Chapter 1

Introduction of Software

1.1 Software Overview

Types

- System Software
- Application Software

Good Software

Functionality	Reliability
Usability	Scalability
Performance	Security
Maintainability	Compatibility
Flexibility	Documentation
Interoperability	Error Handling
Compliance	Feedback Mechanism

1.2 Software Failures

Causes

- Missing Focus
- Content Issues
- Skill Issues
- Execution Issues

Issues that Affect Software

- Heterogeneity
- Bussiness and social change
- Security and trust
- Scale
- ◇ Gen AI:
  - Benifits:
    - Help generate ideas
    - Faster prototyping
    - Increased productivity and efficiency
    - Enhanced code quality
    - Process automation
  - Challenges:
    - Code reliability
    - Security risks
    - Dependence on AI
    - Intellectual property and ethical concerns

1.3 Software Process

- Requirement specification: Define what the software should do
- Development
  - Analysis: Create analysis models of the software system
  - Design: Write detailed design documentation
  - Implementation: Write the program code to implement the design
- Validation
  - Testing: Test the system to ensure it meets the requirements
  - Deployment: Release the software to users
- Evolution (Maintenance): Fix bugs and add new features after the software is in use

Software Process Models

- Waterfall Model
  - Advantages:
    - Easy to monitor and control
    - Documentation is well produced at each stage
    - Structured approach
    - Specialised teams can be used at each stage of the lifecycle
  - Disadvantages:
    - Inflexible to change
    - Time-consuming
    - Not realistic for large projects
    - Decision-making flaws in the early stages have been magnified
  - Only suitable for:
    - Small projects with well-defined requirements
    - Enhancements or adaptation to existing systems
    - High risk, safety critical projects which need high quality
- Evolutionary Development
  - Advantages:
    - Effective
    - Can meet immediate needs
    - Specification can be developed incrementally
  - Disadvantages:
    - Lack of process visibility
    - Poor structure
    - Special skills may be required
    - Scope Creep
  - Only suitable for:
    - Small or medium-sized interactive systems
    - Part of large system (UI)
    - Systems with poorly defined requirements
    - Short-lifetime systems
    - Project with multiple features which need release for testing

- RUP (Rational Unified Process)
  - Advantages:
    - Generic process framework
    - Seprate phases and workflows
    - Risk management
  - Disadvantages:
    - Overhead of documents and diagrams
    - High complexity
  - Only suitable for:
    - Medium to large complex systems
- ↑ Traditional Software Process Models
- ↓ Rapid Application Development (RAD)

Chapter 2

Agile Software Development

2.1 Agile Development Overview

- Focus:
  - Code
  - Based on an iterative approach
  - Deliver working software quickly to meet changing requirements
- Manifesto:
  - **Individuals and interactions** over processes and tools
  - **Working software** over comprehensive documentation
  - **Customer collaboration** over contract negotiation
  - **Responding to change** over following a plan
- Principles:
  - Customer involvement
  - Incremental delivery
  - People, not process
  - Embrace change
  - Maintain simplicity
- Iteration:
  - Planning: Define the scope and objectives of the iteration
  - Requirement analysis: Identify and prioritize user stories

  - Design: Create a high-level design for the iteration
  - Building: Develop the software incrementally
  - Testing: Test the software incrementally

- Extreme Programming (XP)
  - New versions may be built several times per day
  - Increments are delivered to customers every 2 weeks
  - All requirements are expressed as user stories
  - Programmers work in pairs
  - Develop tests before writing code
    * Helps develop common ownership of code
    * An informal review process
    * Encourages refactoring
    * Development productivity is similar to that of two programmers working separately
  - All tests must be run for every build and the build is only accepted if tests run successfully

- Problem:
  - Can be difficult to keep interest of customers who are involved in the process
  - Team members may be unsuited to the intense involvement that characterises agile methods
  - Prioritising changes can be difficult where there are multiple stakeholders
  - Maintaining simplicity requires extra work
  - Contracts may be a problem as with other approaches to iterative development

- Require:
  - Experience in the use of agile methods
  - Working environment that supports the agile process
  - Value thinking
  - Effective and efficient communication
  - Information sharing
  - Tools and Automation

- Suitable for:
  - Small to medium-sized systems
  - Systems with rapidly changing requirements
  - Rapidly evolving systems
  - Customer involvement is possible
  - Not a lot of external rules and regulations

2.2 Requirements

- Who provides → Stakeholders: Any person or group who is **affected by the system** and so who **has legitimate interest** in the system
- Understanding the requirements: Requirements Engineering
- Classification of requirements:
  - Functional requirements: What the system should do
    - Imprecision
    - Need completeness and consistency
  - Non-functional requirements: How the system should perform
    - Some are difficult to verify
    - Need to be measurable (quantitative)
- Conflict of requirements: Choose the best option and this has to be agreed by all stakeholders.
- Software requirements specification (SRS): A document that describes **WHAT** the system should do.
- Capturing requirements:
  1. Background reading
  2. Interviewing
  3. Observation
  4. Document sampling
  5. Questionnaires

In Agile Development – User Stories

- Format: **As a** [user role], **I want** [goal], **so that** [reason]
- Describe **WHAT the user wants to do**, not HOW the system should do it
- Project glossary: A list of terms and their definitions that are used in the project
- Epics: Large user stories that can be broken down into smaller user stories
- Acceptance Criteria: Conditions that must be met for a user story to be considered complete
- Product Backlog: A **prioritized** list of user stories that need to be implemented in the system
  - Prioritisation — MoSCoW Prioritization: A method for prioritizing user stories based on their importance
    - Must have: Critical requirements that must be implemented
    - Should have: Important requirements that should be implemented if possible
    - Could have: Nice-to-have requirements that could be implemented if time allows
    - Want to have: Requirements that are not essential and can be deferred to a later release
  - Estimating — Story Points: A relative measure that allows the team to understand the size of the effort
- Principles — INVEST:
  - Independent: User stories should be independent of each other
  - Negotiable: User stories should be negotiable and not set in stone
  - Valuable: User stories should provide value to the customer
  - Estimable: User stories should be easy to estimate
  - Small: User stories should be small enough to be completed in a single iteration
  - Testable: User stories should be testable and have clear acceptance criteria

Prototyping

- Low-fidelity prototypes: Simple sketches or wireframes that represent the layout and functionality of the system
- Medium-fidelity prototypes: More detailed representations of the system, often created using software tools

- High-fidelity prototypes: Fully functional versions of the system that closely resemble the final product

2.3 Design

- To precise understanding of requirements
- Describe HOW the system works
- Quality guidelines:
  - Meet the requirements
  - Be well structured
  - Be modular
  - Contain distinct repersentations of data, architecture, interfaces, and components
  - Be maintainable and traceable
  - Be well documented
  - Be efficient
  - Be error free
- Foundamental design concepts:
  - Abstraction

	Abstract Class	**Interface**
Defines behavior	✓	✓
Can have Implementation code	✓	
Cannot be instantiated	✓	✓
A class can	Inherit only one	Implement multyple
When to use	All class have same code	

  - Architecture
  - Modularity
    - Functional independence: Independent and interchangeable modules
    - Single responsibility: Each module contain only one specific function
    - Module interface: Define the elements to enable communication
    - Detectability: Other modules can recognize the elements

  - Patterns
  - Information hiding: Encapsulation
    * Restricting of direct access to some of an object's components
    * Bundling of data with the methods that operate on that data
  - Functional independence
    * Coupling: The degree of dependency between modules or components (Loose✓/ Tight×)
    * Cohesion: The correlation of internal functions of a module or a component (High✓/ Low×)
  - Refinement
  - Refactoring: Optimize and improve code structure without changing the external behavior of the code
    * Improve nonfunctional attributes of the software

- Object Oriented Design
  - Advantages:
    * Easier maintenance
    * Potentially reusable components
    * May be an obvious mapping from real world entities to system objects.

- Steps:
  1. Based on the conceptual class diagram produced from the Analisis stage
  2. Identifying Class relationships: Associations/Generalisations
  3. Identify operations
  4. Describe methods
  5. Capture implementation requirements
  6. produce detailed design class diagram

2.4 Implementation

- Design → Code
- Purpose: Implement the system in terms of components
- Build: The result of each incremental step

- Build Plan: Describe the sequence of builds (Functionality & affected part)
- Class Definition:
  - Class Diagram: Class name, attributes, operations
- Method Definition:
  - Show the sequence of messages that are sent in response to a method invocation
  - The sequences of these messages translate to a series of statements in the method definition
  - Parameters, return type, method decomposition
- Unidirectional:
  - One-to-one
  - One-to-many

2.5 Testing

- Aim: Verify the results from the **implementation stage**
- Goal:
  - Validation testing: Demonstrate the software meets its requirements
  - Defect testing: Discover defect
- Process: Unit & System & Acceptance (Alpha)
- Model: Cases (Prediction) & Data (Inputs) & Outputs
  - Test Case: To create a set of tests that are effective in validation and defect testing
  - Format:

Module	`Which Model`
Component	`Which class`
Set up	`Initialize setting`
Input Data	`Enter…`
Expected Result	
Actual Result	

- Defect format:

Defect number	
Defect title	
Bulid number	
Test number	
Description	`Description of the defect`
Assigned to Component Engineer	
Raised by/assigned to Test Engineer	

- Techniques:
  - Black-box testing: mainly on requirements
    - Partition testing: Test critical and intermediate regions
    - Regression testing: Verify that modifications to the code have not had a negative impact on existing functionality
  - White-box testing: mainly on internal program logic
    - Basis Path Testing: Cyclomatic Complexity = number of simple decisions + 1
    - Mutation Testing: To check if the existing tests can detect the errors (Robust)

Test-Driven Development (TDD)

- Red-Green-Refactor Cycle
  - Red: Write test code (Cannot run)
  - Green: Implement functions (Can run)
  - Refactor: Optimized code
- Characteristic:
  - Define interface and specification
  - Writing tests before code clarifies the requirements to be implemented
  - Incremental test development from scenarios
  - Automated tests are run after each change to the code
  - User involvement is essential

***JUnit**

- Method
  - `assertTrue(condition);`
  - `assertFalse(condition);`
  - `assertNull(object);`
  - `assertNotNull(object);`
  - `assertEquals(expected, actual);`
  - `assertArrayEquals(expected, actual);`
  - `assertEquals(expected[i], actual[i]);`
  - `assertArrayEquals(expected[i], actual[i]);`
  - `fail(message);`
- Create new test class: New → JUnit Test Case

Chapter 3

Software Project

3.1 Software Architecture

- Understanding how the system should be organized and how to design its overall structure
- Models: Simple box and arrow models or a UML diagram
- Advantages:
  - System analysis: Understant whether the system can meet its nonfunctional requirements
  - Large-scale reuse: Maybe reusable in other systems & product-line may be developed
  - Stackholder communication: As a focus of discussion
- Agility and architecture:
  - An early stage of agile processes is to design the overall architecture of the system
  - System architecture is expensive to change because it affects a lot of important components in the system
  - Focus on the most important design elements
- Architecture patterns: A stylized description of good design practice
  - Web-based: Client-server architecture (potentially organized into multiple tires, presence of a single point of failure and bottleneck)
  - Distributed systems: Replication and clusters, Load balancing, Caching, Serverless/cloud computing, Hadhoop and Map Reduce
  - RESTful (Representational State Transfer): Defines how to structure Web services (Let web resource can be accessed and manipulated using a uniform and predefined set of stateless operations)

  - RESTful Principles:
    * Client-server
    * Cacheablility
    * Uniform interface
    * Statelessness
    * Layered system
    * Code-On-Demand
  - Mobile applications: Layered architecture
    * Presentation layer
    * Bussiness layer
    * Data layer

3.2 Project Management

- Ensure software is delivered on time, within budget, and with quality
- Characteristic of Software Project:
  - Intangible and flexible
  - Not standardised
  - "one-off" project (Techniques change, Experience is obsolete, Need perceptive sight)
- Activities:
  - Proposal writing
  - Project planning: To make effective management at the start of the project
    * Iterative process: Only complete when the project is complete
    * Process (Bold text means in loop):
      · Establish constraints
      · Estimate parameters
      · Define milestones and deliverables
      · **Draw up schedule**
      · **Initiate activities**
      · **Review progress**
      · **Revice original plan**
      · **Update schedule**

            - **Regegotiate constrains and deliverables**
            - **If problem arise, replan the project**
        * Types:
            - Quality plan
            - Validation plan
            - Configuration management plan
            - Maintenance plan
            - Staff development plan
        * Milestones: Recognisable end-points
        * Deliverables: Project result deliverd to customers
        * Deliverables are usually milestones, but milestones need not be deliverables!
    - Project costing
    - Time management
    - Project monitoring and reviews
    - Personal selection and evaluation
    - Report writing and presentations
    - Risk management
    - Quality management
- Projecy scheduling: Estimate time and resources required to complete each task
    - Split project into separate tasks
    - Organise tasks concurrently
    - Minimise task dependencies
    - Depend on intuition and experience
    - Good rule:
        * Estimate as if nothing will go wrong
        * Add 30% for anticipated problems
        * Add futher 20% to cover unanticipated problems
    - Charts: Task chart, Activity network, Bar chart
- Monitoring: Daily stand up, Weekly reports/meetings, Customer mettings, Demos
- Monitor metric examples:
    - Number of lines of code

  - Number of defects in code
  - Test cases completed and in what time frame
  - Test cases passed/failed
- Most important assets → People
- Factors:
  - Consistency
  - Respect
  - Inclusion
  - Honesty

Agile Project Management

- Scrum approach benifits:
  - Modular development
  - Flexibly respond to changes in demand
  - Transparency and efficient communication
  - Incremental delivery and timely feedback
  - Trust and Positive Culture

Activity Network Diagram in Agile

- Types:
  - Activity on Arrow (AOA): Each activity is represented by an arrow, and the nodes represent the start and end of the activities
  - Activity on Node (AON): Each activity is represented by a node, and the arrows represent the dependencies between the activities
- Construct:
  - Activity: A task or a set of tasks that need to be completed
  - Merge Activity: An activity that has more than one predecessor
  - Parallel Activity: An activity that can be performed simultaneously with other activities
  - Path: A sequence of activities that lead from the start to the end of the project

  - **Critical Path**: The longest path through the network, which determines the minimum project duration
  - Event: A point in time that marks the start or completion of one or more activities
  - Burst Activity: An activity that has more than one successor
  - Dummy Activity: A dashed line that represents a dependency between two activities without consuming time or resources (Only in AOA, Too much use of dummy activities in a network creates confusion)

- Rules:
  - Flow from left to right
  - An activity cannot start until all its predecessors have been completed
  - Arrows indicate precedence and flow and can cross over each other
  - Each activity must have a unique number that is greater than any of its predecessors
  - Looping is not allowed
  - Conditional statements are not allowed
  - Use common start and stop nodes (for AON)

- Conputaion of ES, EF, LS, LF:

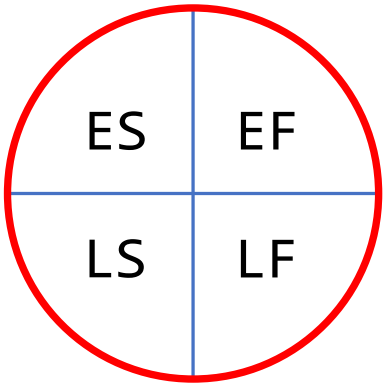

Figure 3.1: Activity Network Diagram

- Forward Pass:
  - ES (Early Start): The time an activity can start, given the completion of its predecessors (choose the largest ES of all predecessors)
  - EF (Early Finish): The time an activity can finish, pass it to its successors

  - Duration: The time required to complete an activity
  - ET (Expected Time): The expected time to complete an activity, calculated as the average of the optimistic, pessimistic, and most likely estimates
  - ES + Duration = EF
- Backward Pass:
  - LS (Late Start): The latest time an activity can start without delaying the project, pass it to its predecessors
  - LF (Late Finish): The latest time an activity can finish without delaying the project, given the completion of its successors (choose the smallest LF of all successors)
  - LS = LF - Duration
  - SL (Slack or Float): The amount of time an activity can be delayed without delaying the project, calculated as LS - ES
- Sensitivity: The likelihood the original critical path(s) will change once the project is initiated

3.3 Project Design Principles

- Trade off:
  - Meet customer needs
  - Be convenient to use
  - Be safe
  - Be efficient
  - Be secure
  - Meet legal requirements
  - Produced within a time limit
  - using available human, software and financial resources
- Liftspan:
  - Customer requirements change
  - Experience with use suggests new features
  - Errors or poor design features need to be corrected
  - Advances in technology open new possibilities

- Principles:
  - Think at a higher level than just code
  - Works correctly
  - Make it easy to develop further
- Poor quality:
  - Cause other parts stop working
  - Inflexible to change
  - Every change may causes all these get worse
  - Have very expensive consequences later
- Security problem: One program interacts with another program in an unexpected way
- Abstract way — UML: The nodes in UML class diagrams correspond to Java classes and interface types
- OO design — SOLID principles:
  - Single Responsibility Principle (SRP): A class should have only one reason to change
    - Disadvantages: Increase complexity, Increase coupling, Decrease cohesion, Create an overenginered system that becomes more difficult to maintain
  - Open-Closed Principle (OCP): Software entities should be open for extension but closed for modification
  - Liskov Substitution Principle (LSP): Objects of a superclass should be replaceable with objects of a subclass without affecting the correctness of the program
    - **Advantages:** Consistency, Flexibility, Maintainability, Robustness
  - Interface Segregation Principle (ISP): Clients should not be forced to depend on interfaces they do not use
    - Violations: Fat interfaces, Low cohesion, Empty method implementations, May break SRP
  - Dependency Inversion Principle (DIP): High-level modules should not depend on low-level modules. Both should depend on abstractions
  - Do not Repeat Yourself (DRY): Avoid duplication of code and logic in the system
  - Least Knowledge Principle (LKP, Law of Demeter): A module should not know about the inner details of the objects it manipulates

3.4 Software Design Patterns

- Design pattern: A general reusable solution to a commonly occurring problem in software design
- Types:
  - Creational patterns: Deal with object creation mechanisms, trying to create objects in a manner suitable to the situation
  - Structural patterns: Deal with object composition, creating relationships between objects to form larger structures
  - Behavioral patterns: Deal with object collaboration and responsibility, defining how objects interact and communicate with each other
- Importance:
  - "Tricks" which experienced programmers have come to use
  - A way of putting together code to solve a particular sort of problem which it has been noted is common enough to set it down as a general technique
  - Applicable in any programming language, but the idea developed in the field of object-oriented programming
  - A way to help keep programs in line with main principles of good program design
  - Learning the main design patterns is one "fast track" way to becoming an experienced programmer
  - Help Structure Code
    - Give ways of structuring code into parts with clear separate responsibilities (Single Responsibility Principle, SRP)
    - Enable code to be written in a generalised way (Open Closed Principle, OCP)
  - Reduce Dependency
    - Enable details connected to implementation to be hidden from code which uses objects (Dependency Inversion Principle, DIP)
    - The code which uses objects does not have to manage when to allow aliasing to save space (Object pool and Flyweight)
    - The code which uses objects does not have to manage the transition between one representation and another when for efficiency reasons it is a good idea to change (State Design Pattern)
  - Give a vocabulary which experienced programmers can use to communicate with each other when designing programs

- Danger and advice:
  - "Over-design" the code which make it more complicated than necessary
  - Over-use which lead to code which is more complex than necessary
  - Better to start with simple code and "refactor" it using one of the design patterns when more requirements are added later
  - Naturally produce code which is easy to refactor later on (principles of good design)
  - Pay attention to SOILD Design Principles

Wrapper Pattern

- Wrapper Pattern: An object (inner object) is wrapped inside another object (outer object) to provide additional functionality or behavior
- Decorator Pattern involves:
  - A class which implements an interface
  - Inside it a variable of the same interface type whose value is set in the constructor
  - Each method from the interface has code which calls the same method with the same arguments on that variable, plus some extra work
- Adapter Pattern involves:
  - An outer object which implements an interface wrapping an inner object which is of a class that does not implement that interface
- Composite Pattern:
  - A collection of objects of a type which is wrapped and given the behavior of a single object of that type
  - "Compose" objects into tree structures to represent part-whole hierarchies

Observation Pattern

- To cover cases where there is a link between two classes
- To connect objects which store data or perform actions to other objects which deal with "input/output"

Strategy Pattern

- Dynamically changes the behaviour of an object by encapsulating it into different strategies
- Choose from multiple algorithms and behaviours at runtime

Factory Method

- Uses factory methods to deal with the problem of creating objects without having to specify their exact class
- Work like constructors (without the `new` keyword)
- Can have an interface type as their return type
- Determine the actual type of a new object at run time
- Be used to hide the actual type of the new object
- Could produce a new wrapped object, or an object which puts a wrapper around an existing object

MVC

- Model-View-Controller (MVC): A software architectural pattern for implementing user interfaces
- Model: The data and business logic of the application
- View: The user interface that displays the data to the user
- Controller: The component that handles user input and updates the model and view accordingly

Immutable View

- Immutable View: A design pattern that ensures that the state of an object cannot be modified after it is created

3.5 Open Source Software

- Open Source Software (OSS): Software that is released with a license that allows users to view, modify, and distribute the source code
- Criteria:
  - Source code must be available
  - Derivative works must be allowed
  - Free redistribution must be allowed
  - Distribution of license must be allowed
  - Integrity of the author's source code must be maintained
  - License must not restrict other software
  - No discrimination against persons or groups
  - No discrimination against fields of endeavor
- Bussiness Strategies:
  - "Loss leader" (in business terms, a product sold at a low price to stimulate sales of more profitable goods or services)
  - Companies make profit from installation of OSS products, from offering training in the use of OSS products, and from providing consultancy advice to businesses about their use (Example: Red Hat)
- Development Strategies:
  - Software produced by volunteers cannot be produced under the forms of management used for commercial software
  - Large scale OSS systems have to be produced by teams
  - Developed by people working across the world, communicating electronically
  - It has some of the aspects of Agile software development, but the Agile methodology emphasises close personal interaction
  - With OSS systems there is typically no developer-customer distinction, the developers are also customers as they will make use of the system themselves
- Linus's Law: Given enough eyeballs, all bugs are shallow (The more widely available the source code is for public testing, scrutiny, and experimentation, the more rapidly all forms of bugs will be discovered)
- Open Source and Bussiness risks:

  - Develop own software: High risk, High cost, High time
  - Buy software from closed source vendor: High cost, Bind to vendor
  - Use an OSS product: Have source code, Several service companies who will help adapt it to your needs, no risk that a company supplying the software to you goes out of business, or makes unreasonable charges (You have no choice but to carry on using it)

- Software freedom:
  - Freedom to run the program
  - Freedom to study how the program works, and change it to meet the needs (Need access to the source code)
  - Freedom to redistribute copies
  - The freedom to distribute copies of your modified versions to others (Need access to the source code)

- Copyright:
  - Only the producers of the original work have the full right to produce copies, and create new work based on it
  - They may grant permission for others to make copies of the work and adapt it
  - Permission to make copies and adapt work will generally be made with the imposition of obligations on those doing it towards the original author, usually include making a payment to the original author, but could also include restrictions on the form of the copy and and any adaption
    - ◇ The idea is that people and organisations will be more willing to produce new work, if they are guaranteed to keep profits made from it

- Copyleft: Guarantees the freedom of others to copy and use their work rather than stopping unauthorised or unpaid copying
  - Involves a free software license, which is associated with free software, use of that software is granted only to those who agree to the terms of the free software license
  - The license will include the requirement that the source code is made available, and also state the extent to which the code may be modified and how the modifications should be acknowledged
  - Any work derived from the Copyleft license must also comply with the provisions of the Copyleft license
  - Example license: GPL, LGPL, Mozilla Public License 2.0, …

/ AOSP (Android Open Source Project):

- People and roles:
  - Contributor: Anyone making contributions to AOSP source code (Google employees and external developers)
  - Developer: An engineer writing applications that run on Android devices
  - Verifier: Responsible for testing change requests
  - Approver: Decides whether to include or exclude a proposed change
  - Project leader: Oversees the engineering for individual Android projects and is "typically employed by Google"
- Open Source (Android): Not very open-source
  - Android is closely controlled by Google, which gives priority access to specific developer groups and organisations, and does not make public its internal decision-making processes
  - Google does not provide public information regarding meetings held and decisions made on the development of Android
  - Google owns the Android trademark, and will not permit it to be used for devices which use Android code but do not pass a set of tests, the Compatibility Test Suite (CTS) which it controls

- Linux: An "open source" computer operating system
- Other examples: (Software)7-zip, Eclipse, Mozilla Firefox, Chromium, (Operating System)FreeBSD, ReactOS, FreeDOS, (Program Language)Python, PHP, PHDL, Perl
- Core and Community
  - Core people control the code base
  - A larger group of people have some involvement in correcting defects & reporting problems
  - Volunteers may submit code to implement proposed new features, the project may benefit from picking the best from several implementations
  - This system works best when the overall product naturally divides into many semi-independent parts
  - A fork is when there is a split, and a separate group starts maintaining a separate version of the product. Developers will prefer to contribute to an established product rather than fork it and develop their own new version
- Open Source Governance: Some organizations or companies are responsible for major major version updates of open source software

- Advantages:
  - Availability of source code:
    - Source code to understand and learn from
    - Do not have to re-invent the wheel
    - Free as in ”freedom”
  - Does not depend on vendor
  - Quality and Customizability in open source is better
  - Costs much less than proprietary counterparts
- Disadvantages:
  - Not generally straightforward to use and requires a certain learning curve to use and get accustomed
  - Incompatibility issue with software and hardware
  - Bad Codes, and some unqualified people who uses it
  - Software quality assurance process is widely not transparent
  - No financial incentive
  - May have security issues (Malicious code)

3.6 Software Development Tools

- Microsoft’s Best Practices:
  - Revision Control System (Version Control)
    - Incorporate roll-back features
    - Check-out
    - Check-in
    - Conflict
    - Merge
  - Daily Build: Having a practice of doing this once a day, usually overnight so the team starts work with the latest version of the software
    - Build: The process of producing a complete working version of the software for the whole system, compiling the source code files and linking them
    - Validation Test — “Smoke Test”
    - Build break: When the build or its tests fail, it should be considered a high priority problem that must be fixed immediately

    - Continuous Integration (CI):
        - Require a complete system build to be executed after check-in
        - Developers are only allowed to submit code after it has passed all tests
        - Purpose: By frequent integration and automated testing, defects can be detected early, improving code quality and team collaboration efficiency
  - Build Verification Tests
    - Unit Testing:
        - Assert statements: A statement involving an expression which is always meant to evaluate to true, if it does not an exception is thrown
        - Use `JUnit`
  - Bug Database: A central record for bugs that have been identified in the system
    - Role: Problem recording, Allocation and tracking, Collaboration and communication, Data analysis
    - Include: Title, Description, Replication steps, Severity, Priority, Status (active, resolved, closed), etc
    - War Team and Bug Bridge
        - Task: Ensure the system is good enough for release
        - Do: Triage remaining bugs
  - Code Reviews: A systematic examination of source code, involves analysing code to check for compliance with predefined sets of coding guideline rules or best practices
  - Coding and Engineering Guidelines
  - Code Analysis Tools
  - Globalization and Localization: Need to adapt to different language text display and corresponding reading order

- Debugger:
  - Enable the programmer to investigate the dynamic state of the program execution when it is suspended, looking at the value of variables and possibly changing them
  - May run code step-by-step or may enable the programmer to set breakpoints at which it suspends
  - Simple way: `print` the value to check
  - 9 essential rules:
    - Understand the system
    - Make it fail
    - Quit thinking and look

    * Divide and conquer
    * Change just one thing at a time
    * Keep an audit
    * Check the obvious first
    * Ask someone else
    * If you didn't fix it then it's not fixed

- IDE (Interactive Development Environments) vs Command line
  - Command line gives closer control over the computer
  - IDE is more convenient and automates many of the routine tasks needed for managing your files while developing software
  - Automation in command line interfaces can be done by writing scripts
  - Use of an IDE can mean you are not aware of the various tools it uses underneath, and have less flexibility over use of tools
  - It is a good idea to gain some experience in command line operation in order to better understand how your computer works

3.7 Ethical — Fair System

- Making the algorithm 'unaware' by obscuring features (Simple solution, Hide some features)
  - Data inherently biased
  - Indirect discrimination still happens through correlations
  - Algorithms are very good at picking up correlations
- Obscuring correlated features (Hide all features)
  - Decisions become inaccurate
- Add more features
- Pre-processing and Post-processing
- Fair Behaviour with Reward (Machine Learning)
  - Need to treat different features differently
  - Not legal in some countries
  - Not clear what the fairness criteria should be

- Auditing Fairness: The UK Data Protection Act (or European GDPR), 'Right to be forgotten'
  - Auditing the outcomes
  - Auditing the process
  - Auditing the algorithms
- Conclusion:
  - Data used in software or to train algorithms is often biased
  - Not mean that fairness in algorithms cannot be achieved
  - Making algorithms unaware does not work, and could make the outcome worse
  - Some solutions require discriminatory processes to compensate for bias
  - As a society we need to decide on fairness criteria, and how we would like to measure these - but more research is needed in understanding the trade-offs and metrics
  - Algorithm decisions are often questioned/audited, and it's clear how this can be done
- Code of Ethics: committed to the health, safety and welfare of the public, and adhere to eight principles
  1. Public: Act consistently in the public interest
  2. Client and employer: Act in a manner that is in the best interests of their client and employer, consistent with the public interest
  3. Product: Ensure that their products and related modifications meet the highest professional standards possible
  4. Judgment: Maintain integrity and independence in their professional judgment
  5. Management: Managers and leaders shall subscribe to and promote an ethical approach to the management of software development and maintenance
  6. Profession: Advance the integrity and reputation of the profession consistent with the public interest
  7. Colleagues: Be fair to and supportive of their colleagues
  8. Self: Participate in lifelong learning regarding the practice of their profession and shall promote an ethical approach to the practice of the profession.of engineering and technology achievements

3.8 Risk Management

- Protect the reputation of the company and/or prevent major potential losses or threats from operating 'business-as-usual'
- Account for the uncertainties in project delivery
- By identifying potential problems in advance means potential risk-handling measures can be planned (and taken as needed) to mitigate against any adverse impacts during the lifecycle of the project (or product)
- Process:
  - Risk Identification
  - Risk Analysis
  - Risk Planning
  - Risk Monitoring
- Types:
  - Project risks
  - Technical (or product) risks
  - Business risks
- Software Risk Metrics
  - Improve the management of software development and product risk
  - Provide indicators for the impact and/or likelihood of future risk, based on past experiences
  - Examples:
    * Technology risk exposure — the potential loss or damage to the business from the delivery or use of software
    * Technology risk maturity — the level of software development or integration practices
    * Number of risk incidents — the frequency, severity, and causes of software-related risk events
    * Technology risk response — the actions and resources required to deal with risk-related events
- Risk Mitigation/Avoidance Strategies

  - Understand the business environment: Be aware of the nature of the business, the market it operates in
  - Balancing innovation and stability: Becareful aboutusing new/unfamiliar products, keep awareness of new products, and do not stick to old tools and systems out of habit or unwillingness to change
  - Regularly synchronize progress: Ensure senior management are well informed on the progress of the project and its importance to the organisation's goals
  - Timely exposure of problems: Actively monitor performance, be prompt to report any slippage, and address/do not "hide" problems from senior management or staff
  - Skill decentralization: Ensurestaff are well trained, and avoid having just one person who has the skills or knowledge of a vital topic
  - Reduce the risk of personnel turnover: Encourage team building, and ensure the working environment is pleasant, so staff are less likely to leave the company

- Use of Contingencies:
  - Alternative resources being identified
  - Possibility of outsourcing work or alternative supply
  - Prioritising which parts of the project to continue, which to postpone or change
  - Identifying aspects of the project which could be modified
  - mean allowing extra time to complete a task, or to correct a problem
  - Damage limitation and Minimisation strategies

- Agile in Risk Management
  - Agile emphasises continuous testing throughout development, which should enable errors to be detected early (help)
  - Agile emphasises frequent interaction with the customer, so the customer can point out if the system is being developed in a way that seems to be unsatisfactory (help)
  - Agile does not emphasise long-term planning, which may mean it is harder to identify some risks (risk)
  - Without long-term planning, it is possible that features may be introduced that could cause problems later on (risk)
  - Agile also typically requires less interim deliverables, which may mean it is harder to identify problems step-by-step (risk)

3.9 Quality Management

- Ensure that software products reach an appropriate level of quality, meeting requirements for functionality, performance, and reliability
- Agile Quality Management:
  - VS Waterfall:

Waterfall	Agile
Quality assurance is an independent post process that undergoes testing and validation only after the completion of system development	Quality assurance is synchronized with development and runs through each iteration cycle
Easy to cause delayed problem discovery and high repair costs	Developing software by continuous release of new versions '' with extra features

  - All staff quality responsibility: Members need to specifically evaluate system quality from a customer perspective or Customer direct participation
  - Advantage:
    * Problems are discovered immediately during the development process, reducing large-scale rework in the later stages
- 'High Quality' Software:
  - Meet its requirements
  - Be efficient and easy to use
  - Be easy to modify to meey changing requirements
  - Pay attention to the main design principles while emphasizing the quality of internal code
  - Focus on 'fitness for purpose' rather than specification conformance
- Fitness for Purpose (Good enough software):
  - Have programming and documentationstandards been followed in the development process?
  - Has the software been properly tested?
  - Is the software sufficiently dependable to be put into use?
  - Is the performance of the software acceptable for normal use?

  - Is the software usable?
  - Is the software well-structured and understandable?
  - "Good enough" does not mean lack of attention to good standards during software development: sloppy standards early-on will lead to delay and more cost later, as well as poor quality products

- "Bug" or "Feature"
  - Bug: An aspect of the system's behaviour which is acknowledged as incorrect
  - Feature: An aspect of the system's behaviour which is acknowledged as correct
  - Usersmaybecomeaware ofaninconsistent or unintended behaviour and find it useful in some circumstances, in this way a "bug" becomes a "feature"

- Attributes: Necessary to trade one off against another

Safety	Understandability	Portability
Security	Testability	Usability
Reliability	Adaptability	Reusability
Resilience	Modularity	Efficiency
Robustness	Complexity	Learnability

- Visible and Invisible Quality:
  - From the point ofview of the customer, the most important quality aspects are what is immediately visible
  - Customer experience of using the system will also demonstrate other important quality aspects such as ease of use of the system, efficiency, quality of its support documentation
  - Quality aspects such as security and reliability are not immediately visible, but are of direct relevance to the customer
  - Code quality is not of direct relevance to the customer, but is important: Good quality code can be **easier** to modify to add new features, or to make required changes; Poor quality code is more likely to **contain bugs**, leading to poor reliability and possible security problems

3.10 Performance

- Poor performance → 'Bug'

- Solve:
  - Employ skilled programmers
  - Use library software for routine algorithms
  - Avoid using complex interactions
- Little point in improving the efficiency of Code

3.11 Reliability

- The probability of a software system operating without failure for a specified period of time in a specified environment
- Changes in the environment may lead to failure
- Not taken to ensure it would deal correctly with all situations encountered when used in practice
- Due to: Not taking account of a situation, or the implementation not meeting the specification
- Consider:
  - Possibility of the failure happening
  - Damage caused if the failure happen

3.12 Security and Resilience

- Core goal:
  - Ensure that software and data are not inadvertently misused or maliciously attacked by users
  - Even in the face of malicious users, software should maintain correct and predictable execution to prevent unauthorized access or data leakage
- Core Requirement:
  - Reduce vulnerabilities: Software should try to avoid exploitable weaknesses, abnormal behavior, or crashes
  - Anti attack capability: able to continue running and quickly recover even under incorrect use or attack

3.13 Standards

- Definition: A set of written rules to ensure that all participants adhere to uniform norms
- Scope:
  - International (such as ISO 9001), national, organizational, or individual projects.
  - Covering all aspects of software development (management, code style, documentation, etc.)
- Role:
  - Promote collaboration: Unified norms make team communication and cooperation more efficient
  - Save time: Adopt mature standards to avoid redefining rules for each project
  - Based on experience: Integrating past successful practices to avoid historical errors
  - Assist newcomers: Provide a clear guidance framework for new employees to quickly adapt to organizational norms
- Problem:
  - Outdated: Standards may become invalid due to technological updates or changes in experience
  - Side effects: Some protective measures in certain standards may solve one problem but trigger other problems
  - Bureaucratic risk:
    * Standards may be seen as tedious paperwork, wasting time rather than improving efficiency
    * If inefficient rules are mixed in the standard, its high-quality content may also be overlooked
  - Improve:
    * Regular review: It is necessary to combine the practical experience of current practitioners and clearly explain the logic of standard setting
    * Tool support: Reduce additional administrative burden through tools, but **poorly designed tools actually increase expenses**

3.14 Outsourcing

- Cause:
  - New workers need to be trained
  - Take time to understand enough aspects of an existing project
  - More possible communications are needed
- Problem:
  - Conflict of development modes: Continuous interaction and iterative development with customers are required, while remote outsourcing is difficult to achieve efficient collaboration
  - Language and cultural barriers:
    - Proficient in the client's language is required to accurately understand requirements, write documentation, and code
    - Cultural differences may lead to misunderstandings, and the efficiency of long-distance communication is lower than face-to-face communication
  - Team collaboration issues: Outsourcing teams are prone to developing a confrontational mentality of 'us vs. them'
  - Management and Cost Issues:
    - Additional administrative expenses
    - High modification costs and project delays
- Agile Development is better