ChairBear Digital Learning

Software Development Essentials

Handbook for Software Development Apprentices

Created 13th May 2020

Chairbear

Foreword

∙∙∙

This book was written for educational purposes, by Jade Lei ©Chairbear, for use by any individual or organization under the BY-NC-ND creative commons licence (Attribution + Noncommercial + NoDerivatives).

This book is particularly useful for those studying a range of qualifications, including software development and IT.

A breadth of topics are covered in this book, with more advanced topics only summarized; it is recommended to take in this basic coverage, and explore particularly heavy subject areas yourself.

Many topics covered include diagrams and some explanations related to these.

And strengths and weaknesses were applicable are summarized with short explorations ascertaining to such.

A glossary is provided, if needed, however for particular instances throughout the book footnotes instead will be utilized to give a more instantaneous understanding if needed.

Recommended further readings for topics discussed here are:

- Developing information systems practical guidance for IT professionals - © 2014 BCS Learning & Development Ltd
- Disciplined Agile delivery: a practitioner's guide to Agile software delivery in the enterprise - Ambler, S. W. and Lines, M. (2012) IBM Press, Indianapolis, IN.
- What is Agile software development? Here are the basic principles - Freecodecamp (2020)
- System designs for interviews - Freecodecamp (2020)

Although there are many more recommendations for reading, these are the main 4 we recommend. However, we acknowledge that not everyone learns the same way, and so suggest you find additional resources that accommodate your learning style.

∙∙∙

Jade Lei
SEO Technical Specialist,
Software Development Apprentice (2020)

The Software Development Lifecycle

The Software Development Lifecycle (SDLC) is a framework describing a process for planning, building, testing and deploying a system. The process can apply to both hardware and software systems.

....
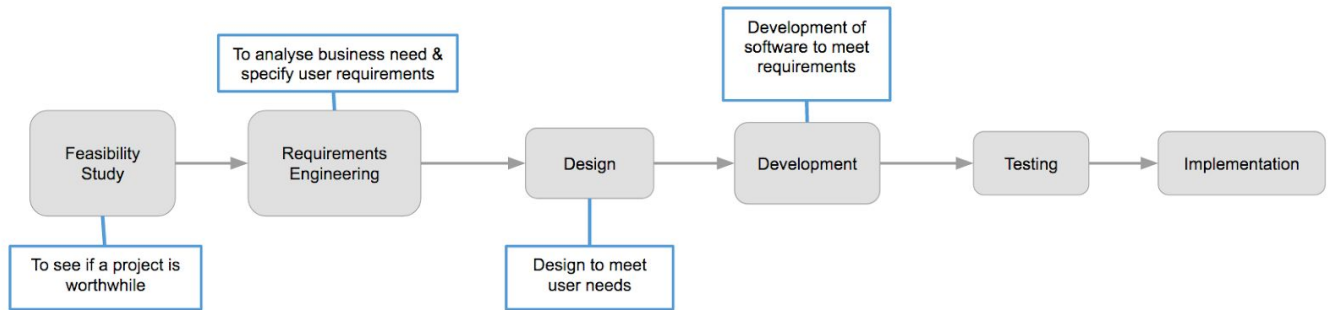
Feasibility Study

Involves investigation and research in order to evaluate the projects potential for success to support decision making, and secure funding. It weighs the resources and costs involved against the project's value. This returns the return on investment (ROI).

ROI Formula

ROI = (Net Profit / Cost Of Investment) X 100

Only projects with a reasonable ROI will be supported.

There are different types of feasibility, explored further below:

Technical Feasibility

- Will the proposed system perform to the required specification?
- Outline technical systems options proposed to use

Social Feasibility

- Consideration of whether the proposed system would prove acceptable to the people who would be affected by its introduction
- Describe the effect on users from the introduction of the new system:
    - Will there be a need for retraining the workforce?
    - Will there be a need for relocation of some of the workforce?
    - Will some jobs become deskilled?
- Describe how you propose to ensure user cooperation before changes are introduced

Economic Feasibility

- Consider the cost/benefits of the proposed system
- Detail the costs that will be incurred by the organisation adopting the new system: consider development costs and running costs

Requirements Engineering

Aims to secure understanding in what the business needs the software to do. Requirements must be elicited, analysed, documented and validated by the business analyst.

The requirements are developed through 4 stages:
1) Elicitation
- Elicitation practices typically include JAD (joint application development) techniques such as interviews, questionnaires, user observation, use cases, role-playing, and prototyping.
- This is likely to involve many different participants.

Interviews

Strengths
- Allows analysts to probe into greater depth
- Analyst can respond in real time to interviewee to ensure clarity of understanding

Weaknesses
- Time consuming and costly
- Subject to bias & interviews may produce conflicting results

Questionnaires

Strengths
- Allows collection of quantitative data
- Faster method of gathering from large sample size

Weaknesses
- Difficult to create well balanced questionnaire
- Difficult to receive good response rates

Observation

Strengths
- Identify peak and quit periods
- Check validity of information gathered in other methods

Weaknesses
- Participants often don't accurately recall everything they do
- People's behaviour changes when they are watched

2) Elaboration

- This stage provides greater depth to the reasoning of each requirement or user commentary, and breaks down each requirement or commentary into technical details.
- Methods used by the business analyst in this stage will include developing use cases, creating flow diagrams, class models, GUI mock-ups, and assigning business rules.
- This phase also assists with addressing known risk factors and establishes or validates the system architecture.

3) Validation
- This stage verifies that requirements are complete (and testable).
- The requirements document should be checked for ambiguities, conflicts and errors, while developers and testers should check the user commentary to ensure it matches up with criteria.
- Techniques used at this stage will include discussions, simulations, and face-to-face meetings.

4) Acceptance
- This is the final stage in requirements definition. It only happens when the requirements have been verified and agreed by all the stakeholders.
- It is during this stage that the business analysts create a 'baseline' of the requirements so that technical development can start and test planning can commence. This is done through an SRS (System/Software Requirement Specification) document which consists of all the product requirements to be designed and developed during the project life cycle.
- The functional and non-functional requirements will also be clearly specified here.

Functional Requirements

In software development, a functional requirement defines a function of a system or its component. A function is described as a set of inputs, the behaviour, and outputs.

An example of a functional requirement would be a system that needs to be able to store user names in a database.

Non-Functional Requirements

A non-functional requirement (sometimes referred to as NFR) is a requirement that specifies criteria that can be used to judge the operation of a system, rather than specific behaviours. Such as speed or scalability.

Non-functional requirements are not any less important than functional.

An example of a non-functional requirement would be a system that should be able to carry out a thousand transactions per second. A non-functional requirement is almost like a SMART objective. It is measurable.

Further Example

A page has a video on it. There should be a facility for the user to change the video audio volume. The audio should be controlled by a slider.

The **functional requirement** will be the ability to change the audio.

The **non-functional requirement** will be the slider. This is to do with the GUI, it is a matter of choice, there are multiple options available to change the volume, you could have just as easily chosen a knob.

If a system calls a function to make something happen, that is a functional requirement, how you make the function happen is not functional, as it is a matter of choice.

Ei; the slider being moved to increase the audio is non functional - it could have just as easily been a knob turned up - however, the audio increasing as a result is functional.

Design

The SRS from the requirements stage is the reference for product architects to decide the best architecture for the product to be developed. Based on the requirements specified in the SRS, usually more than one design approach is proposed and documented in a Design Document Specification (DDS).

This DDS is reviewed by stakeholders and a approach is selected based on variables such as:

- risk assessment
- product robustness
- design modularity
- budget and time constraints

The chosen design is elaborated upon.

System Architecture

In software development, architecture is a description or model of a system that includes all details of the system such as the structure, components, behaviour, and attributes.

It is not just a design and not just about the structure of a system. It encomapsses all aspects of a system, including internal and external interactions.

There are many types of architecture, however the main ones covered here are Enterprise, Business, Solution and Infrastructure Architecture.
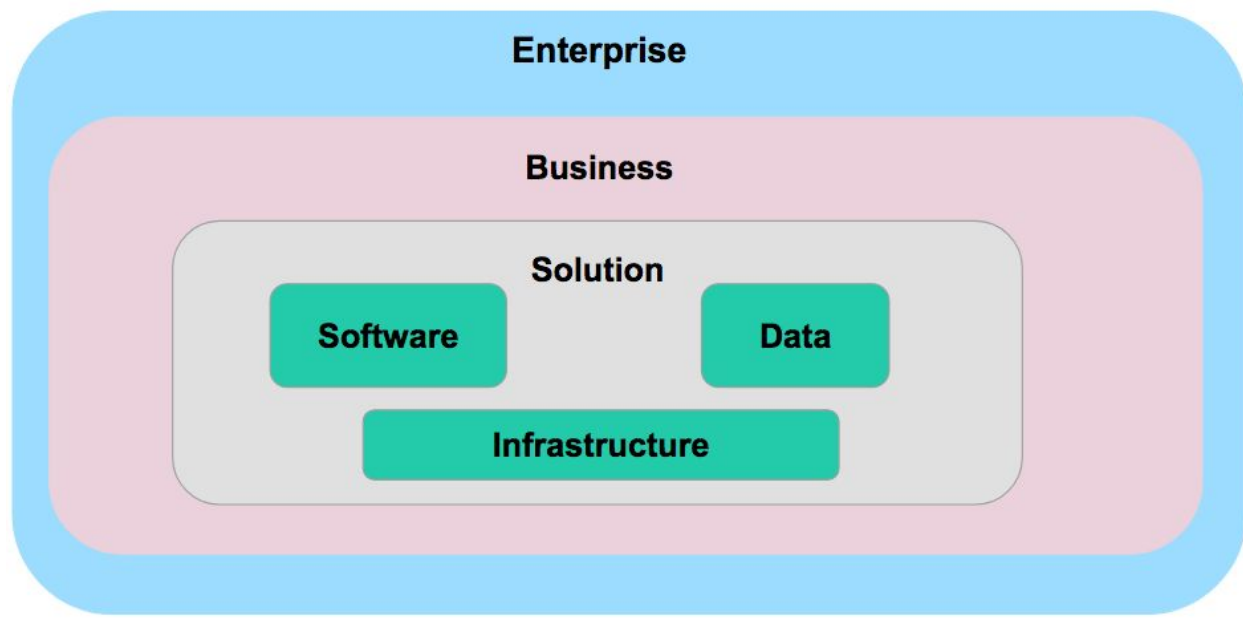
Enterprise Architecture

This is the highest level architecture, and it is all about the organisation as a whole involving planning and managing the structure and strategy of the organisation in all respects - bringing both the business and the IT aspects together. It is primarily concerned with the highest-level, long-term, strategic issues, and it is comprised of human, political, social, software and hardware components.

Inputs To EA

As the scope of EA covers the whole organisation, inputs to this architecture are varied and from all aspects of the business including those from all other architectures – although at this level they are not as specific.

Market And Competitor Analysis	*Financial Data*	*Employee Data*	*Future Projections*
It System Usage And Performance Data	*Materials And Resource Levels/Costs*	*Legal And Governance*	*Strategic Vision*

Zachman Framework

The Zachman Framework is an enterprise ontology[1], not a methodology, and is a fundamental structure for EA which provides a formal and structured way of viewing and defining an enterprise. It progressively describes a system from the top down going from a high level to a more detailed description, and mixes traditional architecture artifacts with detailed design and build artifacts.

---

[1] a set of concepts and categories in a subject area that shows their properties and the relations between them.

	Why	**How**	**What**	**Who**	**Where**	**When**
Contextual	Goal List	Process List	Material List	Organisational Unit & Role List	Geographical Locations List	Event List
Conceptual	Goal Relationship	Process Model	Entity Relationship Model	Relationship Model Of Above	Locations Model	Event Model
Logical	Rules Diagram	Process Diagram	Data Model Diagram	Role Relationship Diagram	Locations Diagram	Event Diagram
Physical	Rules Specification	Process Function Specification	Data Entity Specification	Role Specification	Location Specification	Event Specification
Detailed	Rules Details	Process Details	Data Details	Role Details	Location Details	Event Details

The Levels (Rows)

1. Contextual

What is the domain of discourse[2]?

This is a top level description of the solution system.

Also known as Executive Perspective or Scope Contents, this row usually contains a [Venn diagram](#) to depicted the size, shape, partial relationships and basic purpose of the final structure, and an executive summary to which potential stakeholders or sponsors would use to see an overview of the system, what it would cost, its scope and how it relates to the environment to which it will operate.

2. Conceptual

How might this occur?

This includes the architect drawings that depict the final build from the owners perspective, and correspond to the enterprise models which contain the designs of the business, the business entities and processes and how they relate. This level is also known as the Business Management Perspective or Business Concepts.

3. Logical

What would make sense to build?

---

[2] speak or write authoritatively about a topic.

Also known as Architect Perspective or System Logic, this level includes the architects plans being translated into detail requirement representations from the designers perspective. They correspond to the system model designed by the system analyst who determines the data element, logical process flows, and functions that represent the business entities and processes.

4. Physical

What would it look like?

Also referred to as Engineer Perspective or Technology Physics. The architect's plans are redrawn to represent the builders perspective, with detail to understand the constraints of tools, technology and materials. These plans correspond to the technology models which adapt the information system model to the details of programming languages input and output devices and other supporting technologies.

5. Detailed

What do we actually need to do?

Also known as Technical Perspective or Tool Components. This could include detailed requirements for various COTS, or components of software being procured and implemented rather than built, or it is detailed specifications that will be given to programmers to build and implement the system.

There is a sixth row in the extended Zackman Framework, which is sometimes referred to as Functioning Enterprise, Enterprise Perspective or Operations Instances. This is a view of the functioning system in its operational environment.

Level 1 is the planners view, 2 is the owners, 3 is the designers, 4 is the implementers view, 5 is a sub-constructors view, and 6 is the users view.

The Columns

The columns need to be answered for each perspective:

Why (motivation) are we doing this? Why is the solution the one chosen?

How (function) do we do it? How does the business work (what are the business' processes?)

What (data) is needed? What is the business data, information or objects?

Who (people) will do it? Who are the people that run the business (business units and their hierarchy?)

Where (network) does it need to be done? Where are the businesses operations?

When (time) is it needed? When are the business processes performed (what are the business schedules and workflows?)

Rules

- Columns can be in any order but rows must be in the set order
- Each column has a basic model
    - Each basic model is unique
- Each row represents a distinct view
- Each cell is unique
- Combining the cells of a row should form a complete description from that perspective

Example Framework (Using Extended Model)

	Why	How	What	Who	Where	When
Contextual	List Of Goals : ROI, Efficiency, Functionality, Ect	List Of Types Of Services And Processes The System Carries Out	Basic Description Of System	List Of Responsible Staff And Stakeholders	Locations Of Systems And Resources Used	List Of Events And Cycles Of Activity
Conceptual	Business Plan Related To Development	Physical Data Flow And Business Processes	Entity Relationship Model	Roles And Skills, Access Levels (Security Clearances)	Locations Of Systems And Logistics	Schedule Of Project
Logical	System / Business Rules	Activity Diagrams	Entity Relationship Model	Use Cases For Roles And User Interfaces (UI)	Distributed System Architecture	Dependency Diagram
Physical	Rules Specification	Architecture, Input And Output (I/O) And Algorithms For System	New System Mapped To Older: Registers To Tables, Ect	Security Design, UI & UX	Hardware & Software Requirements, Comms Systems And Apis	Unit Development Timings (Schedule)
Detailed	System Configuration And Rules	System Unit Specification And Design	Document Describing Physical	Detailed Plans For UI, For Roles And	Comms Protocols And System	Implementation / Configuration

Chairbear

	Related To Functions		Storage Design	Associated Data Views	Details (Network Architecture)	Timetable
Functional	System Developed Meeting Goals & Rules	Executable Programs	Resource/ Data Sets Converted From Old System	Trained People For Roles And Responsibilities	System Communications	Resource And System Events

Business Architecture

A subset of EA, business architecture is all about aligning business requirements and operations with the strategic objectives that EA is concerned with - how the organisation's strategic objectives are going to be executed.

Inputs To Business Architecture

As the business architecture is focussed on the business objectives of the organisation, this type of architecture has a narrower set of inputs than EA.

Business Drivers   *Organisational/Structure Model*

Financial Models   *Market/Competitor Analysis*

Solution Architecture

A level down from business architecture, solution architecture is concerned with the structure and behaviour of solutions - solutions typically being systems like software applications and their accompanying processes. These solutions implement the operational business objectives.

Solution architecture therefore is a description of the software, hardware, networks, data, processes and procedures that make up a solution to a business need. And this level can be considered to contain software, data and infrastructure as subsets.

Software Architecture

Software itself has many different architectures and designs, with modern software developments using a multi-tiered approach where the software code itself is split into different components – each with its own purpose and functionality.

Benefits of tiered software architecture are:

- Increased Maintainability & Supportability
- Faster Code Development & Modular, Reusable Code

The 3 tier architecture layers are:

1) User Interface
   This layer is the code for the interface/GUI
2) Business Logic
   This layer is where the processing of UI actions and data is performed according to the business rules
3) Data
   This layer is where the data is stored (e.g. in a database), and code interacts with the stored data

Inputs To Solution Architecture

As solution architecture is focussed on the implementation of business objectives, inputs at this level are related to the needs of the business and its stakeholders.

Business Drivers And Need   *User Requirements*   *Budget And Resources*

Infrastructure Architecture

Also known as technology infrastructure, this architecture is concerned with modelling the hardware elements within an organisation, and the interaction and relationships between them. This includes client devices, servers, all networking and communications equipment, and the location of databases. Without infrastructure architecture, the other higher-level architectures would not be possible, as it is the basis for organisation and system communications, the hosting point for applications, and the connection to external stakeholders and systems.

Inputs To Infrastructure Architecture

These include:

Solution/Software Capacity & Performance Requirements   *User Needs*

Budgets And Resources   *Technologies Available*

Data Modelling

Data modeling is the exploration of data oriented structures, and like other modelling techniques explored further in Deliverables, data models can be used for purposes from high level conceptualization to physical data models. Physical data modeling is conceptually like class modeling, with the goal being to design databases - depicting the tables, the data column in those tables, and the relationship between said tables.

There are 3 types of data models:

- Conceptual

This model defines what the system contains, its purpose is to organise, scope and define business concepts and rules. It is typically created by the business stakeholders and architects.

- Logical

This model defines how the system should be implemented regardless of the database management system(DBMS), its purpose is to develop a technical map of rules and data structures. It is usually created by the architects and business analysts.

- Physical

This model describes how the system will be implemented using a specific DBMS system, the purpose being the actual implementation of the database. This is created by developers and the database administrators.

Interface Design

When designing the software there are two main ways to model the interface design, these are wireframing and [prototyping](#).

A wireframe is a visual layout of how a user interface (UI) should look. They include basic information such as the position of elements like images or forms, and can be anything from a rough sketch to a detailed plan.

Design Principles

Good design should follow these principles:

No Wasted Space, But No Cluttering Either	*Don't Give An Unnecessarily Large Number Of Options (Hick's Law[3])*
Use Of Appropriate Colors (Psychology Of Colours[4])	*Make Controls Large, And Easy To Reach (Fitts' Law[5])*

Design Constraints

There are obvious constraints such as performance capabilities of the hardware, time constraints and staffing limitations. However there are 3 main restrictions that mold a system design, these are:

---

[3]
https://www.interaction-design.org/literature/article/hick-s-law-making-the-choice-easier-for-users

[4] https://en.wikipedia.org/wiki/Color_psychology

[5] https://www.interaction-design.org/literature/topics/fitts-law

Legal

These are constraints on development enforced by law. One example would be 2018s Data Protection Act (GDPR).

Ethical

Ethical refers to moral values and considering what is generally thought to be good practise. One such 'good practise' is developing with the public interest in mind.

Financial

This means there will be a budget to stick to. Things that fall under finances include staff and resource cost. Often cost/benefit analysis is performed, and expenditure monitored at particular development phases (depending on methodology used).

An Example Of Constraints In Practise

A business is developing some email marketing software that allows users to make a marketing email using a template and send it to their emailing list. Funds are running a little low, and so the product is packaged and sold without fully being complete - an email unsubscribe button is left missing out of the templates.

The **ethical issue** here is that the company knowingly is selling an uncompleted product but still selling it as having all the bells and whistles. The **legal issue** is that email recipients have no way of unsubscribing from the emails.

Building In Security

Design also incorporates consideration of quality, security and constraints. Often referred to as 'building security in.'

This consideration leads to secure development. Which is a practice to ensure that the code and processes that go into developing applications are as secure as possible. It includes:

- Threat modeling

  This asks what parts of the software are easily compromised, and what is the impact?

  Threats will be prioritised and if a threat cannot be mitigated with security control the design is changed or the risk is assumed.

- Testing

  This can be static code analysis in which code is analysed without running, or testing the software against a modelled threat.

Security VS Resilience

Security is the pro-active protection against threats, for example firewalls and antivirus software, whilst resilience (also known as threat mitigation) is the reactive countermeasures that limit the damage an attack has on a system and infrastructure, for example system backups.

Common Security Attacks

Cross Site Scripting (XSS)

Typically found in web applications, XSS enables attackers to inject malicious client side scripts into web pages viewed by other users. A XSS vulnerability may be used by attackers to bypass access controls such as the same origin policy.

Example

1) An attacker discovers a XSS vulnerability on a website

   This is usually a user input field that has not been properly configured and restricted which enables attackers to place scripting markup within the input area and run it.

   Preventing this is dependent on the language behind the website.

   For example, with PHP you can use PHP's htmlspecialchars() function to prevent this.

   Whilst using htmlspecialchars() if a user tried to submit the following in a text field:

   <script>something</script>

   It would not be executed as it would be saved in HTML escaped code like this:

   &lt;script&gt;something&lt;/script&gt;

2) The attacker then injects a malicious script that steals each visitor's session cookies - which will say something to the computer akin to 'this is a valid user'
3) The attacker can then use these cookies to gain access to whatever privileges the original user has. If they were admin, for example, the attacker would then have access to any important or sensitive information the administrator had access to

Cross Site Request Forgery

Unlike XSS which exploits the trust a user has for a site, CSRF exploits the trust a site has in a users browser. CSRF is a type of malicious exploitation of a site where unauthorised commands are transmitted from a user that the application trusts, without the user's interaction or knowledge.

Example

1) Jem receives an email from a stranger that asks if they have seen the most recent news about a breaking news robbery in their country. The email contains a link that appears to go to a news site

2) Jem clicks the link
3) The link itself now attempts to access funds in Jems bank account, and since the request comes from Jems computer and it has cookies authenticating them the link has access and successfully transfers funds to another bank account

SQL Injection

An infamous type of attack, SQL injection is a code injection technique similar to Cross Site Scripting in which nefarious SQL statements are inserted into entry fields to attack data driven applications. For example, to send all the database contents to the attacker.

Example

A basic login box:

Username:	
Password:	

 The SQL Injection:

Username:	` OR 1=1;/*
Password:	*/--

What this means to the database:

SELECT * FROM Users WHERE Username='` OR 1=1;/* AND Password='*/--'

Translation:

Select all from the Users table where Username is equal to nothing or equal to true (to the database, if a true value is passed it means 'a value was found'), and Password is cancelled out as the */-- turns the rest of the statement into a comment.

An easy fix is to prevent the input boxes from accepting special characters.

Denial Of Service Attack (DoS)

In a DoS attack, the attacker aims to make a machine or network resource unavailable to its intended users by either temporarily or permanently disrupting the services of a host. It is typically accomplished by flooding the targeted machine or source with a staggering amount of requests in an attempt to overload the system, which will prevent legitimate requests from being fulfilled.

There are also attacks that focus on manipulating a person into divulging information or access to a computer system, this is social engineering.

Below are a few examples of this.

Phishing Emails

This involves getting users to divulge information by tricking them with fictitious emails.

Whale phishing includes targeting a more profitable user (someone who is important or wealthier than a standard target). A juicer target if you will. For example, a CEO of a company instead of a receptionist.

Spear phishing includes using user specific emails, to make users feel it is more reliable. Such as using your name.

Pretexting

This includes using a plausible scenario to gain people's trust.

Water Hole Attack

This includes targeting social websites or online gathering places for a particular sort of person.

As the attacker knows the general subject or topic being discussed here, they can find it easier to coin their attacks by playing to the users weaknesses.

For example, a football fan forum, and the attacker posting a message saying they need money to support their son's football team. They could follow the message up with a plea for other football fans to send money to support the non existing team.

Development

Is the phase where technical components[6] are created, procured or configured.

To build in security developers must work with testers and security experts to better understand their code from an attackers point of view of view. They should also train in secure coding practices and keep up-to-date with the latest cyber attacks and vulnerabilities. Any software that requires user input should also be validated.

Proactive Security Approaches

3 examples of proactive approaches are:

Defensive Programming

Programming that incorporates exception handling, validation and other security features. The worst case scenario is imagined, and then coding measures put in place to try to prevent it.

Permission Settings

---

[6] A part or element of a larger whole. *Analogy - a single puzzle piece that connects with others to make a whole picture.*

This incorporates creating permissions for access to files and data, and setting different user access levels.

<u>Physical Infrastructure</u>

This includes utilizing physical measures of protection such as biometrics, CCTV and locks.

Prototyping

Is the activity of creating early sample versions of software. Typically simulating only a few aspects of the final product. There are two models for prototyping, throwaway model and evolutionary model.

Strengths

- Confusing, missing or difficult functions can be identified
- Errors can be detected early

Weaknesses

- Leads to 'implement & then repair' way of working
- May increase complexity of the system as scope may expand beyond original

- Throwaway Model

  This prototype is thrown away once it has been reviewed. They are quick to produce, and often visually resemble the final product but have no to little functionality

- Evolutionary Model

  This prototype evolves into the final product. The main goal is to build a very robust prototype in a structured manner and then refine it. The prototype forms the heart of the new system and the improvements and further requirements will then be built

Testing

Components are tested to ensure they meet requirements. Levels of testing include unit, component, system and acceptance testing.

Unit Testing

Is a development process in which the smallest testable parts of an application, called units, are individually and independently scrutinised for proper operation.

Unit tests are usually written and performed by developers to ensure code meets its design and behaves as expected. Unit testing should be done before integration testing[7], and can be done via black or white box testing methods.

Component Testing

Similar to Unit testing, component testing is performed on each individual component separately without integrating with other components.

System Testing

Is conducted on a complete, integrated system to evaluate the system's compliance with its specified requirements.

Often carried out by independent or specialist testers.

Acceptance Testing

The purpose of this testing level is to evaluate the system's compliance with the business requirements and assess whether it is acceptable for delivery.

Acceptance testing is done by the user/customer.

The General Testing Process

1. Planning

   This establishes the the scope of testing, and acceptance/exit criteria

2. Analysis & Design

   This is the designing of the test cases

3. Implementation & Execution
4. Evaluation

   Assessing if the testing is complete and reporting of the results

Exit Criteria

Exit criteria is used to determine whether a given test activity has been completed or not. For example, verifying that all tests planned have been run, if so then this exit criteria has been met. When criteria is met, a test summary report will be generated and shared with stakeholders. From this the decision may be made to go onto a new test level or finalise this as the last testing stage.

Why Test Early In Development

---

[7] Testing of combined components to determine if they function correctly.

Problems that are introduced into the system during design or planning can be caught, and the requirements testing can anticipate future problems at a lower cost.

Quality will also be ensured. Test cases written during requirements gathering can be shared with developers and allow them to evaluate more potential problems, reducing the risk of failure in the code.

Finding Defects

Techniques to find defects can be divided into 3 categories:

Static Techniques

Testing is done without physically executing a program.

Dynamic Techniques

This testing includes physically executing system components to identify defects.

An example of this would be executing a test case.

Software Test Cases are a specification of the inputs, execution conditions, testing procedure and expected results that define a single test to be executed to achieve a particular testing objective.

Operational Techniques

This is where an operation system produces a deliverable containing a defect found by a user. The defect is found as a result of a failure.

Defects can be identified at any stage and by anyone in the development process. In order to ensure a quality product, a defect should be well documented. A report should contain:

- A summary
- Description
- Platform or build
- Steps to reproduce
- Expected result
- Actual result

Testing VS Debugging

Testing is the process of looking for errors whilst debugging is the process of correcting the error.

Black Box & White Box Testing

White Box Testing involves checking the way that things happen inside the system, AKA, looking at the code, meaning it is typically a static technique.

Black Box Testing on the other hand involves checking that the output is as expected based on the input.

Example

Black Box

Insert a coin into a drinks machine, and enter the drink code. If the drink is vended correctly the test was successful.

White Box

Open the drinks machine up and check that the money went along the correct verification pathway, and the mixers for the drink were added in the expected sequence.

Performance, Load & Stress Testing

Performance testing is a type of software testing that intends to determine how a system performs in terms of responsiveness and stability under a certain load. Load in this instance being how many transitions a system is performing in a given time.

Stress testing is a test where the load given to a system is at or beyond that which it is expected to be able to work with under normal conditions.

Quality Assurance

There are other mechanisms outside of tests to ensure quality[8]. We will explore 4 below.

Informal Reviews

These are often in the form of peer reviews, such as a developer reviewing the code produced by another developer. A quick and cheap type of review, that helps identify any obvious errors - two pairs of eyes are better than one. However, the review may also not pick up on some errors, as this review relies on humans; it is always open to human error.

Walkthrough

In this, the product is walked through line by line or section by section to identify errors. Often particular scenarios are used to mimic live situations.

The review can be done by an individual or group, and is more likely to identify errors than an informal review particularly if it is done by a group.

Technical Reviews

In this, the product is reviewed by a group of technical experts who usually use checklists of common problems to identify errors.

---

[8] In this instance quality can be defined as a piece of software that meets functional and nonfunctional requirements.

Inspection

This is a formal meeting, with clearly defined roles and objectives. Roles include:

Inspectors

One or more inspectors review the product in detail.

Scribe

The scribe records all errors, actions and decisions.

Moderator

The moderator is responsible for ensuring that the product is reviewed in a structured manner, conflicts are managed appropriately and all points of view are accounted for.

Implementation

Is moving into the live environment. It takes careful planning, and is often conducted by software release engineers. There are many different implementation methods such as installing on a user's machine remotely, uploading to an app store, distributing via email and so on.

Generic Implementation Types

Pilot

The new software is placed on a limited number of machines, and accessible to a small group of users prior to wider use.

Strengths

- Failures can be identified and fixed without widespread issue to the business
- Users have time to get familiar with the new system, and training can be done in pilot groups

Weaknesses

- Scale issues may not be detected, as a system may work well for 10 users but not for 200

Parallel

This is where the new system is used at the same time as the older system.

Strengths

- If there is any issues with the new system, they can be ironed out within a reasonable time instead of rushing as the old system will still be working

- Users can compare the outputs of both systems to ensure they are correct
- Users have time to adjust to the new system

Weaknesses

- Users must take more time to enter data into two systems
- There is a risk of data loss as both systems run in parallel

Phased

This is when small parts of the new system gradually replace the original system.

Strengths

- Training can also be delivered incrementally
- A failure in the new system is less detrimental as it is only a small part of the system

Weaknesses

- Risk of data loss if new system fails
- This method take a long time to implement the new system

Big Bang/Direct

Is the act of implementing the new system without any phased or pilot implementation. The old system is retired and the new system set live.

Strengths

- Only a good method of implementation for non critical systems

Weaknesses

- Risk of data loss if new system fails
- Users do not have time to get acquainted with the new system

Sign-off occurs when the product is handed to the customer and they acknowledge receipt of the product and completion of the project. This usually includes signing a legally binding document, to ensure the customer cannot claim to not have agreed or received the product and the developing party may receive payment. Typically sign off occurs before deployment.

Post implementation reviews(PIR) are conducted at the end of a development. They evaluate the project as a whole, to determine what went well, what didn't and potential improvements for next time. They can include all members of the project team.

An ideal time to do a PIR is shortly after the project has been delivered, and when most of the problems have been addressed, as the project is still fresh in everyone's minds.
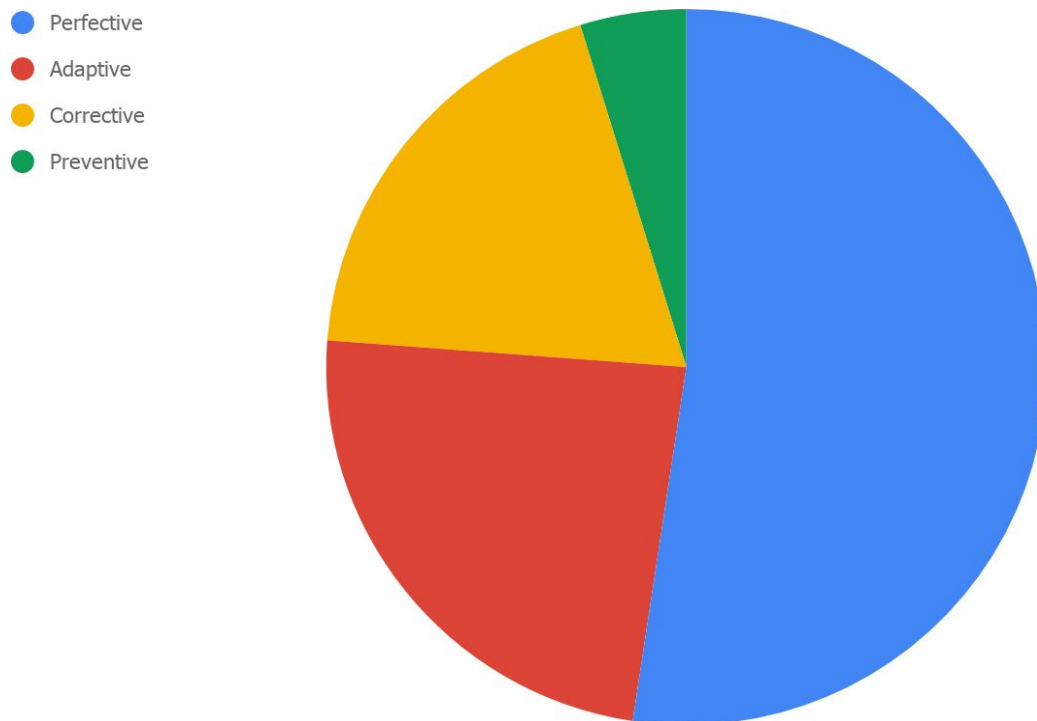
Maintenance

The development lifecycle is not called a cycle for nothing. The development of a system doesn't end when it is released, but rather continues as the system is updated, faults are identified and fixed and improvements are made until, eventually, the system is decommissioned. This is the maintenance stage. Throughout this period, the system needs maintenance and support such as backup, patching, and audits.

Support is to help users such as providing training, tips and helping solve issues whilst maintenance is about keeping the software product working effectively.

Maintenance processes are owned by the IT department, and based upon predefined rules and tasks.

Types Of Software Maintenance

- Perfective
- Adaptive
- Corrective
- Preventive

Perfective Change

50-55% of most maintenance work is attributed to perfective changes.

Perfective changes refers to the evolution of requirements and features to an existing system. This is because as a software gets exposed to users they will think of different ways to expand the system or suggest new features that they would like to see as part of the software, which in turn can become future enhancements to the system.

Perfective changes also include removing features from a system that are not effective or functional to the end goal of the system.

Adaptive Change

20-25% of maintenance work is attributed to adaptive changes.

Adaptive change is triggered by changes in the environment the software lives in such as changes to the operating system, hardware, software dependencies or even organizational business rules and policies. These modifications to the environment can trigger a domino effect of changes.

Corrective

20% of maintenance work is attributed to corrective changes.

Corrective changes address errors and faults in a software, most commonly, these changes are sprung by bug reports created by users. However, be mindful that sometimes problem reports submitted by users are actually enhancements of the system, not bugs.

Preventive Change

5% of maintenance work is attributed to preventive changes.

Preventive changes refer to changes made focusing on decreasing the deterioration of a software in the long run. Restructuring and optimizing code and updating documentation are common preventive changes.

Executing preventive changes reduces the amount of unpredictable effects a software can have and helps it become scalable, stable, understandable and maintainable.

Types Of Software Support

1st Line Support

An end user reports a problem, 1st line support technicians will log the issue and assign a unique reference number to the issue case. 1st line supporters may try to resolve the issue, however if they cannot they hand it over to 2nd line support.

2nd Line Support

2nd line support are more technically knowledgeable, and will try to resolve the issue. If they cannot fix it, it is escalated to 3rd line support.

3rd Line Support

3rd line supporters are typically developers who know a software system. They will almost always solve the issue.

Roles Of SDLC

Business Roles:

Sponsor

A person or organization that pays for or contributes to the cost of the project.

Senior Responsible Owner

Accountable for a project meeting its objectives.

Business Analyst (BA)

Someone who analyzes a business and documents its process and systems, assessing its business model and/or its integration with technology.

- Can be involved in all phases of the SDLC but main responsibilities are:
  - Requirements analysis, capture & tracking
  - Resource estimation and planning
  - Separation of functional and non-functional requirements

Domain Expert

A person who is an authority/very knowledgeable in a particular area/topic.

End User

A person who actually uses a particular product or service.

Project Roles:

Project Manager

A role dedicated to the planning, scheduling, resource allocation, execution, tracking and delivery of a software product.

- Can be involved in all stages of SDLC
- They are the single point of contact (poc) for development teams and the client
- Ensure SDLC standards are upheld, such as ISO or CMMI
  - The International Standards Organization (ISO) and Capability Maturity Model Integration (CMMI) are two standard ratings and guidelines given to organizations that develop software

Team Leader

Someone who provides guidance, instruction, direction and leadership.

Work Package Manager

A work package is a group of related tasks within a project, they are often thought of as sub-projects within a larger one. Work packages are the smallest unit of work that a project can be broken down into. A Package Manager manages these 'sub-projects'.

Technical Roles:

Technical Architect

Responsible for defining the overall structure of a program or system. They act as a project manager, overseeing IT assignments that are aimed at improving the business and ensuring the project runs smoothly.

Designer

Is responsible for designing a software model that fulfils the specifications, usually by using diagramming tools. They are also responsible for documenting the design and choosing the system architecture.

Solution Developer

Creates the computer products needed in order to accomplish particular goals, ranging from custom software to graphical design. Customer needs must be determined by constant communication.

Solution Tester

Tester of above. They will perform and record testing (black-box and white-box testing), and be responsible for quality assurance by using static and dynamic analysis tools such as walkthroughs, automated scripts and performance tools.

Implementation And Support Roles:

Release Manager

Release management is the process of managing, planning, scheduling and controlling a software build through different stages and environments, including testing and deployment.

Database Administrator

This role may include: capacity planning, installation and configuration, database design, migration,security as well as backup and data recovery.

System Administration

Is responsible for upkeep, configuration and reliable operation of computer systems.

Team Working & Structure

Effective team working requires:

- Communication
- Composition

  Team composition involves putting together the right set of individuals with relevant expertise to accomplish the team goals and tasks, to maximize team effectiveness

- Maturity

  Maturity is not just about age, but how someone is mindful of others before they say or act

There are 3 team structures you should be aware of:

Functional Structure
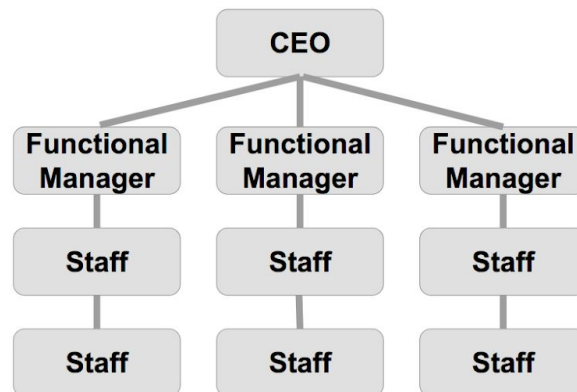
In a functional structure, employees are grouped based on their specific skills and knowledge, with each department structured vertically.

Strengths

- Streamlined Communication
  - With one manager to answer to, employees know who to go and confusion is reduced
- Quality Products
  - Employees who have similar skills and experiences are grouped together, reducing time for learning, and production more efficient resulting in higher quality products

Weaknesses

- Siloing & Rigid Structure

- ○ Employees may lack knowledge about other departments, especially when managers are only concerned with their own teams and do not communicate with each other, which leads to siloing, and in turn make changes, and flexibility hard to handle
- Repetition & Enthusiasm
  - ○ Employees may find it boring to repeat the same task over and over, resulting in less enthusiasm over time

Project Structure

In a project structure, employees are grouped together for a project which results in a mix of skills and resources, with a project manager in charge. When a project is complete, the teams may be re-organized.
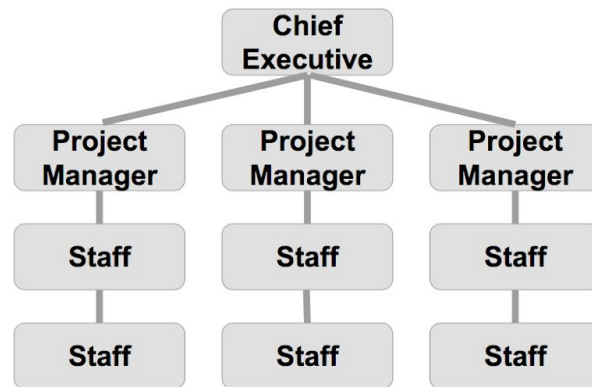
Strengths

- Loyalty & Quality
  - ○ Project structure is the easiest in which to create a strong team culture, which encourages collaboration, enthusiasm and as the whole team is focused on the team's goals, it can result in a high quality product
- Project & Team Control
  - ○ Stronger control over the team, and therefore project, as the team respond to one project manager

Weaknesses

- Job Security
  - ○ Sometimes closing a project can mean job losses, as the business may not have another role available for certain employees
- Resources & Expenses
  - ○ Having a team dedicated to one project is an expensive commitment, especially as they accumulate resources to work on a singular project. This can limit the number of projects a company can do at any one time, especially when different projects require the same skills

Matrix Structure

The matrix structure is a combination of the project and functional structures, which strengthens the strengths of each and tackles the weaknesses. Employees may report to their functional manager but also to their project manager for the duration of a project.

Strengths

- Flexibility & Communication
  - The matrix structure allows the sharing of highly skilled people and resources between functional units and projects, this means communication are open and knowledge can be shared
- Job Security
  - With the matrix structure, employees can widen their experience and skill sets, as they can be apart of various projects which put them in an environment that facilitates learning and gives them an opportunity to grow professionally
  - Teams remain loyal because the structure provides a more stable environment where job security is strengthened

Weaknesses

- Manager Conflict
  - Since the matrix structure results in the team having two managers, there can be confusion on who to communicate certain aspects of the project to, and if the managers do not communicate effectively with each other or disagree on particular aspects, this can result in unnecessary problems
- Cost & Burnout
  - Having multiple people in managerial positions can be costly
  - Team members workload can be heavy, as they're often tasked with their regular assignments and then additional project work, which can lead to burnout or some tasks being ignored

Deliverables Of SDLC

Many deliverables are a way of detailing what is understood about the system in terms of what it needs to do, how it should do it and how it gets delivered.

Deliverables include:

- Requirement Documents

  (PRD) is a document containing all of the requirements of a product. It is written to give an understanding of what a product should do.

- Class/Entity Relationship Models

- ○ [View here](#)
- Use Case Diagrams
  - ○ [View here](#)
- Process Models ei V-model/Waterfall
  - ○ [View here](#)
- Component Diagrams
  - ○ [View here](#)
- State Transition Diagrams
  - ○ [View here](#)
- Sequence Diagrams
  - ○ [View here](#)
- Activity Diagrams
  - ○ [View here](#)
- Data Flow Diagrams
  - ○ [View here](#)
- Test Plans
- Test Scripts
- Implementation Plans
- System Components And Working Software

Stage	Deliverables	People Involved in Stage
Requirements Engineering	- Business Requirement Documentation (Brd) - Software Requirement Specifications (Srs) - Technical Requirement Specifications	Top Level Management Of The Business Are Involved In Gathering The Requirements Of The Software: - Project Managers - Directors - Sales, Marketing & Consulting - Stakeholders - Business Analyst
Design	- Use Case Diagram - Class Diagram - Entity Relationship Diagram - Component Diagram	- Designers - Business Analysts - Architects
Development	- Functional Code	- Developers
Testing	- Test Plans - Test Scripts	- Testers
Implementation	- Working Product	- Release Managers - Developers

Diagram Deliverables

A model is a representation of a subject, it can be a real-world object, imagined subjects (such as a video game spaceship) or a system that either already exists or may exist in the future.

There are many modelling notations and frameworks in existence, but the following are ones you need to be aware of for the BCS Software Development Essentials exam.

The UML models below can be split further into 2 types:

Behavioural Diagrams	Structural Diagrams
Use Case	Class / Entity
Sequence	Component
State Machine (State Transition)	
Activity Diagram	

Entity Relationship Models

ERM is a theoretical and conceptual way of showing data relationships. Most often used to design or debug relational databases.

ER diagrams are composed of entities, relationships (cardinality- which defines relationships in terms of numbers) and attributes.

If you've ever worked with Databases and/or SQL you will find this model very easy to understand.
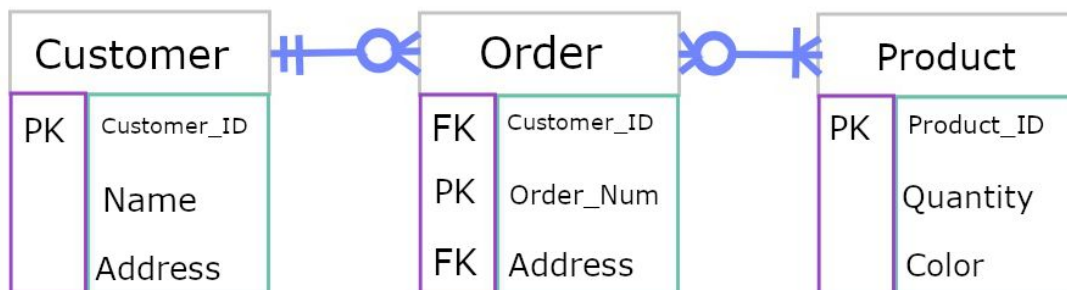
Entities

A definable thing such as a person, object or event that can have data stored about it. Think of them as nouns[9], such as a customer or product. They are shown as rectangles.

Each entity (type) represents an individual table in a database. And in the database, the entities will be the rows whilst the attributes are the columns.

Entity Categories

The available categories for entities are strong, weak or associative.

A strong entity can be defined solely by its own attributes, whilst a weak cannot - it is an entity that depends on other entities as it doesn't have any key attribute of its own.

Associative associates entities within an entity set.

Entity Type & Set

An entity type is a group of definable things such as customers or students whilst an entity is a specific student or customer.

An entity set is the same as entity type, but defined at a particular point in time. Such as customers who purchased in the last 3 months.

Entity Keys (Optional)

Highlighted in the diagram example above as purple.

Entity keys refer to an attribute that uniquely defines an entity.

- Primary Keys *(shown above as PK)*
  - A candidate key chosen by the database designer to uniquely identify the entity
- Foreign Keys *(shown above as FK)*
  - Identifies the relationship between entities - its a term used for an attribute that is the primary key of another table
- Super Keys
  - A attribute or set of attributes that uniquely identifies an entity - there can be many of these.
  - In layman terms this is any combination of attributes that uniquely identify a row/entity where no combination is the same.
- Candidate Keys
  - Otherwise known as a minimal super key, it has the least possible number of attributes to still be a super key

---

[9] A word used to identify any of a class of people, places or things.

Super & Candidate Key Example

book_ID	name	author
1	Learn CSS	Chairbear
2	Learn JavaScript	Chairbear
3	Learn HTML	Chairbear
4	Learn Web Development	Chairbear

The combination of *name* and *author* is a superkey, as none of the 2 value combinations are the same.

[Learn CSS, Chairbear] is different from [Learn JavaScript,Chairbear]

name	author
Learn CSS	Chairbear
Learn JavaScript	Chairbear

If ID 4 had the *name* of Chairbear and *author* of Chairbear, this would then not be a unique combination as both attributes have the same value, and so name and author would then not be super keys.

book_ID	name	author
3	Learn HTML	Chairbear
4	Chairbear	Chairbear

Going back to our original table, book_ID is a superkey as its values are unique - no two values combined are the same.

book_ID
1
2
3
4

If we say no two combinations of *name,book_ID* and *author* are the same, we can say this is a super key.

[1,Learn CSS, Chairbear] is different from [2,Learn JavaScript,Chairbear]

However, when working with relational databases, if you combine all of the attributes like above they will of course make a super key. As no two rows in a database table will be exactly the same. This is called redundancy. Some super keys, like the one above, will have redundancies.

A candidate key is a super key without redundancy, and it is not further reducible. We can see that book_ID is unique throughout, and so there is no reason to say that the combination of *name,book_ID* and *author* is unique, as name and author are made redundant in that combination. book_ID is a candidate key.

Attributes

A property or characteristic of an entity. Shown as an oval or circle.

Attributes Categories

The available categories are simple, composite, derived, multi and single.

A simple attribute is an attribute value that is atomic, and cannot be further divided. Such as a phone number.

Composite, is a sub attribute composed of many other attributes. For example, an address attribute could be composed from street, zipcode and country attributes. A composite attribute is represented by an oval with other ovals branching from it (behind it).

Derived attributes are calculated/derived from another attribute such as age from birthdate.

Multi value attributes, are attributes where the value is or can be multiple, such as multiple phone numbers for one person.

Single value attributes contain just one value.

Cardinality And Relationships

Relationships are how entities act upon each other/ are associated. Think of them as verbs[10].

---

[10] A word used to describe an action, state or occurrence.

For example, a student enrolling for a course. The 2 entities are the course and the student, and the relationship would be the act of enrolling.
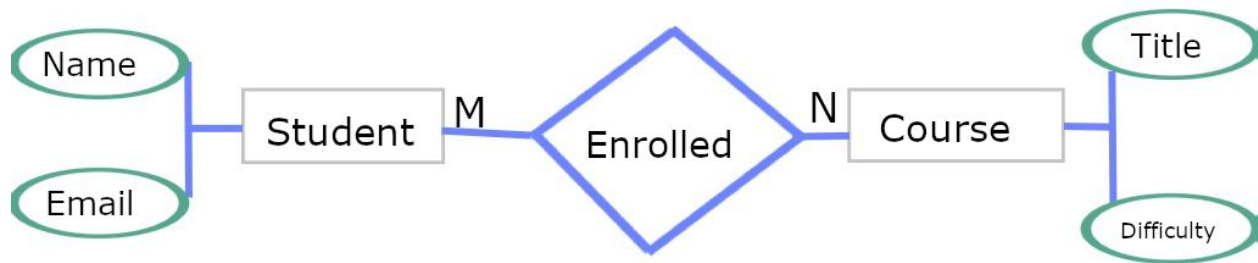

Relationships are shown as diamonds or labels on connecting lines.
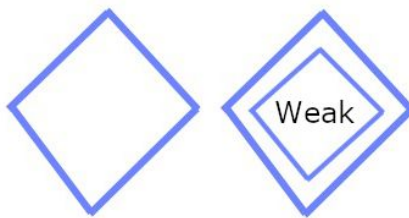
A recursive relationship is where the same entity participates more than once in a relationship.

Cardinality defines the numerical value of the relationship between entities. The main cardinalities you'll see are:

- One To One
  - For example, 1 student to 1 email address
- One To Many/ Many To One
  - For example, 1 student enrolled in multiple courses
- Many To Many
  - For example, students as a group are associated with multiple teachers
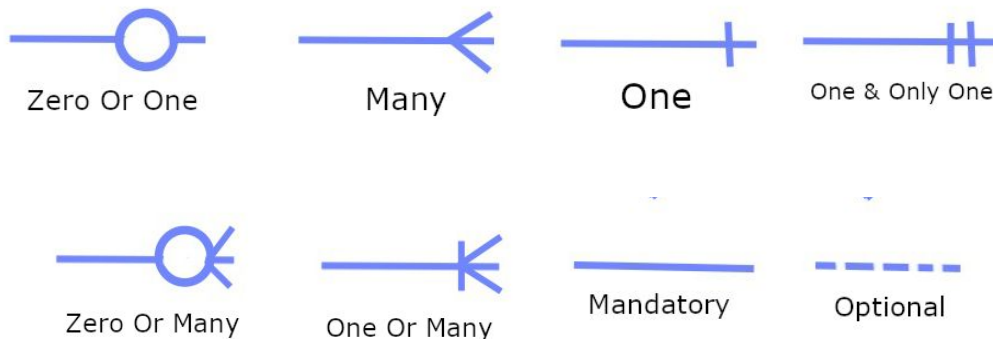
A mandatory relationship will be shown with a solid line, whilst an optional relationship will have a dotted line.

First Example Explained - Working Out Cardinality
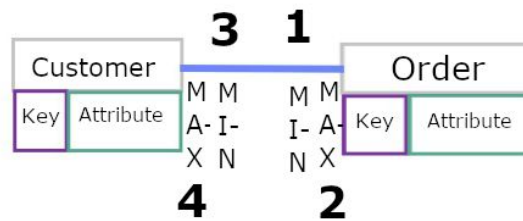
1) <u>What is the MINIMUM number of orders a customer can have?</u>

   a) A customer can exist without any orders. The answer is 0.

2) <u>What is the MAXIMUM number of orders a customer can have?</u>

   a) A customer can exist with many (infinite) orders. The answer is many.

3) <u>What is the MINIMUM number of customers an order can have?</u>

   a) An order needs a minimum of 1 customer to exist. The answer is 1.

4) <u>What is the MAXIMUM number of customers an order can have?</u>

   a) An order can only have 1 customer, as one order having multiple customers would be confusing. The answer is 1.

Second Example Explained
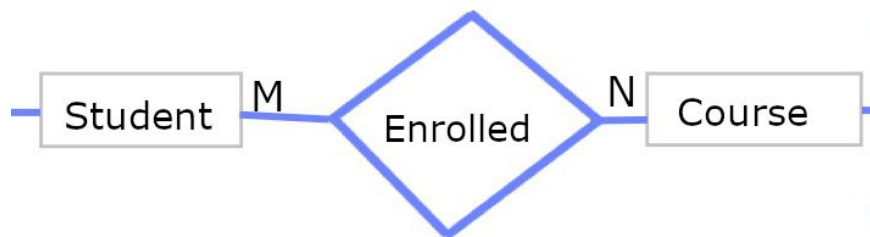
The M and N represent any integer[11], meaning any number of students can study any number of courses. M&N are variables. If it were M --- 1 it would suggest a many to one relationship.

Class Relationship Models

A class relationship model is a static structure diagram that represents the structure of a system through showing its classes, their attributes and methods, and the relationship between them.

A class, in this instance then, is a group of objects all with similar roles in the system.

In class relationship models, the class shape is a rectangle with three rows.

---

[11] A whole number
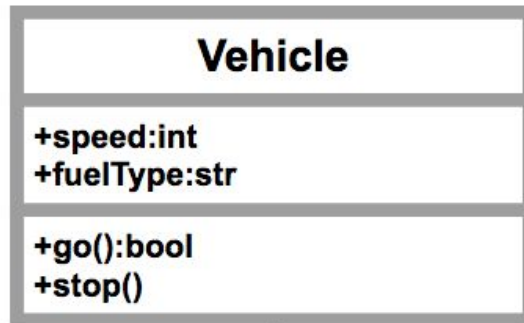
The top row contains the name of the class. This section is always required.

The middle row contains the attributes of the class, each attribute takes up a line. The attribute type is shown after the colon. Attributes (also known as fields, properties or variables) map onto member variables[12] in code.

The bottom section expresses the methods (also called operations) that the class may use, and each operation takes up its own line. The operations describe how a class interacts with data, and map onto class methods in code.

Similar to attributes, the return type of a method is shown after a colon. And if the method contains parameters within it's parentheses, the type will be after the parameter, separated by a colon.

Member Access Modifiers

Classes can have different access levels depending on the access modifier, otherwise known as it's visibility. The visibility of an attribute or method sets the accessibility of such.

Public (+)

An attribute or method is public if the + (plus) sign is before it. This means it can be accessed by another class.

Private (-)

An attribute or method is private if the - (minus) sign is before it. This means they cannot be accessed by any other class or subclass.

---

[12] In object-oriented programming, a member variable is a variable that is associated with a specific object, and accessible for all its methods. For example (This is a Java Example):
public class IntegerClass {
    int anInteger;
    }
The above declares an *integer member variable* named anInteger within the class, IntegerClass.

Resources: http://journals.ecs.soton.ac.uk/java/tutorial/java/javaOO/variables.html
https://en.wikipedia.org/wiki/Member_variable

Protected (#)

An attribute or method is protected if the # (hash) sign is before it. These can only be accessed by the same class or it's subclasses

Package (~)

 If the ~ (tilde) sign is before an attribute or method, this means the visibility is set so they can be used by any other class as long as it's in the same package

Relationships

A class may be involved in one or more relationships with other classes. There are 5 types of relationship: inheritance, simple association, aggregation, composition and dependency.

Inheritance (also known as generalization)

Is the process of a child (or sub-class) taking on the functionality of a parent (or superclass)

The are represented with a solid line with a hollow arrowhead that points from the child to the parent class.
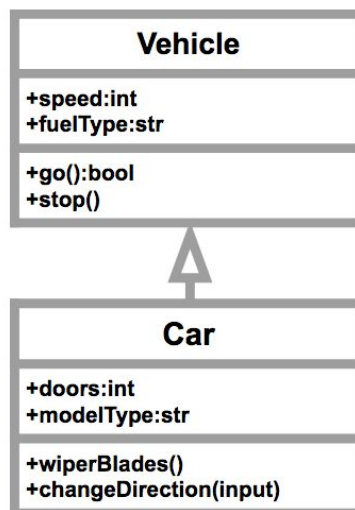
In the above example, the 'car' object would inherit all of the attributes (speed and fuelType) and methods (go() and stop()) of the parent class 'vehicle', in addition to the specific attributes and methods of its own class (such as doors, or changeDirection()).

Association

In association, both classes are aware of each other and their relationship. There is no dependency, it's just a basic association which is depicted by a simple line.

Aggregation

This is a type of association that specifies a whole and its parts - if a part can exist outside of a whole it is an aggregation relationship. This is represented by an open diamond. The unfilled diamond is at the association end.
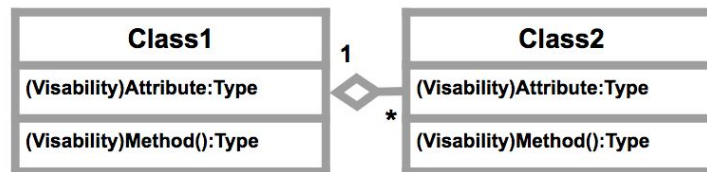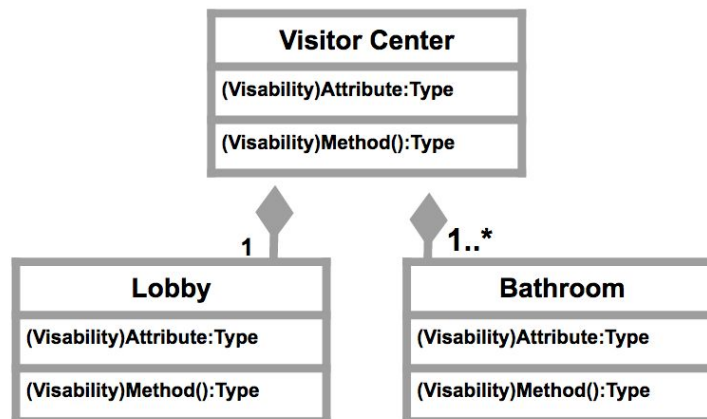

Example Explained

Class2 is a part of class1. Many instances of class2 can be associated with class1, however objects of class1 and 2 have separate lifetimes.

Composition

A special type of aggregation where parts are destroyed when the whole is destroyed. Depicted by a solid line with a filled diamond at the association end.

Example Explained

If the visitor centre is torn down, the lobby and bathrooms would also cease to exist.

Dependency

This exists between two classes if changes to the definition of one may cause changes to the other, but not visa versa. Represented by a dashed line with an arrow, from dependent to independent.
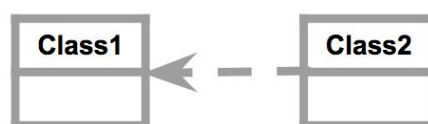
Example Explained

Class2 is dependent on class1.

Relationship Names

Written in the middle of the association line. Good names make sense when you read them
outloud.

Example Explained

Every spreadsheet contains some number of cells.

Multiplicity

Symbol	Definition
0..1	Zero To One
n	A Specific Amount (in the visitor example, it uses 1 for the lobby meaning it can only have one lobby, but can have one or many bathrooms)
0..*	Zero To Many
1..*	One To Many
M..N	A Specific Number Range

Use Case Diagrams

In software development, a use case is a list of actions or event steps typically defining the
interactions between a role (in the Unified Modeling Language(UML) this is called an actor) and
a system to achieve a goal. The actor can be human or an external system.

Common components in a use case include:

- Actors
  - The user that interacts with a system. They must be external objects that
    produce or consume data
- System

○ A specific sequence of actions and interactions between actors and the system. AKA a scenario
● Goals
○ The end result of a use case. A successful diagram should describe activities used to reach the goal

Use Case Notation

Use Cases

These are horizontal ovals that represent the different uses a user may have with a system.

Actors

These are represented by stick figures, that represent the people employing the use case. Always drawn outside of the system boundary box. There are two types of actors, primary and secondary.

Primary actors initiate the use of a system, and are always on the left side of a system boundary box. Secondary actors are reactionary, and always on the right side.

Associations

Associations are represented with lines between actors and use cases.

System Boundary Boxes

A box that sets a system scope to use cases. All use cases outside a box would be considered outside the scope for that system.

Include Relationships

Shows a dependency between a base use case and an include case. Both cases will be executed. Think of it as a base case requiring an include case in order to be complete. IRs are depicted by a dashed line, with an arrow that points at the include case, with 'include' written between double chevrons.

Extends Relationships

When a base use case is executed, the extend use case will happen only when certain criteria are met. ERs are depicted by a dashed line, with an arrow that points at the base case, with 'extends' written between double chevrons.

Generalisation Relationships

Specialised uses cases point at a generalised use case. They represent more options or categories beneath the general case. And are depicted with solid lines and hollow arrows.

Packages

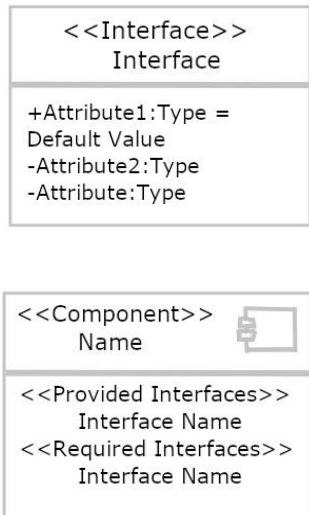A UML shape that allows you to put different elements into groups, these groupings are represented as file folders.
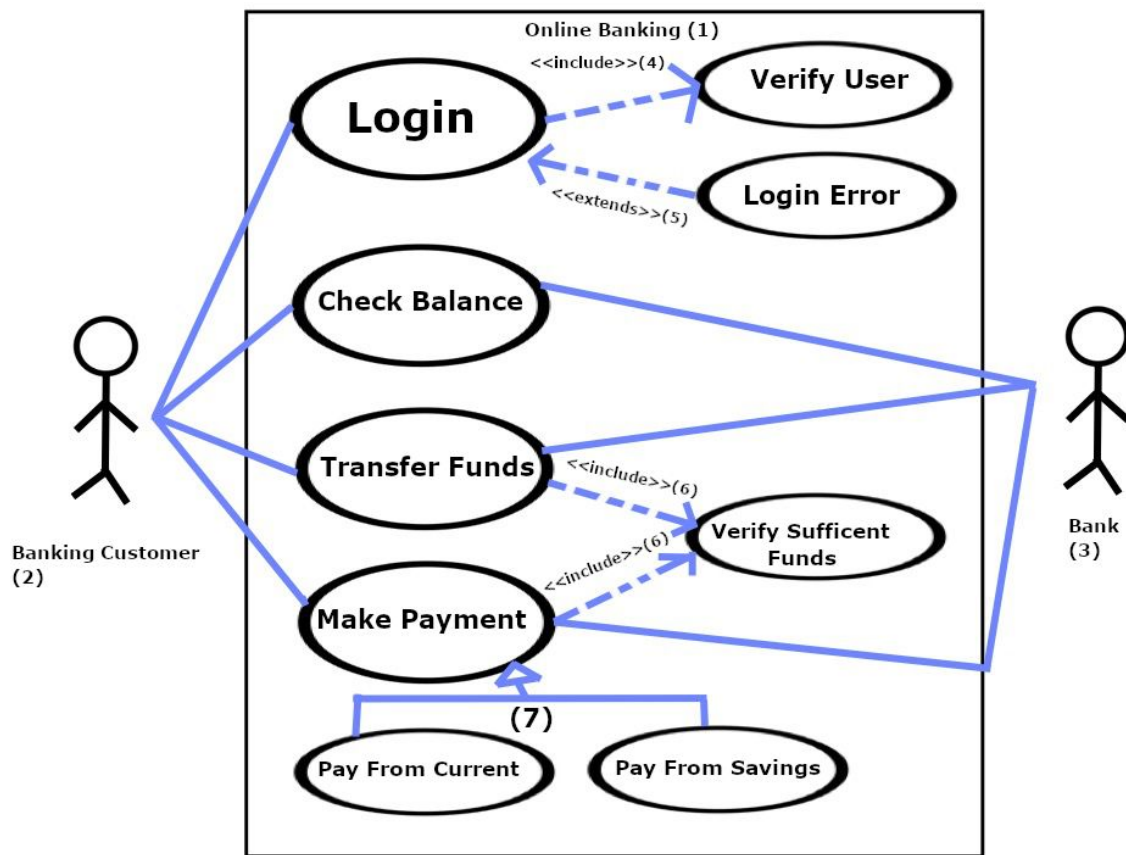
1) Name of system at the top
2) Primary Actors
3) Secondary Actors
4) Include Relationship
5) Extend Relationship
6) Multiple base use cases can point at the same include case
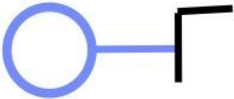7) Generalisation Relationship

Component Diagrams

Are more specialised versions of the Class Diagram, in which it breaks a system down into smaller components and visualises the relationships between.

Symbol	Symbol Definition

	Component An entity required to execute a function. A provides and consumes behaviour through interfaces and/or other components.
	Node This represents hardware or software objects which are of a higher level than components.
<<Interface>> Interface +Attribute1:Type = Default Value -Attribute2:Type -Attribute:Type <<Component>> Name <<Provided Interfaces>> Interface Name <<Required Interfaces>> Interface Name	**Interface** Shows input or materials that a component either receives or provides. Interfaces can be represented by text notes or symbols.
	Port Specifies a separate interaction point between the component and the environment.
	Package This is a group together of multiple elements of the system.
	Note Allows Developers to add a note(additional info) to the diagram.
- - - - →	**Dependency**

|	
	Provided Interface Depicted as a straight line from the component box with an attached circle, this symbol represents the interface where a component produces information used by the required interface of another component.
	Required Interface Depicted as a straight line from the component box with a half circle, or dashed line with an open arrow, these symbols represent the interfaces where a component requires information in order to perform its function.

State Transition Diagrams

State transition diagrams - also known as state machine diagrams - are used to describe all of the states that an object can have. This includes the events under which an object changes state (AKA *transitions* to a new state), the conditions that must be fulfilled before the transition will occur (called guards, explored further below), and the activities undertaken during the life of an object.

Notation

State

This is a condition during the lifecycle of an object in which some condition is satisfied, an action is performed, or waiting on events occurs. Depicted as a rectangle with rounded corners.
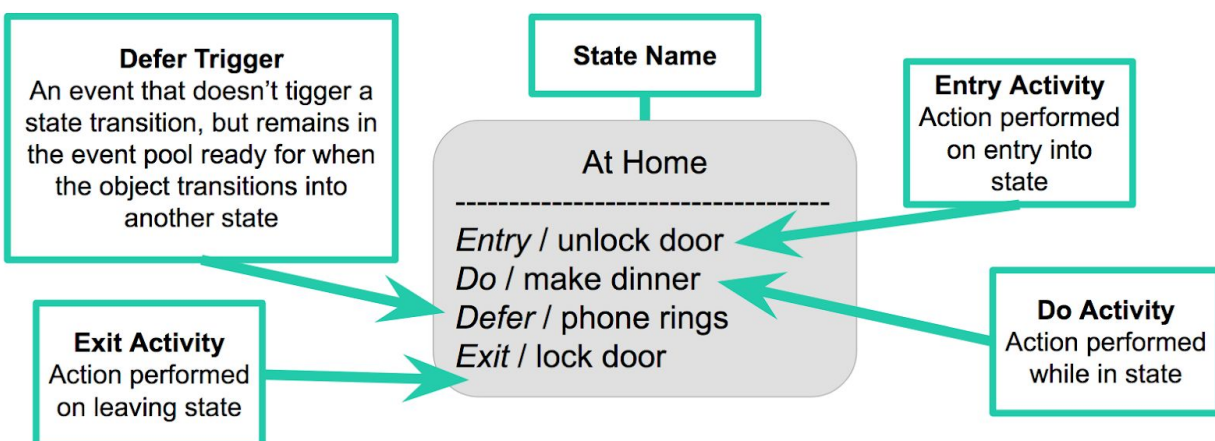

First State

This is a marker for the first state in the process, and is depicted by a dark (filled) circle with a transition arrow.

Events & Triggers

This is an occurrence that may trigger a state transition. There are 4 event types:

- A signal from outside the system
- An invocation from inside the system
- The passage of time
- A condition becoming true

They are labeled above transition arrows.
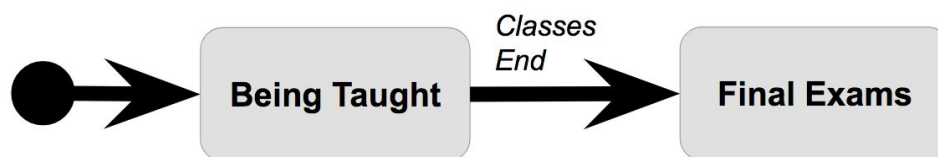
For example, 'classes end' is the event here that triggers the end of the being taught state and the beginning of the final exams state.

Guard

A boolean expression which when true enables an event to cause a transition. These are written above the transition arrow.

Choice Pseudostate

Represented by a hollow diamond, these represent a dynamic condition with branched potential results.

Exit Point

The point at which an object escapes the state machine, it is typically used if the process is not completed but had to be escaped due to an error. Depicted by a circle with an X through it.
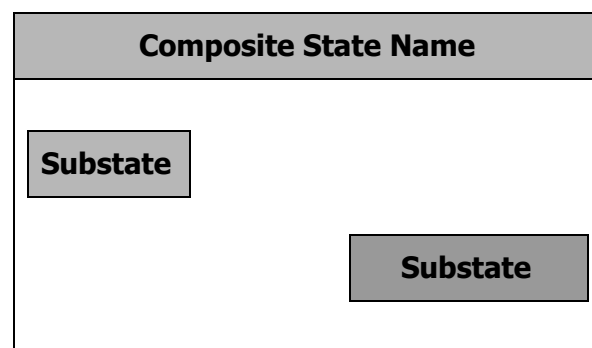
Terminator

A circle with a dot withinit, it indicates that a process is terminated.
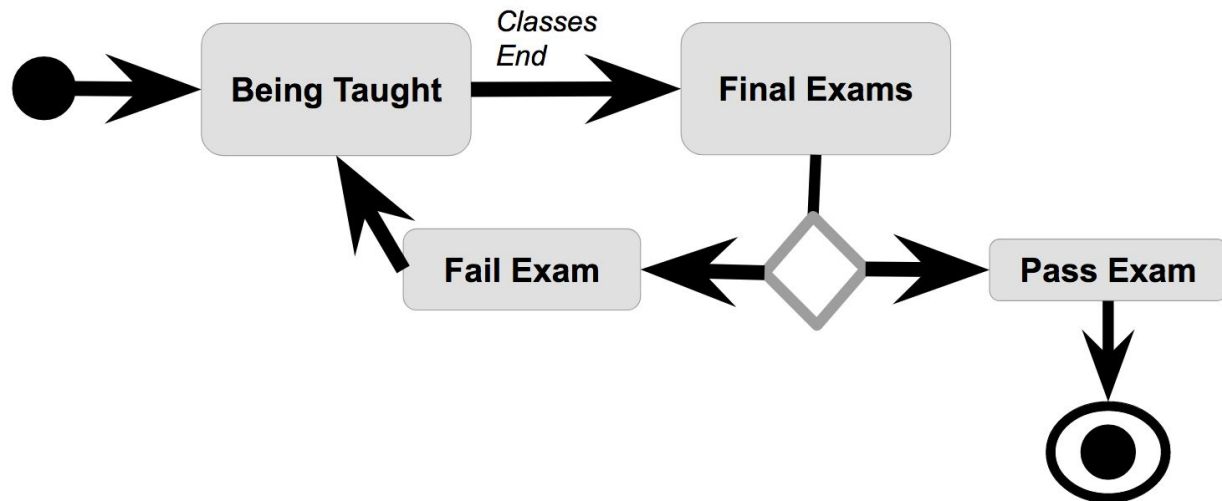
Substates & Composite States

A composite state is a state that has substates nested into it. Often depicted as an encompassing rectangle, with the name of the composite state highlighted within its own section.
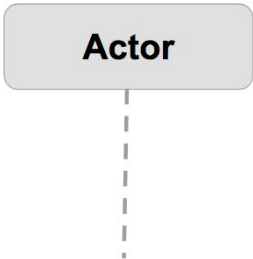
A substrate, then, is a state contained with a composite states region.
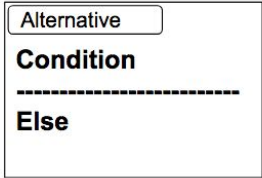
Sequence Diagrams

Sequence diagrams are interaction diagrams that detail how operations are carried out. They depict high-level interactions between users and the system, or between the system and other systems. Sequence diagrams are sometimes known as event diagrams or event scenarios.

They show how elements interact over time, organized according to object (horizontally) and time (vertically).

Notation Symbol	Notation Meaning

	Object Symbol Represents a class or object, and demonstrates how an object will behave in the context of the system. Class attributes should not be listed in this.
	Activation Symbol Represents the time needed for an object to complete a task. The longer the task will take, the longer the activation box becomes.
	Actor Entities that interact with, or are external to the system.
Actor	**Lifeline** Represents the passage of time as it extends downward. Lifelines may begin with a labeled rectangle shape or an actor symbol.
	Delete Message This message destroys an object. It is depicted by a solid line with an X at the end.
Loop **Condition**	**Option Loop** Used to model if/then scenarios. Used for circumstances that will only occur under certain conditions.

Alternative **Condition** ------------------------ **Else**	**Alternative** Symbolizes a choice between two or more message sequences.
	Synchronous Message Represented by a solid line with a solid arrowhead. This is used when a sender must wait for a response to a message before it continues.
	Asynchronous Message Represented by a solid line with a lined arrowhead. These don't require a response before the sender continues. Unlike synchronous messages, only the call should be included in the diagram, not the reply too.
	Reply Message Represented by a dashed line with a lined arrowhead, these messages are replies to calls.

Basic Example
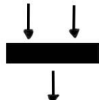
This example includes a Self Message which is a kind of message that represents the invocation of a message of the same lifeline.
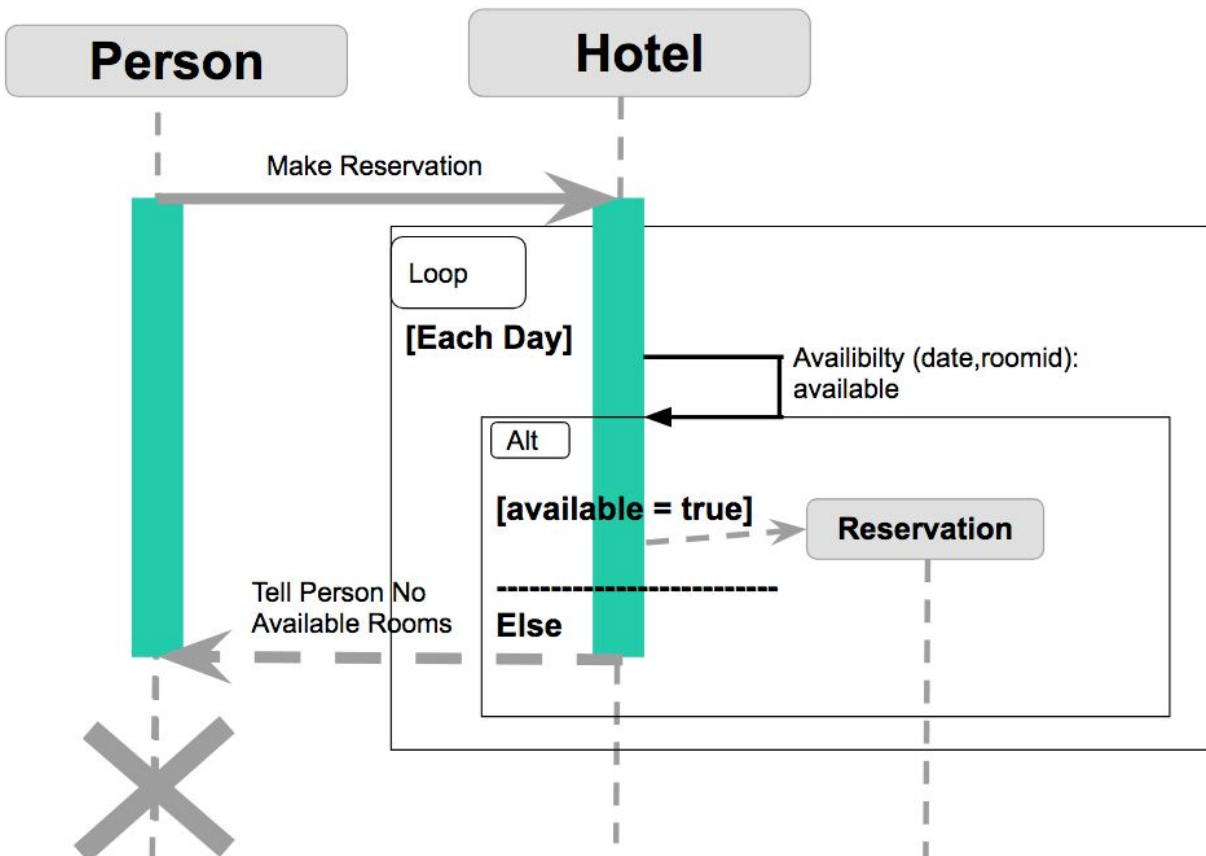
Activity Diagrams

Activity diagrams are very similar to state transition diagrams.

Notation Symbol	Notation Meaning
●	Start Symbol Represents the beginning of a process, and can be used by itself or with a note that explains the starting point.
▭	Activity Symbol Indicated the activities that make up a modeled process, with short descriptions within.
→	Connector Shows the directional flow

↓ ▬ ↓ ↓	**Fork** Splits a single activity flow into two concurrent activities.
↓ ↓ ▬ ↓	**Join Bar** Combines two concurrent activities and reintroduces them to a flow.
◇	**Decision** Always has at least two paths branching out from it with condition text to allow users to view options.
▱	**Note** Used to communicate messages that don't fit within the diagram itself.
⬠	**Send Symbol** Indicates that a signal is being sent to an activity.
⬡	**Receive Symbol** Demonstrates acceptance of the event.
Ⓗ	**Shallow History Pseudostate** Used to represent a transition that invokes the last active state.
⊗	**Flow Stop** Represents the end of a specific process flow
⊙	**End** Marks the end of an activity and the completion of all flows in a process.
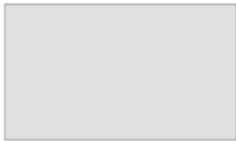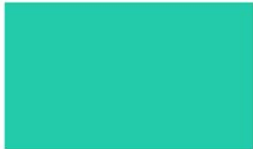
Basic Example
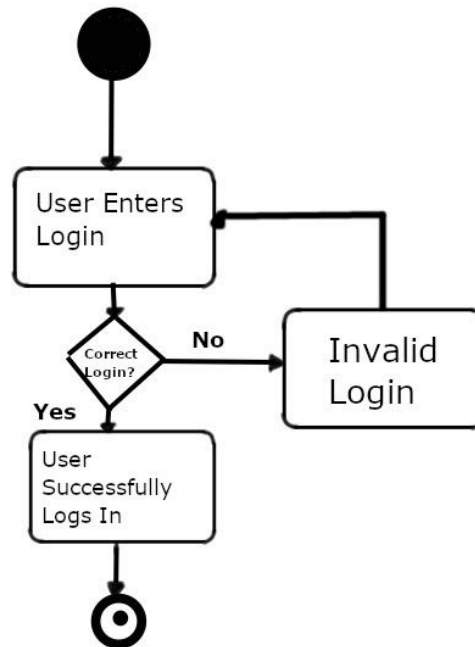
Data Flow Diagrams

Shortened to DFD, these diagrams represent functions or processes which capture, manipulate, store, and distribute data between a system and its components, and the environment and the system.

There are four basic symbols that are used to represent a data flow diagram, named after the models creators.

Notation Symbol	Notation Symbol	Notation Meaning
Yourdon & Coad	Gane and Sarson	External Entity

		Process
(circle)	(green rectangle with horizontal line)	
(grey rounded rectangle)	(green rectangle with vertical line)	Data Store
(arrow)	(arrow)	Data Flow
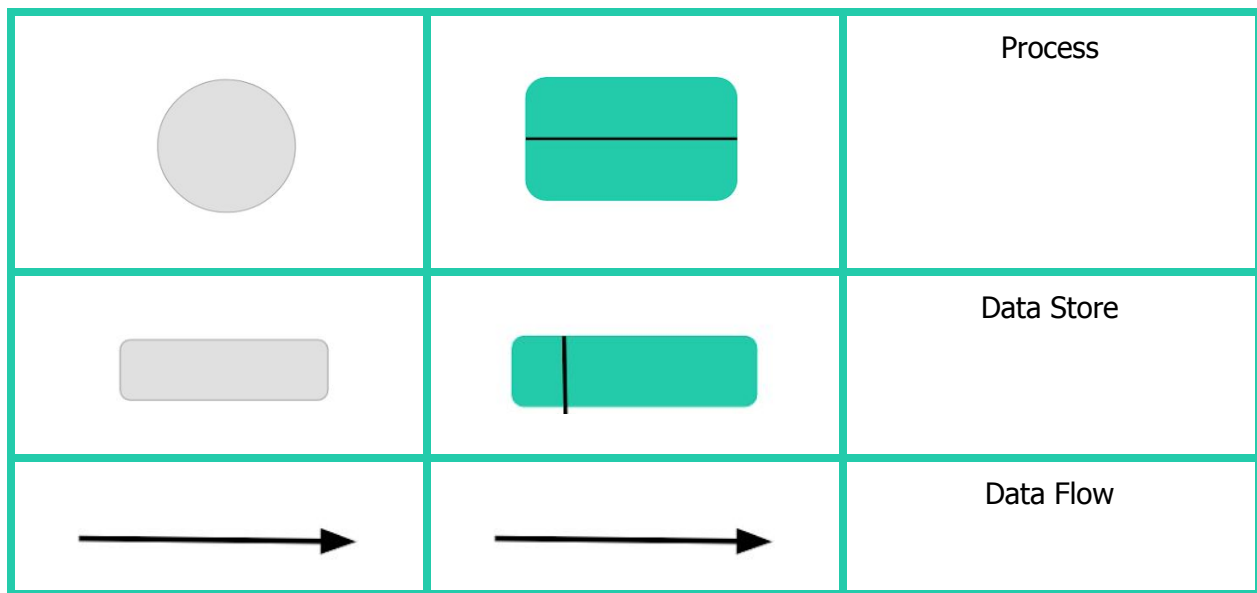
Basic Example
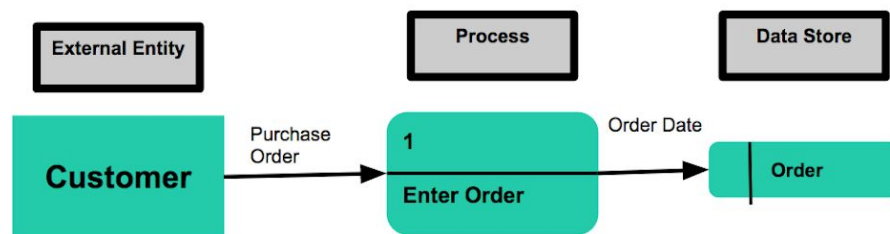
Processes are given IDs (1 in this example) for easy referencing. The process name is placed under the line or within double lines.

A data flow is a path for data to move from one part of the information system to another, it may represent a single data element such as the Customer Address or it can represent a set of data elements. An entity cannot provide data to another entity without some processing occurring, nor can it move data directly to a data store without being processed. Same goes for data stores, they cannot transfer data to an entity or other data store directly, without processing.

A data store (also known as data repository) is used to represent a situation when the system must retain data because one or more processes need to use the stored data at a later time. The store name is on the right side of the symbol division, and a label is on the left. For example, D1,D2, ect.

Methodologies

A software/system development lifecycle is essentially a series of stages that provide a model for the development and management of a piece of software. It is independent of the development methodology used.

There are just five basic methodologies and two approaches.

The two approaches are linear or evolutionary.

Linear

A linear approach describes a sequence of tasks that are completed in order, only moving to the next step once the previous step is complete.

This is suitable for developing where the requirements are well understood, can be agreed upon up front and are unlikely to change.

Strengths

- Manageability & Cost
    - Breaks down problems into distinct stages, all issues are captured and corrected before the next step starts
    - The locking down of each stage before advancing to the next makes it easier to control cost and scope creep
- Simple Development
    - Everything is agreed in advance of being used with no need to revisit later

Weaknesses

- Rigid Structure
    - It doesn't cope well with changing requirements
    - And each stage must be done properly as it is hard or impossible to go back and change it later
- Time & Value
    - For large projects, the time required to be thorough at each stage leads to long timescales
    - If the project is stopped before completion, there will be little or no business value to show for the cost

Evolutionary

An evolutionary approach evolves the solution through progressive versions, each more complete than the last.

The evolutionary approach is the basis of 'Agile' development methodologies, as it focuses on progressing understanding through delivering early prototypes or iteration releases. It assumes

that requirements are not well understood or cannot be well articulated early in the lifecycle, or that early delivery is more important than completeness.

Strengths

- Early Benefits
  - Early delivery of value to the customer – either through working versions or knowledge of project risk
- Flexibility
  - Copes well with complex requirements especially if they are yet to be defined, fast changing or complicated
  - Allows collaboration between the user throughout development

Weaknesses

- Over/Under Predictions
  - Easy to over-promise on early functionality
  - Overall costs can be higher due to the evolving requirements meaning it can be hard to give a definitive cost of a project
- Management
  - Careful management required, as it can be hard due to multiple iterations & especially so with multiple iteration teams or complex products

The 5 basic methodologies are:

- Waterfall
- V-model
- Incremental
- Iterative
- Spiral

Methodologies Based On The Linear Approach

There are 3 methodologies that follow the linear approach:

- Waterfall
- V-model
- Incremental

The Waterfall Method
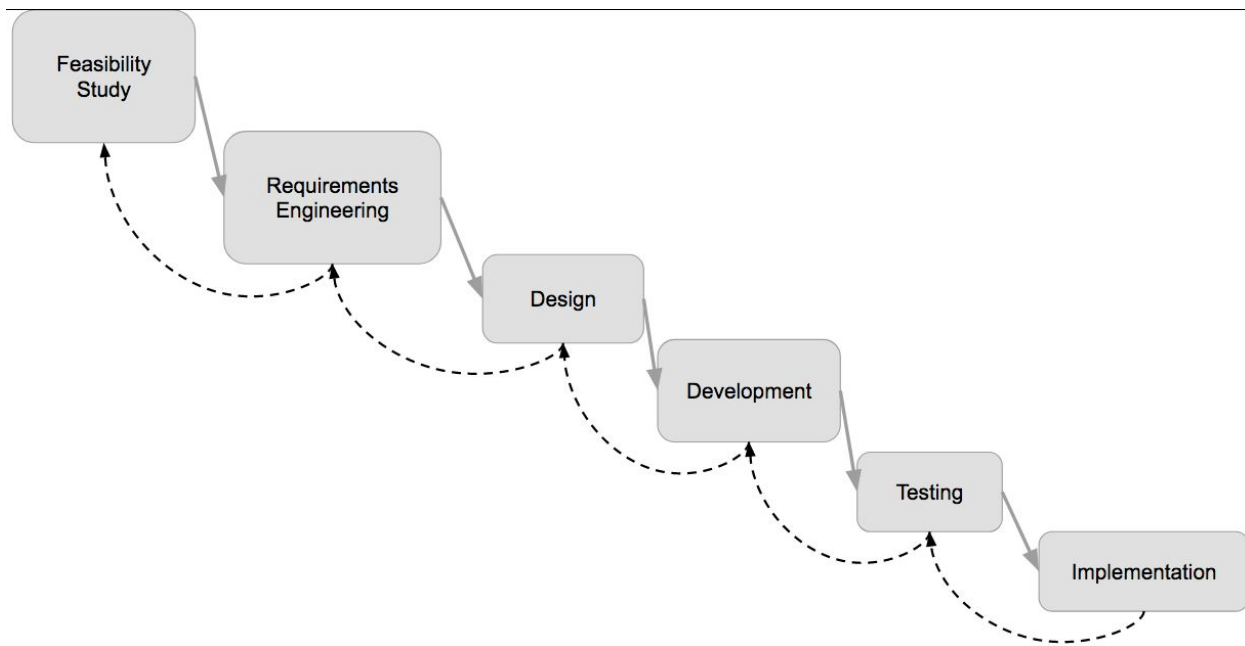
The waterfall method dictates that each step should be completed before moving onto the next step. This results in a quality product as the system is well understood through formal documentation and review.

The dotted arrows show that if a problem occurs in any of the steps, then communication and understanding must also flow backwards to rectify the problem. For example, if during development a problem occurred where a particular requirement could not be satisfied, then development would stop whilst the design step was reconsidered in light of the problem and, if necessary, the requirements engineering ststageep would also be revisited.

Strengths

- Simple
  - Simple to understand and easy to use
- Ideal For Solid Understanding Projects
  - Good for developing systems where the requirements are well understood and unlikely to change
- Easy Project Management
  - Due to the rigour and control around each stage, with obvious hooks[13] for project milestones

Weaknesses

- Rigid Structure
  - Once the requirements stage is complete, changes become increasingly expensive, as the team must revisit the requirements stage and all stages afterwards. Formal change control procedures are often put in place to manage change or even prevent the changes

---

[13] A thing designed to catch people's attention. *Analogy – a book store looking for a sales hook.*

- Deliverability
    - Working software not delivered until the final stage
- User Input Limits
    - Customer collaboration is limited to requirements and user testing
- Cost
    - Poor estimation can severely impact project cost and schedules, and change can be expensive

The V-model
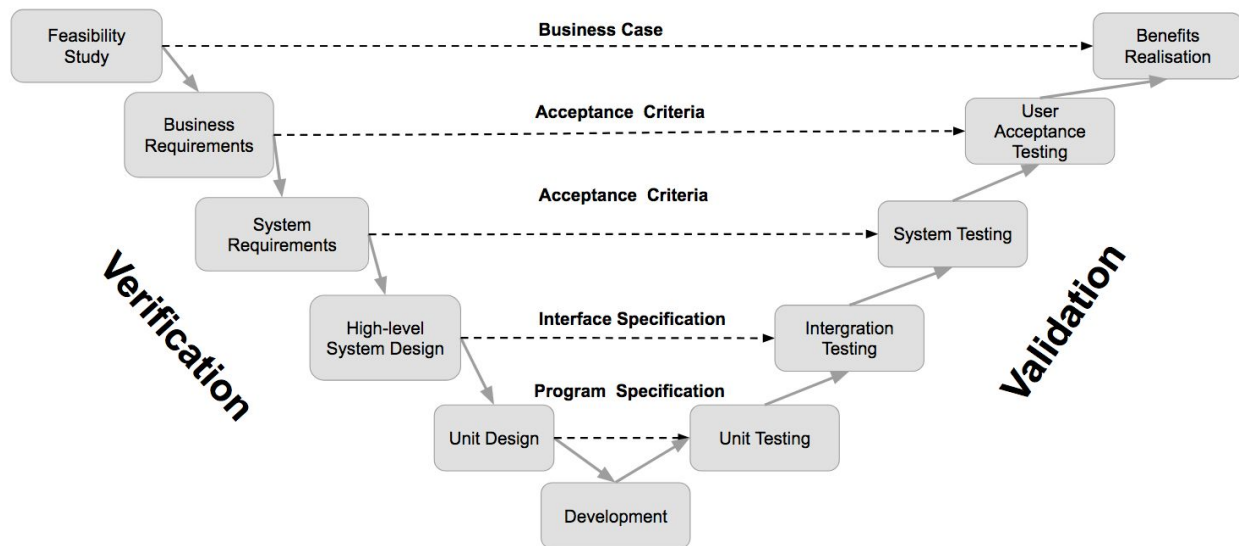
The V-model was introduced as a solution to the Waterfall models time for testing issue, whereby time allocated to testing could often get squeezed during a project if time and money started to run out. The V-model - also known as Verification and Validation model - is an SDLC model where execution of processes happens in a sequential manner in a V-shape.

For every phase in the development cycle, there is a directly associated testing phase.

The extended V-model, illustrated above, introduced the feasibility and benefits realisation stages, as a way of measuring and testing how well the actual business benefits were delivered, as opposed to testing when the requirements were delivered, as a system could verify and validate all the requirements, and yet fail to meet the need of the business because it does not deliver the business benefit.

In software development, validation is the process of checking the accuracy of developed components against a specification.

In layman's terms it is looking at the specification that says what the system should do, and looking at the components being developed, and checking if it will perform a function related to the specification to satisfy or fit the user requirements.

Validation can be done at any stage in the SDLC, however the V-model dictates that it should be done in a sequential manner. There are two types of validation, internal and external.

During internal validation, it is assumed that the goals of the stakeholders were correctly understood and expressed in the requirement artifacts, and so if the software meets the requirement specification, it has been internally validated.

External validation is performed by asking the stakeholders if the software meets their needs, if it does, it is successfully externally validated.

Final external validation of a project requires the use of an acceptance test which is the last validation testing phase in a standard V-model, and the second to last phase in the extended V-model above.

Verification is testing code, and it can only be done by developers or testers - it must be someone who understands code. It is looking at the components and testing to see if it does what it should do.

Validation VS Verification Summary

Validation is the process of evaluating software during, or at the end of development (dependent on the development model) to determine whether it satisfies specified requirements whilst verification is the process of evaluating software to determine whether the components of a given development phase satisfy the conditions imposed at the start of said phase.

Strengths

- High Quality Product
  - Due to the additional focus on testing - great for safety critical systems
- Easy To Manage
  - Due to the rigid structure and control

Weaknesses

- Cost
  - For simpler projects the heavy testing and integration cost can be unnecessary
  - Cost and impact of change increases significantly as you progress further down and up the model
- Time
  - Early stages take longer to complete as there are test artefacts to create
- Deliverables
  - Working system not delivered until the final stage
- Equal Importance
  - Development teams can often omit putting enough effort into planning the right-hand side as they progress down the left which results in significantly increased risk
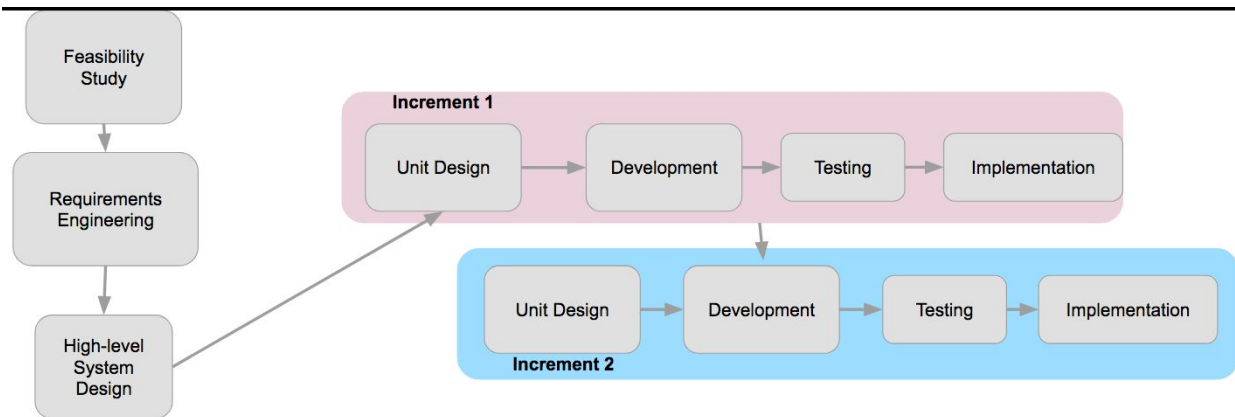
The first three stages of the SDLC are conducted in sequential order, each being completed before the next is started, however, where the incremental cycle differs from the waterfall or v-model is that after understanding and designing the system, decisions can then be made to deliver some parts of the system before others. The Incremental lifecycle, also known as instrumental delivery, is ideal where early capability is required, as high priority requirements can be delivered first.

Strengths

- Cost
  - If the business decides that the delivered functionality of the earlier increments is sufficient, then the project can be stopped and no further cost is incurred
- Deliverables
  - Delivers working software early on

Weaknesses

- Cost
  - Additional releases add to costs, as release costs such as user training and deployment are incurred multiple times
  - Subsequent releases must be regression tested also, against earlier functionality, meaning that features in the first release are re-tested in every subsequent release to ensure that they still work properly which adds to the test cost
- Change Management
  - Difficult to manage change, as requirements are agreed before increments decided

Lifecycles Based On The Evolutionary Approach

Iterative
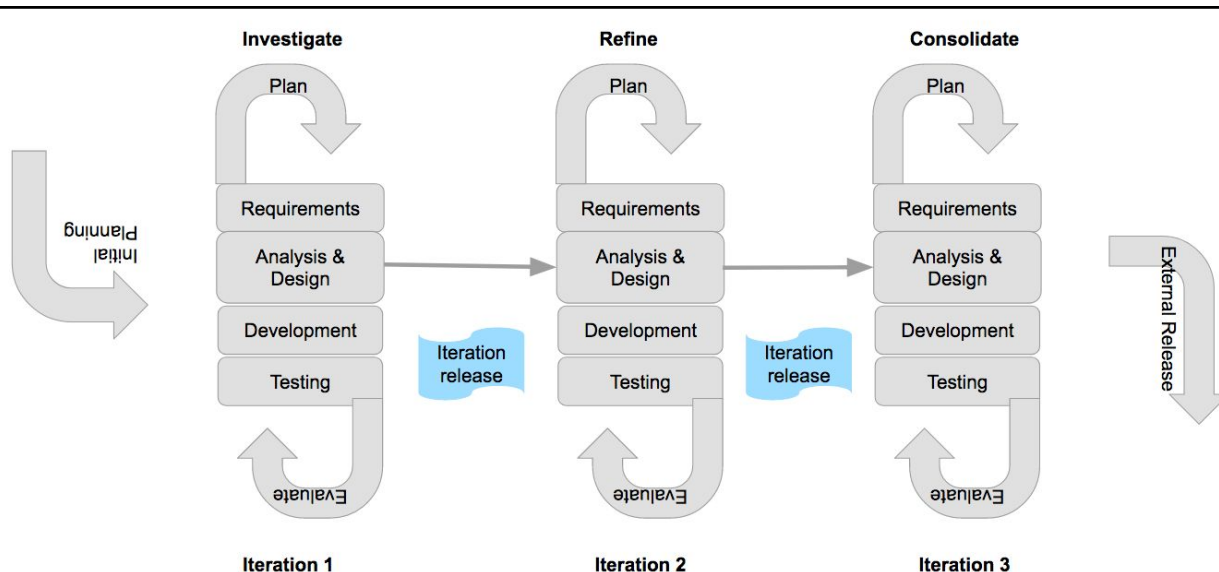
Iterative development is a method where the overall project is composed of several smaller time-boxed developments, called iterations, and each iteration is a self-contained mini project composed of activities such as requirements, design, development and testing.

Requirements are elicited during each iteration, and in doing this the understanding of the overall project evolves as the requirements evolve.

A standard minimum of three iterations would usually be required to achieve an external release, these iterations would consist of:

Investigate

This initial iteration is used to investigate things such as risks around user functionality, and results in more understanding about what is required to deliver the external iteration or first release.

Refine

There are usually multiple refinement iterations, to which the requirements are fully detailed and designed and the system is developed to meet the business goal.

Consolidate

This is the final iteration prior to external release, and it ensures that the code and design developed is stabilised and fully integrated with any other components/iterations so that the overall business goal is achieved in the working system.

Strengths

- Cost
    - A great model for prototyping. Iterative development is good where the problems or solutions are not clear at the beginning, or where the business

doesn't know yet if it wants to fund the project as the early iterations can provide clarity over what is feasible and the resulting cost
- ○ Costs can easily be controlled as activities such as requirements, design, development and testing are recreated every iteration
- Flexibility For Change
  - ○ Change is easier to manage as requirements are not locked down

Weaknesses

- Cost
  - ○ Poor management can lead to increased costs, as evolving requirements can result in scope creep
  - ○ Overall costs can be higher due to the additional integration and tests across multiple teams and iterations
- Management
  - ○ Can be hard to manage due to multiple iterations or multiple iteration teams

Spiral
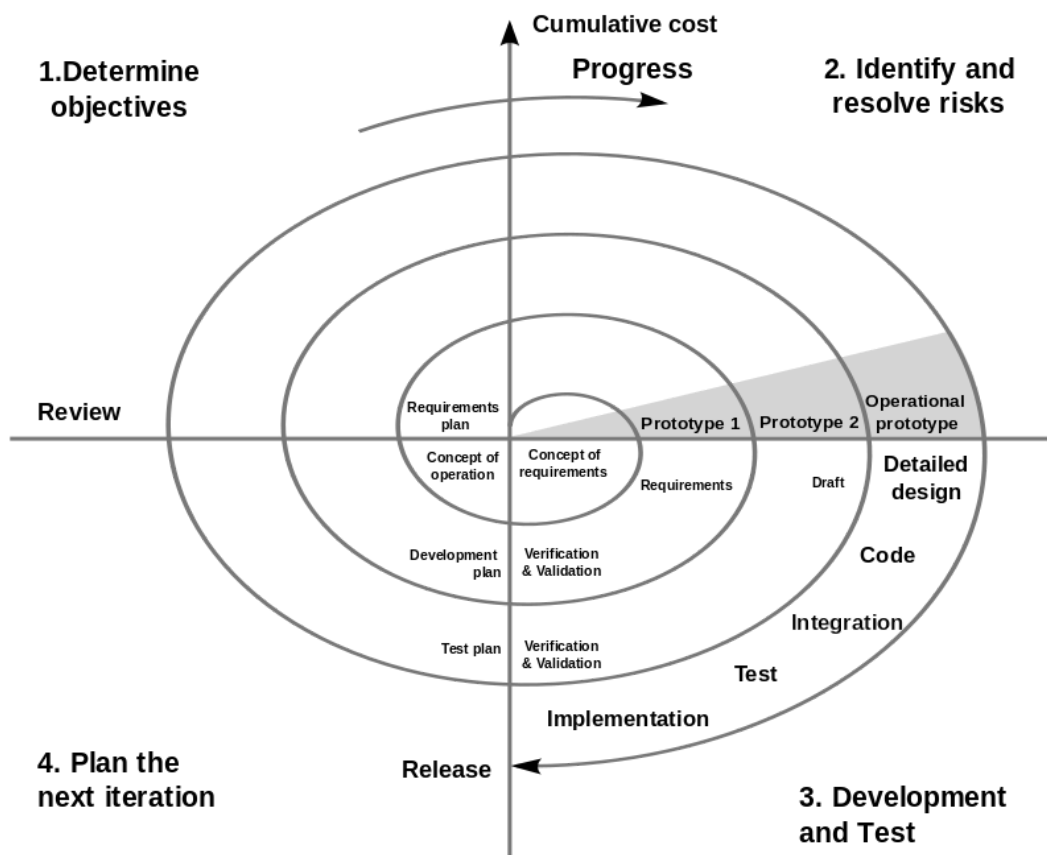
Spiral model (Boehm, 1988)

The Spiral Model is a software development life cycle model which provides support for risk handling by combining iterative development with prototyping, to test and evolve the requirements so that risks can be addressed throughout. The exact number of loops of the spiral can vary from project to project. Each loop of the spiral is called a "phase of the software development process".

There are four main phases:

Determine Objectives

- Requirements are gathered
    - In this first stage, the development team and business owners identify the objectives for development and agree prioritisation of requirements. Throughout the project this is constantly reviewed and revisited

Identify And Resolve Risks

- Risks are identified, as are alternatives, and a prototype built
    - In this stage the developers explore technical possibilities to meet the business goal and assess any risks they may bring.

Develop And Test

- Software is produced and tested
    - The initial prototype is reviewed to assess how well requirements were met, designs are then developed and agreed and another prototype built. After a few iterations an operational prototype will be produced.

Plan Next Iteration

- Output of project thus far is evaluated, and next steps are made
    - In this stage the working system is released into the live operational environment. If this is an incremental release, further releases will be planned until the complete system is developed.

Strengths

- Deliverables
    - Software is produced early in the life cycle
- Change & Flexibility
    - Due to the iterative nature, functionality can be added later and requirements can evolve easily
- Low Risk
    - Extensive risk assessment ensures that the project less likely to fail

Weaknesses

- Cost

- - ○ Poor management can lead to increased costs, as evolving requirements can result in scope creep
  - Documentation
    - ○ As the focus is on a working system, documentation of said system can become de-prioritised, which can result in the system being harder to maintain post-project

What Is Agile

"Agile software development" refers to a group of software development methodologies based on iterative development, following the 'Agile Manifesto' from 2001. The Agile Manifesto states:

> *We are uncovering better ways of developing software by doing it and helping others to*
>
> *do it. Through this work we have come to value:*
>
> **Individuals and interactions** over process and tools
>
> **Working software** over comprehensive documentation
>
> **Customer collaboration** over contract negotiation
>
> **Responding to change** over following a plan
>
> *That is, while there is value in the items on the right, we value the items on the left more.*
>
> (https://Agilemanifesto.org/)

This section reviews some of the more popular Agile lifecycles, which are often hybrid approaches of the methodologies previously discussed.
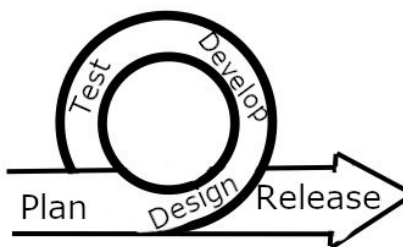
Below are some of the generic strengths and weaknesses Agile methodologies have.

Generic Strengths

- Flexibility
  - ○ Agile approaches are best-suited for a relatively uncertain environment where it is difficult to accurately define the requirements and design for the solution in

detail prior to the start of the project. The requirements and design are able to change as the project progresses

- Cost
  - Agile approaches significantly reduce overhead from reducing unnecessary documentation and control requirements
  - As the projects are typically handled incrementally or iteratively, cost can be managed as requirements are discovered and developed
- Product Meeting Needs
  - An Agile approach should result in a higher customer satisfaction because the customer is heavily involved throughout the development by providing feedback and inputs
- Deliverables
  - An Agile approach typically results in faster time-to-market due to shorter startup times, and software being produced early in the life cycle

Generic Weaknesses

- Scalability
  - It can be difficult to scale an Agile approach to larger complex projects
- Organisational Structure
  - An Agile approach may require some level of organizational transformation to the business structure to make it successful, as they tend to require the business users to work collaboratively with the development team and that requires breaking down some organizational barriers (Silo breaking)
- Training & Skills
  - An Agile approach requires a considerable amount of training and skill to implement successfully

SCRUM

SCRUM is an Iterative and Incremental process that follows an evolutionary approach to the development of software. This approach is recognisable through it's specific concepts and practices that can be divided into 3 categories: Roles, Artifacts, and Time Boxes.

Roles

SCRUM only contains three roles:

1) SCRUMMaster

   is responsible for making the process run smoothly, by removing obstacles that impact productivity, and organizing and facilitating meetings.

   Main Responsibilities:

   ● Teach the Product Owner how to maximize return on investment (ROI), and meet their objectives through SCRUM

● Keep information about the team's progress up to date and visible to all parties

2) Product Owner

represents the business and is the voice of the customer.

Main Responsibilities:

● Works closely with the team to define the user-facing and technical requirements, to document them as needed, and to determine the order of their implementation

● Maintains the Product Backlog (which is the repository for all of the above information), and sets the schedule for releasing completed work to customers

3) Team

is a self-organizing and cross-functional group of people who do the hands-on work of developing and testing the product.

Main Responsibilities:

● Decide how to break work into tasks, and who to allocate tasks to throughout the Sprint

● Producing the product

Artifacts

The core elements of SCRUM are the Product Backlog, the product increment, and the Sprint Backlog. These are the artifacts, which serve to capture the shared understanding of the team at a particular point in time.

The Product Backlog is a prioritised list of requirements that the team will be delivering. Product Backlog refinement happens every Sprint, and can involve reviewing the highest priority tasks, deleting tasks that are no longer needed, writing new tasks and re-prioritizing and estimating tasks.

Iterations called Sprints, are short, perhaps two weeks in length, and focus on delivering a working version of the product. This product increment must align to the development team's "Definition of Done" and be acceptable by the Product Owner. A new Sprint starts immediately after the conclusion of the previous Sprint.

The Sprint is monitored by the Daily SCRUM which is a short daily, usually 15 minute, highly-structured meeting that is focused on removing barriers to success. The Daily SCRUM is held at the same time and place each day to reduce complexity. Only the people doing the work described on the Sprint Backlog need to attend the Daily SCRUM.

A Sprint Backlog is a derivative of the Product Backlog that the team pulls into the Sprint to work on. It is a 'to do' list that the development team will work off of during the current Sprint,

and only the development team can change it as it is a real-time picture of the work that the team plans to accomplish during the Sprint.

Timeboxes

This is a time management technique that limits the amount of time someone can spend on a certain activity in advance.

SCRUM Process	Timebox
Sprint Planning	8 hours or less
Sprint Retrospective	3 hours or less
Sprint Review	4 hours or less
Sprint	One month or less
Daily SCRUM	15 minutes daily

Sprint Planning is for the SCRUM Team to inspect the work from the Product Backlog, find the most valuable or next priority work to be done next and design that work into the Sprint Backlog.

Sprint Retrospective serves for the SCRUM Team to inspect the past Sprint and plan for improvements to enact during the next Sprint.

A Sprint Review is for the SCRUM Team and the stakeholders to inspect the increment resulting from the Sprint, assess the impact of the work performed on overall progress and update the Product Backlog in order to maximize the value of the next Sprint.

KABAN

KABAN is very similar to SCRUM.

The main differences between:

- No two-week sprint: KANBAN is a continuous process
- No Sprint Backlog: the "pull" system in Kanban happens in a different way, via Work In Progress (WIP) limits. (Work In Progress limit related to the team's capacity)
- Scrum Master for SCRUM is known as the Agile Coach for KABAN
- Daily Scrum is Daily Stand Up in KABAN

Principles Of KABAN

- Visualise Workflow

In KABAN, workflow is organized through a KABAN Board, which consists of columns (typically three: Requested, In Progress and Done), and cards. Each column represents a step in the workflow, and each card represents a work item. When work on an item is started, it is 'pulled' into the WIP column, and when it is done it is moved to the 'done' column. This provides a succinct way to track progress and spot bottlenecks.

- WIP Limits

Setting a maximum items limit per stage ensures that a card is only pulled into the next step when there is available capacity. This ensures that the teams focus doesn't switch halfway through a process, as multitasking this way is inefficient. KABAN aims to ensure that a manageable number of active items are in progress at any one time.

- Feedback

Just as Lean development states that regular meetings are necessary for knowledge transfer (referred to as feedback loops), KABAN has Daily Stand Ups. They are held in front of the KABAN Board, and every team member explains to the others what they did the previous day, and what they plan on doing today.

KABAN VS SCRUM Expanded

Each Sprint starts with a Sprint Planning Meeting where high priority items from the Product Backlog are pulled into the Sprint Backlog.	Items are pulled directly from the Product Backlog into the KABAN Board. Each column in the board has Work In Progress limits, to ensure that items move across the board in the shortest possible time.
When the Sprint is complete, it is reviewed (often includes a demo of new features to stakeholders), and an examination of the Sprint happens (what went well, what could be improved).	A nearly empty column signals to the previous column to send another item this is called the pull system.
Then at the end of the Sprint, completed items are packaged for release.	Each item is packaged for release as soon as it is ready, along typically with a demo to stakeholders and a retrospective.

EXTREME PROGRAMMING (XP)

This is an Agile software development framework which aims to produce higher quality software and responsiveness to changing customer requirements.

Principles Of XP

- Communication
  - Collaboration of users and programmers, with frequent verbal communication and feedback
- Simplicity
  - Avoid waste and keep system designs simple so that it is easier to maintain, support, and revise. Address only known requirements, don't try to predict the future
- Courage
  - Courage in the software development sense is knowing when to throw code away such as code that is obsolete - no matter how much effort was used to create that code. It also means persistence in the face of adversity
- Respect
  - Includes respect for others as well as self-respect

Features Of XP

- Programming in pairs
- Extensive code reviews
- Unit testing of all code
- Avoiding programming of features until they are actually needed
- A flat management structure

- Code simplicity
- Frequent communication with the customer and among programmers

RUP

The Rational Unified Process (RUP) is structured around six fundamental principles. However is not a concrete development model but is instead intended to be adaptive and tailored to the specific needs of a project.

Principles Of RUP

- Develop Software Iteratively
  - This means that the software is developed in increments, by locating and working on the high-risk elements first
- Manage Requirements
  - This means that everything should be well documented. This principle shows how to organize and keep track of documentation, decisions, and business requirements
- Visually Model Software
  - Like a wireframe, RUP means to visually model software, including the components and their relationships with one another
- Use Component-Based Architectures
  - RUP emphasises development that focuses on software components which are reusable throughout a project and within future projects[14]
- Verify Software Quality
  - This means testing regularly all aspects of the product throughout the development life cycle
- Control Changes To Software
  - This describes how to track and manage all forms of change that will inevitably occur throughout development, in order to produce successful iterations from one build to the next. Like Commits and Git.

DevOps

Is short for Development and Operations Collaboration.

This methodology consists of the operations and development engineers working together in the entire project lifecycle, from the design and development process to production releases and support.

Features Of XP

- Reliability

---

[14] You may hear people call this keeping code DRY. DRY stands for Don't Repeat Yourself.

- - By producing sufficient change logs, each team can monitor the logs and help themselves stay updated to be able to make real-time decisions quickly
  - Silo Breakdown
  - Speed
    - One of the characteristics of DevOps is the speed in which feedback is requested and responding changes are made. If not properly managed it can become one of the disadvantages as it eliminates all opportunity to "carefully consider" suggested changes.

Lean Development

Sometimes referred to as LSD, this methodology actually stems from lean manufacturing pioneered by Toyota. It is a set of techniques that can be applied to eliminate waste in manufacturing and improve productivity rather than a detailed methodology with stages, and as such can be combined with other methodologies.

Principles Of Lean

- Focus On Customers
- Continuously Seek To Improve
- Deliver And Learn Fast
- Build In Quality
- Energise Workers

RAD

Rapid Application Development (RAD) is a form of Agile software development that prioritizes rapid prototype releases and iterations.

Steps Of RAD

- Define & Finalise Project Requirements
- Build Prototypes
- Gather Feedback
- Test
- Data Conversion, User Training & Then Launch

Software Package Solutions

Commercial Off The Shelf (COTS)

There are times when it is not feasible to develop new software from scratch, and it is preferable to use a software package that already exists. These are known as Commercial Off The Shelf (COTS) solutions.

These solutions are provided 'as-is', meaning that you pay for the software that is already developed and ready to go, support of some kind is usually included.

They may save time but it comes at a cost of not being tailored specifically for the business.

Component Based Development

This is where software applications are built using components, which can come from a number of sources. This development type takes advantage of code reusability.

Strengths

- Time Save
  - Uses DRY principles
- Cost
  - The reduced time saves money on resources
- Improved Quality
  - Using components that have already been in live use reduces the risk, as bugs have most likely already been found and dealt with

Weaknesses

- Harder Testing
  - Unknown uses of components brought in from outside source
- Middleware
  - Technologies could be incompatible

Bespoke Development

This is a method of producing software that doesn't prescribe to common theoretical approaches. Software may be developed using variations or combinations of methodologies.

Businesses choose aspects of approaches that best suit themselves and combine them to form a customised development practice to give maximum benefits.

The disadvantages is that organisations may do whatever is cheapest, meaning the most appropriate practices may not be followed.

Application Lifecycle Management Tools

With system development being more complex now-a-days, tools exist that can help through all stages of the development process.

Benefits Of Tools

- Standardisation
  - The use of templates, specific notation and forms for written documents ensures that any team member can pick up were another left as the process is standardised and the quality of the developed product is always maintained
- Storage
  - Creating computer based files rather than paper ensures not only higher security, but storage space is relatively unlimited, as no physical capacity is enforced
- Availability & Security
  - Typically electronic documents and repositories can be accessed more readily than paper counterparts
  - With electronic documents you can control the access levels and privileges easier than paper
- Version Control (Source Control)
  - As elements evolve over time, the ability to maintain multiple versions and roll back when needed is vital
  - Recording who made changes, when, and what is vital in 'bug corralling' and maintains independent responsibility

A good example of a version control tool is Github[15][16].

Version control tools typically allow multiple people to work on code simultaneously, whilst keeping a central version. They also allow each stage of development to be saved separately, different branches of development to take place, and allow developers to keep track of changes and revert them when needed.

- Impact Analysis
  - A tool can use links and cross references to provide useful views and reports about elements, such as matrices and traceability reports, these help during change control through an impact analysis. If a change is made on a particular element, tools are quick to find other elements that may also have to change
- Configuration management
  - A configuration is a set of related configurable items, each with a specific value or setting, which need to work in specific combinations. Being able to apply

---

[15] https://github.com/
[16] https://learngitbranching.js.org/

effective change control management to these is integral particular with complex systems

Pitfalls Of Tools

Tools are only as useful as those who use them - if you get a tool you know nothing about, you are not likely to reap the benefits of said tool. There is also the danger that too much trust is placed within a tool, that anything output by said tool must be correct.

What To Consider When Selecting Tools

Requirements

What do you expect the tool to do?

Functionality

Be cautious of lists of features, they can be exaggerated by marketing. Look within the business and see what particular roles can do with the tool.

Suitability

Does the tool meet business standards, or will the business have to change processes and notation to align with the tool?

Compatibility & Technicality

Is the tool compatible with existing operating systems? What is the technical requirements for the tool?

Cost & Skills

Is there a cost, or is it open source? Do users of the tool require training in order to be effective?

CASE

Stands for Computer Aided Software Engineering, and covers everything from requirements to testing. Within CASE there are specialist tools:

CARE

Computer Aided Requirements Engineering tools are specialised tools that can sometimes operate as a stand alone tool, or as a function of a broader business modelling or CASE tool.

They can make a lot of things easier including:

Version & Change Control	*Team Support & Access Security*
Traceability	*Configuration Management*

Business Modelling

Tools that support business modelling and architecture help to develop, implement and maintain the models and architecture within such as:

> *Business Functions & Services*   *Context & Links With Economy*
>
> *Internal Structure & Roles*   *Information Systems (IS) & Information Technology(IT)*

In this context, IS/IT are components that provide services to the business and perhaps direct to customers. As the business seeks to change its model it often requires a change in the IS/IT.

Project Management Tools

Project Management tools tend to focus on planning, resource management and progress reporting.

System Modelling Tools

Can be used by a number of roles such as business and systems analysts, designers and developers. These modeling tools support model driven engineering, which is the process of proving from abstract models of a system through to an implemented version of said model.

The diagram below demonstrates how older techniques created a new system - called a 'U-Curve'.
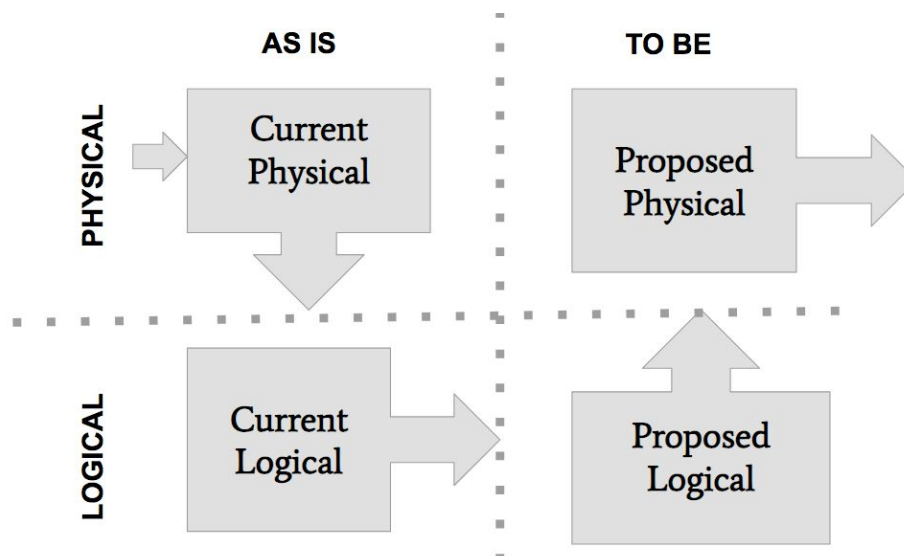
Testers can also use these models as a basis to analyse and design the testing. Or integrate these tools with other testing tools they're using.

CAST

Stands for Computer Aided Software Testing, and there are different tools for different types of testing. They do more than run automated tests, some other features include:

- Test Management
  - Allows managers to plan, eview and report on a testing program
- Test Design
  - Allows testers to analyse requirements and design to design an appropriate level of testing. Including design of test scenarios, sample data and boundary tests
- Test Analysis
  - Allows testers to review the results of a test and determine what feedback to provide the developers about defects, and changes to make to the test designs to increase the effectiveness of the tests

Software Development Tools

Development tools enable developers to take a design, and either write source code or use a visual notation that links components into an assembly to which source code is automatically generated. A good example of this is LucidChart[17].

Integrated Development Environments (IDE)

Code then needs to be able to be executed by the computer. There are various tools for this which fall into two main categories: Interpreter or Compiler.

Interpreter

Here source code is interpreted as a set of instructions at run time. The software that does this is known as a translator. For example, JavaScript embedded on a site is interpreted by the browser.

Compiler

Code here is compiled and linked to produce native executable code.

Source code is taken through a two-stage process, where it is first compiled into an intermediate link-code, which is then linked to similar code in statically-linked libraries to produce a version of the code that is directly executable by a specific platform.

Typical features of development tools include:

Syntax Support	*Integration With Repositories & Version Control*
Highlight Errors	*Model To Code Support*

Service Management Tools

Also referred to as IT Service And Support Management Tools (ITSSM).

---

[17] https://www.lucidchart.com/pages/

These tools are useful for after a solution has been released, as after it's release it still requires monitoring against service levels, support through help desks, documentation of what is deployed and business-as-usual(BAU) maintenance. A configuration management database (CMDB) is a key feature of any ITSSM.

Service Level Agreements (SLAs)

This is an agreement between two or more parties, typically those who build or maintain a system and their customers. It is used to define things such as time to respond to a request for support.

There is a contract between parties that agrees to provide the service to a predefined level and or standard. Penalties may be imposed if the SLA is not met.

Glossary

Word Or Phrase	Definition
Acceptance Testing	is a level of software testing where a system is tested for acceptability. The purpose is to evaluate the system's compliance with the business requirements and assess whether it is acceptable for delivery.
Activity Diagrams	are graphical representations of workflows and actions with support for choice, iteration and concurrency.
Actors	In software development, a use case is a list of actions or event steps typically defining the interactions between a role (an actor) and a system to achieve a goal. The actor can be a human or other external system.
Agile	An umbrella term for many different development methodologies. It is characterised by the division of tasks into short phases of work and frequent reassessment and adaptation of plans.
Backlog	An accumulation of uncompleted work or matters needing to be dealt with.
Big Bang Deployment	is the instant changeover, when everybody associated with the old system moves to the fully functioning new system on a given date.
Bug-fix	A correction to a bug in a program or system.
Business Analyst	is someone who analyses an organisation and documents its business, processes or systems, assessing the business model or its integration with technology.
Business Case	A justification for a proposed project based on its expected commercial

	benefit.
Business Drivers	is a resource, process or condition that is vital for the continued success and growth of a business.
Client GUI	A graphical user interface (GUI) is a human-computer interface (i.e., a way for humans to interact with computers) that uses windows, icons and menus and which can be manipulated by a mouse (and often to a limited extent by a keyboard as well).
Component Diagram	show the dependencies and interactions between software components.
Cross Site Scripting	is a type of computer security vulnerability typically found in web applications. XSS enables attackers to inject client-side scripts into web pages viewed by other users.
Data Flow Models	is a diagrammatic representation of the flow and exchange of information within a system.
Defensive Programming	is a form of defensive design intended to ensure the continuing function of a piece of software under unforeseen circumstances.
Denial Of Service	An interruption in an authorised user's access to a computer network, typically one caused with malicious intent.
Development Methodologies	refers to the framework that is used to structure, plan, and control the process of developing a system.
DevOps	Is a software development process that emphasises communication and collaboration between software developers and operations.
Distributed Denial Of Service	The intentional paralysing of a computer network by flooding it with data sent simultaneously from many individual computers.
Domain Experts	Domain expert is a person who is an authority in a particular area or topic.
Entity	A thing with distinct and independent existence.
Entity Relationship Models	(ERM) is a theoretical and conceptual way of showing data relationships in software development.
Evolutionary Prototyping Model	This prototype evolves into the final product. The main goal is to build a very robust prototype in a structured manner and then refine it. The prototype forms the heart of the new system and the improvements and further requirements will then be built.
Extreme Programming	(XP) is a software development methodology which intends to improve software quality and responsiveness to changing customer requirements.
Feasibility Phase	Feasibility is the measure of how well a proposed system solves the problems.
Feasibility Study	An assessment of the practicality of a proposed plan or method.
Functional Requirements	In software engineering a functional requirement defines a function of a system or its component. A function is described as a set of inputs, the behaviour, and outputs.
Malware	Software which is specifically designed to disrupt, damage, or gain

	authorised access to a computer system.
Manual Testing	is the process of manually testing software for defects. It requires a tester to play the role of an end user.
Non Functional Requirements	is a requirement that specifies criteria that can be used to judge the operation of a system, rather than specific behaviours.
OOP	Stands for Object Oriented Programing
Product Owner	Part of the product owner responsibilities is to have a vision of what they wish to build, and convey that vision to the SCRUM Team.
Project Manager	The person in overall charge of the planning and execution of a particular project.
Prototypes	A first or preliminary version of something from which other forms are developed.
Prototyping	is the activity of creating early sample versions of software. Typically simulation only a few aspects of the final product. There are two models for prototyping, throwaway model and evolutionary model.
Quality Control	A system of maintaining standards by testing a sample of the output against the specification.
Requirements Analysis	Also called requirements engineering, is the process of determining user expectations for a new or modified product. These features,(requirements) must be quantifiable, relevant and detailed. These requirements are often called functional specifications.
Requirements Documentation	(PRD) is a document containing all the requirements to a certain product. It is written to give people an understanding in what a product should do.
SCRUM	SCRUM is a methodology that allows a team to self-organise and make changes quickly, in accordance with agile principles.
SCRUM Master	A SCRUM master is the facilitator for an agile development team. They manage the process for how information is exchanged.
Secure Development	Secure development is a practice to ensure that the code and processes that go into developing applications are as secure as possible.
Social Engineering	The use of deception to manipulate individuals into divulging confidential or personal information that may be used for fraudulent purposes.
Software Architecture	Software architecture is the structure of structures, of a system consisting of entities and their properties, and the relationships among them.
Software Designer	A Software designer is responsible for the process of implementing software solutions to one or more sets of problems.
Software Developer	Develops software. Other job titles used with similar meanings are programmer, software analyst, and software engineer.
Software Release Engineer	concerned with the compilation, assembly, and delivery of source code into finished products or other software components.
SQL	SQL is an abbreviation for structured query language. SQL is a

	standardised query language for requesting information from a database.
SQL Injection Vulnerability	SQL injection is a code injection technique, used to attack data-driven applications, in which nefarious SQL statements are inserted into an entry field for execution.
Systems Architect	Systems architects define the architecture of a system in order to fulfil certain requirements. Such definitions include: a breakdown of the system into components, the component interactions and interfaces, and the technologies and resources to be used in the design.
Technical Architect	Responsible for defining the overall structure of a program or system, overseeing IT assignments that are aimed at improving the business, and ensuring all parts of the project run smoothly.
Test Plan	A test plan is a document detailing the objectives, target market, internal beta team, and processes for a specific beta test for a software or hardware product.
Test Script	is a set of instructions that will be performed on the system under test, to test that the system functions as expected.
Throwaway Prototyping Model	This prototype is thrown away once it is has been reviewed. They are quick to produce, and often visually resemble the final product but have no to little functionality
UML	Unified Modelling language (UML) is a standardised modelling language enabling developers to specify, visualise, construct and document artefacts of a software system.
Unit Testing	Unit testing is a software development process in which the smallest testable parts of an application, called units, are individually and independently scrutinised for proper operation.
Wireframes	A skeletal model in which only lines and vertices are represented.