BCS Foundation Certificate in Digital Solution Development

Syllabus Detailed Guidance

Version 1.0
November 2020

This qualification is not regulated by the following United Kingdom Regulators - Ofqual, Qualification in Wales, CCEA or SQA

Copyright © BCS 2020

Contents

Introduction 4
Syllabus Detailed Guidance 4
Abbreviations 108

Change History

Any changes made to the syllabus shall be clearly documented with a change history log. This shall include the latest version number, date of the amendment and changes made. The purpose is to identify quickly what changes have been made.

Version Number and Date	Changes Made
V1.0 Nov 2020	1st Issue

Introduction

The aim of this document is to provide additional guidance to students and training providers around the content of the syllabus for the BCS Foundation Certificate in Digital Solutions Development. The information contained in this document may form the basis of examination questions.

Syllabus Detailed Guidance

1. The context for digital solution development (10%)

The aim of this section is to ensure that candidates have a solid understanding of the context in which Digital Solutions Development (DSD) takes place, especially the place of DSD among the general business change and transformation activities of a modern enterprise.

The syllabus focusses on the development of applications software that is acquired by the enterprise for its own use or for the use of its Customers, or other stakeholders. It does not therefore cover software that is developed for sale commercially.

1.1. Define the terms *digital service* and *digital solution* and recognise some common characteristics of modern digital solutions.

1.1.1. A *digital service* refers to the electronic delivery of information, including text and multi-media content, across multiple platforms and devices, such as web, desktop or mobile. The service is provided by an enterprise, or contracted for use by the enterprise, as part of the solution to one or more business problems.

A *digital solution* refers to the technical means of delivering a digital service, using information technology, which relies on computer and telecommunications technologies in a modern context.

1.1.2. Candidates are expected to understand that software is simply the idea of having the means to generate instructions to drive computer and peripheral hardware. Therefore, candidates will appreciate that applications software is one of many possible forms of software, including:

- Application software: focussed on solving a specific business problem, via its support for a digital service. Often simply called an ‘app’. Examples include support for mundane, routine tasks like Payroll as well as more innovative tasks involving an Artificial Intelligence (AI) initiative, or the exploitation of Big Data.
- Systems software: software to support the execution of applications, like operating systems, database management systems and communications.
- Embedded software: Internet of Things (IoT), software embedded in machines, robotics.
- Gaming: software allowing users to play 2D and 3D games, single and/or multi-user.

Copyright © BCS 2020

- Virtual reality: software that creates the illusion of a 3D real-world experience.

Candidates should appreciate however that these different forms of software are converging, and already overlap in today's business world.

1.1.3. A major component in a digital solution is *applications software*, and it is the development of such software which is the focus of this syllabus. For the purpose of the syllabus one can think of applications software as the software that is written specifically to deliver one or more digital services. The particular focus of the syllabus is on *enterprise applications software* which is generally data intensive and highly interactive. Such applications are often multi-user and increasingly public facing. These common characteristics form the basis on which the topics in the syllabus have been chosen.

Candidates will appreciate that the word 'enterprise' is bundled with 'applications software' to convey the idea that DSD generally takes place under the sponsorship and governance of a social organisation of some kind, typically a legal entity or a Line of Business within a legal entity. The enterprise aspect of DSD influences the choice of topics in the syllabus too.

1.2. Explain the factors that influence investment in digital solution development

1.2.1. Candidates will be able to define *strategy* as an enterprise's 'plan for success' over the longer term, in view of the mission and essential purpose of the enterprise. Candidates will be able to explain that strategy is influenced by external and internal *drivers*. A 'driver' is some issue of significance that the enterprise needs to deal with:

- Examples of external drivers are legislation and technology innovations.
- Examples of internal drivers are process inefficiencies and data quality issues.

1.2.2. Candidates will appreciate that the desire to implement strategy, and thus tackle these drivers, leads the enterprise to research and commission programmes and projects of change. The generic activities involved in business change are described by the Business Change Lifecycle [1]. Note: the details of the Business Change Lifecycle are out of scope for this syllabus.

1.2.3. Candidates will understand that *Digital transformation* is typically one of the workflows or work streams contemplated within a programme of change. It is therefore quite common for new or enhanced applications software to be seen as part of the solution to business problems.

Candidates will be able to explain that contemporary thinking concerning business change, as codified for example in the Business Analysis body of knowledge [1][7], emphasises the complex nature of bringing about change in enterprises and expects change analysts to take a holistic view of all the elements and options available for solving business problems. Some problems, for example, may be solvable without investing in DSD, at least temporarily, and some problems are simply not solvable by using software at all. Even if software development is

required, it will usually need to fit into a wider set of changes, including, for example, changes to business processes and the organisation of people.

1.3. Describe the scope of digital solution development

1.3.1. Candidates will be able to recognise and describe the phases of a generic Digital Service Lifecycle, as follows:

- Plan: this phase includes the recognition that a digital service is required, and the decision to make the service part of the enterprise's service portfolio.
- Develop: this phase covers the development of the service, getting all the required elements ready.
- Transition: this phase covers the release and deployment of the service, so as to make it available to users who wish to subscribe.
- Operate: includes signing up service subscribers and delivering the service to them.
- Optimise: includes measuring the service performance, dealing with feedback and investing in continuous improvement.
- Retire: covers withdrawing the service.

Candidates will appreciate that DSD, and hence this syllabus, is focussed mainly on topics in the knowledge area underpinning the Develop and Transition phases of this lifecycle. The syllabus however does include links to other parts of the service lifecycle, where appropriate.

1.3.2. Candidates will understand that the success of investing in DSD is more assured, with the outcome likely to be more suitable, if DSD is conducted with reference to a recognised System Development Lifecycle (SDLC). Candidates will be able to explain that an SDLC is a framework, not a prescriptive method, and be able to justify the use of a framework.

The SDLC framework chosen by the enterprise defines the scope of DSD and brings together in one place knowledge that supports DSD. It provides:

- Definition of relevant processes and activities.
- Definition of roles and responsibilities. Identification of relevant disciplines within DSD.
- A set of suggested deliverables and artefacts.
- Tried and tested tools, techniques and good/best practices that have proved their worth under real conditions, over time.

Current examples of SDLC frameworks include SAFe [2] and DSDM [3]. Both of these rely on a combination of waterfall and Agile practices.

1.3.3 Identify the main parameters governing the approach to Digital Solution Development, using an SDLC, based on the following models:

1.3.3.1 Contingency Approach

Candidates should appreciate that, when setting out to develop a software solution in an enterprise context, some form of plan is required as a way of enhancing the chances of success. An SDLC helps with the planning process, since it describes stages of development, associated with activities and defined target states. Since it is a framework, not a prescription, adoption of an SDLC implies a contingency approach to DSD, which takes into account the characteristics of the particular problem to be solved.

Candidates will be able to recognise and describe the following models that illustrate the variables that influence the conduct of a development initiative.

1.3.3.2. The Cone of Uncertainty

This is a model that illustrates the natural variation in the degree of uncertainty over time, given the need to develop or maintain a product. The graphic in Fig. 1 illustrates the concept.

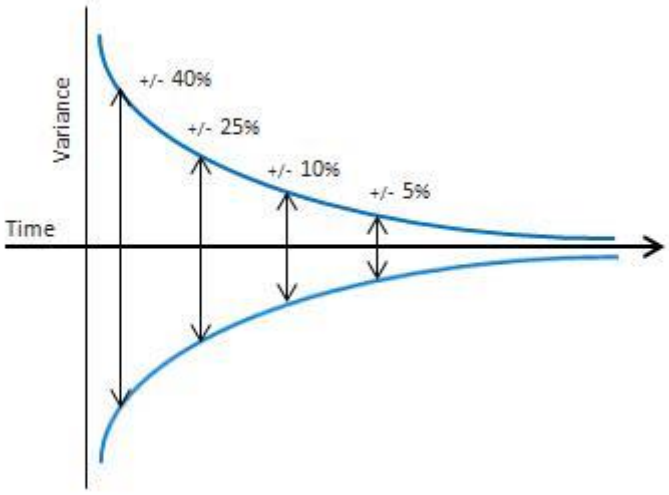


Figure 1: Illustration of the Cone of Uncertainty

Uncertainty in this context concerns the accuracy of parameter estimates, especially of time to execute and cost, relating to a project to produce products to solve a problem. The SDLC recognises the existence of this inherent uncertainty by defining stages or phases that progressively seek to reduce the degree of uncertainty. Uncertainty will be greater in novel situations than in well understood and familiar ones.

1.3.3.3. Time-Cost-Features Compromise

Many years of experience with running projects has led the software development community to understand the effects of the so-called ‘iron triangle’ that rules all project work.

Time, Cost and the set of Features to be delivered are the three basic parameters of any software development project. A project plan is a suitable configuration of these 3 parameters and is always a compromise between these conflicting forces.

Copyright © BCS 2020

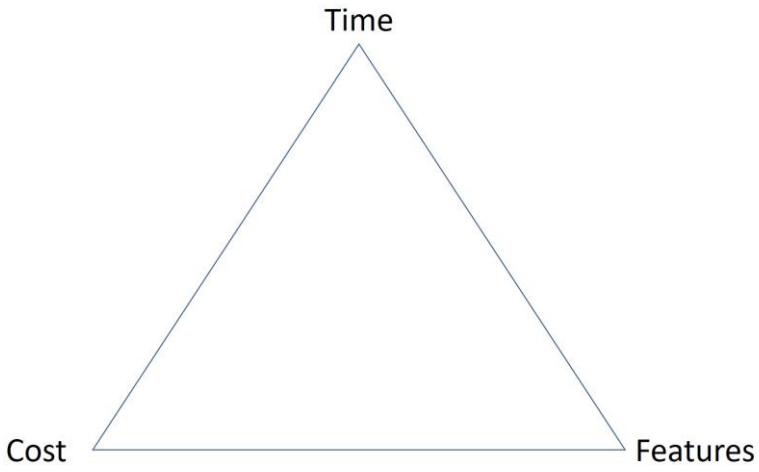


Figure 2: Time-Cost-Features Triangle

Given the inherent uncertainties in development work mentioned above, the planners of a project cannot expect to make accurate estimates of all 3 parameters in this triangle, and therefore have to understand which of the 3 is the most important, the 'stake in the ground'. The other 2 parameters must be given flexible tolerances, otherwise the project is not manageable, and is doomed to fail.

It used to be taken for granted that the default 'stake in the ground' was the set of Features to be delivered (Waterfall approach) but this has changed in recent years and the 'stake' is now regularly seen as Time (e.g. Timeboxes in Agile approaches) unless there is a strong reason to think otherwise. Focussing on frequent deliveries of some features within short fixed timescales has, on the whole, proven to deliver better results, from the point of view of solving business problems.

1.3.3.4. Continuum from Predictive to Adaptive

Bearing in mind the foregoing, the appropriate approach for a DSD development will vary along a continuum from *predictive* to *adaptive*:

- Towards the predictive end of this continuum, planners will want to spend increasing amounts of resources on predicting the set of features required to solve the problem and will engage in more thorough design activities.
- Towards the adaptive end of this continuum, planners will prefer to create solutions quickly, try them out, and adapt the software product based on feedback and experience.

The project plan has to find a suitable position on this continuum, given the nature of the problem to be solved.

1.3.3.5 Continuum from Low Ceremony to High Ceremony

The nature and scope of the software will determine the degree of ceremony appropriate for its development. Ceremony measures the degree of formality of the proceedings that govern the development processes, in particular the

importance of sign-offs, formal reviews etc. It's important to get the degree of ceremony right.

Some software developments are very sensitive to legislation, regulation and health and safety issues for example, and require more ceremony. They may cross internal enterprise boundaries and authority structures. Other software initiatives can be much more informal, for example in 'proof of concept', departmental or local scope projects.

1.4. Recognise examples of constraints that commonly affect DSD from the following list

The candidate will recognise that enterprise application software development may take place under a number of constraints, amongst which are those listed in the syllabus. This is not meant to be a definitive list, but common examples. Such constraints may well impact and influence the considerations mentioned in section 1.3.

1.4.1. *Legal:* both the development process and the end product must comply with applicable legislation and regulation.
1.4.2. *Financial*: development work always takes place within budgetary constraints.
1.4.3. *Technological*: the enterprise will have technical standards that must be respected. There will be limitations on what technology can achieve.
1.4.4. *Ethical*: the development process and the nature of the final product must be consistent with the ethical standards of the society that will make use of it.
1.4.5. *Timescales*: the development may have strict deadlines to meet.
1.4.6. *Organisation policies and standards*: the development process and the final product must be consistent with any applicable organisation policies and standards.

1.5. Recognise and identify the purpose of the following enterprise frameworks, and their relationship with DSD

The candidate will recognise and identify the purpose of the following management frameworks that commonly exist in a modern enterprise and be able to explain the relationship between DSD and these frameworks.

1.5.1. *Enterprise Architecture*: the goal of this framework is to promote business/IT alignment along a strategic roadmap. DSD work may be seen as the tactical investments that are made that realise these strategic plans. It is vital that there is an active connection between personnel working in these areas. A popular framework in use here at the present time is TOGAF [4]
1.5.2. *Programme and Project Management*: these frameworks govern the way that changes are carried out in the enterprise, at the strategic and tactical levels respectively. The DSD SDLC framework chosen must be adapted to fit within these practices. Framework examples include MSP [5] and PRINCE [6].
1.5.3. *Business Analysis and Change Management*: These disciplines ensure that the impacts of business change proposals are thoroughly and holistically investigated. DSD is engaged by roles within these disciplines to provide IT solutions. World

renowned frameworks in this area are the BCS Business Analysis Diploma[1] and the IIBA BABOK [7].

1.5.4 *Service Operations*: This framework covers best practices in supporting the end user with IT services, delivered via an IT infrastructure. DSD is about creating and maintaining an IT software service for the end user, who will be supported by roles and practices defined in this discipline. Examples of such frameworks are ITIL [8] and ISO9000 [9].

1.5.5 *Data Management*: This discipline underpins a function within the business that assists the enterprise to manage its data assets effectively and provides the necessary governance. Applications often use enterprise data, or derivatives of it, and therefore DSD must coordinate their developments with this enterprise authority. A leading body of knowledge covering the area is DAMA's DMBOK [10].

2. Digital service definition (15%)

This section of the syllabus deals with the need to define a set of requirements which specify the abstract characteristics of the software service and related products that are to be built. These characteristics respond to the nature of the business problem to be solved, the role of the software and the environment in which the software is to be deployed.

2.1. Recognise the need for requirements engineering, its key elements and benefits

Candidates are expected to recognise that requirements engineering (RE) for software products and services is no different to similar processes for the development of any product, be it a motor vehicle, a mobile phone, an electric kettle etc. The process involves engaging with all stakeholders to specify the abstract characteristics of the product that will solve a problem for the product's users, within the boundaries of whatever constraints need to be recognised (for example those enumerated in 1.4).

This activity is generally coordinated by the Business Analyst role. Many years of experience have revealed a suitable set of RE processes and a range of suitable techniques and best practices, and these are incorporated into the Business Analysis body of knowledge [1]. This syllabus picks out some of these practices here and considers that RE is an area where Business Analysis and roles in DSD such as Solution Architect should overlap to some extent.

2.1.1. *Explain the rationale for requirements engineering*

Candidates are expected to appreciate that a certain amount of RE related activity is a prerequisite to the development of successful software. *Engineering* is needed because requirements in most cases don't come to the development team in the form needed to create the software product. The user community in particular is often very solution focussed and may not understand what applications software might be able to do to solve their problem.

However, taking into the account the considerations mentioned in 1.3.3, there is a general expectation these days that not all requirements can be, or even ought to be, firmly nailed down before committing to the software production process. The balance between doing too much and too little RE work is a difficult one to judge objectively and requires the application of experience and skill.

Candidates will be able to identify and describe some of the key benefits of RE activity, listed as follows.

2.1.1.1. Alignment of the digital service to business needs and objectives
By adhering to good practices with regard to the management of requirements, in particular with regard to traceability, the analyst can ensure that the solution that is developed and delivered aligns to the business needs and objectives set out in the business case and terms of reference created at the outset of a DSD project. Every requirement has implications for cost, time and risk, and needs to be justified and linked to the business needs and objectives.

2.1.1.2. Completeness of the digital service definition
Focussing on the abstract characteristics of the digital service in its environment helps to ensure that the solution provided is as complete as it needs to be. In the case of applications software functionality completeness can be judged, for example, by modelling business processes and the manipulation of data (see section 2.3).

2.1.1.3. Problem clarification
RE is also ‘problem engineering’ to some extent. Success depends upon being clear about the problem to be solved and making sure everyone involved recognises and agrees this. Through the use of a range of investigation/elicitation techniques the analyst can explore the business problem and help to define and clarify it. This is indispensable for scoping the product, and hence the development effort.

2.1.1.4. Balancing stakeholder interests
RE should be seen as a social process wherein the analyst engages with multiple stakeholders who have an interest in the digital service or the digital solution. The form and function of any product is the result of a compromise between distinct stakeholder interests, which are sometimes in conflict. The RE activity ensures that all stakeholders are identified as far as possible and asked to contribute requirements. This not only sustains the delivery of a successful product, but also fosters a sense of buy in and commitment on the part of the stakeholders.

2.1.1.5. Stability of requirements vs solution
Requirements do not change as often as solutions do. Solutions are volatile, especially in a modern context, where new technologies are constantly emerging. Keeping the definition of requirements and solutions separate from each other proves useful when it comes to considering changes to the software, which are inevitable, since the analyst can assess whether the changes affect the requirements or just the solution.

Preserving the requirements documents records many of the ideas that lead to the solution. These ideas may not change very much over time, even though the solutions do. This knowledge is useful in an enterprise context and means changes may be quicker, easier, cheaper and less risky to carry out. In an enterprise, having solutions in place without access to this knowledge presents serious problems. Many legacy applications are problematic in this respect, and it is incumbent on today's developers not to propagate these problems into the future.

2.1.1.6. Governance of solution acquisition and delivery

Having defined requirements enables the team to evaluate and choose amongst competing solution options.

2.1.2. *Identify the elements of requirements engineering*

Candidates will be able to identify and describe the 5 key elements of a generic RE framework, shown diagrammatically in Figure 1.

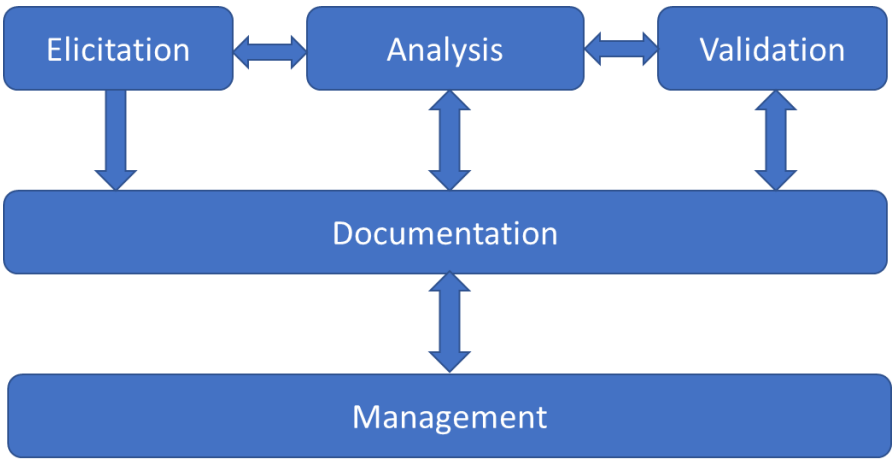


Figure 1: Requirements Engineering framework [1]

Elicitation, analysis and validation essentially form a three-stage iterative process, which is supported by a set of documentation standards and techniques for managing the requirements.

2.1.2.1. Elicitation

Elicitation involves the analyst engaging with stakeholders to focus on what the digital service needs to do and what outcomes it needs to deliver, in order to solve the business problem.

During elicitation the analyst aims to understand the needs of the stakeholders by employing a range of investigation techniques so as to identify the features or functions that need to be provided by the service (functional requirements) and the service levels and constraints that are placed on the delivery of the service (non-functional requirements).

2.1.2.2. Analysis

Requirements that have been elicited will usually benefit for a certain amount of investigation, by the analyst, to get them into a shape that is suitable for the software development team.

Analysis may involve work relating to:

- Ensuring the requirements are expressed appropriately, using the formats that are standard for the team.
- Ensuring that the inclusion of each requirement is justified.
- Ensuring that each requirement is feasible, both technically and financially.
- Ensuring a requirement doesn't duplicate or overlap with another requirement
- Identifying conflicting requirements.
- Understanding the importance and priority for delivery of each requirement.
- Ensuring any given requirement does not contravene any of the recognised constraints.

Models are often drawn during analysis (see section 2.3 below) to confirm and revise scope, check for completeness or re-use and clarify business rules. Where conflicts and inconsistencies between two or more requirements are uncovered, it is often necessary to broker a negotiation between stakeholders in order to resolve the areas of conflict, and reach a suitable compromise.

2.1.2.3 Validation

Validation is the process whereby the requirements that define the service to be built or altered are presented to the Sponsor, Product Owner, and other stakeholders, for their review and sign-off. This is an important aspect of the governance of change, given that the business owns the software service, not the analyst. The business pays for the software and has to live with the consequences of using it.

The formality and ceremony involved in validation varies in accordance with the nature of the software being built. The factors influencing this have already been elaborated in 1.3.3.

2.1.2.4 Documentation

Requirements documentation continues to be a controversial area of software development, with little consensus on what is too little or too much.

Documentation is how knowledge about the software service is captured for future use, so management in charge of digital service development will need to make an appreciation, possibly on a case-by-case basis, of what level of documentation is appropriate, and derive suitable standards, policies and procedures.

Some of the recognised ways of documenting requirements are elaborated in section 2.3.

2.1.2.5 Management
The purpose of requirements management is to plan the RE activity and put in place a series of governance and control mechanisms around the practice of requirements engineering. This ensures for example that:

- there is an accurate record of requirements in accordance with prevailing standards
- changes to requirements are adequately managed
- the delivered solution meets the needs of the business commissioning the development work.

The degree of governance applied will be dependent upon the commissioning organisation, which may be bound by regulation and legislation to ensure that the solution will adhere to a set of rules and practices dictated by the associated regulatory body.

Requirements may be managed through a set of documentation attributes which provide traceability from the source of the requirements to the delivered solution (horizontal traceability), and from solution requirements up to their associated business requirements and objectives (vertical traceability).

In most situations, a change control process should be instituted to ensure that any proposed change in requirements is approved by a suitably authorised representative of the sponsoring organisation, taking into account the potential impacts of the change on the project (timeframe and/or budget), the product and on the business as a whole.

Management also includes the planning of the RE activities within the change initiative and allocating personnel who will carry out these activities. It also includes the provision of suitable tooling and administrative support.

2.2. Identify common sources of requirements and classify different types of requirements

Candidates will appreciate that requirements come in various shapes and forms from different stakeholders with differing interests and perspectives. Consequently, there are different types of requirement that will be elicited, analysed, validated, documented and managed during RE.

2.2.1. *Identify common sources of digital service requirements:*

2.2.1.1. Problem statement from the sponsor
These requirements derive from the business problem that the sponsor of the investment wishes to solve. As such, the business sponsor is a key stakeholder within RE, and their requirements will carry a high priority. Often taken as a ‘starting point’ for RE, or ‘anchor point’, these requirements determine the high-level business and project objectives that the solution development project needs to address, often defined within a business case and/or project initiation document (terms of reference).

These high-level objectives can be de-composed into more granular business needs, that can then be further de-composed into solution level needs.

For example, an objective to implement a new digital strategy for communicating with customers can give rise to a high-level business requirement to enable a self-service facility for customers to manage their own data online. This can be de-composed into solution level functional and non-functional requirements to maintain customer account details and place orders online.

2.2.1.2. Decomposition and analysis of business processes

When analysing business processes to identify potential improvements, one improvement area that is often suggested is automation. This involves acquiring an IT-based solution to support or completely replace tasks within the business process that are currently performed manually. It's also possible that existing software is not fit-for-purpose in some way, and therefore needs altering or replacing. Process improvement is therefore an important source of digital service requirements. The role of modelling business processes is highlighted in section 2.3.

2.2.1.3. Business and technical policies and standards

Organisations define policies that need to be adhered to by members of staff when performing tasks within a business process. Adherence to such policies may therefore be a source of requirements when defining a solution to meet a business need.

Some policies are interpreted in the form of business rules that apply to individual tasks whereas other policies are more strategic and place generic constraints across a range of change initiatives. An example of a business policy of the latter kind is the need to promote the brand wherever possible, both internally and externally.

Technical policies may also be a source of requirements. Such policies may emanate from Enterprise Architecture for example (see 1.5). Policies may be the result of enterprise architecture principles, such as interoperability, and may include the use of particular hardware and/or software platforms. Style guides governing the user interface may also be mandated (see 3.1).

2.2.1.4. Legal and regulatory compliance

Organisations must usually adhere to any legislation imposed by governments within the country/countries of jurisdiction within which the organisation operates, for example, data protection regulations. Furthermore, organisations must also comply with regulations imposed by regulatory bodies that govern the sector(s) within which the organisation is active. Consequently, adherence to the legislative and regulatory regime may be a source of requirements which must be represented in the digital service products.

2.2.1.5. Stakeholder specific needs

In addition to the sponsor, during RE considerations should be given to the needs of all stakeholder groups. Each group is a potential source of requirements.

An example of such a stakeholder group is the target users of the solution, which may be divided into sub-groups with differing needs and access rights. Some users may have disabilities or impairments that affect their ability to use the solution, and hence, specific requirements concerning accessibility features to be built into the solution must be defined. Similarly, depending on the experience and computer literacy of the target users, it may be necessary to define specific requirements concerning usability, navigation etc.

2.2.2. Classify requirements according to the following categories

Candidates will be able to recognise that requirements that define a digital service, irrespective of their origin, will be classifiable as functional and/or non-functional requirements. Candidates will be able to classify examples of requirements into these categories, and the sub-categories mentioned in the syllabus.

It should always be borne in mind that a software service sits within the wider concept of a business service, and must be linkable to it, but the definition of a business service is out of scope for the purpose of this syllabus.

Functional requirements (FR) are specifications of what the software service will be able to do. Collectively the FRs are 'the service'. Since it is a software product the 'doing' is confined to the manipulation of data in terms of some combination of create, read, update, delete (CRUD). This manipulation of data has value for the end user in accessing the enterprise's products and services or in relation to their duties within the enterprise.

Non-functional requirements (NFR) are specifications of how the functionality is to be delivered as a service within defined service levels. Various categories are given as examples below. Some NFRs are developer concerns, others are concerns of service operations, and some are concerns of both communities.

Note that some FRs may have specific NFRs associated with them. Also, some NFRs will imply FRs. The effect is to build up a hierarchy of requirements definitions which link ultimately into the software product's environment, defined by the business service.

2.2.3. *Recognise the following examples of non-functional requirement types:*

2.2.3.1. *Usability*: a measure of how easy/intuitive the service should be to use
2.2.3.2. *Security*: these provisions seek to protect the information/data assets employed in the service from vulnerabilities and threat actors. See section 8.
2.2.3.3. *Performance*: a measure of the speed of processing of an automated task.
2.2.3.4. *Accessibility*: provisions to allow the service to be accessed by persons with a range of disabilities.
2.2.3.5. *Availability*: provisions that determine when the service is to be made available to be used.
2.2.3.6. *Reliability*: a measure of the reliability of the service in terms of functioning correctly as planned.

2.2.3.7. Recoverability: standards that define how quickly service is guaranteed to be restored if it fails.

2.2.3.8. Scalability: a measure of the ability of the service to deal with variations in workload. See also section 4.4.

2.2.4. *Understand the link between service requirements, key performance indicators and service level agreements.*

The need to guarantee the delivery of the service to agreed service levels will mean the creation of service level agreements. These may be formal legal contracts in the case of third-party involvement.

Delivery performance must be monitored and reported upon, so action can be taken if the service levels required are not being achieved. The mechanism for this is usually some form of dashboard, which reports against a number of KPIs, with their target values, and which will have been based around the original FRs and NFRs for the service.

2.3. Describe the use of the following requirements documentation and modelling techniques

Candidates will be able to recognise and explain the use of two common RE documentation techniques, which are User Stories and Use Cases. User Stories are favoured in many Agile production methods, whereas Use Cases are ways of organising and modelling requirements for those developers who find UML [13] useful.

2.3.1. *User stories*

User stories provide a lightweight approach to capturing solution requirements, and are particularly popular within agile solution development projects, where they provide a simple, standardised approach to defining solution/product backlog items.

The original intention behind the use of user stories, which are focussed on functionality, is that stakeholders involved in solution development projects could write their own requirements on cards during a story writing workshop at the beginning of the project to form an initial backlog. User stories originally were thought of as ‘placeholders’ to indicate the need for a conversation between Product Owner and Developer.

The user story structure is simple to understand and quick and easy to use by non-technical stakeholders. The resultant stories encapsulate the basic need without expending unnecessary effort refining and fully defining the underlying requirement, thus reducing wasted time.

Only when a user story has been prioritised for development during a particular iteration of the solution development project, does the detail underlying the requirement need to be explored, and the user story can then evolve into a more complete, fit-for-purpose requirement that can form the basis of development of working software.

2.3.1.1. Structure

The basic structure of a user story is a single statement that comprises three elements, as follows:

As a … this is where the user role that requires the functionality is identified
I want … this is where the required functionality is described
So that … this is where the value that the user derives from the use of the functionality is described

For example:
As a potential delegate **I want** to book a place on an upcoming course **so that** I can gain a qualification.

2.3.1.2. Quality criteria (INVEST)

When the story is first written, using the format described above, it may lack the necessary detail to enable a development team to design and build a piece of working software that realises the requirement. Consequently, stories typically need to undergo various forms of refinement and elaboration in order to be deemed fit-for-purpose and ready for development.

The mnemonic INVEST is widely accepted within the agile development community to be the most useful set of quality criteria for determining whether/when a user story is ready for development (sometimes referred to as the 'definition of ready').

INVEST was originally proposed in an article by Bill Wake in 2003 [11] but was subsequently endorsed by Mike Cohn in his 2004 book User Stories Applied [12]. According to INVEST a good user story should be:

Independent	Stories should be written in such a way that they do not have any dependencies upon other stories. Every user story stands up on its own.
Negotiable	Stories should not be considered as contracts that the software *must* implement. They should be short descriptions of functionality that can be negotiated and prioritised for development in a conversation between the Product Owner and the development team.
Valuable	The functionality described in the story must be valuable to the sponsor, users of the digital service, or some other stakeholder. So, it should be noted that all functionality will necessarily be valued by the end user but it must be valuable to somebody.
Estimable	It should be possible to provide an estimate of the 'size' of a story and therefore the amount of time/effort it will take to turn it into working code. Two common reasons for stories to not be estimable are: 1. The story is too 'big' and requires decomposition into more granular stories.

2. Not enough is known about the story by the developers, and hence, the story requires further elaboration/refinement.

Small — The size of a story should be 'just right'. The general agile principle is that stories must be 'doable' within a single sprint, which is usually considered to be a timespan from 2 weeks to 2 months. Stories that are too big are referred to as Epics (see section 2.3.1.3). Stories that are too small, on the other hand, perhaps won't deliver enough business value to be worth the coding effort.

Testable — The story must contain specifications on how to test it, in order for the developers to know when it has been 'done' (coded). A common approach to ensuring that stories are testable in this way is to define a set of acceptance criteria, which are agreed between Product Owner and developers.

2.3.1.3. *Epic as a story requiring further de-composition*

Epic is the name given to a story that is deemed to be too big to be developed into working software within a single iteration of development. Epics may contain high level and multiple stories and require further de-composition before development work can commence. Often epics capture the initial thoughts of users, which then need further exploration.

2.3.1.4. *Theme as a set of related user stories*

To aid the planning of a solution development project and the management of a solution/product backlog, user stories may be bundled together into groupings of related stories called Themes. Such themes are typically determined by a relevant business concept, for example a grouping of all stories relating to 'bookings' or 'payments'. Especially applicable to larger scale development initiatives.

2.3.2. *Use cases*

A use case is a modelling element defined in UML which seeks to organise all the requirements into discrete business goals that the users ('Actors') wish to achieve using the proposed software. In development frameworks that rely on UML, like RUP [14], the set of use cases drives the whole development process.

Use case models comprise one or more use case diagrams, each identifying a set of use cases, and a set of supporting use case descriptions, one for each use case identified on the diagram(s).

2.3.2.1. Key components of a use case diagram: system boundary, actors, associations, use cases

The notation for a use case diagram is defined within the UML. All use case diagrams comprise at least four key elements or components, although the UML does define additional syntax that can also be used.

- *System boundary* - The system boundary effectively defines a scope for the software system described in the diagram. It represents the system under development. This could also be a sub-system of a wider development if necessary.

- *Actors* - Actors represent anyone or anything that interacts directly with the system under consideration. In the majority of cases actors will represent user roles (as shown in figure 2), although other systems that interface with the system under consideration can also be represented as actors. Some modellers also represent time as an actor to enable time-based 'batch' processing to be shown.

- *Use cases* – each use case identifies a business goal that a user (Actor) wishes to achieve using the proposed software system.

- *Associations* - Each association shows which Actors are associated with each Use Case. The implication of an association is that the Actor is the business role that will be using the system to achieve the goal captured by the Use Case.

An example of a use case diagram is shown in Figure 2.

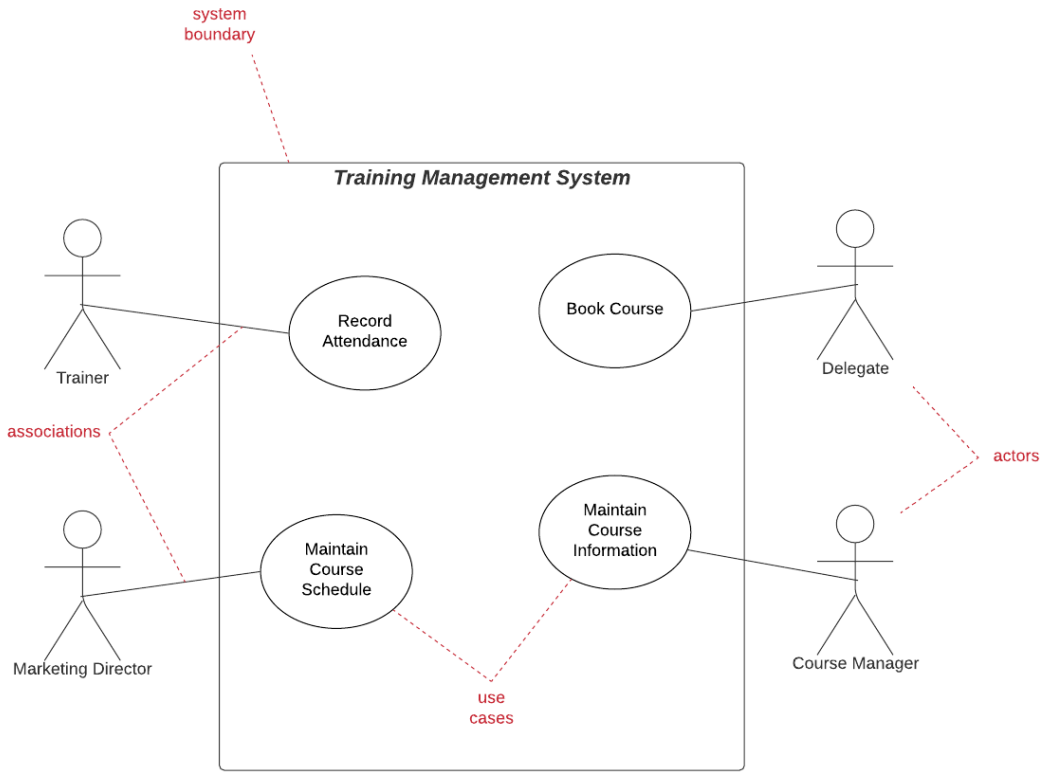


Figure 2: Use case diagram

2.3.2.2. Use case diagrams to agree the functional scope of a digital service

A Use Case Diagram shows the scope of the software application to be built. There is a famous saying that 'the system is the sum of its Use Cases", which indicates that all the software specification is in the set of Use Cases. To put this another way, there should be no requirements specifications anywhere that are *not* included in at least one Use Case.

2.3.2.3. Use case description as an approach to elaborating required system behaviour

The Use Case Description is a text document that describes how the user will interact with the proposed software in order to reach the business goal identified by the Use Case. UML does not specify how this should be laid out, so most practitioners rely on literature such as [15]. Note therefore that there many 'requirements', both FRs and NFRs, might be included in a single Use Case, along the way to achieving the user's goal.

Although there are many fields mentioned by Cockburn that could be included in a Use Case Description, the essential part is a description of the dialogue envisaged between the user and the IT system in order to reach the stated goal. This should be laid out as a sequence of User/System interactions, until the business goal is reached. The following table illustrates the general idea:

Table 1: Format for Use Case descriptions

UC step	*User*	*System*
1.	User does A (B, C)	System does X (Y, Z)
2.	User does …	System does …
3.	…	…

As a guide it is expected that the user should be able to reach their goal within 7+/-2 Use Case steps.

There should be a Main Scenario, which describes what is considered to be the 'normal path' to the goal. AKA 'Sunny Day', 'Happy Path', etc. There should also be a number of Alternate Scenarios, which describe other possible paths. Some alternates may reach the goal by other means, others will mean the user fails to reach the goal.

Other Use Case attributes that are popular include:

- Priority: this field might make use of the MoSCoW prioritisation method mentioned elsewhere in this syllabus (2.4.5).
- Pre-conditions: any significant specifications about the state of the system that the Use Case should assume to be *true* before it commences. If these conditions don't hold the Use Case could fail. A typical example of this is an assumption that the user is authorised to use the function.

- Post-conditions: a brief statement concerning the state of system once the Use Case ends. The state described should have significance for a business audience. So, for example: “Order taken, address confirmed, payment made”

2.3.2.4. Use case description as a mechanism for elaborating test scenarios
The sort of structures suggested above for documenting a Use Case Description, lend themselves to the creation of test scripts – there are in fact tools that can more or less automate this. This is a form of Black Box testing associated with User Acceptance (see section 6).

The usual approach is that each scenario in a Use Case becomes a Test Case, and each step in the Use Case becomes at least one Test Condition. The tester only has to add some suitable test data specifications for each Test Condition, where applicable.

2.3.3. *Data and Process modelling*

The candidate should recognise how data models and process models can contribute to the understanding of software requirements.

2.3.3.1 The role of the data modelling and data models in driving out requirements
The mechanics of data modelling are elaborated in another part of this syllabus (see section 5). The reference to data modelling here is concerned with how data modelling and data models can contribute to the elaboration of application software requirements.

A data model essentially consists of entity types, associations and attributes.

User Stories and Use Case Descriptions will make reference to many business concepts, which should be reflected in the entity types and attributes of the corresponding data model. For example, if a User Story or Use Case mentions a major business concept like ‘Customer’, the analyst would expect to see that concept reflected in the corresponding data model. If the User Story or Use Case specified the capture or amendment of certain attributes, they should be reflected in the model too.

Associations between entity types can help shape requirements too. For example, the existence of an association indicates it is possible to navigate from one entity to another. A User Story or Use Case may require this – for example from knowing the identity of a Customer, the users are expecting to be able to request a history of their Sales Orders.

The multiplicity of the association is a reflection of the business rules that tie objects together in the business domain over time. So for example the multiplicity might indicate that a Customer may have many Sales Orders, but each Sales Order is an order from only one Customer. The specification of requirements for software should respect these business rules or at least identify them and put them up for discussion.

2.3.3.2. The role of the data lifecycle in driving out requirements

All business data has a lifecycle, which traces out how objects get created, what states they can acquire, what modifications to their data can happen during their lifetime, and how they can eventually leave the system.

A UML State Machine or other forms of Entity Life Histories can be used to model this lifecycle. These models can be very helpful to requirements engineers, because the engineer can verify that there is software, actual or proposed, that fully supports the data lifecycle. This area is where the link with Data Management referred to in section 1.5 might need to be activated.

A favourite technique used to help identify gaps and omissions between functional and data requirements is called the CRUD matrix. The principle is illustrated in the table below:

Table 2: Format for CRUD matrix

	Entity Type A	Entity Type B	…
UC or US 1	C	U	
UC or US 2	R		
UC or US 3	D	R	
…	…	…	

A list of Use Cases or User Stories is compared to a list of Entity Types, derived from the data model. The data manipulation by each UC or US is then annotated in each cell where applicable in terms of Create, Read, Update and/or Delete.

Reading down each column of this matrix will reveal all the manipulation of an entity of that type, proposed by the target software. This may reveal gaps in the requirements. For example, an entity is supposed to be Read but is never Created! Perhaps an entity has no Delete entry – is there a missing requirement therefore? It is always possible some other software covers these gaps, but the analyst should check this out. Knowing what is 'unknown' is very valuable to the analyst.

Reading along the rows of this matrix helps the developers see and understand the data access requirements of each Use Case or User Story.

2.3.3.3. The role of process modelling in driving out requirements

Business process models often take the form of flow charts which show a sequence of tasks that finally produce an item of value to the business. Each task in a process requires some form of information. If the decision is taken to provide this information through a digital service, the task definition becomes the basis for digital service requirements.

Process documentation often includes a lot of business-related non-functional requirements too, which may have an impact on the digital service's FRs and/or NFRs.

Many flowcharting notations can show data manipulation directly, for example BPMN [16] or the UML Activity Diagram. The CRUD matrix shown above can also be an effective modelling technique, substituting a list of tasks for the use case or the user story.

2.4. Explain the purpose of some common requirements management techniques

Candidates will be able to identify the following requirements management techniques and explain their purpose.

2.4.1. *Explain the role of a product backlog in managing requirements*

2.4.1.1. User stories as product backlog items

User Stories, and other work items, are collected in a Product Backlog catalogue. In principle every software product in an enterprise has a permanent backlog of requirements that are as yet unfulfilled. The Backlog is therefore a centralised repository. It is the job of the Product Owner to manage this catalogue on behalf of the business.

2.4.1.2. Iteration (sprint) backlog and release backlog as subsets of the product backlog

Stories are associated with a sprint by picking items off the product backlog, in accordance with business priorities. It is highly desirable that each sprint leads to a live environment release, but this may not always be possible, so it may be necessary to plan a release in terms of a collection of sprints.

2.4.2. *Recognise the need to track requirements from origin to delivery in a working solution*

It should be recognised that in an enterprise context a software application may serve the business needs for many years. It is essential therefore that records are kept of how the features that eventually make it into the product got there, and where they came from.

For example, the enterprise may wish to record, for each requirement:

- The origin of the requirement – where/how was it discovered
- The stakeholders who are associated with it
- A history of its treatment, including when/how it made it into the product.

Having these records helps to manage change and the future evolution of the product. Their absence can mean wasted resources trying to discover things that should already be known.

2.4.3. *Explain the need to refine the iteration backlog*

2.4.3.1. Story slicing or splitting

Several of the qualities that make up INVEST reflect the need to make stories small, so they fit inside a sprint, and yet make sure they do have business value. It may be the case that a story taken from the product backlog requires splitting (also known as slicing) in order to meet this criterion. It is important to exercise so-called

‘vertical slicing’ to ensure the business focus is retained. A vertical slice is still a complete standalone story, although it may be one scenario among others.

2.4.3.2. Defining acceptance criteria

Another way of defining and refining stories derives from the requirement to document acceptance criteria for each story. When the software passes these tests the story is considered ‘done’. This process often leads to clarification and modification of the story in important ways, so this approach has come to be known as Test-Driven Development (TDD).

2.4.4. *Explain the use of story points and velocity to plan and manage the work of a development team*

User Stories will vary in the degree of effort required to turn them into code. Hence, to assist with the decision to commit backlog items to development, some estimates of effort will be helpful, along with some means of tracking progress towards the sprint goals.

2.4.4.1. Purpose of the iteration (sprint) planning meeting

The purpose of this event is to decide on the stories and other work that should be carried out in the sprint. Also, the probable ‘size’ of each story and their relative priorities.

2.4.4.2. Story points as a mechanism for the relative sizing of stories

Some form of story point measurement is habitually used to ‘size’ each story. Not all stories imply the same amount of work/effort.

A number of relative scales are in use, for example:

- T-shirt sizing
- Fibonacci Series

Note these are intended to be relative sizes, based on the team’s performance.

2.4.4.3. Velocity as a measure of the capacity of a team to deliver working software

‘Velocity’ is a measure of the rate of progress of a team towards the sprint goals. The principle is to set a steady ‘cadence’ that is sustainable indefinitely and remove the need to rely on ‘heroes’ burning the midnight oil to get the software out.

2.4.4.4. Burndown as a measure of progress

Burndown and Burnup charts are used to regularly measure progress and velocity by measuring actual progress against target. These charts are updated daily and are visible to the whole development team.

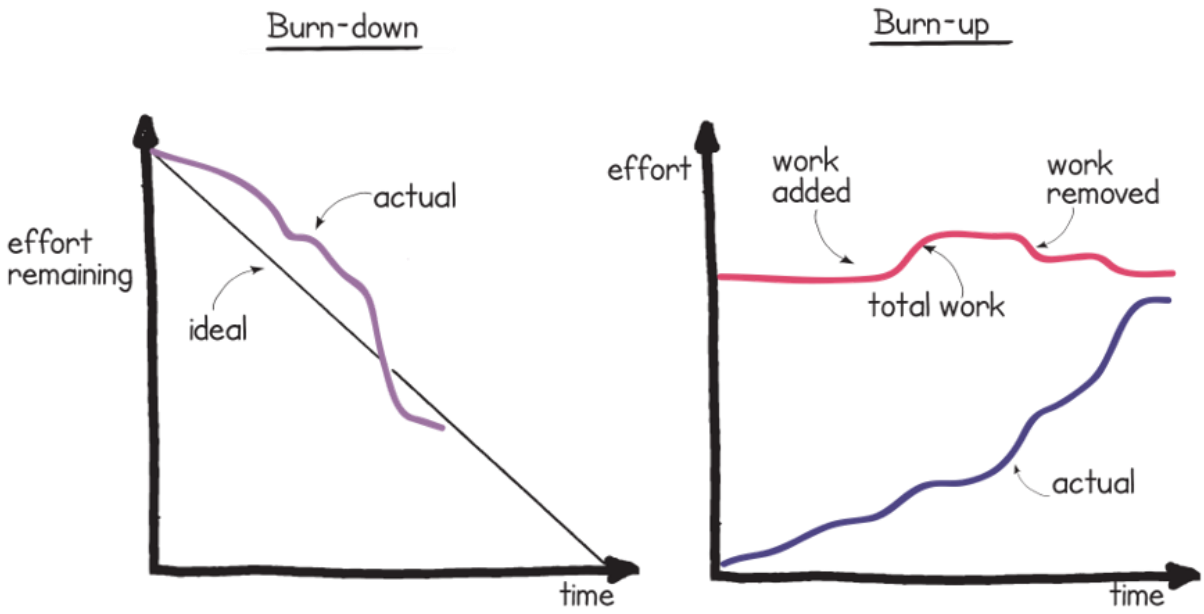


Figure 3: Burn down and burn up charts

2.4.5. *Describe the application of MoSCoW prioritisation to product, release and iteration backlogs*

As a general rule not all UC or US will have the same importance with respect to solving the business problem or meeting an environmental constraint. It may also be true that the business does not need every planned feature from Day 1. If this is added to the need to deal with the 'cone of uncertainty', (see 1.3.3), it often becomes essential to have some form of prioritisation of requirements.

It should be noted that any prioritisation system addresses a specific 'timebox', on the assumption that 'time' is the project variable that has zero tolerance. The time in question could be at the product, release or iteration level. Developers should be alert, however, to the possibility of 'single delivery' projects, which do arise from time to time, and for such cases 'feature' prioritisation, which is referred to here, may not be relevant.

MoSCoW (Must Have, Should Have, Could Have, Won't Have) is a widely recognised prioritisation method, part of the DSDM Agile development framework. The suitable split amongst potential requirements, recommended by DSDM, is M=60%, S=20%, C=20%, measured in effort.

2.4.5.1. Use of 'Must Have' to define the minimal viable product (MVP) or increment

Must Haves are defined as requirements that must be realised in the solution at the end of the timebox. Without these there is not enough value added into the product to merit the expenditure of resources. Especially in the beginning of the development of a new product, the *Must Haves* define the minimum set of features required to address the business problem in any meaningful way. This feature set is called the Minimal Viable Product (MVP).

Copyright © BCS 2020

2.4.5.2. Distinction between 'Must Have' and 'Should Have'
Should Haves are requirements which are important, and they are marked for delivery, but they could be postponed to a subsequent release if necessary. There would be some workaround for the business users to use, albeit as a short-term measure.

2.4.5.3. Use of 'Could Have' for contingency
Could Haves are the main pool of contingency for descoping the feature set that gets delivered. These are often 'nice to haves' which can be slotted in easily with other more important requirements, already being dealt with, and that have a higher priority.

2.4.5.4. Use of 'Won't Have This Time' to confirm out of scope
Won't Haves can take on a number of meanings, depending how developers prefer to use it. These are requirements that definitely won't be delivered as part of the sprint under review.

- It is often the case that making it clear what *won't* be delivered, helps clarify what *will* be delivered. This can help focus the discussion on what to include/exclude.
- Sometimes it is recognised that a feature would be valuable to include but for some reason it can't be included in the timebox being planned. Reasons include:
    - It's 'too vague' in its current form – it needs breaking down and refining, but the team is not prepared to do that in this timebox.
    - It's too big, and therefore warrants another project, more money or a separate team.
    - The team must wait for the results of some other activity, being conducted beyond their control.

It is important to appreciate that requirements that are simply unsuitable, irrelevant etc. are not placed here. Such requirements should be marked in the backlog as 'rejected' or similar, with a brief explanation.

3. User Experience (UX) and User Interface (UI) (12.5%)

Given that most enterprise applications are highly interactive, this section of the syllabus covers some of the basic ideas and terminology commonly used in the analysis and design of the User Interface, alongside a number of techniques that have proved to be effective in this area.

The importance of getting the UI 'right' can hardly be over emphasised. For most users the UI *is* the computer; they will have little notion of what goes behind the interface.

3.1. Recognise the definition of the following fundamental terms commonly used in UI/UX and explain their significance for UI design.

3.1.1. *UI/UX*

Candidates should recognise the terms UI/UX and be able to distinguish between them.

UI=User Interface, UX=User Experience. These are generic terms used to refer to the development area concerned with providing a suitable interface so that users can interact with applications software.

UI focusses on the structures and behaviours present in the interface, while UX focusses on the emotional engagement of users using the application and brings in some of the socio-cultural and psychological aspects of people using computers. UX may be extended to cover the whole of a user's experience ('touch points') as they engage with the enterprise, whether via IT or some other means.

The somewhat old-fashioned term HCI (Human-Computer Interface) is equivalent to UI and is still in use.

3.1.2. *Style guides*

Candidates should understand the need for, and purpose of, Style Guides.

A Style Guide provides information on standards to be used in designing the UI. Standards include such things as colours, fonts, branding and use of controls. Having standards contributes to consistency in the interface, promotes ease of use and usability and cuts down training times. It also cuts down on the specification work required to guide developers and web designers and supports testing the interface. Typically, a different Style Guide is required for each distinct platform used, although some parts of standards may be shared. Different guides may be needed for text-based, GUI, Web and others.

3.1.3. *UI paradigm*

Candidates should recognise and be able to explain the term 'UI paradigm'.

A *paradigm* is an 'accepted way of thinking'. Various paradigms concerning the form of the UI have been in use over the years. A selection of them are included in another part of this section of the syllabus.

3.1.4. *Metaphor*

Candidates should understand and be able to explain what a metaphor is in this context. Candidates should be able to recognise a metaphor.

The term *metaphor* is used to refer to an item in a UI, usually in the form of an icon, that reflects something in the real world. The benefit of using metaphors is that, since they resemble something in the real world, they are instantly recognisable by a user, without any need for training. Example: shopping basket/trolley.

3.1.5. *Idiom*

Candidates should understand and be able to explain what an idiom is in this context. Candidates should be able to recognise an idiom.

An ‘idiom’ is a control in the UI whose purpose, once learnt, is instantly recognisable by a user. Examples include universally well-known controls like Buttons. Standard wording and text like ‘Continue’, ‘Next’, ‘Back’ are included within this idea too.

3.1.6. *UI pattern*

Candidates should understand and be able to explain what a UI pattern is and why it is useful. Candidates should be able to recognise a UI pattern.

A ‘pattern’ is a configuration of controls in the UI which addresses a specific task, or group of tasks. A pattern is presented in the form of a UI template. Thus, for example, there are patterns covering the structure and style of a blogging website. Patterns are easily adapted by developers from the template for their particular problem, with the advantage that all the basic ‘thinking’ has already been worked on and proven to work.

3.2. Recognise and define the following UI/UX principles, centred around usability

Candidates should understand the meaning of each principle and be able to recognise its application.

A number of principles relating to UI/UX design have been formulated, based on experience and research over many years. Adherence to these principles, which are all centred around the basic *usability* of the interface, would contribute to a successful UI/UX. These principles can also be used as the basis for quality assessments on any given UI.

3.2.1. *Nielsen's usability heuristic*. A heuristic is a ‘rule of thumb’ rather than a strict rule. Jakob Nielsen produced a list of heuristics in 1994 [38], which surprisingly remain mostly relevant even today. The following principles, which overlap to some degree, are derived from that list.

Name	Statement	Rationale	Implications
3.2.2 Simplicity	The HCI should be as simple as possible, but not simpler!	People use computers to realise tasks, so all parts of the HCI should be focussed only on what is strictly necessary to complete a given task.	Occam's Razor. Minimalist design. Focussed.
3.2.3 Navigation	The navigation options around the HCI should be obvious and clear.	It is important to a user to know at all times where they are and where they can get to.	Use of intuitive controls like 'back' and 'breadcrumbs'. Break up complex tasks. Show progression through a complex task. Redundant sets of navigation controls. Always enable going to 'home'. Make sure any given page/screen has an obvious context.
3.2.4 Feedback	The HCI should provide feedback based on user actions	Users need to know what the system is doing in response to their actions, so that they feel reassured that the system is doing something.	Any action taken by a user should be followed by a display of some form of feedback, even if just a 'timer' icon. Something in the HCI should have changed as a result of any user action and this should be obvious.
3.2.5 Affordance	Controls in the HCI should 'afford' their usage.	Users should not be puzzled and uncertain about how to use a control.	Maximise use of well-known metaphors and standard idioms. Good use of visual clues on usage.
3.2.6 Intuitive	"Don't make me think" [17]	It is pleasant to use an interface that is intuitive when it comes to performing a task (UX).	It must be obvious how to perform a task that is familiar to the user, using the HCI. Minimise the features that are 'hidden'. Try to implement the '3 clicks' rule. Write the User Manual first is good advice, rarely followed.

3.2.7 Tolerance	The HCI should be tolerant of user preferences and mistakes.	There is sometimes more than one way to perform a task, and to err is human.	Allow for many ways to accomplish a task, starting from different places. Reduce to a minimum the scope for errors to be made, by using appropriate controls (poka-yoke). Minimise input. Provide useful help and support if needed, including helpful error messages. Allow users to recover easily from errors. Use helpful constraints to limit user actions.
3.2.8 Consistency	The 'look and feel' of the HCI should conform to a stylistic guide.	Consistency reduces learning time and improves ease of use.	Define style guides for all applications of a given type and platform. Test and audit against these guides.
3.2.9 Maximise Re-use	"Don't make me remember". [17]	Having to memorise data invites errors and produces negative sentiments in many users. Users may not be in a position to note things down.	Recognition, not recall. Re-use data already entered wherever possible. Ensure users can access data already entered/stored. Use email, text and print to provide a copy of data provided by the user or by the application (e.g. codes etc.).
3.2.10 Accessibility	Using the HCI features should be accessible by all users.	A significant number of users may have a variety of disabilities	The interface should take into account, within reason, users who have colour-blindness, are partially sighted or blind or have physical impairments. Use of alternate texts on images, tool tips, text-as-speech. Recognition of voice commands.

3.3. Recognise the following UI paradigms

Candidates should be able to recognise and explain the various paradigms listed in the syllabus.

UI paradigms are ‘ways of thinking’ about the construction of the UI. The following are all examples of common paradigms, which are not necessarily mutually exclusive.

3.3.1. *Command line interface*: this interface is mostly used by the IT community today to interact directly with the underlying operating system by executing commands as strings with switches. Much loved by the Unix/Linux community. For example, these commands can be used to install and configure software.

CLI is widely used by developers using modern interpreted languages like Python or Javascript. The commands are entered via a ‘console’, which accepts the commands and provides feedback.

3.3.2. *WIMP:* WIMP stands for **W**indows, **I**cons, **M**enus and **P**ointers. Since the advent of the GUI, late 80s, this has been the paradigm most users would have used and would readily recognise. It is still very influential and widely used, although its dominance is gradually being eroded, as technology evolves towards more natural forms of HCI. There are 3 basic variations:

- *WIMP dialogue based*: this interface gets a task done by presenting the user with a series of dialogue windows which the user interacts with. Very popular in client-server applications, and stand-alone desk-top applications, based on the ‘dialogue box’ idea to get input or display information.
- *WIMP forms*: forms are used to get information from a user which is then submitted for back end processing. A form is a group of data entry controls that capture data related to a topic.
- *WIMP manipulation based*: objects are manipulated directly by the user, using a pointing device, for example in simple 2D card games like Solitaire.

3.3.3. *Desktop interface:* Desktop applications are installed locally on a client computer. They are able to function off-line, although they often require an internet connection to achieve full functionality. The interface is often referred to as a ‘rich client’ interface, since the UI can be designed and programmed very precisely for the tasks required, using standard and bespoke widgets. The UI doesn’t depend on the programming features offered by browsers, which are limited.

3.3.4. *Static and dynamic web interface*: the static form of web interface presents static document information only, with links between documents, which was the original purpose of the Internet. The user can read static information and navigate around the site, search etc. but there is no or very little dynamic content. Examples include newsletters, newspapers on-line, local council website etc.

A dynamic web interface reacts and changes its content in line with user actions. Examples are buttons that change their text or colour following a user action. A

dynamic interface has controls like forms to accept data input which have a limited amount of data validation capability.

Compared to rich client, this interface has been rather limited in its versatility in the past, but presently there is something of a convergence of paradigms, with desktop and web interfaces gradually adopting the best features of each other, including their metaphors and idioms.

3.3.5. *Direct manipulation:* paradigm suitable for mobile devices and touch screens, the user is able to do actions like resizing, scrolling, and selections with their fingers or a stylus. For example, expanding a photograph with fingers only.

3.3.6. *VR, AR and MR*: this paradigm comes from the gaming/virtual worlds arena but is creeping into enterprise applications. Some kind of device, say a headset, is worn by the user who enters into a 3D VR world. Gestures are recognised too. AR is 'augmented reality', which is proving very useful in many applications, from medicine to logistics. The user wears some kind of device, like Google glasses, and can see relevant information super-imposed on the real world in real time. Mixed reality (MR) is a combination of VR and AR.

3.3.7. *Voice and gesture recognition*: the problem with WIMP is that it is not very intuitive for the application's user. Considering the final objective of the UI is to control the computer's actions, voice is much more natural for human beings. Examples of voice control are Alexa and similar devices, also voice controlled menu choices over the phone. Gestures recognition is being researched too, since a high percentage of human-to-human communication is actually via gestures of various kinds, including facial expressions.

3.4. Recognise the following examples of UI related models and techniques and their potential use in HCI development

Candidates will be able to recognise and explain the models and techniques listed in the syllabus.

UI/UX design is a team activity, and may involve many roles such as Product Owners, the users themselves, analysts and developers, as well as specialists in this area. The tools and techniques listed here are a selection of those available to a UI/UX team. Many of these techniques reinforce each other and overlap in their purpose.

3.4.1. *User discovery and user analysis*: Not all users of any given application will have the same characteristics or require the same facilities. The HCI designer has to understand what distinct groups of users the interface is going to be used by and decide how to accommodate each group's needs. Ideally each group would be presented with only the features that they need to use, and only see the information they should see. Example: A Holiday Villa Booking Website has very distinct user groups: Would-be Holidaymakers, Villa Owners and Multi-villa Accommodation Managers.

3.4.2. *Personas*: Personas are fictional characters, which the analysis team creates, based, for example, on research and workshop activity. Each persona represents a different user group. Creating personas helps to understand and document the different users' needs, experiences, behaviours and goals. Above all it allows the team to frame the ‘mental picture’ that each user has of the tasks that they must perform with the computer. This mental picture should drive the design of the UI.

3.4.3. *Storyboards/wireframes/prototypes*: Storyboarding comes from the film making industry, where it’s used to map out a section of a film or advert. In UI design a storyboard is often a series of sketches linked together in a cartoon strip that maps out the accomplishment of a task by a user. The focus is on both the business process *and* the use of IT within it. Storyboards help designers to string together personas, processes, epics, user stories and investigation findings to develop requirements for the software product.

A *wireframe* might be described as the ‘skeleton’ of the eventual user interface, particularly in the web environment, where pages are divided into blocks of various kinds. It's generally a low fidelity sketch, sometimes just whiteboard stuff, although there are tools to support this process. Wireframes are intended to convey the main features, functions and content of a UI, without getting too deep into the visual design. They may be used as part of a storyboard, or simply as stand-alone prototypes.

A *prototype* is an early visual sample of UI design proposals used to get feedback and allow rapid experimentation with new ideas. Prototypes range from sketches through to working code that has been hastily assembled as a proof of concept. Note, however, that prototype software is not confined to the UI.

3.4.4. *Task analysis:* Applications are deployed in support of user tasks that sit within business processes, so the UI designer must understand these tasks very well, in order to make the HCI suitable. The aim of UI design is to replicate and reflect the users’ mental model of the task as closely as possible. Task analysis also includes understanding the environment in which the task take place, the required procedure and such things as volumes, distractions, potential for interruptions etc.

3.4.5. *Customer journey*: Customer Journeys (also called User Journeys or Journey Mapping) is a UX methodology used to get insights into a user's experience when interacting with an organisation's service or product across all touch points and channels. The focus is not just on the mechanics of the actions being taken, but also on the emotional engagement with the user. It should feel comfortable and pleasant to use an application, not fearful and stressful.

These ideas have been extended to include all users, for example front-line employees.

3.4.6. *Site and navigation maps:* These are graphical representations of the options available on say a website and shows how options are linked together

hierarchically. These models help to balance and control the width and depth of the option hierarchy.

A desirable feature of structure limits the depth of options in the UI as much as possible. There is a famous rule-of-thumb that a user should be able to get anywhere within '3 clicks'. The site map technique also helps to cross-reference and link options suitably, so users can navigate across any given level without having to retreat back up the hierarchy and down again. These maps help accommodate the needs of differing user groups successfully.

3.4.7. *Data input validation and verification*: The UI designer for enterprise applications is expected to appreciate the need for both data validation and verification. The enterprise's Information System is only as good as the quality of the information that feeds it.

- *Validation* ensures that the data conforms to a defined set of rules, such as format, data type etc. Most development environments include features that make validation a relatively simple affair, so the developer should be able to design the UI in such a way that data quality is assured from this perspective.

- *Verification* attempts to verify the 'truthfulness' of the data. This is much harder. For example, a user could enter a valid date, but is that really their date of birth? Examples of verification steps include double entry of data (like an email address), comparing data entries with other known data, and requiring a 2-step verification process, for example via an email link.

3.5. Recognise a UI is made up of a group of controls. Identify the type of a given control.

3.5.1 Classify controls as Command (C), Data Input (D), Presentation (P), Navigation (N) or Feedback controls (F)

A control is a UI element that displays content or enables interaction. Controls are the building blocks of the user interface. A UI pattern is a template for combining several controls to address a commonly recurring UI problem, for example sign-on.

Most of the types of control used today have come from a style of application called client-server, which is mostly associated with the Windows operating system and desktop applications. Web applications started life as simply displaying pre-formatted text, so the need for controls was very limited. Gradually web UIs have adopted more client-server type controls and invented a few more besides. Meanwhile stand-alone desktop and mobile apps have adopted something of the web UI style. Hence what we have now is a steady convergence towards a common UI and a common set of controls, which must be good for everyone.

For the purpose of the DSD syllabus each control below has been allocated to a single classification, although it is recognised that in some cases other

classifications are possible. It is also recognised that many other forms of controls are in use, and this is an area that continues to evolve rapidly.

3.5.2 Recognise an example of each type, as given in the guidance notes

Window (P)
A *window* is a container for controls that make up a self-contained part of the UI. It is a visual area of the UI and is framed by a window decoration, which are inherent controls that are platform based (e.g. an 'X' to close).

Windows display the output of, and may allow input to, one or more software processes. In a web environment the browser displays a document in a window.

Frame (P)
A *frame* is an area of a window that groups controls together visually based on some relevant criteria. Often there is frame title and a border. An example is a group of radio buttons or a set of data entry fields. In a web UI any control can be framed, there is no separate control as such (see *forms* below).

Menu (N)
Menus allow the user to navigate around the UI by selecting from a list of choices, a metaphor borrowed from the restaurant industry. Sometimes menu options execute commands directly. There are many styles of menu in use, including main menus laid out across a window, drop-down menus, hamburger menus and context menus.

Cursor (F)
A *cursor* is some form of indicator used to show the position in the UI that will respond to the next input from an input device. Also used to display feedback resulting from a user action, such as a timer/hourglass which indicates the action is being processed.

Label (P)
A *label* is a graphical control element which simply displays text somewhere within a window. It is a preset static control with no interactivity. A label is generally used to give information about a nearby control, like a data field.

List (P)
A *list* is a control element that displays a preconfigured list of items. This may be for information purposes only, but they can also allow selection of a single item or multiple items for some purpose relating to the task in hand. The list display may be *drop-down*, with the contents not displayed till the user clicks on it.

Image (P)
An *image* is a control that can contain an image chosen from various standard formats (JPG, PNG etc.). Some variations of this control also allow video and other media.

Data Entry (D)
A *data entry* control may be required to capture input from the user, especially where this is in the form of text that cannot be pre-determined. These controls can have validation conditions attached to them, for example to validate the data type entered by the user. It may be possible to attach a validation code routine, depending on the platform. In a web UI controls like this are contained within *forms*.

Radio Button (D)
A *radio button* is a UI element for getting user input that has a value of *True* or *False*. The user can select the button to indicate True or de-select it to show False. The appearance of True and False conditions will be distinguishable in some way. Typically, radio buttons are arranged in a group and in such cases only one member of the group can be selected as True. Example: “pay by credit card; pay by debit card; pay by PayPal”.

Checkbox (D)
A *checkbox* also accepts user input and allows the user to select True or False, but in a group more than one member may be selected as True. Example: “contact me: by email, by text, by phone”.

Combo box (D)
This control is a combination of a data entry and list control. Typically, users may select an item from the list or enter free-form information.

Spinner (D)
A *spinner* shows a list of valid data values that could be selected, with a means of scrolling through the list quickly to find the correct value. For example, find a particular year.

Slider (D)
A *slider* is a control that allows the user to pick a value from a continuous range, for example a volume control.

Button (C)
A *button* is a control, usually labelled, that allows a user to execute a command. Examples are ‘Next’, ‘Submit’ etc. In a web form this is used to submit the form’s data entry fields to the server for processing.

Hyperlink (N)
A *hyperlink* is a control containing a reference to the location of some resource, like another page. The resource could be located across the web or in the local environment. These controls can be placed almost anywhere in a web UI. Indicated visually by underlining, distinct colors and fonts. Menu options are usually hyperlinks in a web UI.

Breadcrumb (N)
A *breadcrumb* is a navigational aid in user interfaces, especially used in websites. It allows users to keep track and maintain awareness of their locations within programs, documents, or websites, showing the 'trail' of how they got to where they are. Often the user can select any part of trail to return back to where they were.

The term is a reference to the trail of breadcrumbs left by *Hansel and Gretel* as they wandered through the woods, so they could find their way back.

Progress Bar (F)
A *progress bar* gives a visual clue about the progress of some software process towards completion, for example a download. It is also used in media graphics to show the progress of the playing of things like songs and videos.

4. Digital solution architecture and design (15%)

This section of the syllabus covers the internal organisation of a digital solution that will deliver a digital service. Architecture is a high-level design activity that determines what components should be part of a digital solution and how these components are organised and collaborate. Different styles of architecture have been used for enterprise applications over the years and each has benefits and drawbacks. Note that the focus of this syllabus is on logical architecture and design rather than on physical design using specific products.

4.1. Define architecture and understand the importance of architecture in developing a successful digital solution. Candidates will be able to recognise certain architectural styles and patterns and describe their benefits and drawbacks.

4.1.1. *Definition of architecture*

Candidates will recognise the definition of architecture according ISO 42010 [18] and be able to describe the significance of key phrases within it.

The definition is: *"the fundamental concepts or properties of a system in its environment embodied in its elements, relationships, and in the principles of its design and evolution" [18].*

There are many useful aspects of this definition, as applied to DSD:

- A software system sits in an environment, which is the enterprise and its processes. Many aspects of the software architecture will be connected to and influenced by policies and constraints that are recognised and formulated within the enterprise.
- Architecture describes *fundamental* elements and relationships, which in the case of an application include the components that compose the software, their organisation and how they interact. An example of a fundamental element is a UI, another is a database. Relationships describe the role that each element plays in the overall system.

- ‘Principles of design and evolution’ refers to the need for the software architect to think about the inevitability of change as they design the system. The architect must try to ensure that software can be changed quickly at low cost and low risk, although it is evidently impossible to ‘future proof’ anything completely.

4.1.2. *The connection between architectural decisions and the realisation of requirements.*

Candidates will recognise the importance of architecture in determining the ability of the solution to meet all the specified requirements.

A key step in the realisation of requirements for software is to create the software architecture. There may be many designs that fit a given set of requirements, and the solution architect has to make these decisions, taking into account a number of factors, while liaising with both the technical and the business communities. Factors affecting design decisions may be technical, business or project related, and will include financial considerations.

The need to make these decisions is one of the reasons why adherence to recognised architecture patterns is beneficial, as outlined elsewhere in this syllabus.

4.1.3. *Contrast the effects of 'good' and 'bad' architecture*

Candidates will recognise the importance of creating a ‘good’ architecture and the effects of having a ‘bad’ architecture.

The main effect of ‘bad’ architecture, one that doesn’t follow best practices, is to make change difficult, costly and dangerous. It may mean the software is brittle and easy to break.

‘Good’ architecture has the opposite effect, change becomes easier, more agile, with low cost and less risk.

On the other hand, what constitutes ‘good’ architecture will depend on the problem in hand, which brings in the consideration of monolithic vs. distributed architectures.

4.1.4. *Define a monolithic style of solution architecture and list its benefits/drawbacks*

Candidates will be able to recognise the term ‘monolithic architecture’ and list its benefits and drawbacks.

Digital solutions in the early days of software development had what are called *monolithic* architectures, because that was all that the technology allowed at the time. Monolithic means something like ‘the application is self-contained, everything happens inside a single distributable object, which runs on a single computing device’. The metaphor refers to those huge slabs of stone in places like Stonehenge.

Originally such applications suffered from poor programming practices too and would have had poor internal structures, so-called ‘legacy’ applications. These days

a monolithic application may have a more modular structure internally, but nothing can be deployed independently of the whole thing. There are many modern applications that make use of this architectural model, like stand-alone PC and mobile applications for example.

For the purpose of the syllabus a monolithic style of architecture is thus defined as:

"A style of architecture that bundles all of the application functionality into a single self-contained distributable object."

Some benefits of this style are:

- Simple to create small stand-alone applications, worked on by a single developer.
- End to end testing and debugging is less risky to do, as it doesn't rely on anything outside the monolith.
- Simple to deploy/distribute/replicate.
- Easy to scale horizontally if the enterprise is willing to keep adding computing resources like PCs. Easy to scale vertically, up to a point, by upgrading local computing resources.
- Likely to perform better for interactive applications, since it usually doesn't depend on network resources.
- Often, they can work off-line.

Some drawbacks of this style are*:*

- No reusability of common functionality between distinct applications. The same code might be present in various monoliths, presenting a maintenance problem.
- Can result in a large and complex code base over time, which can be hard for developers new to the application to follow and maintain.
- The whole application has to be rebuilt and redeployed every time there is even a minor change, which could impact business schedules.
- Testing, especially regression testing, can take a lot of time.
- No fault tolerance. Anything going wrong inside the monolith will crash the whole application.

In an enterprise context extensive use of this style of application is increasingly rare, with distributed applications becoming much more prevalent. For this reason, the syllabus emphasis is on distributed applications. There is nonetheless extensive use of this style of architecture, and hybrids, in a modern context in the form of stand-alone mobile 'apps'. There will always be a place for this style of architecture.

4.1.5. *Define a distributed style of solution architecture and list its benefits/ drawbacks.*

Candidates will be able to recognise the term 'distributed architecture' and list its benefits and drawbacks.

The opposite of monolithic is *distributed* architecture. In this architecture the responsibilities for executing different parts of the software processes are distributed across various computing devices, linked by a network. The need for this style started with the advent of networking, when the business wanted multi-user access to applications that shared common data. This trend was reinforced by Client-Server technology which allowed the processing to be split between PC clients and centrally located database servers.

As the number of distributed parts of this style of architecture proliferated, developers needed to de-couple the connections between them, in order to manage the complexity of making changes. This is when Service-oriented Architecture (SOA) ideas began to be used (see references further on in this section). By having part of an application invoke the functionality of another part through a service interface, the coupling effect is greatly reduced.

The advent of the WWW has accelerated the distributed and SOA architecture trends, since it's possible for applications to access and reuse private and public resources located across the web and published as services. For example, invoking a third-party Payment Service. This ushers in the era of DSD we now find ourselves in, which makes extensive use of APIs and micro-services.

The final step to date in this evolutionary path is the use of Cloud technology, which offers to take on the task of hosting and managing distributed architectures on behalf of the enterprise, thus relieving the enterprise of the burden of maintaining complex IT infrastructure.

For the purpose of this syllabus, a *distributed* architecture is defined as:

"a style of architecture where the functionality of the software is divided into logical parts, with the parts intended to be distributed across many computing devices. The parts communicate via a network along which messages travel back and forth. Each part performs a specific role in the application's overall functionality and may be re-used by many applications in a similar role."

Some benefits of this style are:

- Platform independence. Each part of the application could be hosted on a distinct platform allowing for the deployment of the best technology to do the job.
- Development in smaller 'chunks'. It may be possible to develop a part with independence from other parts, especially if the SOA style is applied, as explained elsewhere in this document. This cuts down development time.
- Deployment of individual parts is quicker and easier than deploying the entire application.
- Reusability. Parts can be reused in many applications if they do the same job. Cuts down development time and risk.

- Better use of computing resources, which can be adjusted to cope with the workload variations corresponding to particular components. Horizontal and vertical scalability are built into this style of architecture.

Some drawbacks are:

- Requires sophisticated and expensive technology to scale automatically. E.g. load balancers and hypervisors.
- Complexity of integration of parts and compatibility of parts.
- Testing end-to-end and debugging is more complex and finding out where problems are coming from may be difficult, due to the many possible combinations of factors. It is necessary to have comprehensive audit information and tools for tracing transactions.
- Performance issues, especially if the application is highly interactive. For example, bandwidth contention.
- Typically, more 'teething problems' in developing new applications, until the appropriate distribution pattern emerges and 'settles down'. This means longer development lead times and more upfront cost, and perhaps a period of user frustration.

4.1.6. *Recognise and define the system properties known as 'cohesion' and 'coupling'.*

Candidates will recognise and be able to explain the notions of 'cohesion' and 'coupling' as they applied to systems of components.

These terms refer to important properties of any arbitrary system, where a system is defined as any grouping of components that collaborate towards some common goal.

Coupling is the property of a system which measures the dependencies that exist between the system's components. Coupling is a measure of how much components have to 'know' about each other for the system to operate successfully.

Cohesion is the property of a component of the system which measures the degree to which everything inside the component is dedicated to doing 'one thing and one thing only'.

As a general rule many benefits accrue to systems designed using components that are highly cohesive but loosely coupled.

4.1.7. *Explain what is meant by 'service', 'interface' and 'service-orientated' style of architecture (SOA). Explain how this style benefits distributed systems.*

Candidates will recognise and be able to explain the terms 'service', 'interface' and 'SOA'.

A *service* defines behaviour that is exposed publicly by a system, or a component of a system, and which can be invoked by some external agency.

An *interface* defines the access points through which the service behaviour can be invoked. The terms ‘internal’ and ‘external’ are significant here; the service definition and its interface set boundaries on what can be known about a system or component from the ‘outside’.

Distributed architectures benefit from a Service Oriented (SOA) style of architecture, which is an architecture based around a collection of services and interfaces. A SOA style of architecture decouples the service consumer from the service provider as much as possible, while each service is highly focussed internally on performing a single well-defined behaviour.

The use of a SOA style in the context of a distributed architecture tends to reinforce the benefits of distributed applications while mitigating some of their drawbacks.

4.1.8. *Recognise what an architecture pattern is and describe why it is beneficial to use one*

A pattern in general terms is a template solution to a recurring problem. Patterns are used in all aspects of solution development from analysis through design and on into the code (see section 7).

Architecture patterns are tried and tested arrangements of the components that make up an application. By adopting a pattern, the designer can benefit from the experience of others, and communicate more easily with other collaborators. The application can be put together quickly, with confidence in its robustness.

4.1.9. *Recognise the components of an MVC (Model View Controller) pattern of architecture and describe its merits.*

Candidates will recognise the components of an MVC style of application architecture and be able to describe its key components.

A distributed system is made up of components that expose interfaces to each other. However, there are many types of component possible in an application, for example UI components, control components and data components.

Model-View-Controller (MVC) is an application architecture pattern, one of the first attempts to design *layers* to separate distinct concerns in an application. These early ideas were instrumental in supporting the move towards distributed architectures.

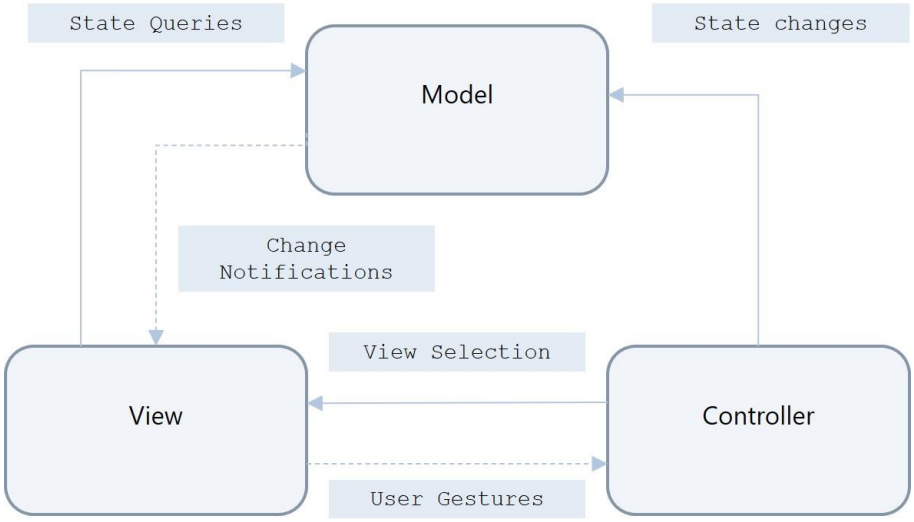


Figure 4: Original MVC architecture pattern [20]

The application roles identified here are:

View: This is what the user sees, so this layer groups together all the UI-related components.

Model: The Model is the layer that has the business knowledge, including business rules, data and domain knowledge.

Controller: Components in this layer broker requests from the View to the Model and ensures they can't interact directly, and therefore don't have coupling dependencies.

Based on MVC and variations of it, a digital solution architect may therefore decide to group application components together into 'layers' or 'tiers' on the basis of their role in the application. This may be called 'n-tier' architecture in a modern context. Note that although the terms *layer* and *tier* are often used synonymously, for some *layer* is a logical grouping, while *tier* is used to indicate a physical grouping. This terminology difference is not examinable, since it isn't standardised, and this syllabus treats the terms as synonyms.

The benefits of using these or similar layers is the ability to change something in a layer and not have that affect other layers. An example of this is illustrated by the need for many modern applications to support a number of different UIs. For example, an on-line Banking Application, which must be deployed for a web UI and a mobile UI. Only the UI layer needs to change in such a case, the other layers that make up the application can remain the same and be re-used. The same is true of using a range of databases to implement parts of the Model, which can be changed freely without affecting any other layers.

A literal interpretation of the original MVC as shown above is rare. The term MVC is often use nowadays as a generic term to cover any style of multi-layered architecture. There are therefore a number of standard variations on the original

Copyright © BCS 2020

theme, which the architect can choose from, especially for organising web applications. The choice of styles should be governed at the enterprise level to ensure consistency.

4.1.10. *Recognise the components of a hexagonal pattern of architecture and describe its merits*.

Candidates will recognise the components of a hexagonal style of application architecture and be able to describe its key components and merits.

The hexagonal architecture, or ports and adapters architecture, is an architectural pattern which is an alternative to the layered MVC pattern [19].

Layers inevitably create a certain amount of coupling – some layers have to 'know' about other layers and what they do. The hexagonal approach aims at creating a more loosely coupled application in which core application components that can be easily connected to their software environment by means of ports and adapters. This arrangement makes components exchangeable anywhere outside the core, and facilitates test automation. It is a very promising style, since it allows the introduction of any port/adaptor as 'plug-ins', even those not contemplated by the original designer. It also makes it clear what the central locus of the application really is.

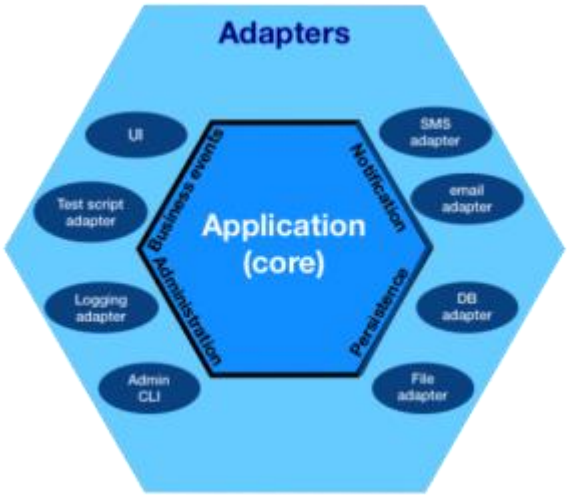


Figure 5: Hexagonal architecture pattern [19]

4.2. Identify the key elements of a typical contemporary digital solution and explain the role that each part plays in the overall architecture.

A contemporary digital solution may consist of many 'moving parts'. Some of the more common components are listed in the syllabus, based on the graphic below. This list is provided as an illustration of contemporary design, which, however, continues to evolve and adapt.

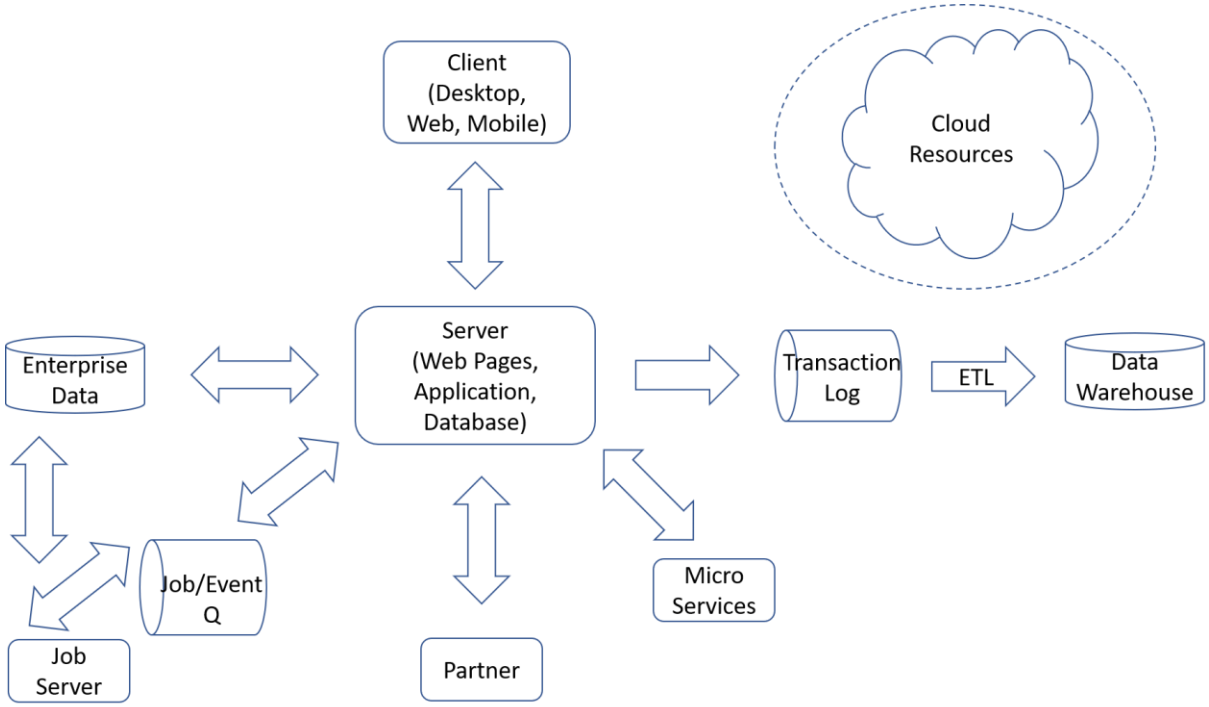


Figure 6: DSD – Distributed Enterprise Application - Logical Architecture

The description of these items is as follows.

4.2.1. The entire arrangement depicted in Figure 6 is heavily dependent on the use of networks and internet technology, which provides the transport needed for passing messages around the system. Time, timing, messages and behaviour are the salient features of a modern enterprise application complex.

4.2.2. The *Client* hosts the interface with the user and may have local code and data installed. This is where the browser would be, if used.

4.2.3. The Client communicates with a *Server* at an address located on a network or the internet. The communication is generally synchronous. This 'contact' server might employ other servers to perform various tasks. Distributing tasks across servers wisely is good practice in a distributed system, as it provides for optimisation, flexibility and resilience. Note that the word 'server' here is being used in a logical rather than in a physical sense.

4.2.4. Enterprise applications often need access to *Enterprise Data*, which is data designed and maintained under the control and governance of the enterprise (see Data Management in section 1.5.5). This data is potentially shared by many applications and forms the core of the Information System. Caches (not shown on the graphic) may also be used, which store data temporarily for use by applications, to avoid having to do database accesses too frequently. Database access is very expensive in resource terms. However this may present currency issues (see section 4.4).

Copyright © BCS 2020

4.2.5. Most enterprise applications generate transactions that the business would like to store, mine and analyse for business intelligence (BI/Analytics). These transactions are logged and copied or removed into a *Data Warehouse* environment by a process called *ETL* (Extract, Transform, Load).

4.2.6. Various forms of queue are used in distributed applications as a way to decouple components and manage differing workloads across the system. For example, an enterprise application may have a requirement to trigger 'batch processing' performed by a Job Server. This may be done asynchronously via a Job Queue. Both batch and interactive jobs may generate events placed in Event Queues, also known as Message Queues, that trigger other jobs or affect interactive processing in some way. Data Pipes are a form of queue used to shift data around the system especially into the data warehouse environment.

4.2.7. Distributed enterprise applications adopting a SOA style will make use of *APIs* and *microservices* to perform certain tasks in the application, as appropriate. This communication is normally synchronous, but occasionally asynchronous. For this to be a reality some form of services catalogue has to be managed, perhaps in the form of an Enterprise Service Bus (ESB - not shown on the graphic). This software is known generically as *middleware*. The application invokes any given service via the middleware, reducing the coupling effect throughout the system.

4.2.8. Applications may need to communicate with *Partner* organisations in real time or asynchronously.

4.2.9. Any or all of these application parts could be hosted by a *Cloud solution*.

4.3. Recognise the term 'web service' and the value this represents.

4.3.1. A system is a collection of components, but each component could itself be regarded as a system. Where does this hierarchy stop? This is a question digital solution architects have to face up to.

The answer to this lies in the direction of the idea of a *microservice*. This idea is the logical extension of the SOA and hexagonal architecture ideas mentioned elsewhere in the syllabus.

A *web service* is simply a service accessible across the internet. The means of access is through a published *API* (Application Programming Interface), which the programmer can use in their code to invoke the service. This may include passing data into the service invocation and receiving data back. In most cases the invocation will be synchronous.

It is important for a developer to be aware that there are 2 basic types of API in widespread use today for invoking web services. They are *SOAP* (Simple Object Access Protocol) and *REST* (Representational State Transfer). The details of these protocols, however, are outside the scope of this syllabus and will not be examined.

The term microservice is used to denote a web service that is narrowly focussed on doing a very specific job. Extending the basic SOA ideas, a 'big' problem is broken down into lots of 'smaller' problems. It is not easy to define precisely what the boundaries should be of a microservice, but it will be characterised by the following considerations [21] [22]:

- Simple/quick to test
- Simple to extend
- Independently deployable from other services
- Loosely coupled with other services
- Domain agnostic

Performing a software task within a distributed enterprise application may involve invoking a large number of such services. It is said for example that loading a typical Amazon web page uses around 200 microservices!

4.3.2. The value of this architecture should be obvious from the comments made above concerning services and distributed architectures. Microservices are simply an evolution of these ideas taken a step further, attempting to find the logical 'bottom' of the component hierarchy.

4.3.3. An important concept in designing web services is the concept of *'state'*. Ideally, and this would be the case for microservices, a service is 'stateless', which means that each invocation is completely independent of any other invocation; the service has no 'memory' of previous invocations.

So, for example a payment service takes a payment from a customer. The next time the same customer makes a payment, the invocation of the service is no different to the previous occasion. The service doesn't 'remember' the previous invocation. In practice this mean that no persistent data is held by the service itself between invocations.

Contrast this with a software process underpinning a complete business task. Imagine a process for booking a flight. This process would be labelled 'stateful' because it is relatively long running and must maintain the current state of the booking all the way through to the end of the transaction, via all the steps involved. Each step might invoke one or more 'stateless' microservices. This style of service becomes quite explicit in those tasks where users can save data midway through a process and resume the process at a later date.

4.3.4. A service whether stateful or stateless can invoke and use other services. This brings in the topic of service composition styles. There are 2 basic service composition styles, although they can be mixed. They are *orchestration* and *choreography.*

Orchestration conveys the idea that there is a conductor in front of an orchestra directing the efforts of others. The graphic in Figure 7 illustrates the application of orchestration:

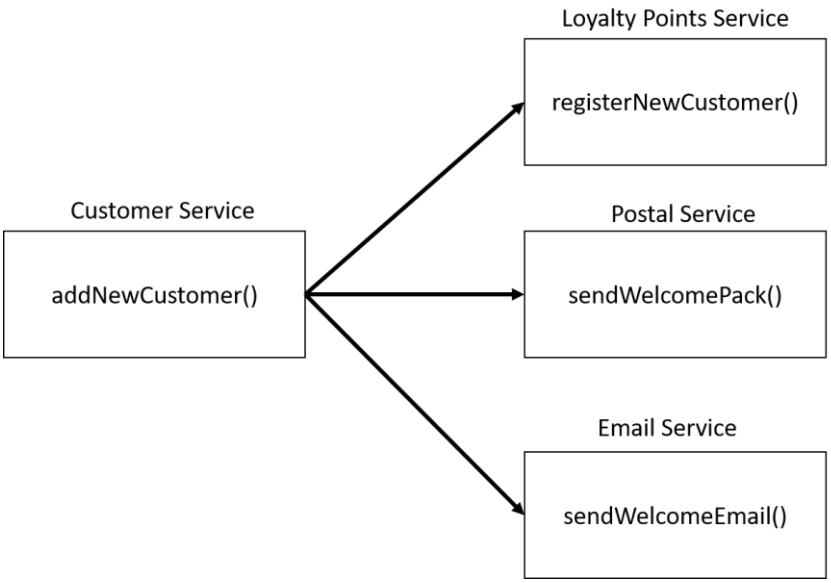


Figure 7: Orchestration illustration [21]

In this example, a software module invokes 3 services synchronously. Each service is invoked in turn by the module – the module is the orchestrator. This arrangement has the advantage that control of the process is centralised, so for example errors could be handled, but has the disadvantage that the controller has to 'know' about the services it should invoke (i.e. there is some coupling). Another disadvantage is that the main module has to wait for all these other processes to finish before it can move on.

An alternative arrangement is called *choreography*. In a choreography there is no central conductor, but rather a group of components that react to events. A well-used metaphor for this is the performance of a ballet, where each dancer's performance is a planned reaction to events occurring on the stage. In Figure 8 is a graphic depicting the same software process as a choreography:

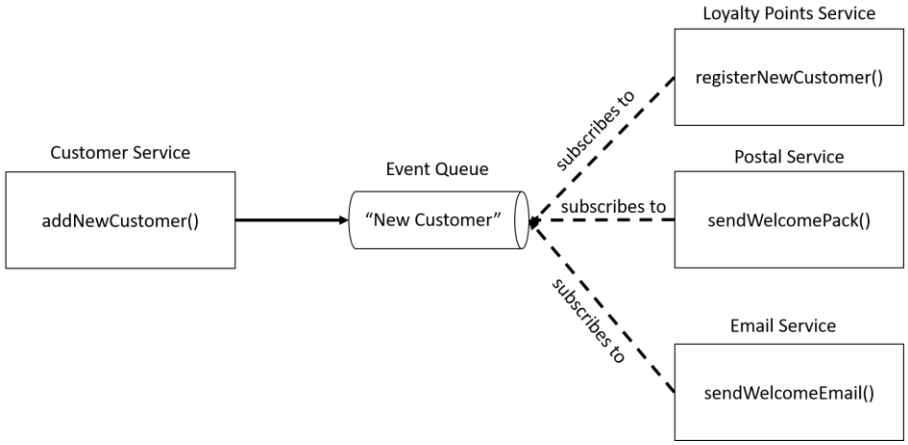


Figure 8: Illustration of choreography [21]

In this arrangement the main software module places an event on an event queue. The other services are subscribers to this queue and react appropriately to the events place in it.

This is the usual basis for implementing asynchronous communications but note that choreographies don't have to be asynchronous. The term also applies to any peer-to-peer relationship which involves message exchanges [16].

The advantage of the choreography arrangement is that there is no coupling between the components involved. They just don't know about each other. A major disadvantage though is not knowing when everything will be finished or indeed did everything get done.

In practice solution architects will want to use a mixture of these styles depending on the task in hand. Ideally large parts of software processes would be designed as choreographies, to promote low coupling, but that just isn't practical for many tasks, especially interactive ones.

4.4. Recognise and identify some of the issues that are inherent in a distributed systems architecture, and explain typical design trade-offs and solutions to these issues

"The internet *is* the computer" is a famous saying that celebrates the convergence of distributed computing and the WWW. There are tremendous benefits that we derive from bringing together these evolutionary paths, but they come at a price. Managing distributed systems is much more complex than managing monolithic architectures. We must remember Murphy's Law at all times - "if it can go wrong it will".

The digital solution designer must be conscious of the following considerations:

4.4.1. *Security*: having many moving parts increases the 'attack surface' exponentially.

4.4.2. *Complexity*: there are many different technologies that need to work together.

4.4.3. *Scalability*: predicting and coping with huge variations in workload is a necessity in this type of environment. These adjustments are need in real time, so it has to be automated, increasing the technological complexity. Nearly every day we are presented with news of websites crashing because they couldn't handle the traffic.

4.4.4. *Failure handling:* finding out where it all went wrong can be a complex and time-consuming process. Reacting to failures must usually be done in real time too, switching between failing resources automatically.

4.4.5. *Concurrency and Latency*: this refers to coping with multi-user access to single resources and synchronising sources.

- In a very simple example of a concurrency problem, imagine 2 users accessing the same Customer record in a database, and then having this data passed back to their respective client systems. One user changes the data and saves it. The other user wants to save their changes too, but by now they are working on an old copy! Should they be allowed the update?

- Latency is a feature of all networks, and it refers to the amount of time it takes to pass information around the system. The larger and more distributed the system is, the more latency problems can become a real issue. The designer must be conscious of this problem and takes measures to eliminate its effects.

The answer to these kinds of problem involves time-stamping and record locking strategies, which, however, bring their challenges too. The details of the various strategies available here are out of the scope of this syllabus.

4.4.6. *Testing challenges*: the challenge is how to provide end-to-end testing of distributed systems where there are so many parts involved, and so many possible combinations of parts. Again, there are answers to these problems, but they are out of scope for this syllabus.

4.4.7. *Cloud Services as a solution to some of these issues:* A major selling point of using *cloud services* is the ability of the service provider to take on and manage a large number of these and similar issues – for a price! This frees up the digital solution developer to focus on solving their enterprise's particular business problems.

5. Data and Information Architecture and Design (12.5%)

Data is an essential resource for businesses and organisations of all types which rely on it for operational transactions, management decision-making and strategic planning. Each enterprise needs an information system, which is a collection of components that undertake the collection, storage and processing of data and the information that is generated from it.

In a modern context, enterprise information systems are supported by computer and communications technology, which has elevated the power of data and information to become a truly strategic resource.

Digital solution development professionals need to be concerned with data and information architecture and design for a number of reasons, amongst which are:

- The data-intensive nature of the applications that are the focus of this syllabus implies that developers will need to concern themselves with where the data for the application is going to come from, and how will the applications manipulate it. Developers will also need to be aware of constraints on the use of data imposed by legislation and enterprise security concerns.
- The needs of the application for data and information should be checked for consistency with the enterprise's view of data and information. Enterprises are increasingly aware of the need to manage and govern data from a centralised perspective, so the application's view of data should be reconciled with this. See section 1.5.5 of this syllabus.
- If 3rd party packages and services are part of the solution, what mappings and transformations are going to be required?
- Transaction data feeds analytics, so the developers must think wider than their own application's scope in many cases.

5.1. Distinguish between data, information and information systems and recognise distinct views of the components of an information system

5.1.1. *Recognise and explain a definition of data.*

It is surprisingly difficult to come up with a single definition of the word ‘data’, before becoming embroiled in academic arguments. A pragmatic definition of data and the one that is used in this syllabus is:

“data is a collection of data items, each of which is a useful fact about some entity of interest to the business.”

An *entity* is any ‘thing’ that has a distinct identity from the business’s point of view. So, a business might hold data about Customers, for example, or Sales Orders.

Each data item represents a fact of significance to the business about the entity and will have some kind of value. So, a Customer has a name, which is a fact with a text value, and a Sales Order has a date on which it was placed, which again is a fact this time with a date value.

These data item values about ‘things’ are the basic building blocks of information and an information system. Such values can be recognised and interpreted by a human or a computing system and can be stored, processed and transmitted by and between information systems.

Each data item must be associated with *metadata*, which is ‘data about data’. Metadata describes the data item, what it represents in the ‘real world’, what sort of data is it, what values can it take etc. Further on in the syllabus metadata is mentioned again in the context of a *data dictionary*.

Metadata is needed in order to interpret the data for business purposes. Just having the fact ‘160173’ associated with an Employee is meaningless until we know it is their date of birth.

5.1.2. *Recognise and explain a definition of information*

Information is a slippery concept too, hard to define. From a practical point of view again, information is defined for the purpose of the syllabus as:

“data set in a context and organised so as to convey meaning to an interested party” [24]

So, one or more facts about an entity or an event of interest may be communicated as information to a human or to a computing system and is interpretable for their purposes.

Hence, for example, the phrase “the birth date of Employee John Smith is 16/01/73” is information if it is made available to someone who is interested in knowing that. In an enterprise context at least, information, which is based on data, is used to take decisions.

The same data, and data items, may be used and re-used to produce multiple forms of information for different purposes and interested parties. Information may be produced by using arithmetic and statistical techniques to combine data with other data. Other forms of modern analytical processing, such as pattern matching and machine learning, may also be used to produce information from data.

So, a list of Employees whose birthdate is before the end of 1973, is information for someone, that re-uses the data item which is the birthdate of John Smith.

5.1.3. *Recognise and explain a definition of information system*

For the purpose of this syllabus an Information System is defined as:

"An integrated set of human and digital components that manipulate specified data and information resources, including the collection, storage, processing, communication and retrieval of data and information."

Data and information are part of a model of Information Systems known as the DIKW pyramid with data as the base of the pyramid and information building upon it. See Figure 9.

Information that has been communicated to interested parties becomes *knowledge*, whilst *wisdom* is needed in order for the information system to use this knowledge effectively to take decisions and solve problems. This implies the information system of an enterprise includes the knowledge and wisdom of its people as well as its machines.

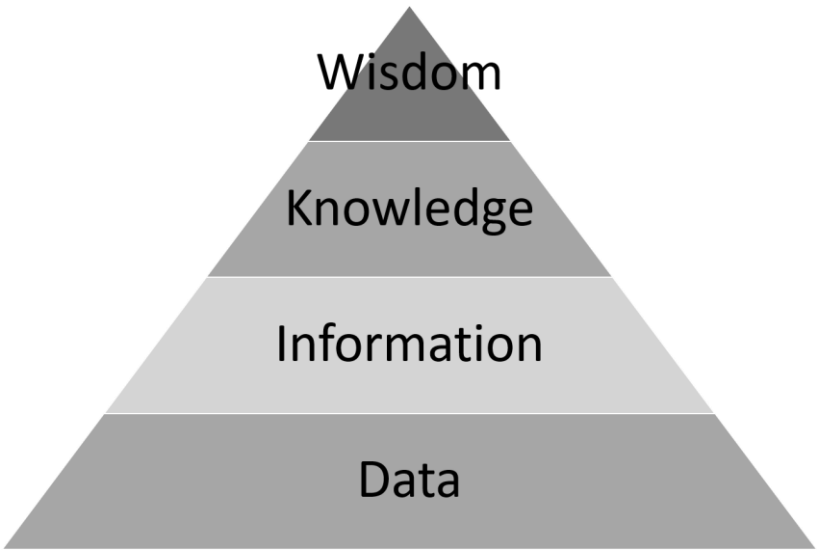


Figure 9: DIKW Pyramid

IT has long been a key technology applied to the Data and Information layers of this pyramid, but it is now also seen as intervening usefully in the Knowledge and Wisdom layers too, alongside human actors.

The significance of Information Systems for developers lies in the role of enterprise applications in implementing a fit-for-purpose Information System. It's easy to lose sight of the fact that the overall fundamental purpose of applications software is to manipulate data and produce information efficiently and effectively in support of business activities.

There are several interesting distinct and overlapping views of the Information System, some of which are listed in the syllabus.

5.1.4. *Distinguish structured from unstructured data*

A modern Information System contains both structured and unstructured data.

Structured data is data held in the Information System about an entity in accordance with a known and pre-defined format. This data is often organised into fixed record structures like database tables. This type of data is easy to query using technologies like SQL. An example of this would be Customer data held in a relational database.

For a long time, automated Information Systems could only handle structured data. However, the desire to exploit unstructured data is now very common in modern organisations. Examples of this include emails, instant messaging, social media, human-written reports, documents, images, audio recordings and videos.

The key characteristic of unstructured data is that it contains many potential facts of different kinds in multiple dimensions that are all 'jumbled up' together and not organised into a fixed format, making interpretation and interrogation of the data for business purposes more difficult. A query like "how many times did lamp posts appear in this video" cannot be answered using traditional query methods. The same video includes images of red cars, but are we interested in that?

It should be borne in mind that unstructured data will typically have some structured data associated with it such as title, subject, author, date created etc.

Data has been held traditionally in information systems based on certain suppositions about how it would be used to create information. Increasingly businesses are aware that both structured and unstructured data contains interesting information apart from its formal or intended use. AI Techniques, such as natural language processing (NLP) and pattern recognition, are increasingly being used to extract interesting information from all sorts of data which can then be organised into structured forms for storage, processing and analysis. A lot of attention is being focussed on social media, for example, for this purpose, analysing things like product mentions, ratings, comparisons with competitors etc.

5.1.5. *Explain the concepts of master data*, *reference data and transaction data*

Master data is a term applied to establishing a standardised definition for data associated with entities that are used across the enterprise and should therefore

mean the same thing everywhere. For example, the data for an entity like a Customer could be treated as master data.

Master data is often referred to as a “single source of the truth” across the organisation about a key entity.

Master Data Management (MDM) is a data governance initiative, part of Data Management, which will seek to define the identity and data of master data concepts and may well govern the availability and use of the associated data, through data services.

Master data and its management both depend on the concept of business ownership of data. It is possible to separate the ownership of the definition of data, its structure and the associated business rules, from ownership of the operational data which includes the responsibility for data quality, concurrency and correctness.

For application developers master data is very significant, since to the extent that applications need master data, it will have to obtained from the master source. Any extension to master data may have to be approved.

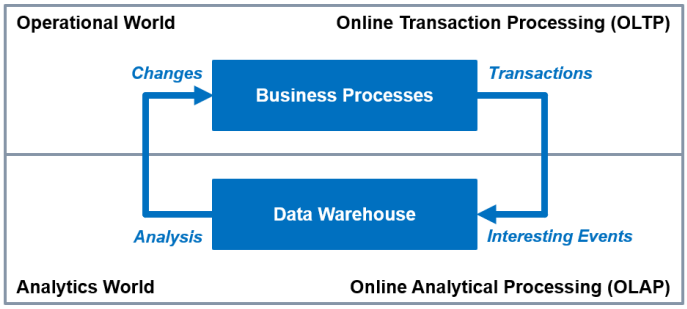
Reference data refers to sets of values or classification schemas that are standardised across an organisation or enterprise and are used by the information system. Reference data changes infrequently and may be externally sourced. Examples include lists of valid values, codes, abbreviations and categories but is not limited to descriptive data and may include other infrequently changing data such as locations, organisational units and the names of products and services. A list of valid postcodes in the UK is an example of external reference data.

Transaction data refers to records of internal or external events that occur as part of business operations. Examples include data captured about sales orders, holiday requests, payments to suppliers, and stock issues etc.

A major goal of many enterprise applications is to capture transaction data. These transactions, which record interesting events, are the main ‘raw material’ of an Information System. Transaction data can be thought of as associating a number of master data subjects with a date/time event. Reference data may be used as part of the transaction record. Transactions are accumulated in data warehouses over time and are therefore the basis on which BI analytics is performed. This theme leads into the next topic.

5.1.6. *Recognise 2 views of data processing in Information Systems*

Another way to view the Information System is as a ‘game of two halves’; OLTP and OLAP.



	OLTP	OLAP
Applications	Transaction Processing Systems (TPS)	Management Information Systems, Decision Support Systems
Typical Users	Staff	Managers, Executives
Horizon	Days & Weeks	Months & Years
Refresh	Immediate	Periodic & NRT
Emphasis	Create, Update, Delete	Read

Figure 10: Comparing and contrasting OLTP and OLAP

In Figure 10 one can appreciate the main differences between these two halves of the Information System. Their characteristics are quite different, so much so that it is usually necessary to physically separate these environments, so they don't interfere with each other.

OLTP

Online transaction processing (OLTP) means handling and recording transactions, as defined in the previous section, that generally take place in real time as part of the business operations of an organisation. In the commercial world, examples include e-commerce and in-store point-of-sales systems that must be able to process large volumes of transactions as quickly as possible as any delay could have a detrimental effect on the volume of sales made.

OLTP systems typically require fast processing of create, update and occasionally delete operations, with a much lower requirement for read operations. A normalised form of data is generally used, stored in relational databases (RDBMS), because data integrity needs to be guaranteed.

OLAP

Online analytical processing is a means of providing easy access to transaction data for reporting and analysis purposes. Often this data is highly summarised. OLAP systems underpin the production of management information (MI) reports on a regular basis and provide business intelligence (BI) feeds that are visualised with a dashboard. Other uses include applying data analytics which uses mathematical and statistical techniques to look for patterns and trends in the operational data, which has been accumulated over time.

OLAP systems get their data from various OLTP system sources, both internal and external, via a periodic process known as ETL (Extract, Transform, Load). Ideally the same Master Data and Reference Data definitions would be used in both environments.

Copyright © BCS 2020

OLAP systems are ‘read-only’, based on historic data, so the risks surrounding update anomalies are eliminated. For this reason, OLAP systems are optimised for read activity, which means for example using non-key indexes, aggregation and de-normalisation as ways to speed up the performance of specific queries and make them easy for end users to manipulate.

5.2. Explain the rationale for the architecture and design of data

Data should be treated as one of the fundamental strategic resources available to any enterprise. As such there is a need to plan and control data assets effectively, in the same way that people and money are managed, for example.

There are a number of themes, therefore, that justify the creation of a data architecture and data designs that span across the whole enterprise. Some of these are listed in the syllabus below.

5.2.1 Alignment of the application view of data with the corporate data architecture

Corporate data architecture:

- Forms part of the overall enterprise architecture of the organisation
- Contains high level entities in a corporate data model
- Aggregates and shares requirements, models and designs for organisational data
- Promotes consistency of the use of data across the organisation or enterprise

Each application will have a view of data which is narrowly focussed on the problems it needs to solve. The danger here is that a local interpretation may be at odds with the bigger picture, so it is beneficial to have a corporate view with which to validate the application view. The application may need to extend the corporate view, but it shouldn’t contradict it.

5.2.2 Discovery of data requirements

The need to create a data architecture leads to the discovery of data requirements that are consistent with the enterprise strategy, and the influence of external factors.

Data requirements may be discovered for example by:

- Using bottom-up techniques such as document analysis
- Examining existing information systems
- Systematic examination of business needs and issues, including the implementation of strategy
- Using top-down conceptual analysis and design of the business domain
- Elaborating of the design of information system components to discover their individual data requirements

5.2.3 Definition and communication of data design

Discussions centred on the definition of data entities and attributes contribute to useful communications amongst stakeholders about the actual and required business use of the data resource.

5.2.4 Compliance with regulations and legislation

Increasingly, the use that enterprises can make of the data and information they hold is controlled by regulation and legislation. Compliance has become one of the major reasons for engaging in the architecture and design of data.

For example, data designs will need to take into account:

- Data may only be kept for a legitimate business use
- Data that is not used in a business process needs to have a justification to be retained
- Data must be kept if it's needed for regulatory compliance
- Auditing data utilisation is a legitimate reason for keeping additional data

Corporate data architectures should maintain a list of relevant regulations and legislation and the design decisions that have been made to comply with them.

5.2.5 One version of the truth

If data and information is to be truly useful to the enterprise, there is a general requirement that there should be 'one version of the truth'. It is surprisingly difficult in many organisations to get clear answers to even very simple questions such as 'how many widgets did we sell last week?' The principle to follow is that although there will always be many sources of data in an enterprise, it should be possible to reconcile these into a single coherent view. So queries on the same facts from anywhere in the enterprise should yield results based on the same underlying truth.

Data architecture and design can assist by providing clear data definitions that are used across the enterprise. This limits the possibility of misunderstanding which can lead to the inconsistent use of data or metadata in different parts of the enterprise. Data that is business-critical and shared across an organisation is the most important target for consistency and is known as master data (see 5.1.5 above).

5.2.6. Support for Data Analytics

Data analytics is the discipline of analysing raw data to draw conclusions and provide information and intelligence that is useful to the business. The techniques and processes involved in data analytics may be automated to deliver output in real-time. It is associated with visualisation techniques such as the use of dashboards to facilitate and speed up the consumption and use of data by recipients (see 5.1.6 above).

Data architecture and design can support data analytics by clearly defining corporate metadata so that data can be correctly identified and interpreted. This provides the basis upon which data marts can be constructed and their contents analysed.

5.2.7 Support for Data Management and Data Governance

Data management is a function and a discipline that brings into the enterprise best practices for managing data throughout its entire lifecycle within the organisation. It includes planning for the acquiring, validating, storing, protecting, and processing of data to ensure the accessibility, reliability, and timeliness of the data for its users. It does this within the rules established by data governance.

Data governance is the rules framework that controls the use of the data assets, based on internal data standards and policies that control data usage. Data governance will take into account relevant legislation and seek to leverage the value of data while controlling the risks. Effective data governance ensures that data is consistent and trustworthy and is not misused.

The artefacts produced by data architecture and design support the goals of both data management and data governance.

5.3 Describe the modelling data and information

Candidates must be able to recognise and explain the main elements of a data model.

5.3.1 The main elements of a data model

5.3.1.1 Entities/entity types

An entity is something of significance to the business, that has a distinct identity, about which data and information is required to support business operations. Similar entities are grouped together as entity types, which define their common characteristics. An example of an entity type is ‘Customer’. An example of an entity is Customer ‘Smith’.

5.3.1.2 Relationship/relationship degree

A relationship expresses a significant fact that ties entities together in the business domain. Relationships are important in data modelling as they reveal the means by which navigation can occur between entities in the domain. Navigation will ultimately allow information to be derived from the data. The existence of a relationship also serves to clarify the business meaning of the entities involved, and the role they play with respect to each other.

An example of a relationship is the phrase “Customer places Order”.

The relationship *degree* expresses for **each** entity of one type how many entities of the related type can be associated minimum to maximum over time. Note the minimum could be zero, which would make the relationship optional.

An example of relationship degree is “Each Customer places one to many Sales Orders”.

5.3.1.3 Attribute

Any property that serves to qualify, identify, classify, quantify or express the state of an entity. An example of an attribute is “Customer Name”. Attributes are the data items referred to in another part of this syllabus.

5.3.1.4 Metadata/Data Dictionary

Information that describes the structure, content or use of data is called metadata. Metadata can be created about entity types, attributes and relationships. In practice

there are many different categories of metadata, which serve different purposes in the enterprise, for data management and for governance.

An example of metadata would be defining the data type of the “Customer Name” attribute of the ‘Customer’ entity as alphanumeric with a maximum of 50 characters, and marking it as mandatory, so it must always have a value.

A Data Dictionary is a database in which metadata is defined, manipulated and stored.

5.3.1.5/6. Data Models using ERD and UML

The vast majority of data models are built using one of two standards: UML Class Diagrams or Entity-Relationships Diagrams (ERD; aka ‘crow’s foot’). Essentially the models do the same modelling job, but they use distinct notations. Candidates are expected to be able to recognise and interpret these modelling notations.

In the following table there is a mapping of the UML Class Diagram terminology to the ERD terminology:

ERD	**UML**
Entity Type	Domain Class
Entity	Object
Attribute	Attribute
Relationship	Association
Relationship Degree	Multiplicity

In the graphic of Fig. 11 is an example of an ERD Model. This diagram uses Information Engineering notation (IE) [26], but note there are many variations on this in use. It is not customary to show attributes in this type of diagram as they could be numerous (identifying attributes are sometimes shown however).

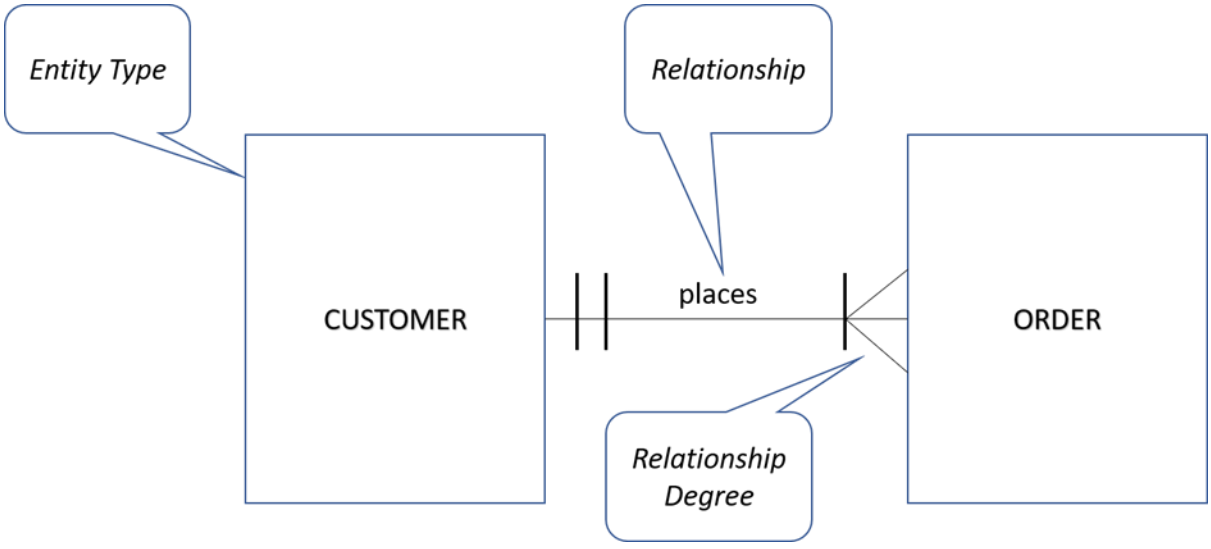


Figure 11: ERD notation (IE).

In the graphic of Fig. 12 is the same example as a UML Class Diagram. Again, attributes are usually not shown.

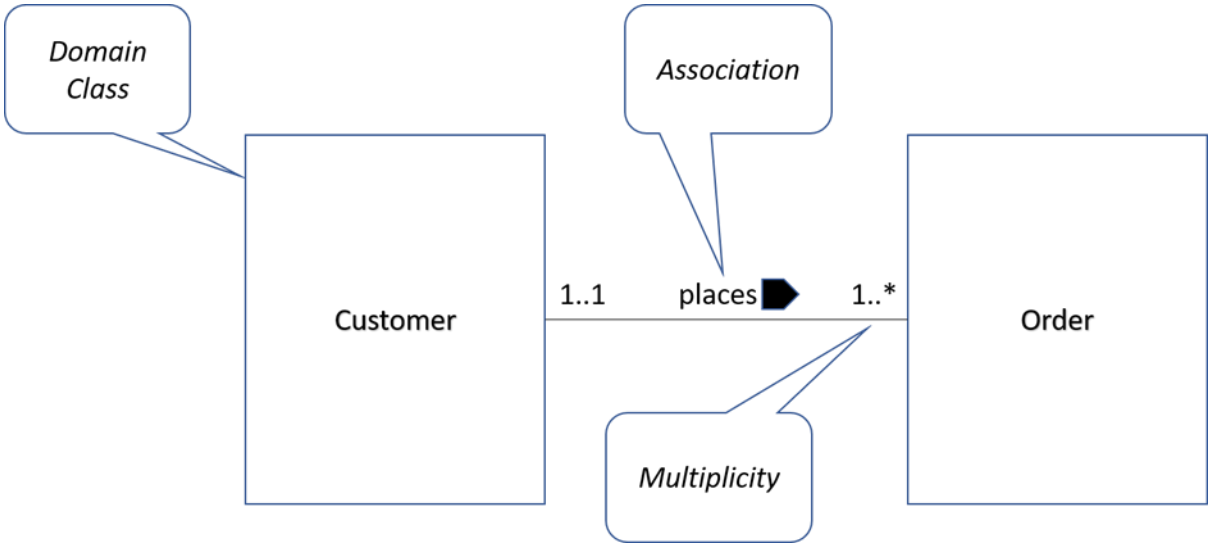


Figure 12: UML Domain Class Model notation

Both of these models represent the business data in the same way, i.e. in the example:

"Each Customer places one to many Orders. Each Order is placed by one and only one Customer"

5.3.2 *Relational modelling (normalisation) as a data analysis and design technique*

Relational analysis is focussed on the identity, structure and meaning of the data items found in the data sources being investigated. It is a bottom-up modelling technique in that it starts from the inputs and outputs of the information system, and 'reverse engineers' these to their underlying data structures.

Firstly, the analysis seeks to identify what entities exist in the source(s) and those attributes whose values make each entity unique. This is the entity's Primary Key (PK), which may have to be an anonymous system code to guarantee uniqueness.

Next, normalisation seeks to organise the non-key attributes of entities in the sources. Attributes are said to be normalised if:

- The value of each non-key attribute is determined only by the value of the PK
- Each non-key attribute represents a single atomic fact about the entity
- Each non-key attribute has only a single value for each value of the PK (or is permitted to be null)

Relational analysis obliges the analyst to really understand the business significance of attributes and what precisely determines their value.

Normalisation in design is a technique for organising data items into simple, stable data structures, based on relational theory. Normalisation is often used with relational database management systems (RDBMSs) where the data items are stored as columns in one or more tables which are related to each other using keys. It may also be used with entity relationship modelling where the data items are

organised as attributes of one or more entity types. Some of the aims of normalisation in this regard are to minimise redundancy or duplication of data and to promote consistency by preventing update anomalies.

5.3.3 Dimensional modelling for OLAP applications

Dimensional modelling includes a set of methods, techniques and concepts for use in data warehouse design.

Dimensional modelling is based on the concepts of facts (measures), dimensions (context) and attributes (characteristics). Facts are typically numeric values that can be aggregated using various mathematical functions. Dimensions and attributes give context to the facts in terms that are useful in some way to the business.

For example, 'number of passengers' could be a numerical fact in a public transit system. Time and location could be contextual dimensions that are of interest to the business. Date and time could be attributes of the time dimension, while transit station and zone could be attributes of the location dimension. This model would allow users to run reports on the number of passengers at a particular location on a specified date, for example.

It is good practice to base dimensional models on knowledge of events in transaction systems (OLTP) and on a single Enterprise Logical Data Model that underpins both OLTP and OLAP models.

5.3.4. Describe the differences between conceptual, logical and physical data models and explain the need for these different views of data

Conceptual, logical and physical models represent different levels of idealisation and realisation of data and information and exist on an abstraction continuum.

The Conceptual Data Model (CDM) is the most abstract form of data model. It is the level generally associated with enterprise architecture, identifies high-level information requirements and often forms the basis of Subject Areas as a way of organising top-down views of the enterprise's data.

Logical models are the detailed design of the enterprise's data structures, seen from the perspective of the enterprise itself. This is captured in an Enterprise Logical Data Model (ELDM/LDM), which is referred to by both OLTP and OLAP. It should be reconcilable with the CDM. Each development project should create its own LDM but should make sure it is compatible with the centralised Enterprise Logical Data Model.

Physical models are an interpretation of the LDM for the technology that will be used to process, store or transmit data; for example an RDBMS like Oracle or SQL Server. These models are frequently referred to as *schemas*.

The application developer is most likely to have indirect contact with the ELDM and direct contact with the physical technology models.

5.3.4 Modelling data at rest and in motion

Modelling data at rest means the physical modelling referred to above and in particular the specification of the data structures for storage.

Data in motion means modelling the specification of data structures for transmission. This includes large-scale transmission such as serialisation and streaming and also small-scale transmission of messages, parameters and commands. Data flow diagrams are one of the modelling techniques used here.

5.4 Explain the relationships between applications software and the information system

This syllabus recognises application software as one of the main technologies that are used in support of a modern information system. This implies a close symbiosis between software and data.

5.4.1 Recognise the four basic operations performed on data (CRUD)

In order for an enterprise to make use of a digital information system it is necessary to be able to create, read, update, and delete data and information. These operations are performed by applications software on some form of persistent data storage.

Hence applications software is used to capture and manipulate all the parts of the Information System mentioned so far, including master data, reference data, transaction data, data in OLTP and in OLAP. Applications are used to capture data, turn it into information and make this available to decision makers.

5.4.2 Recognise the following data storage technologies used with digital solutions and the circumstances within which they might be used

5.4.2.1 Relational technology and the use of SQL

The dominant form of database technology has been relational for many years. RDBMS is the abbreviation used for ‘relational database management system’, which stores data in normalised tables which are linked to each other using primary and foreign key relationships.

This type of technology still dominates in the OLTP environment. One of its main advantages is the integrity of data, which is stored, in theory in only one place, although it may be replicated to many places. This means updating a single fact in a single place is possible and have that update visible everywhere. This promotes data integrity.

SQL is a query language that is used to access and manipulate structured data in a relational database. This way of structuring and storing data is reflected in the syntax of SQL, which is defined in the ISO/IEC 9075 [25] internationally accepted standard.

All application developers at some stage of their careers need to know SQL. SQL statements can be written into all modern programming languages and is the standard way of manipulating structured, normalised data.

5.4.2.2 Non-relational technology (Big data and No-SQL)
Modern demands for processing huge volumes of unstructured data led to what is now called Big Data. The outstanding example of this need was/is the Google search engine, and Google were pioneers in this field.

'Big Data' is a rather ill-defined, broad term to cover a variety of data that is read-only large in volume, mostly unstructured, often being created and/or transmitted at high velocity and usually being generated in multiple locations by diverse systems. Managing this sort of data, quickly stretches the limits of relational technologies, so new database paradigms have had to be invented. A major limitation of relational technology is that it can only run a single computing device, and it performs very poorly in read-intensive environments.

No-SQL (which stands for 'not-only SQL') databases store data in a variety of different ways, quite different from the relational model of the RDBMS. There are various architectures currently being used, but they all rely on a non-relational record structure, de-normalised data and physical pointer systems to link records up. Records do not have to have similar content or even similar structures [39]. The query performance of such systems is extraordinarily fast especially when combined with technologies like Hadoop, which can distribute processing across thousands of physical devices.

5.4.2.3 Data warehouse
A data warehouse is a central repository optimised for analytics. It collects data from many different sources within an organisation and often combines data from different systems to gain a broader or higher-level insight into business operations. The process of acquiring the data is called extract, transform and load (ETL) and data is usually transformed into a non-normalised form which is optimised for reporting and analysis as well as being combined with data from other sources.

5.4.2.4 Data lake
A data lake is a single store of all enterprise data including raw copies of source system data as well as transformed data used for tasks such as reporting, visualisation, advanced analytics and machine learning. The storage of original, source data means data lakes are larger than data warehouses, but the advantage is that changing requirements can be easily accommodated in a data lake whereas data warehouses are built with specific requirements in mind. Additional metadata is stored for data entering a data lake such as logs that record the timestamp and origin of data items.

5.4.2.5 Data mart/cube
A cube is a structured multidimensional dataset, produced as a subset of a data warehouse, via a second ETL process. It is focussed on a specific business area, containing summarised data for analysis and consumption by business users.

5.4.2.6 Blockchain

A blockchain is an immutable time-stamped series record (ledger) of data that is distributed and managed by multiple computers. The data can be about anything and the main benefit of blockchain is that it provides a secure, anonymous, non-repudiable record of transactions of any type. As such it can be used in place of signed contracts between parties.

5.4.3 Recognise the following data transmission standards used with digital solutions and the circumstances within which they might be used

5.4.3.1. XML

An abbreviation for extensible mark-up language, it is a meta-language that is used for the definition, transmission and validation of data. The structure is similar to HTML and consists of elements, with start and end tags, and attributes but whereas HTML elements are pre-defined, XML is completely customisable to suit the data sets that it will be used for. The main types of XML file are:

- XML (.xml) which contains the data set
- Schema (.xsd) which contains the definition and constraints for a data set
- Transformation stylesheet (.xslt) which contains instructions for transforming XML data into a different format which could be XML or not. Note: XSL stands for extensible stylesheet language, originally designed as a generalised presentation language.

5.4.3.2 JSON

Java Script Object Notation (JSON) is an open standard file and message format that uses human-readable text to store and transmit data objects consisting of attribute-value pairs and multi-value array data types. Converting a data object to a transmissible form is called serialisation and JSON may be used to transmit streams of serialised objects between applications. It is also used to transmit messages in the form of service requests and commands as part of a service-oriented architecture (SOA).

5.4.3.3 EDI

Electronic Data Interchange (EDI) is the electronic exchange of business information using a standardised format, allowing business partners to send and receive documents electronically rather than with paper. A large number of document types have been defined. Those for global trade were defined and ratified by the UN under the collective heading EDIFACT (Electronic Data Interchange For Administration, Commerce and Transport). Other document sets have been defined by standards organisations and industry bodies.

5.4.3.4 YAML

YAML is a human-readable data-serialisation language. It is commonly used for configuration files and in applications where data is being stored or transmitted. YAML targets many of the same communications applications as XML and JSON but has a minimal syntax which improves communication speed.

5.5 Explain some of the issues surrounding concurrency of data in multi-user environments, and identify the main strategies for dealing with this

Concurrency, in the context of a software application, is the handling of multiple users (or processes or threads) attempting to access the same data (or code) at the same time. This problem is handled by application developers through the idea of *transactions*.

Transactions group together multiple steps in a software process and ensure that either they are all completed successfully or that if any one of them fails for any reason, that they are all reversed so that the situation is the same as it was before the transaction started.

5.5.1 ACID transactions

The basic purpose of transaction controls is to try to ensure that any database transaction conforms to the ACID test as defined below by the IBM Knowledge centre [42]:

Atomicity
All changes to data are performed as if they are a single operation. That is, all the changes are performed, or none of them are. For example, in an application that transfers funds from one account to another, the atomicity property ensures that, if a debit is made successfully from one account, the corresponding credit is made to the other account.

Consistency
Data is in a consistent state when a transaction starts and when it ends.
For example, in an application that transfers funds from one account to another, the consistency property ensures that the total value of funds summed across both accounts is the same at the start and end of each transaction.

Isolation
The intermediate state of a transaction is invisible to other transactions. As a result, transactions that run concurrently appear to be serialised. For example, in an application that transfers funds from one account to another, the isolation property ensures that another transaction sees the transferred funds in one account or the other, but not in both, nor in neither.

Durability
After a transaction successfully completes, changes to data persist and are not undone, even in the event of a system failure. For example, in an application that transfers funds from one account to another, the durability property ensures that the changes made to each account will not be reversed by a subsequent technology failure.

5.5.2 Transaction controls: commit and rollback

Transactions have a transaction start boundary that the developer can establish. Once all the steps in the transaction have been confirmed to be successful, a *commit* instruction is issued and the transaction is completed. If any error is identified with any of the steps however, then a *rollback* instruction is issued and all

the steps completed so far are reversed, so that the data is in the state it was just before the transaction started.

SQL and other languages, as well as APIs, have commands that start, commit and rollback transactions. These instructions must be included in the applications software.

5.5.3 Need for timestamps and locking strategies

The way to handle concurrency and contention issues typically makes us of:

Timestamps

S.K. Singh [43] defines time stamping as a method of concurrency control in which each transaction is assigned a transaction timestamp. A transaction timestamp is a sequential number, often based on the system clock. The transactions are managed so that they appear to run in a timestamp order. This timestamp indicates the order in which the transaction occured, relative to the other transactions. Each data object has two timestamps representing the last read and the last write. By comparing these timestamps, it is possible to tell if another user, process or application has changed the data. This provides a system where the data does not have to be locked during the transaction, but the application will need sophisticated logic to control concurrency and contention issues.

Locking

Locking is the most common mechanism for dealing with data concurrency issues. For example, if a record is accessed for update by a user, it can be 'locked' so that other users cannot access it for update to. DBMS can be configured to provide locking mechanisms on tables, records and other resources. Locking information can be sent back to the application that makes requests for the data, and this information may be passed on to the users.

One could refer to these as physical locks, as they are provided by the physical implementation of the data (DBMS). However, it is usually more useful and better practice for applications to use logical locks which place locks on logical business entities. A logical lock is a lock held outside the database structures that represent that entity. It is implemented in software by the application or better still as a data service.

6 Quality assurance and quality control (10%)

This section of the syllabus covers several techniques and practices which have proved their effectiveness in the areas of quality assurance and quality control, as this relates to software development.

For the purpose of this syllabus 'quality' is defined as *"The totality of features and characteristics of a product or service that bear on its ability to satisfy stated or implied needs"* [41].

The famous phrase ‘quality is free’ [27] was never truer than in the context of software. Lack of quality can have severe financial and other consequences. However, in a modern context we understand that quality is not an absolute, and therefore the development team and their sponsors have to judge what is an acceptable level of quality, a level that is good enough to get the business problem solved successfully.

6.1 Distinguish between quality assurance and quality control

Candidates are expected to be able to distinguish between practices and techniques that are directed at Quality Assurance (QA) and those that are directed at Quality Control (QC).

Quality Assurance (QA) covers the measures taken to ensure that the product in focus, and the methods of producing it, will be of sufficient quality. Quality Control (QC) covers the measures taken to check the extent to which the product, once produced, does in fact meet the quality criteria set.

A lot of the best practices in software QA/QC today are derived from the Japanese quality schools of thought that revolutionised manufacturing in the 60s and 70s, based on the ideas of W. Edwards Deming. Deming’s overriding idea was that everything that contributes to the production process must be scrutinised from a quality perspective proactively looking for possible sources of defects. Quality is not, according to Deming, ‘people in white coats’ standing at the end of a production line; it is making sure that ‘quality is certain’. This essential philosophy has been built into Agile practices, for example into Scrum.

6.2 Recognise the following quality-oriented activities

Candidates will be able to recognise and explain the following quality-oriented activities and practices.

6.2.1. Inspection and adaptation

Inspection and adaptation are two of the three pillars of the Scrum methodology, the third pillar being transparency. These are relevant for both QA and QC as they are designed to support the underlying Agile principles of iterative, value-based incremental delivery by frequently gathering customer feedback and embracing change resulting in enhanced software quality, and improved risk management.

Transparency means presenting raw facts without modification to everyone in the team, including the customer. Everyone trusts each other and has the confidence to share good and bad news. Everyone is working towards the same objective, so no one has any interest in keeping things hidden.

Inspection means examination by every team member and the customer so that things can be continuously improved. Everything is open to inspection including the product, processes, skills, practices, and continuous improvements. For example, at the end of each Sprint, the team shows the product to the customer with complete

transparency and gets honest feedback. If this results in changes to the requirements, the team adapts, using this as an opportunity for further customer collaboration to improve the product.

Adaptation is the process of continuous improvement based on the transparency and inspection. Any adaptations must be linked to an Agile objective, product quality, early delivery, business value etc.

6.2.1.1 Product Review

The product review is also known as the sprint review and is a session at the end of a sprint where the team and other stakeholders (users, other teams, managers, investors, sponsors, customers, etc.) reviews the work that has been done and not done and the product is demonstrated and feedback obtained to assist with decisions about how to proceed.

The product review is an opportunity for inspection and adaptation including demonstrating the product and reviewing the product backlog.

6.2.1.2 Retrospective and continuous improvement

The sprint retrospective occurs after the product or sprint review and prior to the next sprint planning event. The purpose of the sprint retrospective is to:

- Inspect how the last sprint went with regards to people, relationships, process, and tools.
- Identify the major items that went well.
- Identify any potential improvements.
- Create a plan for implementing some improvements in the next sprint.

By the end of the sprint retrospective, the scrum team should have identified improvements that it will implement in the next Sprint. Implementing these improvements in the next sprint is the adaptation to the inspection of the scrum team itself. Although improvements may be implemented at any time, the sprint retrospective provides a formal opportunity to focus on inspection and adaptation.

6.2.2. Refactoring

Refactoring is a disciplined technique for restructuring an existing body of code, altering its internal structure without changing its external behaviour [28].

The aim of refactoring is to improve the quality of the software. A key aspect of this is to make code easy to read, understand and maintain. Another aspect of this is adherence to technical standards based on the criteria mentioned in Sections 4 and 7 of this syllabus.

Code that has been refactored is known as clean code. Code that has not been refactored accrues ‘technical debt’ (see Section 7) because extra work will eventually be required to fix it. This could be in maintaining the code itself because badly structured code is harder to maintain, but poor code may have a big impact on testing and quality control. One reason for this is that poor code is more likely to contain errors because they are harder to find. Another related reason is that more

tests are required to provide an acceptable level of test coverage as poorly structured code usually contains duplication and other forms of redundancy.

Refactoring is part of the test-first approach in test-driven development (TDD) which consists of a cycle of 'red-green-refactor', where red indicates a failed test and green indicates a passed test. Following green the code should be inspected to see if it needs to be refactored.

6.2.3. Adoption of a recognised SDLC

Adopting a recognised SDLC is key to quality assurance (QA) as it ensures that processes are followed methodically and systematically and ensures that appropriate QA/QC activities can take place, including reviews and testing.

In any software development lifecycle model, there are several characteristics of good approaches to testing:

- For every development activity, there is a corresponding test activity.
- Each test level has test objectives specific to that level.
- Test analysis and design for a given test level begin during the corresponding development activity.
- Testers participate in discussions to define and refine requirements and design, and are involved in reviewing work products (e.g., requirements, design, user stories, etc.) as soon as drafts are available.

No matter which software development lifecycle model is chosen, test activities should start in the early stages of the lifecycle, adhering to the Agile principle of 'test early, fail fast'.

6.2.4. Test planning and execution

Testing is one of the main QA/QC activities that will take place within a software development initiative. Some of the fundamentals of testing are elaborated in 6.3.

Testing activities can be split across the SDLC into static testing and dynamic testing.

6.2.4.1. Static testing

Static testing is testing activity performed on requirements documents before the code is run or even built. This is a QA activity, whose purpose is to find defects early, when it is cheap and quick to fix them. For example, a tester will scrutinise a Use Case to determine how to test it and turn it into a Test Case.

Also included in static testing is the planning for testing and the creation of test cases, test scripts, test conditions, test data and acceptance criteria.

Static testing of code that has been built can be automated and integrated within an integrated development environment (IDE). Quality reports can be produced.

6.2.4.2. Dynamic testing

Dynamic testing is testing that involves the execution of a test item under controlled conditions and recording the results. This is a QC activity, based on the plans created as part of static testing. The ultimate purpose of dynamic testing is to determine whether the software does in fact do what it should do in the way it should do it.

Unit, integration, system and acceptance tests are all types of dynamic testing.

6.2.5. Adoption of best practice in architecture and software development

Many of the best practices that are included in this syllabus help to improve and maintain software quality. Examples include:

- Use of recognised patterns for the software architecture and design.
- Develop iteratively so that the customer can provide feedback frequently and the software can be adapted accordingly.
- Develop software to meet requirements and do not introduce any additional features.
- Use component architecture to break software into units that can be produced, tested and maintained independently.
- Model software to aid understanding by all team members.
- Verify quality at every stage by testing.
- Control change so that it does not introduce errors by thorough testing.

6.3 Describe the fundamentals of software testing

6.3.1. Explain why testing is necessary

Rigorous testing of components and systems, and their associated documentation, can help reduce the risk of failures occurring during operation. When defects are detected, and subsequently fixed, this contributes to the quality of the components or systems. In addition, software testing may also be required to meet contractual or legal requirements or industry-specific standards.

On the other hand, it must be recognised that the extent of testing activity must be proportionate to the risks of failure and their consequences. In common with other aspects of quality, testing is not an absolute.

Unfortunately, throughout the history of computing, and even today, it is quite common for software and systems to be delivered into operation and, due to the presence of defects, to subsequently cause failures or otherwise not meet the stakeholders' needs. However, using appropriate test techniques can reduce the frequency of such problematic deliveries, when those techniques are applied with the appropriate level of test expertise, in the appropriate test levels, and at the appropriate points in the software development lifecycle.

The approach to testing is also important. Examples include:

- Having testers involved in requirements reviews or user story refinement could detect defects in these work products. The identification and removal of

requirements defects reduces the risk of incorrect or untestable features being developed.
- Having testers work closely with system designers while the system is being designed can increase each party's understanding of the design and how to test it. This increased understanding can reduce the risk of fundamental design defects and enable tests to be identified at an early stage.
- Having testers work closely with developers while the code is under development can increase each party's understanding of the code and how to test it. This increased understanding can reduce the risk of defects within the code and the tests.
- Having testers verify and validate the software prior to release can detect failures that might otherwise have been missed and supports the process of removing defects that would cause failures in operation. This increases the likelihood that the software will meet stakeholder needs and satisfy requirements.

6.3.2. Distinguish between error, defect and failure

A person can make an error (mistake), which can lead to the introduction of a defect (fault or bug) in the software code or in some other related work product. An error that leads to the introduction of a defect in one work product can trigger an error that leads to the introduction of a defect in a related work product. For example, a requirements elicitation error can lead to a requirements defect, which then results in a programming error that leads to a defect in the code.

If a defect in the code is executed, this may cause a failure, but not necessarily in all circumstances. For example, some defects require very specific inputs or preconditions to trigger a failure, which may occur rarely or never.

Errors may occur for many reasons, such as:
- Time pressure
- Human fallibility
- Inexperienced or insufficiently skilled project participants
- Miscommunication between project participants, including miscommunication about requirements and design
- Complexity of the code, design, architecture, the underlying problem to be solved, and/or the technologies used
- Misunderstandings about intra-system and inter-system interfaces, especially when such intra-system and inter-system interactions are large in number
- New, unfamiliar technologies

In addition to failures caused due to defects in the code, failures can also be caused by environmental conditions. For example, radiation, electromagnetic fields, and pollution can cause defects in firmware or influence the execution of software by changing hardware conditions.

Not all unexpected test results are failures. False positives may occur due to errors in the way tests were executed, or due to defects in the test data, the test

environment, or other testware, or for other reasons. The inverse situation can also occur, where similar errors or defects lead to false negatives. False negatives are tests that do not detect defects that they should have detected; false positives are reported as defects, but aren't actually defects.

6.3.3. Define the seven principles of testing

According to the ISTQB [30] the following are all useful test principles. Recognition of these principles in test planning should lead to better testing outcomes.

6.3.3.1 Testing shows the presence of defects, not their absence. Testing does not guarantee the software will not fail.
6.3.3.2 Exhaustive testing is impossible. It is impossible to test every possible combination of moving parts.
6.3.3.3 Early testing saves time and money. Test early, fail fast. Static testing can save time and money.
6.3.3.4 Defects cluster together. It is usually a relatively small number of features that cause the majority of defects. Also known as the Pareto Principle, where 80% of the defects are caused by 20% of the code.
6.3.3.5 Beware of the pesticide paradox. If same tests are run time and again the same results will be produced. Useful tests must vary the tests performed.
6.3.3.6 Testing is context dependent. The testing that should be carried out depends on the context in which the software will be deployed.
6.3.3.7 Absence-of-errors is a fallacy. The software could pass all its tests, but that won't help if the requirements were 'wrong' in the first place.

6.3.4. Explain the concepts of test condition, test case and test basis

A *test condition* is *"an item or event of a component or system that could be verified by one or more test cases, e.g. a function, transaction, feature, …" [30].* A test condition is some aspect of the software product that should be tested. This might be functional or non-functional.

In a use case-based approach, test conditions can be based on steps in each use case. In a user story-based approach, test conditions can be based on acceptance criteria and/or behaviour scenarios.

A *test case* is *"a set of preconditions, inputs, actions (where applicable), expected results and postconditions, developed based on test conditions" [30].* It is usually very inefficient to devise tests for individual test conditions, so these are generally grouped into test cases, each of which follows a scenario.

In a use case-based approach test cases can be based on use case scenarios, the main flow and alternates. In a user story-based approach, test cases are devised taking into account acceptance criteria and/or behaviour scenarios. However other scenarios might be tested too, based on business process workflows or data lifecycles, for example.

A *test basis* is defined as *"all documents from which the requirements of a component or system can be inferred. The documentation on which the test cases are based." [30]*

In a use case-based approach the test basis is therefore provided by the Use Case Model, which includes the Use Case Diagram, with supporting descriptions. In a user story-based approach, the test basis is the sprint backlog and story documents.

6.4 Identify a range of common testing practices and processes

6.4.1. Identify the test levels used to organise testing activities within a typical software development initiative

6.4.1.1 Explain the Agile Test Pyramid model

Software systems are tested at different levels and at different volumes. This is represented as a pyramid with many unit tests at the base, with gradually fewer integration, system, and acceptance tests towards the top. The benefit of this approach is to ensure that every part of the software works as specified, all the way from the smallest block or unit of code through groups of units working together to the whole system providing the end-user functionality it was designed to provide. Since there are many units and only one system, the number of tests naturally decreases as the code units are built up into a complete product.

The test pyramid emphasises the fact that development teams should aim for a large number of tests at the lower levels, at the bottom of the pyramid, but that, as development moves to the upper levels, the number of tests should dramatically decrease.

There are several versions of the test pyramid but nearly all have unit tests at the base of the pyramid with more tests than at any other level. At the top of the pyramid are tests for the business functionality of the application. These are sometimes described as user interface (UI) tests but are at the same level as acceptance tests. In between code-level and business-level tests are tests of component parts of the application, which could be systems or subsystems. These are aligned with integration and system testing but because the functionality being tested is a subset of the overall functionality and is normally organised into application services, tests in this layer are also known as service tests.

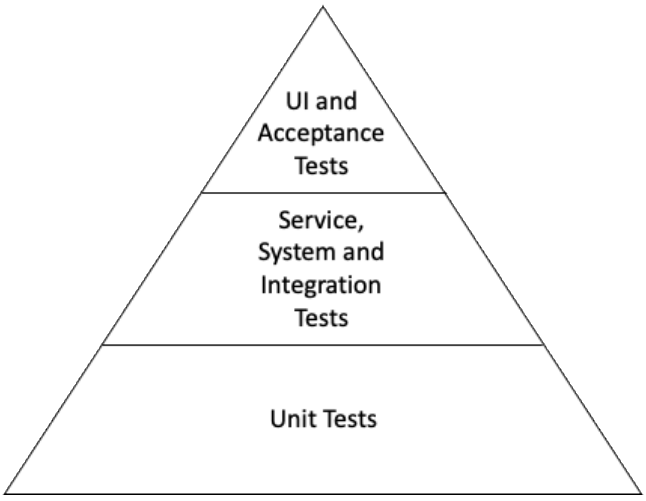


Figure 12: Agile Test Pyramid

Unit and integration level tests are automated and are created using API-based tools. Automated acceptance tests are created using GUI-based tools. The test pyramid concept is consistent with the 'early testing' principle which aims to eliminate defects as early as possible in the development lifecycle to save re-work and wasted resources.

The test pyramid is consistent with test-first development including test-driven development (TDD) and behaviour-driven development (BDD).

6.4.1.2 Component/unit testing

Unit testing is performed on the smallest piece of testable software in the application. Depending on the language, this may be a class, method or function.

A component may be the same as a unit but may also refer to a larger subsystem of an application.

The aim or unit and component testing is to isolate the system under test (SUT) from the remainder of the code and determine whether its behaviour matches the requirements under all conditions and with a range of inputs. Each unit or component is tested separately before being integrated with other parts of the system. This approach helps to isolate any faults that do occur, by eliminating units and components already proven to work from the list of suspects. Unit and component tests become part of regression testing.

Unit tests are usually written by developers using a unit test framework such as JUnit or NUnit. Subsystem component tests are usually written by testers and form part of integration testing.

Both code units and subsystems have dependencies on other code and so these need to be accounted for and eliminated from tests. A code unit that contributes to the work of a system under test (SUT) is called a depended-on collaborator (DOC). DOCs are usually replaced with a mock-up or stub that has a fixed or predictable behaviour so that the SUT can be tested working with its DOCs but without any possibility that errors are occurring in them. Subsystem component tests may depend on other subsystems or external systems including other business systems and databases. Internal subsystems can usually be mocked up or simulated but the usual practice with external systems is to have a special test version that can be made more predictable to strengthen the isolation of the test being performed.

Multiple test cases are normally required, even for simple classes or subsystems, both because there are multiple functions being tested and because for each function there is a range of test conditions, each of which requires separate test code. It is bad practice to test multiple conditions in a single test.

6.4.1.3 Integration testing

Integration testing combines individual code units and tests them working as a group. Integration test cases try to expose faults in the interaction between integrated units and subsystems. Integration testing starts from the assumption that

individual code units work correctly according to their specification and so depends on the quality and effectiveness of unit testing.

6.4.1.4 System testing

System testing is performed on a complete and fully integrated software product. System tests are performed against end-to-end system specifications or requirements.

Usually, the software is only one element of a larger computer-based system and the system under test (SUT) needs to be interfaced with other software/hardware systems to perform its function. This means that test versions of any other systems must be set up with sufficient data available to provide realistic tests. Often a sample of live data is imported and anonymised. Data in all systems needs to be sufficiently representative and varied to cover all the test cases that will be run during system testing.

System tests are written and performed by testers working closely with other members of the development team and using scenarios drawn from the business.

System tests can be automated or manual and are included in the set of regression tests that are run every time new software is added to the build or existing software is modified.

6.4.1.5 Acceptance testing

Acceptance testing is very similar to system testing in that the tests target the whole software system or application and the scenarios are end-to-end rather than just fragments of a task, as in unit and integration testing.

Acceptance testing aims to build confidence with business users and stakeholders which ultimately leads to the product being released to the live environment and being used by business users to perform business processes and tasks.

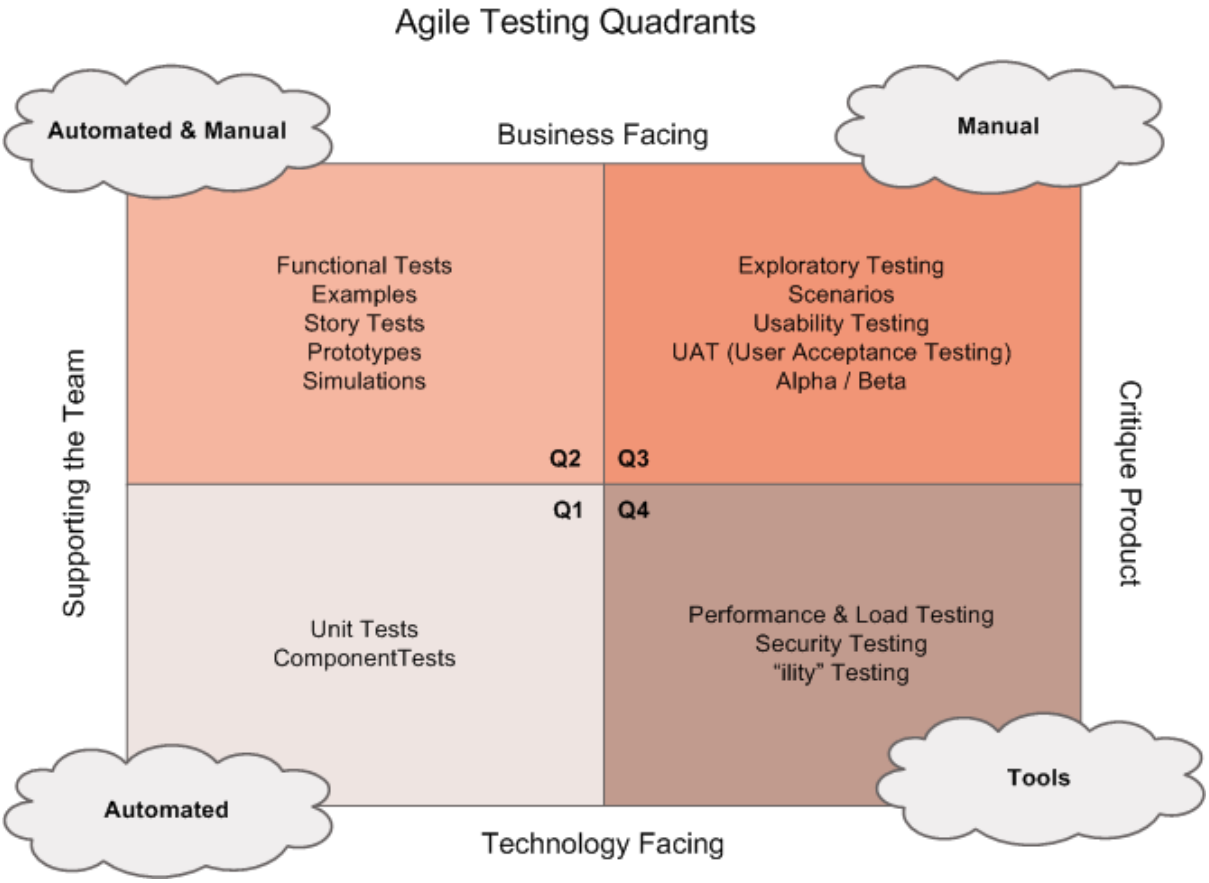
The system under test (SUT) has to appear as if it were live but cannot be allowed to affect live data or interact with external systems. The same approach as with systems testing may be used by providing test versions of external systems and importing anonymised data. In some cases, live data is imported without being anonymised, meaning that users can work on data that is familiar to them. This has additional associated risks but provides a more real experience for the users.

Testers provide scenario-based scripts, but users are encouraged to deviate from scripted test cases to ensure that all aspects of the system are tested. Testers monitor these manual tests to record any actions not so far covered by test cases and add new tests as appropriate.

Users should have received training on the use of any new or improved software before being involved in acceptance testing so that the tests can be run as realistically as possible, otherwise, if development staff have to help users too much and the independence of the testing activity is reduced.

6.4.1.6 Explain testing quadrants and how they align test levels and testing types

Testing quadrants align the test levels with the appropriate test types in the Agile methodology so that all necessary test types and levels are included in the development lifecycle. This model also provides a way to identify and describe test types to stakeholders, including developers, testers, and business representatives.



This Photo by Unknown Author is licensed under CC BY-SA-NC

Figure 13: Agile Testing Quadrants (Redo!)

Referring to Figure 13, tests may be business or technology facing, meaning they are performed by users or members of the development team respectively. Some tests support the work done by the Agile team and confirm software behaviour. Other tests can verify the product. Tests can be fully manual, fully automated, a combination of manual and automated, or manual but supported by tools.

The four quadrants are described as follows:

- Quadrant Q1 is unit level, technology facing, and supports the work done by developers. It contains unit tests that should be automated and included in the continuous integration (CI) process
- Quadrant Q2 is system level, business facing, and confirms product behaviour for users. It contains functional tests, examples, story tests, user experience prototypes, and simulations. These test types check the acceptance criteria and can be manual or automated. They are often created alongside user story development and improve the quality of the stories. They are useful when creating automated regression test suites.

- Quadrant Q3 is system or user acceptance level, business facing, and contains tests that demonstrate the product, using realistic scenarios and data. It contains exploratory testing, scenarios, process flows, usability testing, user acceptance testing, alpha testing, and beta testing. These tests are often manual and performed by users
- Quadrant Q4 is system or operational acceptance level, technology facing, and contains tests that demonstrate the end-to-end operation of the product under realistic conditions. It contains performance, load, stress, and scalability tests, security tests, maintainability, memory management, compatibility and interoperability, data migration, infrastructure, and recovery testing. These tests are often automated.

During any given iteration, tests from any or all quadrants may be required. The testing quadrants describe dynamic testing rather than static, although it may also be used to plan required tests in advance.

6.4.2. Identify the following test types and explain their purpose

The following are common test types which in principal might be used at any level of testing.

6.4.2.1 Functional testing

Functional testing tests functionality or features that have been defined by functional requirements. It is classed as black box testing because the test cases are designed against requirements which are implementation independent and so only inputs and outputs are taken into account.

Functional tests are performed at all levels and test at different levels of granularity. Unit and subsystem component test cases are against the functional specification of packages classes and functions and test API calls directly. System and acceptance test cases are against use cases and user stories where the end-to-end functionality can be tested often using the GUI but may also test APIs as in the case of web services and microservices.

6.4.2.2 Non-functional testing

Non-functional testing tests against non-functional requirements (NFRs). This is also classed as black box testing as the internal workings of any components are not tested, only the external behaviour.

Non-functional testing is usually performed at the system or acceptance levels, although it is useful to decompose some categories of NFR such as performance and resilience to identify bottlenecks and points of failure for example.

Some types of non-functional testing test the configuration as well as the software. These include scalability, which can be effected using virtualisation and resilience where failover components are involved.

Types of non-functional testing include:

- Performance Testing: Testing to determine the performance efficiency of a component or system (ISTQB)
- Load testing: A type of performance testing conducted to evaluate the behaviour of a component or system under varying loads, usually between anticipated conditions of low, typical, and peak usage (ISTQB)
- Stress testing: A type of performance testing conducted to evaluate a system or component at or beyond the limits of its anticipated or specified workloads, or with reduced availability of resources such as access to memory or servers (ISTQB)
- Soak testing: Also known as endurance testing, capacity testing, or longevity testing, this involves testing the system to detect the performance-related issues such as stability and response time by maintaining the designed load on a system and measuring its performance over an extended period. Soak testing is a type of load testing.
- Accessibility testing: Testing to determine the ease by which users with disabilities can use a component or system (ISTQB)
- Volume/capacity testing: Testing to evaluate the capacity of a system (ISTQB)
- Compliance testing: Testing to determine the compliance of the component or system. Also known as conformance testing, regulation testing and standards testing (ISTQB)
- Concurrency testing: Testing to evaluate if a component or system involving concurrency (shared multi-threaded access to code and/or data) behaves as specified (ISTQB)
- Scalability testing: Testing to determine the scalability of the software product (ISTQB)
- Spike testing: Testing to determine the ability of a system to recover from sudden bursts of peak loads and return to a steady state (ISTQB)
- Security testing: Testing to determine the security of the software product.
- Penetration testing: A testing technique aiming to exploit security vulnerabilities (known or unknown) to gain unauthorised access (ISTQB)
- Fuzz testing: A software testing technique used to discover security vulnerabilities by inputting massive amounts of random data, called fuzz, to the component or system (ISTQB)
- Usability testing: Testing to evaluate the degree to which the system can be used by specified users with effectiveness, efficiency and satisfaction in a specified context of use (ISTQB)

Where it is established that NFRs cannot realistically be tested prior to release, the associated specifications should be subject to SLAs signed up to by the service delivery organisation. Common examples of these are [30]:

- Availability: The degree to which a component or system is operational and accessible when required for use (ISTQB)
- Failover: The backup operational mode in which the functions of a system that becomes unavailable are assumed by a secondary system (ISTQB)
- Reliability: The degree to which a component or system performs specified functions under specified conditions for a specified period of time (ISTQB)

- Maintainability: The degree to which a component or system can be modified by the intended maintainers.

6.4.2.3. Black box testing (behavioural)
Black-box test technique: A test technique based on an analysis of the behavioural specification of a component or system (ISTQB). The 'black box' refers to the fact that the testing is only on the external behaviour of the module under test.

6.4.2.4. White box testing (structural)
White-box testing: Testing based on an analysis of the internal structure of the component or system. Also known as clear-box, code-based, glass-box, logic-coverage, logic-driven and structure-based testing (ISTQB)

6.4.2.5 Regression testing
Regression testing: A type of change-related testing to detect whether defects have been introduced or uncovered in unchanged areas of the software (ISTQB)

Regression testing is normally run whenever any change is made to any part of the software. Where automatic builds or continuous integration (CI) are being used, testing is automated to occur during each build of the software. Regression tests include unit, integration and system tests. The set of regression tests increases throughout development as new tests are added, either to test new or existing software. New tests may be added for existing software as a result of the discovery of faults or bugs or simply to strengthen the existing set.

6.4.3. Describe the practice of Test Driven Development TDD

Test-driven development (TDD) is a style of programming involving three tightly interwoven activities: unit testing, coding and code design or refactoring.

It consists of the following activities:
- Write unit and integration tests to match the requirements specification
- Run the tests, which should fail because the program lacks the necessary code
- Write just enough code in the simplest way possible to make the tests pass
- Refactor the code until it conforms to accepted coding standards
- Repeat the process, accumulating an increasing set of unit tests over time

Multiple unit and integration tests are nearly always required for even the smallest requirement.

Requirements specifications must be elaborated into test cases and unit and integration tests must cover all the test cases.

Failure to refactor means that the code is unlikely to conform to coding standards and will require re-work in the future. This postponed re-work is known as technical debt. Poor quality code is harder to maintain, and so technical debt may increase exponentially if refactoring is not performed systematically.

TDD has a number of benefits including:

- Self-documenting code: Tests plus code equal documentation means that the purpose and limits of the software are entirely contained within well-designed and conformant code that is easy to read together with the tests that specify what the code must and must not do. Therefore, there is no need for additional documentation to explain the code.
- Comprehensive test set: There is always a complete set of tests for any part of the software that has been written and these tests form the basis of regression testing that helps to maintain high-quality software throughout the development lifecycle.
- Low technical debt: The process of testing and refactoring limits the amount of future re-work required to maintain the software when adding functionality of fixing faults.

6.4.4. Describe the practice of Behaviour-driven development BDD

BDD is a three-step, iterative process:

- Take a requirement to change the software in the form of a user story or use case and establish the acceptance criteria or expected behaviour in the form of scenarios or test cases
- Document the scenarios or test cases in a way that can be automated and validate them against the requirements
- Implement the behaviour described by each documented scenario, usually by entering a TDD cycle

Each change should be as small as possible and the development process should be iterating rapidly, moving back up a level, either to the requirement or to consult the business, each time more information is required. Automated tests written during BDD are system tests and form part of the set of regression tests. The scenarios can also be used as part of acceptance testing.

Test scenarios are written using a three-section format called “given-when-then”. Gherkin is an example of a language that can be used to write scenarios that can be executed by the Cucumber automation system.

- ***Given*** describes the state of the world before the scenario begins and is the set of pre-conditions for the test
- ***When*** specifies the events that occur or behaviour by a user or other actor
- ***Then*** describes the changes that must take place and represent post-conditions for the test

7 Digital solution acquisition, deployment and maintenance (15%)

In this section of the syllabus, the emphasis is on a range of themes concerning how applications software gets acquired and deployed to the end users.

7.1 Describe the benefits and drawbacks of various approaches to software acquisition

Candidates are expected to appreciate the different options for acquiring applications software, with their benefits and drawbacks.

7.1.1 Building bespoke software components

Benefits:

- Meets all requirements including integration with existing systems and processes.
- May be the only viable solution if there is no existing software available that meets the requirements
- Not paying for unused functionality
- Complete control over build and maintenance
- Can provide a competitive edge as no other organisation can acquire the same software

Drawbacks:

- Significant up-front costs
- Time to deliver is longer than implementing some existing software and there is a risk that the situation will have changed by the time the software is delivered
- Uses internal business resources to specify and make decisions about design
- Requires more internal technical expertise
- Higher risk that what is built will not be suitable

7.1.2 Buying commercial software – COTS (Commercial Off-The-Shelf) and MOTS (Modified Off-The-Shelf)

A COTS (commercial off-the-shelf) product is one that is acquired from a vendor or integrator and used without modification, although configuration options may be available to change the way it works. The term COTS may apply to hardware and software and in some cases, both are part of the solution.

COTS benefits:

- Can be lower cost due to economy of scale of sharing build cost with other customers (although profit margin must be taken into account)
- Mature COTS software will have been tested and used in multiple environments where most if not all of the functionality will have been explored and improved, but this applies less to newly produced software products
- Support and training may be available depending on the supplier
- Maintenance updates are provided
- Software sold publicly must comply with legal and regulatory requirements
- A supplier contract may provide some legal indemnity

COTS drawbacks:

- Unlikely to meet all business requirements completely
- Control of software maintenance (e.g. prioritising bugs) is ceded to the supplier
- Higher ongoing costs such as licence fees
- Integration with other software may not be possible or may require expensive customisation
- May be hard to extract data when moving to a different system if the decision is made to stop using the COTS solution

- May require changes to business practices and procedures
- High reliance on the vendor and the risk that the vendor or product may not survive in the long-term

Modified off-the- Shelf (MOTS) is a type of software solution that can be modified and customised after being purchased from the software vendor. Like COTS, MOTS is available to buy and deploy quickly and provides the core functionality of its business domain. MOTS software enables customisation to extend or modify the functionality by allowing access to some or all of the source code or by programmatic addition of code using an API or scripting language. MOTS in common with COTS also allows for configuration, which can modify or constrain functionality by changing settings or using configuration files.

MOTS benefits and drawbacks:
MOTS combines some of the benefits and drawbacks of building bespoke software and buying a COTS solution. The fact that most of the functionality is pre-built and tested means that the implementation and deployment can be achieved more quickly although work is required to provide the missing functionality. One potential disadvantage is that modifications can conflict with product updates. This may mean that customised modules have to be rebuilt before upgrades to the core software can be applied. In extreme cases, a company may be forced to use an unsupported version of the software due to incompatibility of newer versions with customisations that have been deployed

7.1.3 Hybrid approach: Component based architecture

A component is something that can be deployed individually or as part of a subsystem and that provides specific functionality through a clearly defined interface. Components are usually encapsulated so that the internal working is independent of the interface.

Application and system software can be component based and a solution can be built by combining components to provide the functionality required. This approach can be used to gain the benefits of both COTS and bespoke software by combining components obtained from the most appropriate source. In addition, components may be reused in different solutions and so another source might be ones that are already available within the organisation. In order for a component-based solution to work, the components need to be high-quality and the components owners must guarantee to maintain the interface in the long-term. This solution lowers the dependency on external suppliers because it is easier to replace a component than an entire application.

Components can be linked with APIs and microservices, which may be available from third parties or developed in-house.

7.1.4 Decision factors affecting whether to buy or build software

The following are factors to consider when making the decision to buy or build software

	Bespoke	COTS
Problem uniqueness	Good at solving unique problems	More useful for standard problems
Total Cost (full lifecycle)	High cost over the full lifecycle	Lower cost over the full lifecycle
Timescale (to build, install and operate)	Longer timescales	Shorter timescales
Degree of risk	Very much higher	Lower
Possibility of competitive advantage	More likely	Less Likely
Level of training and support	Likely to be poor	Likely to be good
Quality of documentation	Likely to be poor	Likely to be good

7.1.5 Building solutions with CRM and ERP platforms or "Best of Breed"

Enterprise resource planning (ERP) is a category of business process management software that allows an organisation to use a set of integrated applications to manage the business and automate many back-office functions related to IT, support services and human resources. ERP software is module based, and vendors usually sell the modules separately to suit the requirements of the customer. There are common modules that are shared between line-of-business modules such as sales, HR etc. and that facilitate data sharing and interoperability. Shared components include an enterprise service bus (ESB) and a data warehouse that can be used to combine data and workflows from non-ERP corporate systems.

Customer relationship management (CRM) is a category of business software that automates business processes related to customer-facing activities including targeting, acquiring and communicating with customers and potential customers of the business. Given how central to business operations CRM and ERP systems are, there is a large overlap between their functionality. Both are module based and offer facilities for extension and configuration. However, both ERP and CRM are generic and usually require a great deal of configuration before they can be deployed within an organisation.

"Best of Breed" software is an alternative approach to buying and configuring ERP and/or CRM systems. It means being prepared to buy software from different vendors for each line-of-business, or for each business function, on the basis that it is the 'best solution on the market', or the closest fit to the business requirements. Going down that route implies integrating these packages with components such as an enterprise service bus (ESB) that is acquired from, perhaps, yet another vendor. Additional components such as a data warehouse to provide reporting and management information (MI) may also be purchased from specialist vendors and integrated.

Deploying a multi-vendor solution may be beneficial in that it allows individual applications to be selected on the basis of how closely each matches the specific requirements and ways of working of the business area concerned. Another benefit is that there is less dependence on a single vendor and the customer has more bargaining power as it is easier to switch application components than to give up on a major ERP or CRM implementation that will have taken a great deal of commitment, work and investment over an extended time period.

However, there is much more work and risk involved in integrating multi-vendor systems as they will have diverse interfaces and data sets. Buying from a single vendor does simplify support, licencing and billing arrangements

7.1.6 Open source software and frameworks

Open source software (OSS) is fully working software where the source code has been published so that anyone can view it and, depending on the licence, modify, enhance and re-publish. OSS can be used as part of a solution but there may be legal implications of agreeing to the terms of the licence, for example, some OSS licences require any modified software code to be re-published and the intellectual property rights (IPR) are often retained by the original publisher. Risks associated with using OSS as part of a corporate solution are similar to the use of COTS software components in that any modifications made by the customer organisation may lead to divergence with the published OSS as new versions are released. This may mean that improvements in the published OSS, for example security enhancements, may not be available to customers who have made extensive modifications to the original software.

OSS frameworks such as Bootstrap, Spring framework, Apache Spark or Hibernate provide support facilities for specific aspects of software such as UI development and data handling. These can be built into a corporate application and provide useful components to speed up development and promote consistency.

Using OSS can be beneficial in:

- solving common problems with readily available open source technology
- allowing more time and resource for customised solutions to solve the rarer or more unique problems
- lowering implementation and running costs

Publishing source code can be beneficial as it encourages:

- clearer documentation making it easier to maintain the code
- cleaner and well-structured code
- clarity around data that needs to remain protected and kept separate from code
- suggestions from others about how the code can be improved especially around security

7.1.7 Cloud-based development

Deployment of software on cloud infrastructure has the following benefits:

- Organisations are able to pay only for the use of infrastructure components when needed. This includes servers and software such as DBMSs and application servers as well as middleware such as enterprise service buses (ESBs) and message queues.
- Total cost of ownership (TCO) is lowered and becomes much more visible as billing can be detailed and broken down by application as well as other dimensions.
- Scalability is improved as resources can be more quickly allocated and do not need to be provided in advance.

- Global availability is improved as cloud vendors can provide a global infrastructure including regional data centres and caches to support demand from users.
- High resilience and speedy recovery can be provided at scale and speed as the infrastructure is available on demand rather than needing to be procured.
- Software services can be provided to agreed levels by agreeing contracts covering service level agreements including monitoring and exception reporting.
- Software components can be integrated using open interfaces such as web services, APIs and microservices

7.2 Describe the following software engineering concepts and practices

7.2.1 Programming paradigms

7.2.1.1 Procedural programming is a based on the concept of sequential code with repeated operations placed in procedures (also known as subroutines or functions). Procedures may be called at any point during a program's execution, including by other procedures or even recursively. Fortran, ALGOL, COBOL, PL/I, BASIC, Pascal and C are all procedural languages. They are characterised by large blocks of code although some facilities are available to break applications down into reusable parts, for example, procedure libraries of commonly used functionality.

7.2.1.2 Functional programming is a declarative style in which programs are constructed by applying functions to values and linking them together, for example by passing a function as the argument to another function and thus modifying its behaviour. Operations are stateless and values immutable meaning that there are no side-effects and errors are easier to find. Languages such as Lisp, Clojure and F# are functional languages, although Lisp also supports other paradigms. Many non-functional languages support programming in a functional style including Python, Kotlin and Java. Functional programming is supported in serverless cloud applications with services such as AWS Lambda and Azure Functions.

7.2.1.3 Object-oriented programming (OOP) is “based on the concept of objects, which contain data, in the form of fields (also known as attributes or properties), and code, in the form of procedures (often known as methods,” [44] operations or functions). A feature of objects is encapsulation where an object's procedures can only access and modify the data of the object with which they are associated. In OOP, programs are built from objects that interact with one another. Many OOP languages are class-based, meaning that objects are instances of classes, which are used as data types. OOP languages include Java, C++, C#, Python, R, JavaScript, Ruby, MATLAB, and Smalltalk.

7.2.2 Software engineering cycle

7.2.2.1 Coding is the process of transforming the design of an application into a computer programming language in the form of source code. Different parts of the application may be written (or coded) in different programming languages that are suited to the required functionality. For example, data handling is often written using SQL and user interface (UI) code for web-based applications will usually be

written at least partly using JavaScript. Source code is often written using an integrated development environment that assists in the coding process and highlights potential errors to the coder. Source code may be stored in a source code repository or source code control system (SCCS) and may be subject to version control which ensures that only the current version is used and keeps previous versions if errors are discovered and a decision is made to revert to a previous version.

7.2.2.2 Compiling is the process of converting source code into object code that can be executed by a computer operating system such as Windows or Linux. Languages such as C++ and Java need to be compiled before they can be run. If any errors are found by the compiler program then these are reported to the coder and compilation will fail. Java and some other languages compile to an intermediate state that is further compiled on the target machine making the part-compiled code portable between operating systems. Some languages are interpreted such as JavaScript and Python and do not need to be compiled.

7.2.2.3 Building is the process of putting together all the object code components that are required for the program or application to run. This usually involves some compilation of code and some configuration of the environment, for example specifying the location of API libraries. The build process may be manual or automated.

7.2.2.4 Testing is essential for producing high quality software, as seen in section 6. It is better to find faults as early as possible in the software lifecycle as this reduces the impact and the cost of correcting them. Therefore, testing usually starts with the source code (a type of static testing) which may be reviewed by a peer or supervisor or special software may be used, and this is often integrated with an IDE. The compiled code of each module is tested in isolation using a technique called unit testing. Once multiple components have been individually tested, they are put together and tested using a technique called integration testing. Unit and integration testing are called dynamic (as distinct from static) as they operate on running object code rather than source code. The entire application may be tested following a series of user pathways or scenarios. This is called end-to-end or system testing. Once the development team is convinced there are no faults in the system it will be signed off by users and other stakeholders during acceptance testing.

7.2.2.5 Debugging is the process of eliminating faults. Special software called a debugger allows the coder to visualise the running software and simultaneously follow the flow of the source code so that when an error occurs, it can be linked to the location of the fault in the source code. Debuggers are usually an intrinsic part of an IDE.

7.2.2.6 Integration may mean two things in the context of software development, first the combination of several components of the application under development and second the linking of the current application with other systems with which it will need to communicate. Both are essential for the application to function correctly.

Integration with external systems may be lesser or greater depending on the situation and the requirements.

7.2.2.7 Packaging software means combining all the resources (usually files) that are required to make the application run in its target environment. This includes compiled object code, interpreted code and scripts, configuration files and media files such as images and videos. These may then be compressed into a single file containing the entire package. Examples are MCI (Media Control Interface) files and jar (Java archive) files.

7.2.2.8 Release or deployment is when a new application or an update is made available to users. A release management tool may be used to control the process and it may be done in a phased (limited functionality) or piloted (limited user base) way rather than direct transfer or big bang. Whichever method is used, all the resources need to be available in the target or live environment before users can be allowed to access the new software

7.2.3 Coding standards and examples of best practice in software development

7.2.3.1 Design patterns

In software engineering, a design pattern is a generalised, reusable solution to a commonly occurring problem within a specific context in software design. It is a description or template for how to solve the problem rather than a finished design that can be transformed directly into source code. New design patterns become refined and perfected over time as developers use them in various situations. It is good practise to use design patterns as they promote consistency and embody lessons learned from previous solutions.

A book, Design Patterns, Elements of Reusable Object-Oriented Software was published in 1994 by the 'Gang of Four' [31] contained 23 design patterns, many of which are still widely used in software production today.

Design patterns provide a structured approach to application design bridging the gap between a programming paradigm and a specific algorithm.

7.2.3.2 SOLID principles

Five principles of class design that apply to any component-based style, but particularly object-oriented design, were identified by Robert C. Martin in 2000 [40]. They are known by the acronym SOLID and are listed here:

- The ***S****ingle Responsibility Principle* is that a class should have one, and only one, reason to change
- The ***O****pen Closed Principle* is that classes should be closed for modification but open for extension
- The ***L****iskov Substitution Principle*, named after computer scientist Barbara Liskov is that derived (sub) classes must always be substitutable for their base (super) classes

- The ***I****nterface Segregation Principle* is that client classes should not depend on interfaces they do not use so interfaces should be small and specific to the function they provide
- The ***D****ependency Inversion Principle* is that high-level classes should not depend on lower level classes, rather they should both depend on abstractions

7.2.3.3 Code quality metrics

Code quality can be measured with:

Test coverage:
- Functional
- Statement
- Branch

Complexity:
- Lines of code
- Cyclomatic (counts the number of independent paths through the source code)
- Essential (cyclomatic complexity with loops reduced to a single point)
- Halsted volume (calculated from the number of operators and operands)
- Maintainability index (calculated from the above measures)
- Depth of inheritance (relevant to object-oriented languages)
- Class coupling (coupling between unique classes through parameters, variables, return types, method calls, inheritance, dependency injection and decoration)

Effective quality:
- Average Percentage of Faults Detected (APFD)
- Fault Severity
- Production Incidents
- Quality Over Release Life Cycles

7.2.3.4 Code smells or anti-patterns

Code “smells” or anti-patterns have been identified for object-oriented code and fall into the following categories:
- Bloaters – lead to large unwieldy code blocks
- OO Abusers – turn object-oriented code into procedural code
- Change Preventers – make code unmaintainable
- Dispensables – unnecessary code
- Couplers – examples of tightly coupled classes

The following general coding principles have been identified:
- DRY (“Don’t repeat yourself!” i.e. eliminate repeated code by refactoring into methods and other classes)
- KISS (“Keep it simple stupid!” i.e. if there are multiple ways to do something, always choose the simplest)
- YAGNI (“You ain’t gonna need it!” i.e. do not code for the future, just for the present set of requirements)

Simplicity of code means the following:

- Code + tests must communicate everything (and be understood) so no need for comments
- No duplicate code
- Fewest possible classes/modules
- Minimum data (attributes, arguments)
- Fewest possible methods/operations
- Shortest possible methods/operations

7.2.3.5 Managing technical debt

Technical debt is an analogy representing the trade-off between the short-term benefit of rapid delivery and long-term value of high-quality code. The analogy is with financial debt in that it needs to be repaid at some point in the near future otherwise it will just get worse.

Negative effects of technical debt are that it can compromise the following aspects of software:

- Reliability
- Security
- Performance Efficiency
- Maintainability

Technical debt can be avoided by following best practices such as:

- Using design patterns
- Following SOLID principles
- Measuring and maximising code quality through metrics
- Avoiding code smells
- Minimising code complexity
- Refactoring

7.2.4 Code management techniques

7.2.4.1 Version Control

A version control system (VCS) is a tool that manages changes to source code over time. It keeps track of individual modifications to the code and allows developers to examine and compare current and earlier versions of the code to help find and eliminate faults and, if necessary, revert to an earlier version.

VCSs can be local, centralised or distributed:

Local version control is suitable where developers will not be working on the same code at the same time. Centralised systems allow developers to work on the same code by checking code out, working on it until completed and then checking it in. Other developers can see who is working on the code and cannot check it out at the same time. All requests for code go to a central server. Distributed version control systems (DVCSs) keep a copy of the code base and version history on all developer machines so files can be checked out and managed immediately but any clashes need to be resolved. For large code bases, replication times can be long and there may be security issues with having source code stored on multiple

machines. These issues can be addressed by breaking systems into smaller subsystems.

7.2.4.2 Code branching allows for parts of a software system to be developed in parallel within a single code base. Branches are often created for feature sets or individual features so that they can be developed separately. If the new feature works according to the specification and is accepted by the business, then its branch can be merged with the main product branch. However, the feature can be withheld or withdrawn for any reason. Branches can be created for any reason, not just for features, for example, successive releases may each have a branch and bug fixes are also often isolated in this way.

7.2.4.3 Flags or toggles can be applied to a branch. These annotate or mark code that has changed within a branch. These are often known as feature flags or toggles as they are used to associate specific code changes with a feature although they can be associated with any type of branch. Feature flags or toggles can be used to turn on and off the features they are associated with after release. It can also be used to release two versions of a particular feature and switch between them to get feedback in the live environment or to target different users with alternative versions of the same feature to see which is preferred. This is known as an A/B release as users are polled to see if they prefer version A or version B of a feature.

7.2.4.4 Software configuration files which give instructions for the initial setup can be included with source code and be subject to version control. With virtualised environments, including cloud, instructions for the infrastructure can be provided using Infrastructure as Code (IaC) which can also be managed alongside the application source code and released along with the application components to which it applies

7.2.5 Development environments

7.2.5.1 IDE. An Integrated Development Environment (IDE) facilitates software development by providing automated assistance to developers with the tasks they need to perform when producing software.

Typical features of IDEs include:

- Text editor with language support
- Syntax checking
- Code highlighting and formatting
- Code completion
- Integration with software version control systems for checking code in and out
- Debugging tools to assist in identifying and correcting faults
- Built-in compiler to test code as it is written and identify compilation errors
- Console and UI simulators
- Emulators for operating systems, devices and browsers
- Static testing and code quality metrics
- Integration with unit testing frameworks

Individual developers work on code using an IDE including writing unit tests and performing code reviews with peers and supervisors before submitting code to the shared repository.

7.2.5.2 Integration Environment. The first location that code from multiple developers is brought together is the integration environment. Code is tested using integration tests where multiple components are made to work together. Whenever new code is submitted to the integration environment, all previous tests, both unit tests and integration tests are re-run to ensure no new faults have been introduced. This process is known as regression testing. New integration and unit tests may be added as well.

7.2.5.3 Quality control environment is where complete system tests are run. Software is also integrated with other systems and the interoperability tested. Exploratory testing may be performed by testers and business analysts as well as more formal end-to-end scenario testing, vulnerability or penetration testing and performance testing focusing on non-functional requirements.

7.2.5.4 Pre-production. Once testing has been completed, the software is deployed to a pre-production or staging environment that mirrors the live environment. The application can be tested at scale and the size and configuration of infrastructure can be established and refined.

7.2.5.5 Live Environment. At the point where the decision to release the application to users has been made, the software is deployed to the live production environment. It may be tested here before users are given access.

7.2.5.6 Full stack development is an approach where developers do not solely develop front-end (UI) or back-end (business logic, persistence and integration) software but are able to work on any part of the complete product.

Benefits of full-stack development:

- Flexible team members
- Visibility of all layers
- Fewer developers required
- Easy skills upgrade
- Faster delivery
- Faster problem resolution
- Lower technical debt

However, this usually requires the use of a framework that simplifies development by, for example, reducing the number of programming languages used. This can mean that the development team becomes dependent on the full stack framework being used.

7.3 Describe the key aspects, benefits and considerations of the following practices and techniques

7.3.1. *Iterative development* is the process of repeating and refining a cycle or way of working to continuously improve the product. Typically, iterative development involves a number of steps that are repeated in each iteration:

- Planning, where requirements are confirmed with the business or customer and an initial approach is agreed
- Risk analysis, where risks and their mitigations are identified and agreed
- Engineering, where prototypes and software are produced
- Evaluation, where feedback is obtained from the business or customer

Arguably the most significant step is evaluation as this drives the next iteration or may stop development altogether. Agile development calls for feedback at the earliest possible point as the view of the customer or business is paramount.

The product at the end of any single iteration may or may not be ready for use by the business or customer. Typically, iterative development is used where requirements are unclear and need to be elaborated by prototyping. Therefore, several iterations may be required to achieve something that is ready to be released.

7.3.2. *Incremental delivery* means delivering software in small increments which build upon the previous delivery by adding more functionality and making corrections. There is also the aim here of minimising work-in-progress, which reduces risk. Since the software needs to be usable it needs to contain a minimum set of functions and this is known as a minimum viable product (MVP).

7.3.3. *Product focus.* The valuable commodity at the end of the day is the working code, so it is beneficial to keep focussed on this. Ideally there would be working code at the end of each iteration and increment.

7.3.4. *Customer collaboration.* Software is everyone's business not just the developers. It is useful to keep in close contact with the software's customers and users, and get them involved in the development as much as possible.

7.3.5. *Self-organising teams.* Small teams work more efficiently and effectively if they organise their own work, on a daily basis, within any guidelines set out for the project.

7.3.6. *Continuous improvement (Kaizen)*
Kaizen is a Japanese term meaning "continuous improvement." It is a Japanese business philosophy which focuses on the processes that continuously improve operations and involve all employees.

Although Kaizen approaches vary, there are generally three main pillars:

- Gemba or workplace, ensuring the working environment is clean and free of clutter and that the team has the right tools

- Muda or waste, eliminating anything unnecessary either in the process or the product such as rework, delay, bottlenecks in the process, hand-offs etc.
- Standardised change, usually stated as: Plan, Do, Check, Act. This supports the Agile concept of short, iterative sprints that drive business change based on feedback

7.3.7. Kanban

Kanban, a Japanese word meaning signboard, is a scheduling system often used by Agile software development teams to manage work items and balance work requirements with available capacity. A Kanban board is used to make progress visible to all team members by placing each work item on a separate card that is moved through the development process usually from 'backlog' to 'done' although the granularity of sections in between can vary and may be as simple as a single section called 'in progress'. This approach helps everyone to visualise the progress of the team as well as the amount of work to be done.

7.3.8. Pair programming

Pair programming is a technique in which two programmers or developers work together at a single computer. One writes code while the other reviews and comments on the code, adding suggestions for improvement and considering strategic aspects of the software. It is important that the roles are switched frequently.

Benefits include:

- Higher quality software
- Shorter elapsed time to deliver software
- Raised morale of developers
- Knowledge transfer between developers
- Team building

Pair programming may increase the resources required to deliver a given amount of software, but the quality should be higher so less time is required to correct errors and the understanding of how the software works is greater within the team so less communication will be required.

7.3.9. Agile manifesto

The Agile manifesto [45] lists the following assertions, in which the authors state a preference for the things on the left over the things on the right:

- Individuals and interactions over processes and tools
- Working software over comprehensive documentation
- Customer collaboration over contract negotiation
- Responding to change over following a plan

7.4 Identify and describe the following concepts used in the deployment of software:

7.4.1 Deployment strategies

The following deployment strategies, which may be combined, have been found to be useful in delivering software to the end users. Release and deployment are always the riskiest parts of any effort to invest in digital services, so this needs to be managed with care.

7.4.1.1 Direct changeover (big bang) where the old system is switched off at the same time as the new system becomes available
7.4.1.2 Phased deployment where functionality is released gradually rather than all at once
7.4.1.3 Pilot deployment where software is released to a limited set of users (by location, job role or other categorisation) before being rolled out to the entire user base
7.4.1.4 Parallel running where the new and old systems are kept running together for a period of time.
7.4.1.5 Blue/green deployment where the new system is deployed on separate infrastructure so that the old system can be brought back quickly, usually by redirecting users using a router or gateway. This is popular in cloud deployments where infrastructure can be set up and shut down quickly. Once the new (green) deployment has been accepted by users the old (blue) infrastructure can be released so that costs are minimised.
7.4.1.6 Canary release where a blue/green deployment targets a small percentage of users to find any serious problems before releasing to the entire user base.
7.4.1.7 Dark launch where new features are released to a small percentage of users using feature flags or toggles. The new features are not publicised until users are happy with the functionality provided. This requires close monitoring of users to see how they react to the new features.

7.4.2 Deployment automation and DevOps techniques

7.4.2.1 The DevOps cycle consists of eight stages:

- Plan: Define business value and requirements
- Code: Design and create software code
- Build: Compile and package code and configuration
- Test: Continually test using manual and automated tests to ensure optimal code quality
- Release: Manage, coordinate, schedule, and automate product releases into pre-production
- Deploy: Deploy the code into the production environment for live use by the business
- Operate: Manage software in production
- Monitor: Identify and collect information about issues from a specific software release in production

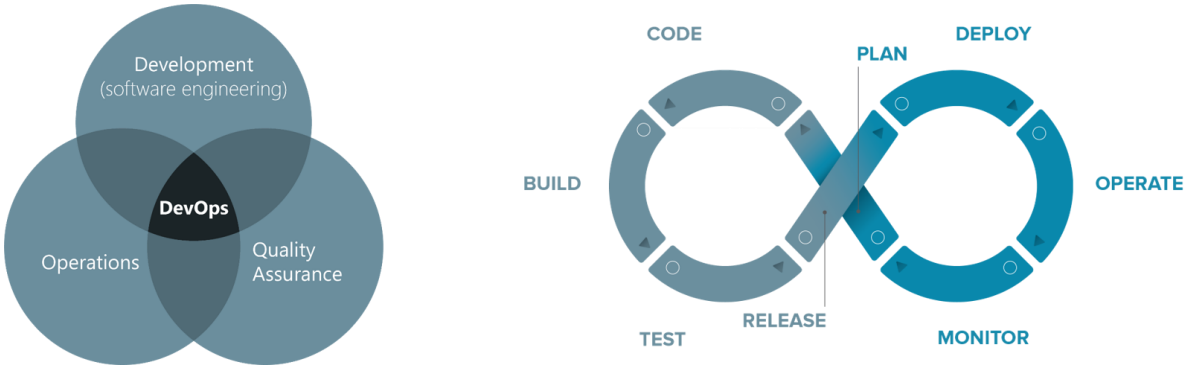


Figure 14: DevOps

DevOps therefore intends to cover the full digital service lifecycle, as depicted elsewhere in the syllabus, with an emphasis on a continuous 'rolling road'. Two qualities that are emphasised in DevOps are CI and CD.

7.4.2.2 Continuous Integration (CI)

Continuous Integration is a software development practice where developers submit code frequently, usually at least daily to an integration environment where the code is verified by an automated build and test with feedback generated immediately on any errors.

7.4.2.3 Continuous Delivery (CD)

Continuous delivery is closely associated with and dependent on continuous integration, often used together as "CI/CD". It is the practice of moving integrated software modules and systems into the pre-production environment ready for deployment. CD does not make the code available to users without a manual authorisation step but does make sure it is always ready for deployment when the decision is made to do so.

CI/CD emphasises the automation of the code-build-release-deploy cycle. Related to DevOps, the left-hand side of the graphic in Figure 12 is CI, the right-hand side is CD.

7.5 Identify and distinguish between the following types of maintenance and monitoring techniques

7.5.1 Service operation activities

Service Operations is the generic term used to describe the support for end users who use IT applications. Service Operations includes activities such as:

- Maintaining a Help Desk as the first port of call for assistance with any IT related problems
- Providing incident management, for example resolving incidents with the IT infrastructure
- Deploying software services to end users.
- Administering the Disaster Recovery (DR) and Business Continuity plan.
- Planning capacity and installing equipment
- Administering security, such as data backups

- Running routine operations tasks like data replication

The best practice framework for running service operations is ITIL [8]

7.5.2 Types of maintenance

Maintenance refers to software development activity on products already deployed to the live environment and which need minor corrections. There are four major categories of maintenance activity, which is usually handled outside of the full SDLC framework, by a separate team.

7.5.2.1 Corrective, fixing bugs, flaws and defects in the software
7.5.2.2 Adaptive, keeping the software aligned to the technical and business environment by accommodating changes
7.5.2.3 Perfective, continually improving the software so that it more closely meets functional and non-functional requirements
7.5.2.4 Preventative, futureproofing software and preparing for any potential changes ahead

7.5.3 Application monitoring techniques

The following techniques are widely used to monitor running applications and identify the need for maintenance. They are also used to measure performance against SLAs especially for NFRs.

7.5.3.1 Health monitoring is used to track the inputs and outputs of an application based on key metrics, logs and traces in order to watch how an application performs over time. A health check on an application is analogous to medical health checks in that acceptable “healthy” values for each of the metrics is agreed so that alerts can be triggered when the measure is outside these ranges. The baseline for health monitoring is the set of service level agreements (SLAs) that back up the non-functional requirements (NFRs) of the application. Health monitoring generally goes beyond these measures to give advanced warning of the need for infrastructure or software maintenance.

7.5.3.2 Service logging is used to identify faults and locate the causes so that they can be fixed quickly, and damage can be limited. Events are logged by multiple components of a system and so a mechanism is put in place to collect and aggregate them so they can be correlated and associated to see patterns of behaviour that transcend individual components. Log collections are also indexed to facilitate searching and analysis. Apart from its use for troubleshooting when issues become apparent, log data can be used to provide automatic notifications, regular reports and can be visualised using dashboards and other techniques.

7.5.3.3 Application KPIs are used at the macro level to measure business objectives, for example digital take-up of previously non-digital processes, user satisfaction, task completion rate vs. drop-outs and cost per transaction to the business.

8 Cyber Security (10%)

This section of the syllabus covers security concerns, with a focus on enterprise software applications. The section recognises the need for a holistic approach to security, and therefore emphasises that cyber security sits within the wider context of Enterprise Risk Management (ERM).

8.1 Explain the importance of having secure systems. Recognise that Cyber Security sits within the wider context of ERM. Identify standards applicable in this area and have an overview level of understanding of the scope of each.

Enterprise risk management (ERM) is a business function that aims to identify, assess, and prepare for any dangers, hazards, and other potential risks that may interfere with the organisation's operations and overall objectives.

Cyber security is a significant area of risk for many organisations in the context of ERM because of the critical dependence on automated information systems for operational delivery of services as well as the potential for breaches of data governance which can lay the organisation open to legal action.

8.1.1 Explain the importance of having secure systems, by recognising some common consequences to the enterprise of security breaches

Some common consequences to the enterprise of cyber security breaches include:
8.1.1.1 Financial Loss through theft and fraud
8.1.1.2 Service outages and the inability to operate normally
8.1.1.3 Reputational exposure and loss of brand value.
8.1.1.4 Legal consequences and compliance fines, for example breaches in GDPR.
8.1.1.5 Damage to customer confidence, for example if credit card details are stolen.

8.1.2 Recognise the following hierarchy of security regimes, define the terminology used and understand the key relationships between the concepts involved

8.1.2.1 Enterprise Risk Management: management of risk in general across the enterprise. Risk in ERM frameworks tend to define risk as ‘uncertainty of outcome’ and contemplates both negative outcomes (downside risk) and positive ones (upside risk). The overall purpose of risk management is to control risks so that enterprise goals are achieved. One of the frameworks widely used in this area is Management of Risk (MoR) [32].

8.1.2.2 Business Security: Business security is a general topic within risk management dealing with safety and threats to the value and usefulness of assets of all types. The enterprise invests in assets because they are necessary for the achievement of goals. Therefore anything that diminishes their value is a security concern.

8.1.2.3 Cyber Security: Cyber narrows the focus of security to threats to Information System assets and Information Technology assets. Cyber security is concerned with protecting networks, devices and data from unauthorised access or criminal use and has the general aim of ensuring the Confidentiality, Integrity and Availability

(CIA) of information resources. An example of a security architecture framework that relates to these concerns is SABSA [33].

8.1.2.4 Application Security: This is the process of developing, adding, and testing security features within applications to prevent and detect security vulnerabilities against threats such as unauthorised access and modification. The ISO series 27000 [34], which is classed as an Information Security Management System (ISMS), is applicable here.

ISO 27034 in particular states: *"Applications should be protected against vulnerabilities which might be inherent to the application itself (e.g. software defects), appear in the course of the application's life cycle (e.g. through changes to the application), or arise due to the use of the application in a context for which it was not intended."*

ISO 27034 further states: *"Throughout its life cycle, a secure application exhibits prerequisite characteristics of software quality, such as predictable execution and conformance, as well as meeting security requirements from a development, management, technological infrastructure, and audit perspective. Security-enhanced processes and practices-and the skilled people to perform them-are required to build trusted applications that do not increase risk exposure beyond an acceptable or tolerable level of residual risk and support an effective ISMS."*

8.1.3 Explain the need for security controls and to balance the use of security controls in application software against factors such as risk, cost and usability

Security is managed in practical terms through the installation of appropriate controls. A control is any mechanism that is intended to provide a degree of protection against an identified risk. A 'control objective' is a statement describing what is to be achieved as a result of implementing a control.

It is not realistic to expect to operate in totally risk-free environment. Security controls introduce additional complexity into the user experience and can lead to increased levels of dissatisfaction amongst users. Hence, their introduction should be carefully balanced against the potential vulnerabilities that may arise from not including the control, and any additional development cost that may be incurred designing, coding and thoroughly testing the control.

8.2 Recognise and explain the following generic approach used by security experts for managing cyber security risks

8.2.1 Define cyber security risk

Cyber security risk is defined as any threat that undermines desirable properties of an information system, including CIA:

Confidentiality: property that information is not made available or disclosed to unauthorised individuals, entities, or processes.

Integrity: property of accuracy and completeness.

Availability: property that information is accessible and usable on demand by an authorised entity.

This actually covers a wide range of risks, including both logical and physical security.

8.2.2 A generic 4 step approach for managing cyber security risks

Risk in this context should be seen as a possible future event that if it occurred would cause some damage to the enterprise's interests or those of its stakeholders. Cyber risks are no different to any other type of risk in the general approach to managing them, something that has been well researched over the years. A generic approach to managing risk involves 4 steps:

*8.2.2.1 Identify: '*Threat modelling' is the process of finding where the application software has vulnerabilities that could be exploited by ill-intentioned entities. Such vulnerabilities give rise to risks.

8.2.2.2 Assess: Assessment of a risk that has been discovered is generally measured in terms of likelihood and impact. How likely is the threat event to occur, and what would be the consequences?

8.2.2.3 Mitigate: Also known as 'treatment'. Appropriate treatment of a risk is based on the assessment of it. Recognised ways of mitigating risk include:

- Reduction: taking actions to reduce likelihood and/or impact
- Transfer: insurance or sharing of risk with 3rd parties
- Contingency: deciding on actions to be taken should the risk event occurs. In this regard it is essential that there is designated ownership for each risk.
- Acceptance: some risks may be acceptable if the likelihood and impact are known and small. It is not realistic to expect to eliminate risk completely from the use of software applications. In some cases, it may be far more expensive to treat a given risk than the possible dollar impact the threat event can cause. Also, there may be risk which is difficult or impossible to remove.

8.2.2.4 Monitor: The panorama of cyber security is changing all the time, with new threats and threat agents emerging all the time. It is important therefore to reassess cyber risks frequently, as software gets deployed, developed and renewed. It is also vital to assess the efficiency and effectiveness of the controls put in place to mitigate cyber threats.

8.3 Identify and describe the purpose of the following range of security related techniques and best practices, which are especially related to digital solution development.

8.3.1 *Forensics, audit, activity logging*.

Computer forensics is an activity that is similar to the forensic activities carried out a crime scene, and has the same purpose, which is to establish what happened in the case of a breach of cyber security and to gather evidence. Forensics for example has revealed the identity of hackers and led to criminal prosecutions.

Activity logging is the main means of audit that forensics can exploit. Logging software can be installed as part of the IT infrastructure. Logging involves recording events and other activity that occur in the IT environment where an application or information system operates. Activity logging includes recording actions by users and external systems and configuration changes to the system itself. Unusual activity can be flagged up in real time for operators to take action.

Although most of the evidence required by forensics will be provided by IT infrastructure systems (see SIEM below), application developers must be prepared to participate in this too and include in the software suitable audit trail information concerning the use of the application, where required.

An important security principle is *non-repudiation*; i.e. it should not be possible for a user to deny that they took certain actions. Evidently from an ethical standpoint, users should be informed about what information the system record about their actions.

8.3.2 Security Information and Event Management

Security Information and Event Management (SIEM) is a software solution that aggregates and analyses activity from many different sources across an organisation's environment including network devices, user devices and servers. A SIEM system stores and aggregates multi-source activity log records and can apply analytics to that data to identify trends, detect threats, and alert security personnel. These systems are allied to Intrusion Detection Systems (IDS) and Intrusion Prevention Systems (IPS).

8.3.3 Penetration testing

Penetration testing (abbreviated to 'pen' testing and also known as ethical hacking) is a process where the enterprise itself probes vulnerabilities in its own applications and networks to find weaknesses that criminals could exploit. Any untreated vulnerability can then be mitigated with a cyber security control. To make the tests more authentic, some organisations designate the penetration testers as the 'red team' and the security staff as the 'blue team'. The red team can attack at any time and the blue team must be have everything in place to defend against them.

8.3.4 Encryption of data in motion and at rest. Key and Certificate Management.

Encryption is an effective security measure for data security and can be applied to data in motion (communications) and to data at rest. A number of strategies for encryption are available, including:

Symmetric keys: used for data at rest: the same encryption key is used to both encrypt and decrypt the data.
Asymmetric keys: used for data in motion: a pair of keys for the encryption and decryption of the data. Both keys in a key pair are related to each other and created at the same time. They are referred to as a public and a private key.

- Public key: used to encrypt data and is freely published

- Private key: used to decrypt data encrypted by the public key, and must be kept safe

Asymmetric keys are used for secure communication such as over a virtual private network (VPN).

A digital certificate provides information about the identity of an entity. A digital certificate is issued by a Certification Authority (CA). Examples of trusted CA across the world are Verisign, Entrust, etc. The CA guarantees the validity of the information in the certificate.

The main purpose of a 'digital certificate' is to ensure that the public key contained in the certificate belongs to the entity to which the certificate was issued; in other words, to verify that a person sending a message is who he or she claims to be.

8.3.5 Identity Access Management (IAM)

A key aspect of security is checking whether users wanting to use an application, or an application feature, are allowed to do so. This set of controls is known as Identity and Access Management (IAM). It is widely recognised that people are the most important security problem for information systems, in many cases employees, or ex-employees.

Carter (2017) [41] explains that identity and access management systems are typically composed of three major elements: users, systems/applications, and policies. Policies define how the users can interact with the different systems and applications.

Most identity and access management (IAM) products provide a variety of methods for implementing the policies to control access to applications, with varying terminology being used to describe these methods. There are four classic models for IAM:

Discretionary Access Control (DAC) controls access based on the requestor and on access rules stating what the requestors are or are not allowed to do. For example, an administrator can manually give another user access to an application at his or her discretion.

Mandatory Access Control (MAC) controls access based on comparing security labels with security clearances (e.g. Confidential, Secret, Top Secret) and is typically utilized in protecting classified information in military systems.

Role-based Access Control (RBAC) controls access based on the roles that users have within the system and on rules stating what access is allowed for what users in given roles. Typically, with modern IAM solutions, a role is roughly equivalent to a User Group in the directory system and is simply a logical grouping of one or more users that have some common affiliations, such as a same department, grade, age, physical location, or user type. For example, with RBAC, a user in the accounts payable clerk position would automatically get added as a member (i.e. dynamic membership) to the AP Role, granting him or her access to AP functions in the accounting system.

Attribute-based Access Control (ABAC) can control access based on three different attribute types: user attributes, attributes associated with the application or system to be accessed, and current environmental conditions. An example of ABAC would be allowing only users who are type=employees and have department=HR to access the HR/Payroll system and only during business hours within the same time zone as the company.

ABAC is the most flexible and powerful of the four access control models, but is also the most complex. ABAC is capable of enforcing DAC, MAC, and RBAC. ABAC enables fine-grained access control, which allows for more input variables into an access control decision, but this adds complexity to the task of administering the user community.

8.4 Recognise that every coding platform has inherent security weaknesses and explain the need for secure coding practices.

8.4.1 Security weaknesses inherent in the coding platform

There is a recognition that each coding language presents certain security vulnerabilities, sometimes unique to the platform. There are quite famous examples in the "Writing Secure Code" book (Le Blanc/Howard) [35]. Applications based on a mixture of coding languages can multiply these dangers.

8.4.2 Secure Coding Practices (OWASP)

OWASP 'The Open Web Application Security Project' is an online community that produces freely available articles, methodologies, documentation, tools, and technologies in the field of web application security. This organisation promotes the use of secure coding practices. A wide range of security areas relating to applications are covered including:

- Input Validation
- Output Encoding
- Authentication and Password Management
- Session Management
- Access Control
- Cryptographic Practices
- Error Handling and Logging
- Data Protection
- Communication Security
- System Configuration
- Database Security
- File Management
- Memory Management

8.4.3 Use of Code Analysis tools, integrated with the IDE

Static Application Security Testing (SAST) is an automated application that performs static, white-box testing focused on security to find security vulnerabilities in the application source code and ensure conformance to coding guidelines and standards. SAST can be used throughout the development lifecycle as it does not require the code to be compiled or integrated. SAST software can be integrated into an IDE.

Dynamic Application Security Testing (DAST) is an automated application that performs dynamic tests on running applications and software modules to find security vulnerabilities and weaknesses. Since it does not look at the internal workings of software modules, nor look at the source code, it is considered to be a type of black box testing. However, DAST uses fault injection techniques to feeding malicious data to the software (fuzzing), to identify common vulnerabilities, such as SQL injection, cross--site scripting, authentication and server configuration issues and so tries to break the encapsulation of modules and attack the inner working of the software.

8.4.4 Risks arising from the use of 3rd party components

In another part of the syllabus the benefits of using 3rd party components (TPC) were enumerated. However, there are also a number of security related risks that arise from using them, for example:

- TPC may contain unknown security vulnerabilities
- TPC not tested to user organisation standards
- Attackers may already know about TPC vulnerabilities of popular components
- It may not be clear what TPCs are included in a given software product, or even in a TPC.
- Harder to keep TPC security patches up to date

8.4.5 Adherence to recognised Security Principles which apply to secure code application development, in particular web development

Security principles from the OWASP Development Guide, 'Security by Design':

1. Minimise attack surface area

Every time a programmer adds a feature to their application, they are increasing the risk of a security vulnerability. The principle of minimising attack surface area restricts the functions that users are allowed to access, to reduce potential vulnerabilities.

2. Establish secure defaults

This principle states that the application must be secure by default. That means a new user must take steps to obtain higher privileges and remove additional security measures (if allowed). Example is the strength of passwords

3. The principle of Least privilege

The Principle of Least Privilege (POLP) states that a user should have the minimum set of privileges required to perform a specific task

4. The principle of defence in depth

The principle of defence in depth states that multiple security controls that approach risks in different ways is the best option for securing an application.

5. Fail securely
Failure of the application and error conditions should not give the user additional privileges, and it should not 'break' and show the user sensitive information like database queries or logs.

6. Don't trust services
This means that the application should always check the validity of data that third-party services send and not give those services high-level permissions within the app.

7. Separation of duties
Separation of duties can be used to prevent individuals from acting fraudulently. Role conflicts must be sought out and eliminated. For example, an Administrator should be allowed to behave like a Customer.

8. Avoid security by obscurity
For example, if your application relies on its administration URL to be hidden so it can remain secure, then it is not secure at all. Someone will find it.

9. Keep security simple
Developers should avoid the use of very sophisticated architecture when developing security controls for their applications. Having mechanisms that are very complex can increase the risk of errors.

10. Fix security issues correctly
If a security issue has been identified in an application, developers should determine the root cause of the problem. They should then repair it and test the repairs thoroughly.

8.5 Recognise that threat modelling takes place as part of software design.

Security is everyone's business and it is down to application developers to do their part and recognise that threat modelling should be an integral part of software design, not an afterthought.

Threat modelling is a way of understanding what vulnerabilities exist that might be exploited by ill-intentioned entities. It has developed its own vocabulary to describe threat situations, and some models to help focus thinking about them.

8.5.1 Define the following threat modelling vocabulary and techniques

Threat Agent: a malicious entity that has means, motive and opportunity to devise and launch a threat action.

Threat Actor: entity that carries out a threat action, maybe unwittingly.

Threat Target: identifies the object of the attack. For example, the target might be information that criminals wish to steal, or to be able to threaten an organisation with disruption to their operations.

Threat Action: the action taken by a threat actor with respect to the threat target.

Threat Event: Some event that threatens one or more aspects of CIA, or a security property.

Attack Vector: a path or means by which an attacker can gain unauthorised access to a computer or network to deliver a payload or malicious outcome. Attack vectors allow attackers to exploit system vulnerabilities, install different types of malware and launch cyber-attacks.

Vulnerability: a state of being exposed to a particular threat event. Vulnerability can be measured from zero (no vulnerability) to high (highly vulnerable).

Abuse Case: a scenario or other description of how threat actors could exploit vulnerabilities, and their goals in doing so. Similar to the idea of a Use Case but focussed on malicious intent.

8.5.2 *Use of STRIDE and DREAD models*

STRIDE is a model, produced by Microsoft, used to assess common threats to a system such as an application or applications in general. The model gives the developer a structure for thinking about vulnerabilities.

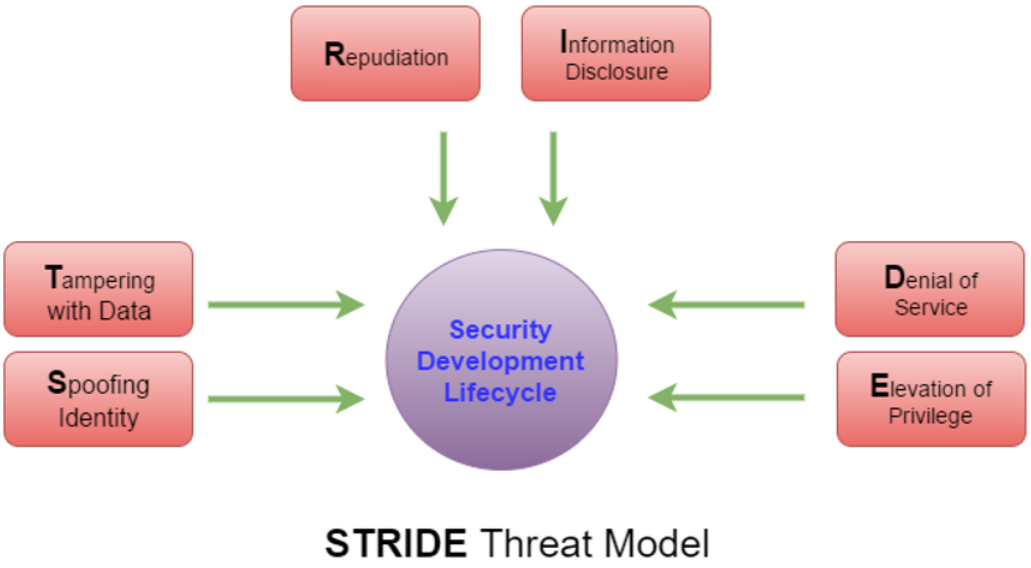


Figure 15: STRIDE Threat Model

It is an acronym for:

- **S**poofing or gaining authenticity through deception. Spoofing threatens *authenticity*, which is what IAM systems are supposed to assure.
- **T**ampering or interfering with the integrity of data or other system information. Tampering threatens *integrity*.
- **R**epudiation or performing actions and avoiding attribution. Repudiation threatens *non-repudiability*.

- **I**nformation disclosure or bypassing confidentiality controls. Disclosure threatens *confidentiality*.
- **D**enial of Service or reducing the availability of a system to legitimate users. Denial threatens *availability*.
- **E**levation of Privilege or gaining a higher level of authorisation by illegitimate means. Elevation threatens *authorisation.*

DREAD is a model used to assess the impact or seriousness of an attack, based on detecting a vulnerability through STRIDE. It is an acronym for:

- **D**amage. How much damage would occur if the attack were successful?
- **R**eproducibility. How difficult is it to reproduce this attack?
- **E**xploitability. What is the likelihood that the threat can move from theory into practice?
- **A**ffected users. How widespread would the impact be among the user community?
- **D**iscoverability. How easy would it be for a malicious entity to discover this vulnerability?

8.5.3 Make use of a recognised 'threat library' like the OWASP Top 10.

OWASP publishes, and periodically refreshes, the 'Top Ten' threat actions that are current in the cyber security space. The list keeps changing but it is a useful resource for developers to review against their application, so that they can ensure that they take any preventative steps required.

Abbreviations

API	Application Programming Interface
AR	Augmented Reality
BDD	Behaviour Driven Development
CAP	Consistency, Availability and Partition tolerance
CD	Continuous Delivery
CI	Continuous Integration
CIA	Confidentiality, Integrity and Availability
COTS	Commercial Off-The-Shelf
CRM	Customer Relationship Management
CRM	Customer Relationship Management
CRUD	Create, Read, Update, Delete
DBMS	Database Management System
DevOps	A set of practices that combines software development (Dev) and IT operations (Ops) in order to shorten the systems development life cycle and provide continuous delivery of high-quality software.
DFD	Data Flow Diagram
DREAD	**D**amage, **R**eproducibility, **E**xploitability, **A**ffected users, Discoverability
DSD	Digital Solution Development
EDI	Electronic Data Interchange
ERD	Entity Relationship Diagram
ERM	Enterprise Risk Management
ERP	Enterprise Resource Planning
ESB	Enterprise Service Bus
GUI	Graphical User Interface
HCI	Human Computer Interface
IAM	Identity and Access Management
IDE	Integrated Development Environment
IE	Information Engineering
INVEST	Independent, Negotiable, Valuable, Estimable, Small & Testable
ISMS	Information Security Management System
JSON	JavaScript Object Notation
Kaizen	The concept of continuous improvement.
Kanban	A scheduling system for lean production, usually managed using a Kanban board that everyone involved can see and that shows work items, often on cards, moving through the steps of a process.
KPIs	Key Performance Indicators
MOLAP	Multidimensional Online Analytical Processing
MoR	Management of Risk
MoSCoW	Must have, Should have, Could have, and Won't have this time
MOTS	Modified Off-The-Shelf
MR	Mixed Reality
MVC	Model View Controller
MVP	Minimum Viable Product

NCSC	National Cyber Security Centre
OLAP	Online analytical processing
OLTP	Online transaction processing
OWASP	Open Web Application Security Project
REST	Representational State Transfer
ROLAP	Relational Online Analytical Processing
SAFECode	Software Assurance Forum for Excellence in Code
SDLC	Software Development Life Cycle
SIEM	Security Information and Event Management
SOA	Service Oriented Architecture
SOAP	Simple Object Access Protocol
SOLID	**S**ingle Responsibility Principle, **O**pen/Closed Principle, **L**iskov Substitution Principle, **I**nterface Segregation Principle, **D**ependency Inversion Principle
Sprint	A usually short, fixed time period during which a team commits to complete certain work. Associated with the Agile and Scrum methodologies.
SQL	Structured Query Language
STRIDE	**S**poofing, **T**ampering, **R**epudiation, **I**nformation Disclosure, **D**enial of Service, and **E**levation of Privileges
TCO	Total Cost of Ownership
TDD	Test-Driven Development
TFD	Test First Development
UI	User Interface
UML	Unified Modelling Language
UX	User Experience
VR	Virtual Reality
WIMP	Windows, Icons, Menus, Pointer
XML	Extensible Mark-up Language
YAML	A recursive acronym for "YAML Ain't Mark-up Language"

Additional Information

https://owasp.org/www-project-top-ten

https://www.ncsc.gov.uk/collection/cloud-security/implementing-the-cloud-security-principles

https://safecode.org/safecode-principles

https://owasp.org/www-pdf-archive/OWASP_SCP_Quick_Reference_Guide_v2.pdf

JSON

Consistency

Availability

Partition tolerance

Window decoration

Processes

Graphical control element

Control element

User interfaces

ISO/IEC 25010:2011

ISO9000

ISO42010

ISO/IEC 9075

ISO 31000

ISO 27001

ISO 27034

ISO 27000

References/Accreditations

Index	Reference
1	Paul, D. (2020) Business Analysis. BCS Learning and Development.
2	SAFe®. Scalable Agile Framework, v5.0.
3	DSDM®. Dynamic System Development Method.
4	TOGAF®. The Open Group Architecture Framework, v9.
5	MSP®. Managing Successful Programmes.
6	PRINCE2®. Projects in Controlled Environments.
7	IIBA®. International Institute of Business Analysis™/BABOK® Guide to the Business Analysis Body of Knowledge.
8	ITIL®. Information Technology Infrastructure Library.
9	ISO 9000:2015. Quality Management Systems (QMS).
10	DAMA. Data Management Association. DMBOK®. Data Management Body of Knowledge.
11	Wake, W (2002). Extreme Programming Explored. Addison-Wesley.
12	Cohn, M (2002). User Stories Applied. Addison-Wesley.
13	Object Management Group (OMG®). Unified Modeling Language (UML®).
14	Rational Unified Process (RUP®).
15	Cockburn, A (2000). Writing Effective Use Cases. Addison-Wesley.
16	Object Management Group (OMG®). Business Process Modeling and Notation (BPMN®).
17	Krugg, S (2005). Don't make me think!
18	ISO 42010:2011. Systems and Software Engineering – Architecture Description.
19	Cockburn, A (2005). https://alistair.cockburn.us/hexagonal-architecture.
20	Model-View-Controler (MVC) Pattern. Trygve Reenskaug, SmallTalk-79.
21	Newman, S (2015). Building Microservices. O'Reilly.
22	Erl, T (2007). SOA – Principles of Service Design. Prentice-Hall.
23	ISO/IEC 2382-1:1993:2015. Information Technology Vocabulary.
24	Based on the entries for 'information' in Wikipedia
25	ISO/IEC 9075: 2016. Information technology - Database languages – SQL.
26	Martin, J (1989). Information Engineering. Prentice-Hall.
27	Crosby, P (1978). Quality is Free.
28	Fowler, M (1999). Refactoring. Booch, Jacobson, Rumbaugh.
29	ANSI/IEEE 1059:1993. IEEE Guide for Software Verification and Validation Plans. "IEEE Guide for Software Verification and Validation Plans," in IEEE Std 1059-1993 , vol., no., pp.1-87, 28 April 1994, doi: 10.1109/IEEESTD.1994.121430.
30	International Software Testing Qualifications Board (ISTQB®).
31	Gamma et al (1994). Design Patterns. Addison-Wesley.
32	Management of Risk (M_o_R®).
33	Sherwood Applied Business Security Architecture (SABSA®).
34	ISO 27000 series of standards on Information Security Management.
35	Howard, M (2004). Writing Secure Code. Microsoft.
36	Open Web Application Security Project (OWASP®). https://owasp.org/www-pdf-archive/OWASP_SCP_Quick_Reference_Guide_v2.pdf
37	OWASP Top Ten/https://owasp.org/www-project-top-ten
38	Jacobsen, N (1994).
39	Fowler, M (2012) No-SQL Distilled. Addison-Wesley.
40	Martin, R (2000). Design Principles and Design Patterns.
41	Carter, S (2017). RBAC vs ABAC Access Control Models - IAM Explained https://blog.identityautomation.com/rbac-vs-abac-access-control-models-iam-explained
42	IBM Knowledge Center (2020) https://www.ibm.com/support/knowledgecenter/SSGMCP_5.4.0/product-overview/acid.html
43	S.K.Singh (2009) Database Systems: Concepts, Design and Applications

44	Based on the entries for ‘object-oriented programming’ in Wikipedia
45	Agile Manifesto (2001) https://agilemanifesto.org/