs to be tested for logic errors.

```
BEGIN
  input amount
  IF amount > 25 THEN
    IF amount ≥ 80 THEN
      discount = 30%
    ELSE
      discount = 20%
    END IF
  ELSE
    discount = 10%
  END IF
  output discount
END
```

Which of the following sets of test data would provide the best test for the algorithm?

- (A) 15, 25, 50, 80, 100
- (B) 24, 26, 50, 79, 81
- (C) 10, 20, 25, 30, 80
- (D) 5, 10, 15, 20, 25, 30, 35

Interface design in software solutions

The design of the interface is influenced by the nature of the problem. In **event driven approaches** (a program where user input determines the order of processing) the screen designs are crucial to the operation of the product whereas a programs where most of the work takes place in the background i.e. **sequential programs** (sequential programs are those where the code itself determines the order of execution, they have a set start and end and they may even direct the user) e.g. drivers, the interface design will have a miniscule effect on the productivity of the system.

Identification of data required

Once the required data to be used on each screen is acquired (from previous modelling tools), you will need to decide which screen elements to use to display this data. These may be:

- Menus e.g. dropdown
- Command buttons
- Check boxes
- Radio buttons etc.

Current popular approaches

- The design of the screen elements is usually dictated by the OS.
- Internet solutions will be executed on a large and unknown selection of systems so they require a suitable design across all computers.
- As computers become more powerful, higher interactivity and resolutions can be used.

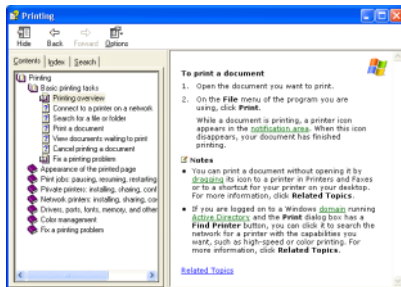
Design of help screens

Software should aim to be consistent and easy to use, inevitably however, some users will require assistance. Some common help screens are:

- **Context sensitive help:** self-explanatory, shows help for what they user is currently working on e.g. hovering over a textbox and a little contextual popup shows up.
- **Procedural help:** shows how a task is to be accomplished. Usually contains an index,

contents and search options. Usually for 'how' tasks are to be completed. Look at image below.

- **Conceptual help:** usually for 'why' tasks are to be completed. For example, 'why' files need to be saved. Usually accessed from a procedural help screen.
- **Tours, tutorials and wizards:** you know what these are. Tours shows the features of the product, tutorials walk you through them step-by-step, and wizards automate complex tasks using dialog boxes.



Audience identification

The interface needs to be design to suite the intended audience. For example, the design of a program written for pre-schoolers will differ than a program written to carry out complex calculations for engineers.

Consistency in approach

Consistent user interface allow users to transfer their existing skills to new systems.

Examples of consistent design include:

- Names of commands - common commands should keep the same names e.g. copy, cut and paste.
- Use of icons - e.g. the save button is always a floppy disk.
- Placement of screen icons - e.g. save button is always in the toolbar.
- Feedback - use of loading bars and icons.
- Forgiveness - warnings about potentially dangerous operations and their recovery methods such as deleting a file should always be included and be kept consistent.

Customisation of existing software solutions

Many COTS (Custom off-the-shelf) packages allow customisation to be made using an included programming language. So for example, you may purchase a software with limited functionality, but allows you to add to the program with your own code.

Selection of language to be used

Algorithms should be written in such a way that they could be implemented in a variety of languages (think of an app made for Android, which in the future, may also be implemented on iPhone, this should be made easy to do from the beginning). Here are some questions to consider:

- Will the language be event-driven or sequential? (the order of processing is chosen by the user or by strict rules set by the programmer? e.g. MS Word or windows drivers)
- Will the language provide all required features?
- Is a GUI required?
- Is the language suitable for the hardware and operating system ? (does it run on the OS? Does it support system settings and hardware?)
- Do the developers know the language?

Factors to be considered when selecting the technology to be used

Sometimes software being produced is part of an entire information system, in this case,

system designers must determine the type of technology to be used.

- **Performance requirements** such as CPU and HDD/SSD speeds should be determined. Also fault tolerance should be considered such as reliability, recovery from power spikes etc.
- **Benchmarking** is the act of running a program in order to assess its performance against some standard. For example, testing the ability to render 3D graphics in HD at a particular frame rate would be a good performance measure for gamers.

5. Implementation of Software Solutions

This is the third stage of the SDC where the previously planned algorithms (planned using pseudocode, flowcharts, diagrams etc.) are actually written, along with the rest of the source code, and into a form where it can finally be processed by the computer.


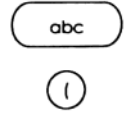
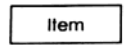
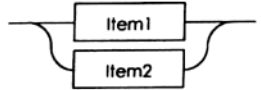
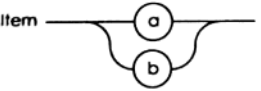
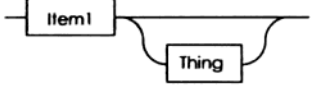

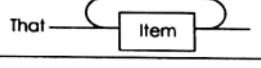
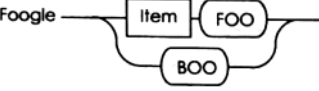
Choosing an appropriate programming language

Here are some questions to consider when choosing an appropriate language:

- What is the nature of the program? (e.g. will it run complex AI algorithms or will it be an online e-commerce store?)
- Will the language be event-driven or sequential? (the order of processing is chosen by the user or by strict rules set by the programmer? e.g. MS Word or windows drivers)
- Will the language provide all required features?
- Is the language suitable for the hardware and OS? (does it run on the OS? Does it support system settings and hardware?)
- What are the developers most familiar with?

Language syntax required for software solutions

The rules that tell us how to construct a statement in a specific programming language is called the syntax. Metalanguages are used to represent and define the syntax of a programming language, they describe the rules of the language.

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	

^^p227 Davis text.

*Note: take a look at the textbook for solid examples of these diagrams.

- Terminal elements are used exactly as they appear
- Non terminal symbols (<Item> or a square in RRD) are defined elsewhere.

EBNF

- Extended Backus-Naur Format is a metalanguage which supplies a formal method of describing the syntax of a language.
- It differs slightly from BNF as it has added repetition and options (i.e. where '{}' means repetition and where '|' means 'alternative').

Railroad diagrams

- These are a visual alternative to EBNF, which means it works in a very similar way.

Translation methods in software solutions (syllabus: the need for translational to machine code from source code)

Definition: translation is the process of converting high-level source code into executable machine code (or object code). The two common methods of translation are interpretation and compilation.

Interpretation

Interpretation is where source code is translated, line by line, into an intermediate language and then immediately executed. So basically, it translates the first line and runs it, then translates the second line and runs it etc. The intermediate language isn't necessarily machine code.

Commonly used on web servers and languages such as Python and Ruby.

Advantages:

- Easier to debug because translation occurs line by line until there is an error, which it will then stop and can be quickly corrected by the developer; that single line will be retranslated & execution will continue.

Disadvantages:

- Slower program than in compilation as each line of source code is translated right before it is executed.
- For a program to be executed, the user must obtain the source code itself, which raises the issue of intellectual rights.
- Users must also have an interpreter installed, which uses resources i.e. memory.
- The intermediate code can take longer to process compared to machine code.

Correct Answer C

An advantage of using an interpreter to translate source code is that

- (A) ☐ it uses less space in RAM.
- (B) ☐ the source code is protected.
- (C) ☐ syntax errors are located easily.
- (D) ☐ the machine code can be used by any CPU.

Compilation

Compilation is where the source code, as a whole, is translated into machine code. The

executable code can then be executed at a later time without the need for a translator. When there is an error, the translation will fail and a list of the errors will be displayed.

Commonly used in programming languages such as C, C++.

Advantages:

- Program runs faster because code is already translated.
- Program will be distributed in machine code, thus protecting intellectual rights, and avoiding many issues of interpretation.
- Compiled/machine code is usually smaller than source code and interpreted code, thus reducing file size and resource requirements.

Disadvantages:

- Harder to debug as the entire program runs before the errors are produced.
- The whole program needs to be recompiled when changes are made.
- Because the code will be in machine language, the code will be machine specific, which means the program will need to be recompiled using a compiler for the specific processor or OS.

Steps in the translation process

There are three major tasks when translating code: lexical analysis, syntactical analysis and code generation.

Lexical analysis

A lexical analysis is the process of converting source code, character at a time, into a sequence of tokens.

Each character is read according to the rules of the specific programming language being used. Characters such as spaces, indentation and comments are discarded. Programming languages will have a predefined **table of symbols (a symbol table)** which source code will be compared to, character by character. If there is a match e.g. WHILE, that part of the source code (called a lexeme, which is a string of characters) will be replaced by a token. Symbols created by the programmer, which aren't predefined, will be given a token, and will be added to the symbol table. This will continue until the entire source code has been tokenised.

Error messages may be produced by a lexical analysis as a result of incorrect syntax, naming, undeclared identifiers etc. This is because they don't conform with the specific programming language's syntax.

Second try at explaining (after 2nd read):

The lexical analysis is the process of examining each word in the source code to ensure it is a valid part of the language. So, it ensures that all reserved words, identifiers, constants etc. are actually part of the language. It doesn't make sure everything is in the right order or syntax, it just makes sure that any elements used are actual, legitimate language elements.

Syntactical analysis

Syntactic analysis, or parsing, is an examination which 'tests the validity' of whether the identified elements (those assigned with a token) of a statement are combined together in a way that is legal according to the syntax of the language. This is because the syntax not only consists of rules to determine the elements of the language, but also the ways these in which these elements interact e.g. `age = 17` is syntactically correct as it is assigning a value

to a variable, `age = "ABC" + 17` is obviously syntactically incorrect.

There are two aims of a syntactical analysis: (1st) to ensure the source code adheres to the specific languages' syntax rules and (2nd) to check the validity and integrity of data types.

Step 1: **Parsing** is the process of actually checking the syntax of a sentence (I'm not quite sure which, but either the source code or the tokens is what's being 'parsed', it makes sense that it's the tokens though). A parse tree is created using the tokens from the lexical analysis, then each statement will be compared against the language's specific syntax rules. If any errors occur at this stage, i.e. the tokens cannot be parsed, then it means the tokens are arranged incorrectly and therefore a syntax error.

Step 2: **Type checking** "tests the validity and integrity of data types (i.e. makes sure strings aren't added to integers) appropriate storage locations are also allocated. "

Code generation

Code is generated once the lexical and syntactical analyses are completed without error. This is where the tokens are converted to machine code, there will also be no errors found in this stage.

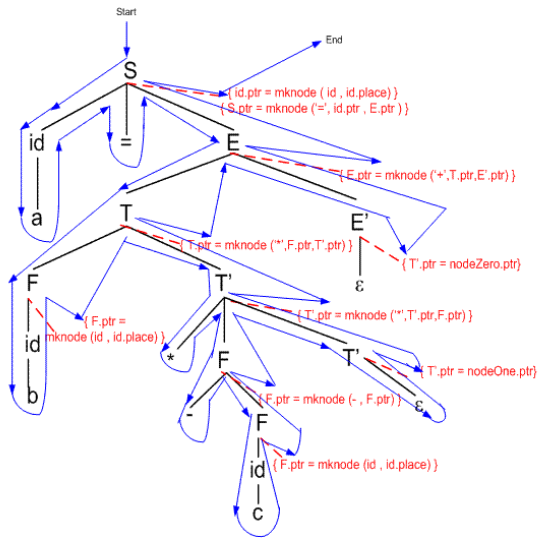
If an interpreter is used, then each statement will be executed as soon as its translated. In a parse tree, it will be traversed from left to right when generating the code.

Second try at explaining (after 2nd read):

This is where each token is converted into machine code. No errors occur in this stage as any error would have been detected in previous stages, i.e. lexical and syntactical analyses. **Traversing** is when the parse tree is 'traversed' from left to right (it sort of works its way around the tree starting from left side of the root node and ending on the right side of the root node...look at the blue lines in the picture below), and each token is converted to machine code.

If a compiler is being used, the resulting machine code is stored in an object file. A **linker** links runtime libraries and dynamic link libraries (DLLs) required for execution. (A dynamic link library -DLL- is a file containing object code subroutines that add extra functionality to high level languages. They need to be distributed -may just exist in a folder or may be installed- with applications so that they are present on the users computer.)

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



The role of machine code in the execution of a program

Machine code

1. **Word size:** the amount of bits the processor can process in one go. Generally 32 bits or 64 bits in modern computers.
2. **Register:** temporary storage location within the CPU. The size of the register is the same as the word size of the processor.
 - **Accumulator:** general purpose register used to accept the results of computations from the ALU.
 - **Program counter:** stores the address of the next instruction being executed by the CPU (it is only ever incremented by one, or is reset).
 - **Instruction register:** stores the current instruction being executed by the CPU.

Less important definitions:

1. **Interrupt:** how the CPU communicates with outside devices. The device responds by reading or writing to the accumulator (top of p251)
2. **Microcode instructions:** permanent instructions built into the CPU which carry out specific processes. Machine language instructions typically translate into one or more microcode instructions.
3. **Instruction set:** a list of microcode procedures permanently stored in the processor.

How machine code is executed

Each machine code instruction activates a microcode procedure. Each microcode procedure is executed using a complete fetch-execute cycle.

Every CPU contains three basic units (which are all combinations of circuits):

- **Arithmetic Control Unit (ALU):** carries out all the arithmetic (e.g. +, -, /, x) and logic (AND, OR, NOT, NAND, XOR etc.) in the CPU.
<https://www.techopedia.com/definition/2849/arithmetic-logic-unit-alu>
- **Control Unit (CU):** decodes instructions to be executed (which determines the nature of the instruction).
- **Registers:** explained above.

Techniques used in developing well-written code

The use of good programming practices

1. **Use of a clear and uncluttered mainline and structure:** well structured mainlines

should utilise modules. This utilises the recommended top-down design. Structure charts aid in planning this.

2. **One logical task per subroutine:** keeps things easier to understand, debug and maintain.
3. **Use of stubs:** this enables higher routines to be tested before lower routines are created.
4. **Appropriate use of control structures and data structures:** (control structures i.e. sequence, repetition, subprograms, selection etc.) the conditions determine the only exit for the loop, it shouldn't exit unconditionally or unexpectedly. (data structures i.e. arrays, records, files, tables etc.) should assist in processing.
5. **Writing for subsequent maintenance:** should be considered before coding commences. Documentation, proper naming and indenting should be used.
6. **Version control and regular backup:** code should be saved and backed up regularly to prevent loss of data and to revert to older modules if needed. Version control allows entire applications to be developed while regularly releasing versions to users.
7. **Recognition of relevant social and ethical issues:** when writing source code, intellectual rights should be respected, thorough testing should occur so the software acts as expected, and the code should be documented for future maintenance.

Errors occurring in software solutions

Three types of errors are syntax, logic and runtime.

Types of errors:

1. **Syntax:** syntax errors are those that break the rules of the specific programming language. They prevent the translation of code and are found during the syntactical analyses. Examples would be typing and spelling mistakes of data types.
2. **Logic:** these are syntactically correct code which doesn't produce the expected output or complete the desired task. These are the most difficult to debug as there is technically nothing wrong with the code. An example would be $\text{Avg} = 1 + 2 / 2$ instead of $\text{Avg} = (1 + 2) / 2$. If the developer cannot find the issue, peer checking and desk checking may help.
3. **Runtime:** these aren't detected until the program is run. They can be caused by:
 - **Arithmetic overflow:** when the result of a calculation is too large for the allocated storage area. For example, in most languages, an integer can only be approximately equal to ± 32767 , so creating an integer larger than this would cause an arithmetic overflow (a runtime error).
 - **Division by zero:** mathematically, dividing any number by zero is undefined and cannot be done.
 - **Accessing inappropriate memory locations:** for example, creating an array with 10 items, and trying to access the 11th.

Techniques for detecting errors:

1. **Flags:** a variable that indicates whether an event has occurred or not. They are usually a Boolean.
2. **Stubs** are a small subroutine, put in place of a yet to be coded routine. They test to see whether it actually runs and tests to see whether all modules work together.
3. **Drivers:** a driver provides an interface between two components. In software, a driver is a subroutine that tests another subroutine. They usually set the value of a variable(s), call a subroutine, and output the results from the subroutine. These are especially useful in bottom-up design as each subroutine will be tested as they are coded.
4. **Debugging output statements:** this is where the developer places an output

statement in the code to help isolate a problem. They may contain the value of a variable, a location of the statement, or simply a statement that the variable was called.

5. **Peer checking:** other developers/team members can often see the code from a different perspective and can be very effective in finding errors in code.
6. **Desk checking:** the process of working through an algorithm by hand, and writing down the value of variables as they change. They can aid in finding errors and can also help in understanding the algorithm.

Debugging tools:

1. **Breakpoints:** temporarily halting the execution of a program. Often used to inspect the program at a specific point/line.
2. **Resetting variable contents:** sometimes, it can be helpful to change a variable to one that is known (e.g. simple, easy numbers as opposed to large, complex numbers in a mathematical calculation) to easily test the output.
3. **Program traces:** allows the order of execution of statements to be tracked or the changing contents of variable to be tracked. It is a log about the programs execution. They analyse the flow of execution. Some development environments display each line of code as its executed, others allow you to analyse the call stack (shows info about any active subroutines).
4. **Single line stepping:** halting execution after each statement is executed.

Documentation of a software solution

User documentation

User documentation needs to be suited for their level of knowledge and expertise, i.e. simple. This is even furthered depending on the type of user e.g. kids will need documentation which is different than adults'.

1. **Installation guide:** inform the user on how to install the software, usually a step by step guide. Notes should also be included to resolve common installation errors.
2. **User manual:** provide comprehensive info about the operation and purpose of the software. They are commonly contained in a help file. Each topic should contain a conceptual explanation (e.g. saving stores a permanent copy of the file your editing) and a procedural explanation (e.g. to save a file click the 'file' menu, then click 'save').
3. **Reference manual:** designed to be an efficient source of information. They describe each command in an application. They are not made to be read from start to finish, rather the user will quickly scan and find what they need. (screenshot of a reference manual is below, from the Davis textbook.)
4. **Tutorials:** provide step by step instructions of the software using example scenarios. They take the user through the practical, real world use of the software. Often sample data, or even the users actual data, will be used in the tutorial. They are usually split into individual lessons.
5. **Online help:** all of the above user documentation can either be printed, provided with the software, or available online.

The ATAR Estimator Quick Reference Guide		
Menu Command	Shortcut Keys	Function
File – New	Ctrl-N	Create a new file
File - Open	Ctrl-O	Open an existing file
File – Save	Ctrl-S	Save the current file
File – Print – Individual	Ctrl-P	Print the current students results
Edit – Delete	Shift-Delete	Delete the current student record
Help – Using	F1	Open help
View – Individual	F2	Switch to individual student view
View – All Students	F3	Switch to all student view
View – By Course	F4	View students in a particular course

Technical documentation

As the name suggests, technical documentation is designed for a knowledgeable and proficient audience. They describe the engineering behind the software.

1. **Log books** or process diaries, record the actions that have occurred during the development of the software. They are used and created during the design and development of the software. They are in chronological order. They can be use to determine the cost of the project, and can be used as a reflective device (by examining mistakes, delays or inefficiencies during development, this can be prevented in future projects).
2. **Source code documentation:** or internal documentation is possibly the most important technical documentation. Two common forms of source code documentation:
 - a. **Comments:** these are included in the source code, usually before any function or process, which explains what the code is doing. They should explain 'what' the code is doing rather than 'how' it is doing it. This allows other programmers (or the same programmer at a future date) to understand the intention behind the code.
 - b. **Intrinsic documentation** is documentation that is part of the code. (think of: it is the source code itself, things like indenting and using useful names act as documentation itself). Two types are indenting and using meaningful identifiers. Identifiers (the names of variables) should be descriptive yet concise, they should describe the purpose of the variable.

Hardware environment to enable implementation of the software solution

1. **Minimum hardware configuration:** minimum hardware requirements should allow the software to operate successfully with acceptable performance and response times.
2. **Possible additional hardware:** is hardware which is supported by the software but not required e.g. more RAM.
3. **Appropriate device drivers or extensions:** hardware requires drivers to appropriately interface/communicate with the system.

Emerging technologies

Davis textbook doesn't say much about this, only (slightly) notable info is spoken about below

- Game consoles which include various motion, video and sound sensors.
- Biometrics e.g. finger prints, iris, face recognition and other biometric scanning devices.
- Scanning pens to scan text, phone camera to scan QR codes, smart card reader within eftpos, RFID for stock control.
- Mind control devices which detect facial expressions, emotional state and cognitive

process using 14 sensors.

6. Testing and Evaluating Software Solutions

Testing the software solution

Generating relevant test data for the complex solution

1. **Black box testing** doesn't try to find the source or the cause of a problem, rather it aims to identify that a problem exists. Usually done by comparing actual outputs with expected outputs.
2. **White box testing** comes after black box testing, and aims to find the cause of the problem and then fix it, which requires explicit knowledge of the internal workings of the system.

Levels of testing

Module testing:

- Each module is tested independently to make sure it works on its own, without any external influences.
- A **driver** may need to be used to substitute the mainline in order to provide an input and accept an output.
- Will usually use both black and white box testing.

Program testing:

- Tests to see that all modules (the entire program) work together as a whole.
- Concentrates on the interface between modules (because after module testing, everything should be working, right?).
- Program testing is usually done using bottom up or top down testing:
 - **Bottom-up testing** incorporates each module into the program for testing, one at a time, from the lowest/deepest modules up to the higher modules. Drivers will be used to sub for the mainline. The program as a whole will be tested last.
 - **Top-down testing** starts with the main program and uses stubs for some modules as it incorporates each module into the program.
 - The choice of testing is usually a matter of preference.

System testing:

- System testing is the testing of the program outside of its development environment, so basically, real world testing. This is usually done by those outside of the development team.
- This is done because the software may work perfectly in the controlled, or high-end, developers environment and computers.
- Usually black box testing (as you cant really white box test a user's computer)
- **Acceptance testing** is to determine whether the program has met its requirements. So basically, if it is of acceptable quality for delivery.

Using live test data to test the complete solution

1. **Larger file sizes:** the developer needs to test the software to ensure the program can handle (input) files larger than those usually expected, in case it is ever necessary. It is also important to test these large files to find the limits of the software. Any slow downs would highlight inefficiencies.
2. **A mix of transaction types:** test data should attempt to test/simulate as many possibilities/scenarios as possible. For example, the way I might use a software may be different to the way someone else does.
3. **Response times:** these are the amount of time taken to complete a process(s). These

need to be tested under light and heavy loads.

4. **Volume data:** or load testing, is to enter large amounts of data into the system to test the software under extreme load conditions. (think of stress testing).
5. **Interfaces between modules:** each module will need to be tested independently and dependently to ensure it outputs the correct data without any errors.

Reporting on the testing process

- It is crucial to **document** the test (what was tested, how it was tested, the result, and a recommendation) as it may need to be referred back to. For example, in the future, the developers may find an error, and they will need to see what was tested as something may have been missed.
- CASE tools provide assistance, and sometimes automate, the creation of documentation. They also aid in the creation of test data and can even simulate an environment for testing.

Communication with those for whom the solution has been developed

1. **Test results:** results should be communicated to both developers and clients and should cover both positive and negative results. Test results should contain info about problems identified (e.g. bugs, long response times etc.) and limitations of the program.
2. **Comparison with the original design specification:** the final report should detail the way in which the solution meets its design specifications.

Evaluating the software solution

1. **Benchmarking** is the comparison of the program against established standards. These standard may come from similar software (maybe a competitors'), from industry standard, or from in-house standards.
2. **Quality assurance:** aims to guarantee the software is of high quality. International Standards Organisation (ISO) developers standards for quality assurance.
3. **Acceptance tests:** spoken about previously.

7. Maintaining Software Solutions

Modification of code to meet changed requirements

1. **Identification of the reasons for change in source code:** Common reasons for a change in source code are bug fixes, new hardware, new software or added functionality.
2. **Location of section to be altered:** A thorough understanding of the software as well as DFDs, structure charts and other documentation aid in finding the location of the code to be altered.
3. **Determining changes to be made:** changes may need to be made to data structures, files, UI and to the source code.
4. **Implementing and testing the solution:** basically chapter 5 and 6.

Documenting changes

1. **Source code documentation:** includes comments and intrinsic documentation.
2. **Updating associated hardcopy documentation:** hardcopy documentation is rarely used anymore, updating this to online help is becoming more popular.
3. **Use of CASE tools to monitor changes and versions:** CASE tools aid in tracking changes such as documentation, testing, data structures, bug fixing etc. Version control CASE tools keep track/manage different versions of each module in the software. This is done to the extent that previous versions of the software could be completely reconstructed/restored if the developer wishes. (If asked about CASE tools to monitor changes and versions, speak about tracking changes and version control.)

8. Developing A Solution Package

more of a practical chapter which integrates previous chapters. Notes don't seem necessary, rather just read over the chapter to know how all the theory is implemented

10. The Interrelationship Between Software and Hardware

*Note: I didn't find it very practical to write notes for this chapter. It is heavily based on fundamental concepts and your ability to understand and make use of these concepts.

Rough/random notes

ASCII vs Unicode (HSC sample answers)

- Unicode can represent many more characters than ASCII (more than 1 million compared with 128). Unicode is a 'superset' of ASCII and can therefore represent all ASCII characters.
- The characters of most languages can be represented using Unicode, however the letters represented by ASCII are in English only.

<http://www.differencebetween.net/technology/software-technology/difference-between-unicode-and-ascii/>

<http://stackoverflow.com/questions/19212306/difference-between-ascii-and-unicode>

Programming of hardware devices

- Both the header and trailer are used to ensure that data is delivered correctly.
- A **data stream** is a sequence of packets being transmitted (look at Fig 10.82, p472).
 - Hardware manufacturers release specifications to specify the specific format of the data stream for each of their devices.
- **Control characters** are used to cause some predefined action by the hardware e.g. move a printer head.

A data packet is made of three components:

1. **Header:** this precedes the data block and, with hardware devices (connected directly to the computer), usually contains a single start bit. This signals to the receiving devices (e.g. computer) that a new data packet is coming (e.g. mouse was moved).
2. **Data:** the actual data being sent. Often contains control characters (e.g. a control character sent to a modem may tell it to go off-hook or dial a number).
3. **Trailer:** the information following the data block, usually contains error checks (often just a parity) and stop bits (to indicate the end of the packet).

Quick notes

1. A **latch** is a circuit that is used to store 1 binary digit. (it's the S/R flip flop which just loops around).
2. The **clocking component** allows us to control when a latch is allowed to change state (where a 0 means no changes can occur and a 1 means changes are allowed to occur...think of it as a gate where 1 is open and 0 is closed)
3. A **D-latch** is where the R component is removed, as R is always the inverse of Q, you can achieve the same outcome by replacing R with a NOT gate, as long as there is the clocking component).
4. The **edge trigger component** is used to ensure the flip-flop only changes state once for each tick (a tick is a 1) of the clock. They consist of two latch circuits, a master and a slave.

Shift registers are used to store binary data. They allow for data to be moved into and out of the register and they all share a common clock.

- a. This (shift registers) is what is used to perform **binary multiplication and subtraction**.

A shift of one bit to the left multiplies by two and a shift of one bit to the right divides by two.

- b. They are just flip flops grouped together in a chain.