Software Engineering

Chapter 1: Introduction

- The economies of all developed nations are dependent on software.
- More and more systems are software controlled
- Software engineering is concerned with theories, methods and tools for professional software development
- Software costs dominate computer system costs. The costs for software are often greater than hardware costs.
- Software maintenance costs more than to develop
- Causes of Software Project Failure:
    - Increasing system complexity
        - Demands change
        - Systems have to be built and delivered more quickly
        - Larger even more complex system required
    - Failure to use soft eng methods
        - Fairly easy to write computer programs without using software engineering methods and techniques. Consequently their software is often more expensive and less reliable

Professional Software Development

- What is software?
    - Consists of computer programs and associated documentation
    - Can be developed for specific customer or for a general market
- Attributes of good software?
    - Functional: delivers required functionality and performance
    - Maintenance: Easy to update and modify
    - Dependable: Reliable and secure
    - Usable: Intuitive and user friendly
- What is software engineering?
    - Engineering discipline that is concerned with all aspects of software production
- Fundamental Software Engineering Activities
    - Specification (defining requirements)
    - Development (designing and programming)
    - Validation (ensuring software meets requirements)
    - Evolution (adapting software to changing needs)
- Difference b/w Software Engineering and Computer Science
    - Software Engineering focuses on practicalities of developing and delivering software
    - Computer Science involves theory and fundamentals
- Difference b/w System Engineering and Software Engineering
    - System Engineering is concerned with all aspects of computer based system development including hardware, software and process engineering
    - Software Engineering is a subset focused on software specific process
- Key challenges in Software Engineering
    - Coping with Diversity: Software must work across different platforms and environments.

    - Reduced delivery time: Faster developments without compromising quality
    - Trustworthiness: Ensuring software reliability and security
- Cost of Software Engineering
    - 60% developments
    - 40% testing
- Best Software Engineering Techniques and Methods
    - You can't say one method is better than another
    - Methods vary by system types
- Impact of Internet on Software Engineering
    - Enabled development of distributed, service-based systems
    - Supported growth of mobile app industry, transforming software economics
- Program is about writing code while software engineering involves designing, developing, testing, deploying and maintaining software in systematic way
- Software Products
    - Generic Products
        - Standalone systems that are marketed and sold to any customer who wishes to buy them
        - Specification of product is owned by software developer and decisions of software changes is also made by developer
    - Customized Products
        - Software that is commissioned by specific customer to meet their own needs
        - Product specification owned by customer and they make decisions on software changes that are required
- Attributes of Good Software
    - Acceptability
        - Understandable, easy to comprehend
        - Simple to interact
        - Works well with other systems
    - Dependability and security
        - Consistently performs its intended functions
        - Prevents physical or economic harms in case of failure
        - Protects against virus
    - Efficiency
        - Ensures responsive and fast processing times
        - Minimize memory and process usage
    - Maintainability
        - Efficient updates
        - Easily adaptable to meet customer needs
- Software engineering is an engineering discipline (apply theories & tools *selectively*, under budget/schedule constraints, searching for workable compromises, not perfection) that is concerned with all aspects of software production from early stages of system specification through to maintaining the system after it has gone into use
- Importance:
    - Able to produce reliable and trustworthy systems economically and quickly
    - Usually cheaper to use software engineering methods and techniques for software systems
- Software Process Activities

    - **Specification** – define *what* & constraints.
    - **Development** – software is designed and programmed
    - **Validation** – testing & verification.
    - **Evolution** – maintenance & enhancement.
- Issues that affect software
    - Heterogeneity
        - Systems must operate across diverse platforms
        - Integration with older legacy systems
    - Business and Social Change
        - Rapid technological/social changes demand quick software evolution
        - Traditional methods are slow, delaying value delivery
    - Security and trust
        - Software must protect against malicious attacks
    - Scale
        - Software ranges from small scale to global cloud
- Software Engineering Diversity
    - Different systems have unique requirements and constraints
    - Stand alone applications
        - Run on personal devices without requiring network
    - Interactive transaction based system
        - Remote systems accessed via web
    - Embedded control system
        - Software embedded in hardware devices
    - Batch processing system
        - Process large amount of data
    - Entertainment system
        - Games and other personal use system
    - Data collection system
        - Collect and process environmental data
    - System of systems
        - Combination of different software system working together
- Four Fundamentals Common to All Systems
    - Managed, understood process
    - Dependability + Performance focus
    - Rigorous specification & requirements management
    - Strategic reuse of existing software assets
- Internet software engineering
    - The web is now a platform for running application and organizations are increasingly developing web based systems
    - Web services allow application functionality to be accessed over web
- Web software engineering impact
    - Software reuse
        - Web systems are often assembled using pre-existing components and framework
    - Incremental development
        - Systems evolve over time
    - Service oriented SE
        - Web based systems are built using standalone web services

    - Interface development testing
        - Use of AJAX and HTML5 to create rich, browser based interface

Software Engineering Ethics

- Software work takes place inside a **social + legal framework**; engineers' freedom is constrained by laws, contracts, and professional duty.
- **Public impact:** Code can cause economic loss, privacy leaks, or physical harm → society expects engineers to behave beyond mere "coding skill."
- Four baseline obligations (beyond the law):

#	Duty	Practical meaning	Red-flag examples
1	**Confidentiality**	Guard employer / client secrets, designs, data.	Sharing pre-release source or customer data with friends, speaking at a conference without clearance.
2	**Competence**	Accept only work within your true skill set; disclose limits.	Overselling your machine-learning expertise, leading to an unsafe medical-diagnosis app.
3	**Intellectual-property respect**	Follow copyright, patents, licences; protect client IP.	Copy-pasting GPL code into a proprietary project.
4	**No computer misuse**	Don't exploit systems for personal gain or disruption.	Casual game-playing on employer servers, writing malware or data-scrapers for shady actors.

- ACM/IEEE Software Engineering Code:

#	Principle	Essence
1	**Public**	Put public interest & welfare first.
2	**Client & Employer**	Act in their best interests *consistent with the public interest*.
3	**Product**	Deliver high-quality, standards-conformant software.
4	**Judgment**	Maintain independent, honest professional judgment.
5	**Management**	Leaders must foster an ethical development culture.
6	**Profession**	Promote integrity and good standing of SE discipline.
7	**Colleagues**	Be fair, supportive, share knowledge, credit others.
8	**Self**	Lifelong learning, ethical self-improvement.

Chapter 2: Software processes

- A **software process** is a set of related **activities** that lead to the **production of a software system**.
- Four software processes:
    - Specification - defining what systems should do
    - Design and implementation - defining the organization of system and implementing system
    - Validation - checking that it does what customer wants
    - Evolution - changing the system in response to customer needs
- Software process descriptions

Element	Description
Activities	What people *do* (e.g., implement, test, model).
Products/Deliverables	What is *produced* (e.g., source code, UML diagrams, test reports).
Roles	*Who* is involved and what responsibilities they carry (e.g., tester, architect).
Preconditions/Postconditions	Constraints that must hold *before and after* an activity. Example: Before architecture design → all requirements approved; after → design reviewed.

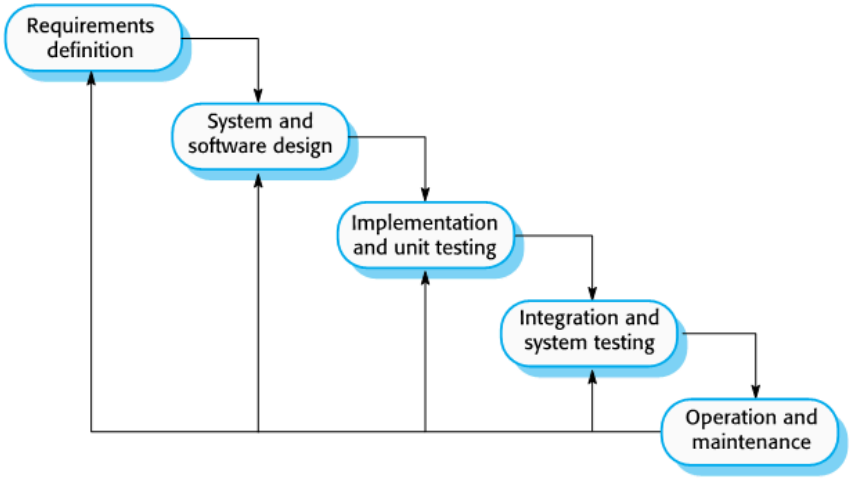
- Plan-Driven vs. Agile Processes

Plan-Driven	Agile
Process and schedule **fully planned** in advance	Planning is **incremental** and reactive
Progress measured against predefined milestones	Progress adapts to changing needs
Emphasis on **predictability, documentation, and control**	Emphasis on **flexibility and working code**
Ideal for **large-scale, safety-critical systems**	Ideal for **dynamic, fast-changing business needs**

Software Process Models

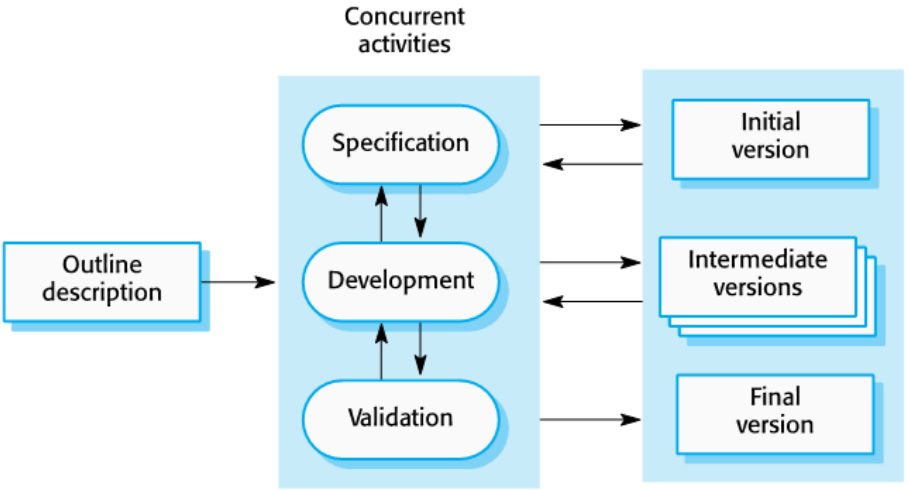
- A **software process model (SPM)**, or **Software Development Life Cycle (SDLC)**, is a **simplified representation** of a software process.
- 1) Waterfall Model
    - Plan driven
    - Separate and distinct phases of specification and development
    - Linear and sequential

- Development progresses through FIXED stages each completed before moving to next
- Requires detailed planning and documentation at beginning



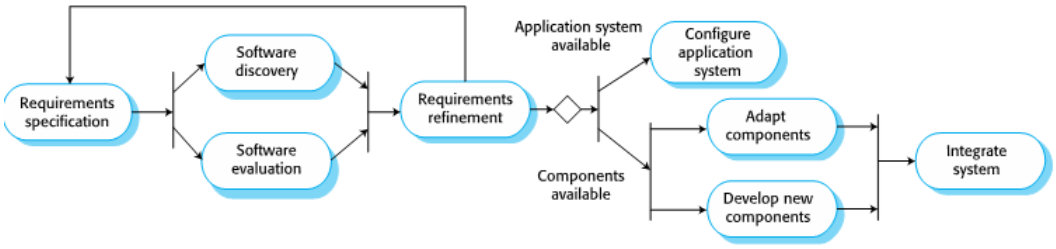
- Phases
  - Requirements analysis & definition
  - System and software design
  - Implementation & unit testing
  - Integration & system testing
  - Operation & maintenance
- Drawbacks:
  - Difficulty of accommodating change after process is underway
  - Phase has to be complete before moving onto next phase
  - Hard to adopt changing requirements
- Mostly used for large systems engineering projects

- 2) Incremental development
  - Plan driven or agile
  - Requirements divided into stand alone modules of software development cycle



- Benefits
  - Cost of accommodating to customer requirement is reduced
  - Easier to ge customer feedback on development work that has been done (Agile)
  - More rapid delivery and deployment of useful software to customer is possible
- Problems
  - Process is not visible
    - Managers need clean deliverable to assess progress
    - No comprehensive documentation since its costly and time consuming
  - Systems structure tends to degrade as new increments are added
    - Code structure deteriorates no clean code
    - Messy unorganized code makes system harder and more expensive
    - Regular refactoring agile teams refactor code frequently

- 3) Integration and Configuration
  - Based on software reuse where systems are integrated from existing components
  - Even though components are reused, they can be configured and adopted to match specific user needs
  - Types of reusable software
    - Stand alone applications
    - Component packages
    - Web services

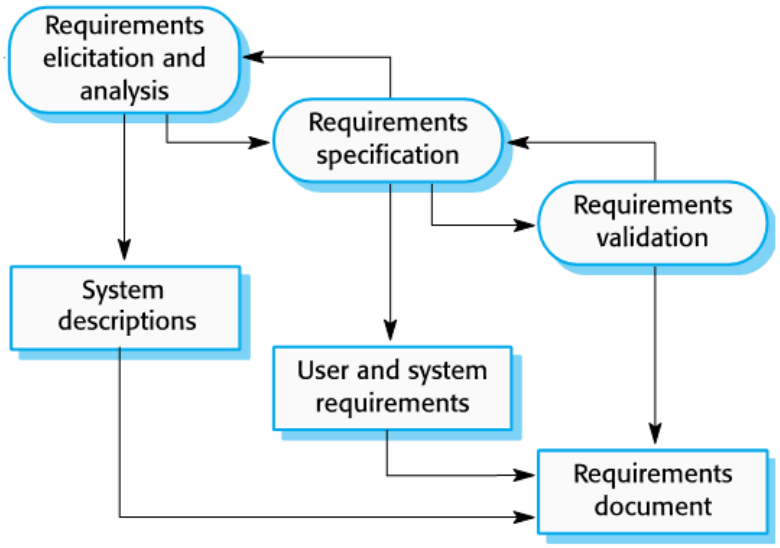


- **Initial Requirements Specification** – Brief and flexible
- **Software Discovery and Evaluation** – Find candidate components
- **Requirements Refinement** – Modify based on what's available
- **Application System Configuration** – Adapt entire systems (e.g., ERP)
- **Component Adaptation and Integration** – Use, extend, or compose modules
- Advantages
  - Reduced costs and risks as less software is developed from scratch
  - Faster delivery of system
  - Loss of control over evolution of reused system elements
- Disadvantage
  - **Requirements compromises** likely

        - **Limited control over component evolution**
        - Dependency on third-party versioning and support

Process Activities

- The four basic activities are organized in sequence in Waterfall Model while in incremental they are interleaved
- Software Specification
    - Goal: Determine *what* the system should do and *under what constraints*.



Requirements Elicitation & Analysis

- Observing systems, interviews, task analysis, prototyping.

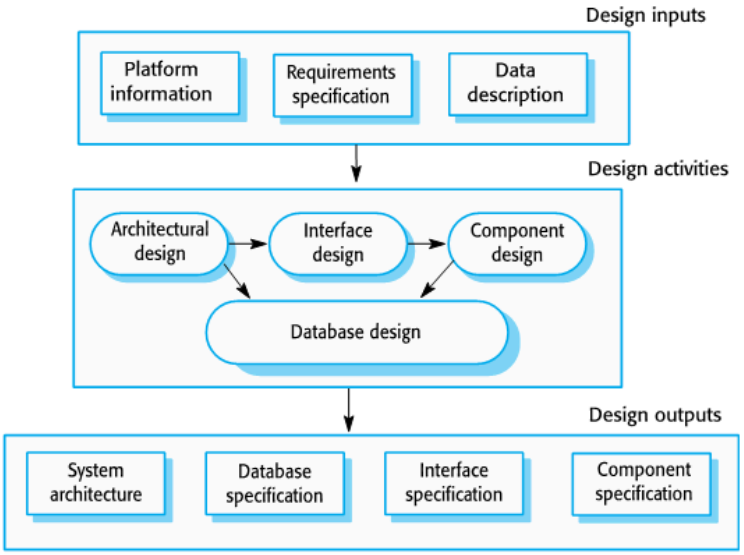
Requirements Specification

- Translate gathered info into:
    - **User requirements** (abstract, readable)
    - **System requirements** (detailed, for devs)

Requirements Validation

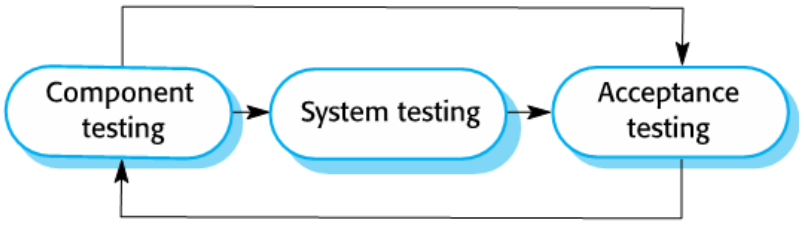
- Check for **completeness**, **realism**, **consistency**.

- Software Design and Implementation
  - Process of converting system specification into an executable system



Activity	Description
1. Architectural Design	Define high-level structure, modules, their relationships
2. Database Design	Define data structures and DB schema
3. Interface Design	Define how components interact (precise, modular interfaces)
4. Component Selection & Design	Reuse where possible; otherwise design new components

- Software Validation (V&V)
  - V&V is intended to show that a system confirms to its specification and meets user requirements
  - Involves checking and review processes and system testing



Stage	Focus
1. Component Testing	Test individual modules (unit tests, often automated e.g., JUnit)
2. System Testing	Test integration and emergent behavior
3. Customer Testing	Final acceptance testing, real data, beta users

    - Agile vs. Plan-Driven Testing:
        - Planning
            - Agile: Test cases created before development (TDD)
            - Plan-driven: Test plans based on spec & design
        - Responsibility
            - Agile: Developers write & run tests
            - Plan-driven: Independent testing teams execute test plans
        - Feedback loop
            - Agile: Rapid
            - Plan-driven: Formal & structured
- Software Evolution
    - Ongoing modification of a system in response to new requirements or issues.
    - Why software must evolve:
        - Changing business rules
        - Bug fixes
        - Performance/security updates
        - Customer needs shift

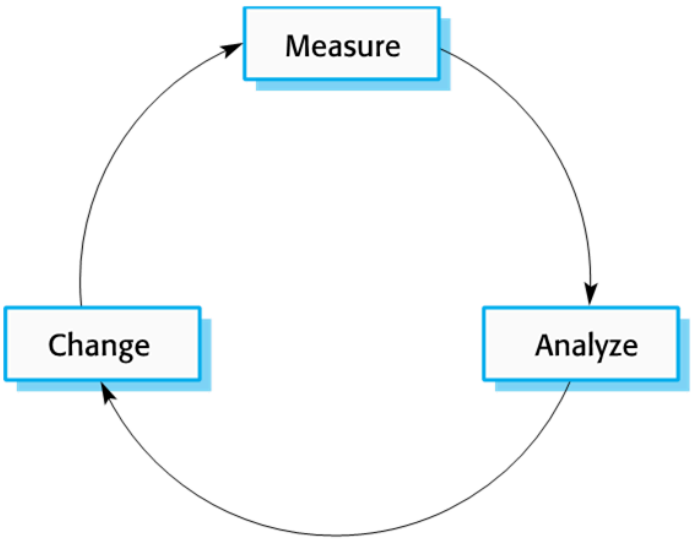
Coping with the change

- Change is inevitable in large software products due to evolving business needs
- Change leads to rework costs, new feature implementation
- Reducing rework costs:
    - Using early prototypes to predict possible changes before development progresses
    - Using incremental developments allowing changes in future increments
- Coping with changing requirement (same thing as above):
    - System prototyping: quickly developing a system version to verify customer needs and design functionality
    - Incremental delivery: delivering system in parts
- Software prototyping
    - A prototype is an early version of a system used to demonstrate concepts and validate design decisions
    - Benefits:
        - Better usability and design quality
        - Closer alignment with actual user needs
- Prototype developments
    - Uses rapid prototyping tools
- Throw away prototype
    - Prototypes should be discarded after development as they are not good basis for production system
- Incremental delivery
    - Deliver software **in small, functional increments** over time.
    - User requirements are prioritised and highest priority requirements are included in early increments
- Incremental development and delivery
    - Incremental development:
        - Develop system in increments and evaluate each increments before proceeding to development of next increment
        - Used in agile method
        - Evaluation done by user
    - Incremental delivery
        - Deploy an increment for use by end user
- Advantages
    - Customer value can be delivered with each increment so system functionality is available
    - Early increments act as prototype to help elicit requirements for later increments
- Problems:
    - Can't easily replace existing full systems
    - Contracts often require full specifications early (conflicts with incrementalism)

Process Improvement

- Why Improve Software Processes?
  - **Industry demands**:
    - Cheaper software
    - Better quality
    - Faster delivery
  - **Goal**: Increase **product quality**, reduce **costs**, and **accelerate development**
  - Enhance quality of the software
- Two Major Approaches to Process Improvement

Approach	Focus	Strength	Challenge
1. Process Maturity Approach	Improve project/process management	Enhances **predictability** and **product quality**	Adds **overhead**, requires detailed documentation
2. Agile Approach	Reduce overhead through iteration and feedback	Emphasizes **speed**, **customer responsiveness**, and **lean practices**	Skeptical of heavy process formalization

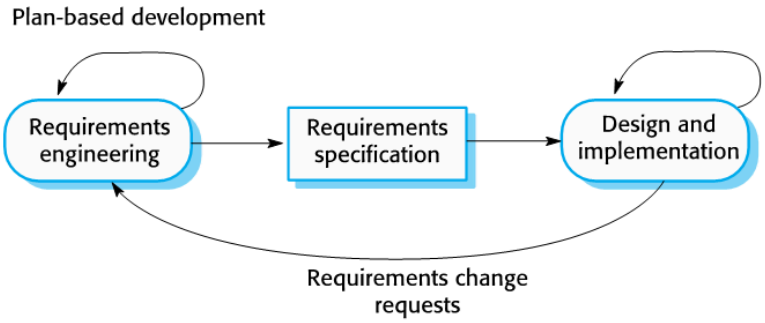


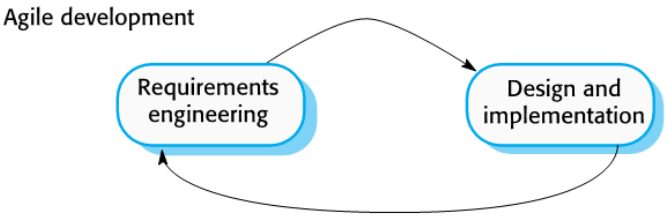
- Measurement: Collect baseline data on process/product (e.g., defect rates, delivery times)
- Analysis: Identify process bottlenecks, weaknesses, and delays

- Change: Propose and introduce changes to improve process → start next cycle
- Process Metrics
    - Time taken for process activities to complete
    - Resources required for processes or activities
    - Number of occurrences of particular event

Chapter 3: Agile software development

- Rapid software development
    - Rapid development and delivery is now often the most important requirement for software engineering
    - Software has to evolve quickly to reflect changing business needs
- Problems with plan-driven
    - Changing requirements not easy to accommodate
    - Late delivery
    - Heavy documentation
    - Does not meet business needs
- Agile development
    - System developed in series of versions or increments with stakeholders involved in version specification and evaluation
    - Minimal documentation
    - Quick delivery of new versions for evaluation





- Plan-driven development
    - Structured approach where software development process is divided into separate stages, with outputs defined and planned in advance
- Agile development:
    - In agile specification, design, implementation and testing occur in parallel and these activities are interwoven

- Agile methods were introduced
    - To focus on code rather than design
    - Intended to deliver working software quickly and evolve this quickly to meet business needs
    - Aim is to reduce overheads in software processes and be able to respond quickly to changing requirement
- Agile manifesto
    - Values individuals, working software, customer collaboration and responding to change over tools, documentation, contact negotiation and following plan
- Agile principles
    - Customer closely involved throughout development process to evaluate iteration
    - Software developed in increments with customer specifying requirements
    - Expect system requirements to change and so design system to accommodate changes
    - Focus on simplicity in both software being developed and in development process
- Agile is ideal for
    - Product development where software company is developing small medium sized product
    - Custom Systems – Where the customer is actively engaged and fewer external constraints exist.
- Agile development techniques
    - Extreme programming
        - Introduces incremental development with small releases often delivering new versions every two weeks
        - Takes extreme approach by frequently producing new versions of software, testing every build and delivering updates often
        - Software build and tested regularly to ensure it works and meets evolving requirements
        - * XP Practices
            - Incremental planning
                - Stories are broken down into tasks and release is planned based on priority and available time
            - Small releases
                - Systems minimal useful functionality is developed first
                - Each release adds more feature incrementally
            - Simple design
                - Only necessary design is carried out (design is minimal)
            - Test first development
                - An automated unit test framework is used to write tests before implementing functionality, ensuring that every new feature is properly tested
            - Refactoring
                - Continuous code improvement
                - Refactoring keeps code clean and maintainable
            - Pair programming

        - Developers work in pairs, providing continuous peer support and ensuring quality through real time feedback
    - Collective ownership
        - All developers are responsible for entire codebase, allowing anyone to make changes to any part of system
    - Continuous integration
        - After completing tasks a code is integrated into whole system immediately
    - Sustainable pace
        - No excessive overtime is allowed
        - Maintaining sustainable pace ensures high quality code over long term
    - Onsite customer
        - Customer representative is available full time to provide constant feedback ensuring system meets user requirements
- Influential XP practices
    - Not easy to integrate with management practice in most organizations
- User stories for requirement
    - A customer/user is part of XP team and is responsible for making decisions on requirements
    - User requirements are expressed as user stories or scenario
    - These are written on cards and the development team breaks them down into implementation tasks. These tasks are bases of schedule and cost estimates
- Refactoring
    - Traditional software engineering suggests designing system to anticipate changes, reducing costs when changes occur
    - XP rejects anticipating changes in advance as its impossible to reliably foresee what changes will be needed. XP focuses on constant code improvement to make future changes easier
    - Changes cant be always predicted since its hard to foresee all changes
    - Benefits
        - Developers regularly improve software even when there is no need ensuring system is clearer and easier to modify
        - Well structured code improves understandability
    - Challenges
        - Costly and complex
- Test first development
    - Testing is central in XP where the program is tested after every change
    - Writing tests before code helps clarify requirements that need to be implemented
    - Tests are written as programs not data making them executable automatically and ensuring they work correctly
    - All tests (new and old) are run each time a new functionality is added ensuring no errors are introduced
    - Key Features
        - Tests are written before code is implemented

        - Tests are developed incrementally based on user scenarios
        - Customers help in test development and validation
        - Automated test harnesses are used to run all tests after each change and release
    - Customer involvement
        - The customer helps develop acceptance tests for stories to be implemented in next release
        - Customer is part of team writing tests as development progresses
        - The customer may have limited availability and may not be fully involved in testing process
    - Test automation
        - Tests are written as executable components before task is implemented
        - Tests are run quickly and easily whenever new functionality is added
    - Problems
        - Developers may prefer code over testing leading to incomplete task
        - Writing test can be challenging
        - Developers may skip edge cases
- Pair programming
    - Developers work in pair developing code together
    - Benefits
        - Common ownership of code and shared knowledge
        - Acts as informal review process as each line of code is examined by more than one person
        - Encourages refactoring as whole team benefits from improved system code
        - Reduces risks when team member leaves
        - Pairs are created dynamically ensuring all team members work together during development
        - Crucial for reducing risks when team members leave
        - Efficient
- Agile project management
    - Agile project managers focus on delivering software on time
    - Typically plan driven where entire process and deliverables are planned in advance
    - A different approach is required in agile focusing on iterative and incremental development
- Scrum
    - Is an agile method that focuses on managing iterative development
    - Phases
        - Initial phase
            - Outlining planning and establishing software objectives
        - Sprint cycle
            - Develop an increment of system in each cycle
        - Project closure phase
            - Wrap up project, complete documentation and asses lessons learned
    - Scrum terminology (a)

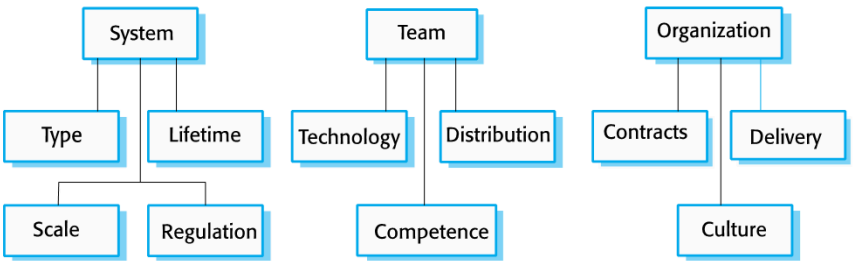
        - Development team
            - No more than 7 software developers
            - Self organising team responsible for developing software and project documents
        - Potentially shippable product increment
            - Product increment delivered from sprint ideally in finished state
        - Product backlog
            - List of tasks that the scrum team needs to complete including software features and requirements
        - Product owner:
            - Person responsible for defining features and continuously prioritizing product backlog
    - Scrum terminology (b)
        - Scrum
            - Daily meeting of scrum team to review progress and prioritise work for the day
        - Scrum master
            - Ensures scrum process is followed and guides the team in using scrum effectively
        - Sprint
            - Development iteration usually lasting 2-4 weeks
        - Velocity
            - Measures amount of product backlog effort that the team can complete in a sprint
    - Sprint Cycle:
        - Review backlog.
        - Plan sprint (select items, estimate time).
        - Daily Scrum for updates.
        - Sprint review for feedback and process improvement.
    - Scrum sprints have fixed lengths of 3-4 weeks. The planning begins with product backlog and selection phase involves team working with customer to choose the features to be developed during sprint
    - Teamwork in scrum
        - Scrum master facilitates these daily meetings, tracks project progress and any arising problems. Entire team attends these meetings to share their progress and problems
    - Distributed scrum
        - Videoconferencing
        - Shared whiteboards / Scrum boards
        - Real-time tools (Slack, Jira)
        - Regular cross-time-zone meetings
- Scaling agile methods
    - They work well for small and medium sized projects but face challenges when scaling up. Scaling involves adjusting principles for larger projects
    - Scaling up vs scaling out
        - Scaling up involves applying agile to larger, more complex systems while scaling out involves applying agile across multiple teams or departments

- Practical problems with agile methods
  - Agile developments informal approach is not well suited to legal contract definitions used by large companies
  - Agile methods work best for new software development but most software costs in large companies arise from maintaining existing software systems where agile may be less effective
  - Designed for small co located teams but modern software development often involves globally distributed teams/best for small teams working closely together
  - Large companies usually follow strict legal approaches for contracts in big companies while agile is more flexible and informal
  - Contractual issues
    - Most custom software contracts are **specification-based**, defining exact deliverables.
    - This clashes with agile norms which interleave specification and development (i.e., evolve requirements).
    - Agile-friendly contracts:
      - Focus on developer time rather than fixed functionality.
      - ⚠️ Risk: Legal departments see this as risky—outcome cannot be strictly guaranteed.
  - Agile Methods and Software Maintenance
    - Software maintenance is more expensive than new development.
    - For Agile to be sustainable, it must support both initial development and ongoing maintenance.
  - Agile Maintenance Issues
    - Key Problems:
      - Lack of documentation.
      - Low customer involvement over time.
      - Risk of losing team continuity.
    - Agile relies heavily on **team knowledge** instead of external documents
  - Agile vs. Plan-Driven Methods
    - Projects often mix **Agile** and **Plan-driven** methods.
    - Need for **detailed specifications** → go Plan-driven.
    - Use of incremental delivery + fast feedback → go Agile.
    - System scale: Small systems suit Agile; large, distributed systems often require Plan-driven.
  - Agile Principles & Organizational Practice

Principle	Real-World Practice Challenges
Customer involvement	Customers may lack time or full context; stakeholders like regulators can't easily engage with devs.
Embrace change	Hard in multi-stakeholder systems; conflicting change priorities.

Incremental delivery	Short-term cycles often clash with long-term marketing & planning.
Principle	**Real-World Practice Challenges**
Maintain simplicity	Schedule pressure discourages devs from simplifying systems.
People over process	Agile may not suit teams with **weak interpersonal dynamics** or mismatched personalities.

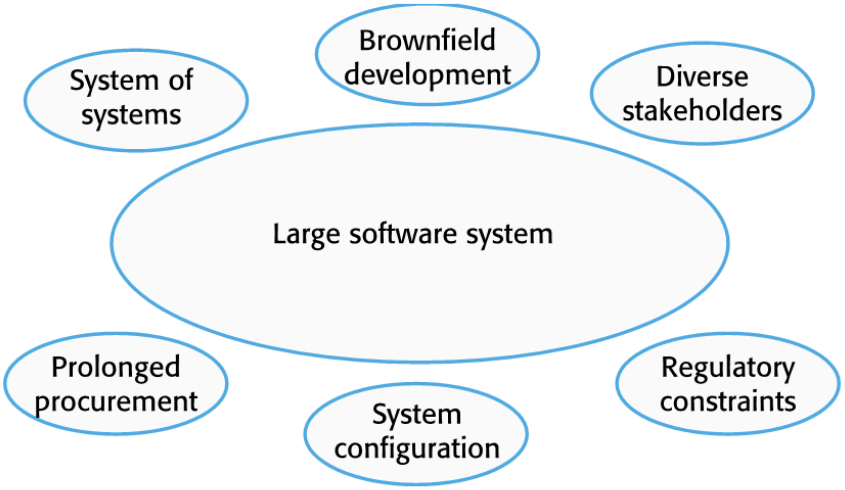
- Agile and Plan-Based Factors



- System Issues
  - Size: Agile suits small, co-located teams.
  - Type: Complex systems needing detailed analysis may require plan-driven methods.
  - Lifetime: Long-living systems need solid documentation for maintainability.
  - Regulations: Compliance requires formal documentation → less agile.
- People and Teams
  - Skill level: Agile needs **highly skilled** developers.
  - Distributed teams: Require **formal design docs** for clarity.
  - Tools: IDEs and visualization tools are essential when **documentation is minimal**.
- Organizational Issues
  - Traditional engineering orgs favor **plan-driven** culture.
  - Important to consider organizations approach to system specification, customer involvement and agile's informal nature

- Agile Methods for Large Systems?? Idk bro
- Large System Development Challenges
  - Regulatory constraints affect how systems are developed.
  - Long procurement timelines → difficult to retain consistent team members.
  - Stakeholder diversity → hard to involve everyone in the process.

- Key Factors in Large Systems



- IBM Agile Scaling Model (ASM):
  - Disciplined Agile Delivery
  - Agility at Scale (governance, regulation, legacy integration)
  - Scrum at Scale:
    - Multiple Scrum teams
    - Scrum of Scrums
    - Aligned releases
    - Team-specific Product Owners
- Multi-Team Scrum
  - **Role replication**: Each team has its own Product Owner and ScrumMaster
  - **Product architects** ensure aligned system architecture.
  - **Release alignment**: Teams coordinate on product release timelines.
  - **Scrum of Scrums**: Daily coordination meetings across teams.

- Daily standup
  - Is a quick time boxed meeting 15 min where team discusses what they've accomplished
  - Can be held virtually via video or audio
- Sprint planning
  - Meeting where scrum team collaborates to determine which user stories they'll work on
- Sprint review
  - Is held at end of each sprint to demonstrate the work completed
- Sprint retrospective
  - A Sprint Retrospective is a meeting where the Scrum Team **inspects how the last sprint went** (in terms of people, processes, and tools) and **identifies improvements** for the next sprint.

Chapter 4: Requirements engineering

- Process of determining what services is required by user and constraints under which system operates
- **Outcome**: System requirements — clear descriptions of services and constraints generated during requirement engineering process
- What is the requirement?
    - Ranges from abstract statements to detailed specifications.
    - Dual Purpose:
        - For bidding (general/abstract)
        - For contract (detailed/spec-specific)
- Type of requirements
    - System requirement (how system works for developers)
        - Structure document setting out detailed descriptions of system functions, services and operational constraints
        - Used by developers and contractors
    - User requirements (what system does for customers)
        - Statements in natural language plus diagrams of services
        - Written for customers
- System stakeholder
    - Anyone impacted by the system or has a legitimate interest.
    - Types:
        - End users
        - Managers
        - Owners
        - External stakeholders
- Agile methods and requirements
    - Many agile methods argue that producing detailed system requirements is a waste of time as requirements change so quickly
    - Requirement document is always out of date
    - Agile methods usually use incremental requirement engineering and may express requirements as ‘user stories’
- Functional vs Non Functional Requirements
    - Functional
        - Statements of services the system should provide how the system should react to particular input and how system should behave
        - May state what system should not do
        - Describes what systems should do in terms of services, behaviours and responses to input
    - Non Functional
        - constraints/functions/services offered by system
        - Define what constraints and qualities of system
        - Define limitations and constraints on system like HOW QUICKLY it should respond, how much time it can take
- Functional requirement
    - Describe functionality or system services
    - Depend on type of software, expected users and type of system where software is used

    - Functional user requirements may be high level statements of what system should do
    - Describe system services in detail
    - Example in Mentcare system
        - A user shall be able to search appointments lists for all clinics
        - System shall generate each day, for each clinic, a list of patients who are expected to attend appointments that day
    - Problems arise when functional requirements are not precisely stated
    - Ambiguous requirements may be interpreted in different ways by developers and users
    - Consider the term search in requirement 1
        - User intention - search for patient name across all appointments in all clinics
        - Developer interpretation - search for a patient name in an individual clinic. User chooses clinic then search
- Non functional requirements
    - These define system properties and constraints
    - Process requirements may also be specified mandating a particular IDE, programming languages or development method
    - Maybe more critical than functional requirements
    - May affect overall architecture of a system rather than the individual components
    - Product requirement
        - Specify that delivered product must behave in particular way
    - Organisational requirement
        - Consequence of organisational policies and procedures
    - External requirements
        - Requirements which arise from factors which are external to system and its development process
    - Usability requirements
        - System should be easy to use by medical staff and should be organized in such a way that user error are minimised
- Metrics for specifying non-functional requirements
    - Speed
        - Processed transactions
        - user/event response time
        - Screen refresh time
    - Size
        - Mbytes
        - No of RAM chips
    - Ease of use
        - Training time
        - No of help frames
    - Reliability
        - Mean time to failure
        - Probability of unavailability
        - Rate of failure occurrence
        - Availability

    - Robustness
        - Time to start after failure
        - Percentage of events causing failure
        - Probability of data corruptions on failure
    - Portability
        - Percentage of target dependent statements
        - Number of target system
- Requirement engineering processes
    - The processes used for RE vary widely depending on application domain, the people involved and organisation developing requirements
    - However there are number of generic activities common to all processes
        - Requirement elicitation
            - Gather info on what system needs to do
        - Requirement analysis
            - Breaking down and understanding gathered info
        - Requirement validation
            - Ensuring requirements are correct and meet projects needs
        - Requirement management
            - Managing and tracking requirements
    - RE is an iterative activity in which these processes are interleaved
    - Requirement Elicitation
        - Involves technical staff working with customers to find out application domain the services that the system should provide and systems operational constraints
        - May involve end-users managers engineers involved in maintenance domain experts trade unions etc. These are called stakeholders
        - Work with range of system stakeholders to find out about application domain
            - Stages include
                - Requirement discovery - interacting with stakeholders to discover the requirements, domain requirements are also discovered
                - Requirement classification and organization - groups related requirements and organises them into coherent clusters
                - Requirement prioritization and negotiation - prioritising requirements and resolving requirements conflicts
                - Requirement specification - requirements are documented and input into next round of spiral
        - Problems
            - Stakeholders don't really know what they want
            - Express requirements in their own terms
            - Different stakeholders may have conflicting requirements
            - Organisational and political factors may influence system requirements
            - Requirements change during analysis process
            - New stakeholders may emerg and business environment may change

- Interviewing
  - Formal or informal interviews with stakeholders are part of RE processes
  - Types of interviews
    - Closed interviews based on predetermined list of questions
    - Open interviews where various issues are explored with stakeholders
  - Effective interviewing
    - Be open minded avoid preconceived ideas about requirements and are willing to listen to stakeholders
    - Prompt interviewee to get discussions going using a springboard question, requirements proposal, or by working together on prototype system
  - Interviews in practice
    - Normally a mix of closed and open-ended interviewing
    - Interviews are good for getting an overall understanding of what stakeholders do and how they might interact with system
    - Interviewers need to be open minded without preconceived ideas of what system should do
    - You need to prompt user to talk about system by suggesting requirements rather than simply asking then what they want
- Ethnography
  - A social scientist spends a considerable time **observing and analysing how people actually work**
  - People do not have to explain/articulate their work
  - Social and organisational factors of importance may be observed
  - Ethnographic studies have shown that work is usually richer and more complex than suggested by system model

BAS HOGYI AGAY KHUD PRHO

Chapter 6: Architectural Design

- Focuses on organizing and structuring the software system
- Acts as a bridge b/w design and requirements engineering
- Identifies the main structural components in a system and their relationships
- The result of architectural design process is an architectural model
- Describes system's organization as set of communicating components
- Agility and architecture
  - Early stages of agile processes involve designing the overall system architecture
  - Refactoring the architecture later can be expensive because it affects multiple system components
- Architectural abstraction
  - Deals with the internal design of **individual programs**, breaking them down into **components**

    - Involves **enterprise-level systems**, including multiple **subsystems**, **distributed components**, and often spans **multiple machines or organizations**.
- Advantages of Explicit Architecture
    - Helps stakeholders discuss and understand the system.
    - Allows checking if non-functional requirements (e.g., performance, security) are met.
    - Enables reuse across different systems; supports creation of product-line architectures.
- Architectural Representations
    - Block diagrams: Common, simple diagrams to show entities and relationships.
    - Often **lack semantics**, don't show detailed **relationship types** or **properties** of components.
    - Depends on **how the model is used**
- Uses of Architectural Models
    - Discussion: High-level overviews help stakeholders talk about system design without getting lost in detail.
    - Documentation: Models serve as formal records showing system components, interfaces, and connections.
- Design Decisions
    - Architectural design is creative and depends on the system.
    - But some choices (like how to manage performance or security) apply to many systems.
- Architecture Reuse
    - Similar systems in the same field use similar architecture.
    - A **core design** can be reused with small changes to fit customer needs.
    - Some systems use common **design styles** or patterns.
- System qualities

Quality	How Architecture Helps
Performance	Keep critical tasks close together, reduce chatter
Security	Use layers, put secure parts deep inside
Safety	Put safety-critical parts in small subsystems

Availability	Add backups and fault tolerance
Maintainability	Use parts that are easy to replace

- Architectural Views

  You need different **views** of the system:

  - **Logical view** – what functions it does.
  - **Process view** – how parts interact during execution.
  - **Development view** – how it's broken into modules for developers.
  - **Physical view** – where components run (hardware).
- How to show these views?
  - Some use **UML diagrams**, but the book says it's not ideal.
  - **ADLs (Architecture Description Languages)** exist but aren't popular
- Architectural Patterns
  - Reusable solutions that **share good design ideas**.
  - Help standardize and **reuse** proven methods.
  - Can be shown as tables or diagrams.
- 1) Layered Architecture
  - System is divided into layers (e.g., UI layer, logic layer, database layer).
  - Each layer does specific tasks.
  - Easy to change one layer without affecting others.
  - But it can sometimes be **unnatural** or forced.
- 2) Repository Architecture
  - All subsystems **share data** through:
    1. A **central database** (common).
    2. Or each has its own database and shares manually.
  - Useful when **huge amounts of data** are involved.
- 3) Client-Server Architecture
  - Distributed system model where processing is split across clients and servers
  - **Components**:
    - Stand-alone **servers** (e.g., printing, data services)
    - **Clients** that request services
    - A **network** connecting them
- 4) Pipe and Filter Architecture
  - Data flows through components (filters) connected by pipes.
  - **Key Features**:
    - Used in batch data processing (e.g., UNIX shell)
    - Sequential transformations
    - **Not ideal for interactive systems**
- 5) Application Architectures
  - Designed to meet organizational needs.
- Uses of Application Architectures
  - Design checklists

    - Guide development teams
    - Help reuse components
- 6) Information Systems Architecture
    - **Layered design** for transaction-based systems.
- Language Processing Systems
    - **Function**: Transform language input (natural/formal)
    - **Includes**: Interpreters and algorithm-based tools
    - **Use Cases**: Meta-case tools (rules, methods)

Chapter 7: Design and Implementation

- **Design** = thinking and planning how software should work
- **Implementation** = actually writing the code
- Both often happen together (interleaved).
- In small systems, design = implementation. In large systems, design is one stage of many.
- Use UML for design
- Understand how your system connects to its environment.
- Use two models:
    - System Context Model (what systems are connected)
    - Interaction Model / Use Cases (how it interacts)
- Architectural Design
    - Shows major parts (components/subsystems) of your system and how they work together.
    - Weather stations have subsystems: Data collection, Communications, Config Manager, etc.
    - Uses **listener architecture**: subsystems listen for commands/messages.
- Object Class Identification
    - Identify main objects and classes in the system.
    - Group similar objects into superclasses using inheritance.
- Design Models
    - Two main types of UML models:
        - **Structural Models** – classes and relationships (e.g., inheritance)
        - **Dynamic Models** – runtime behavior (e.g., sequence, state diagrams)
- Interface Specification
    - Define how components talk to each other (methods, not data).
    - One object can have multiple interfaces.
- Design Patterns
    - Reusable solutions to common problems.
- Implementation Issues
    - Reuse
    - Use existing:
        - **Designs (patterns)**
        - **Objects** (libraries)
- **Components** (e.g., UI frameworks)
- **Systems** (off-the-shelf apps)

Pros: Faster, cheaper, reliable
Cons: Compatibility, integration, cost of adapting

- Configuration Management
    - Keeps track of:
        - Different **versions**
        - **System builds**
        - **Bug tracking**
        - **Releases**
    - Use tools like **Git**, **GitHub**, **Bugzilla**, etc
- Host-Target Development
    - Develop on one machine host
    - Run it another target especially for embedded system
    - May use simulators if hardware not available
- Open-Source Development
    - Code is **public** and can be modified by others
    - Projects often have a **core team** + contributors
    - **Benefits**:
        - Free/cheap
        - Reliable
        - Big community
    - **Challenges**:
        - Licensing issues
        - Not all projects get help
- Open-Source Licensing

License	Rule
GPL	If you use it, your code must be open too
LGPL	Can link to it; only changes to original code must be open
BSD/MIT	Do anything you want, just credit original author

Chapter 9 : Software Evolution

- Testing Strategy
  - Guide how software testing should be planned and executed.
  - **Types**:
    - **Static vs Dynamic**: Without running code vs with running code.
    - **Preventive vs Reactive**: Test early to prevent bugs vs test after issues appear.
    - **Hybrid**: Mix of both approaches.
- Program Testing
  - Testing checks if the software does what it's supposed to **and** finds defects before launch.
  - Uses **artificial data** to test the system.
  - Testing shows the **presence** of bugs, not their **absence**.
  - Part of the broader **verification and validation (V&V)** process.
- Program Testing Goals
  - Test all features mentioned in the requirements.
  - Detect wrong or unexpected behavior (e.g. crashes, incorrect outputs).
- Validation vs Defect Testing
  - Ensures the system behaves **correctly** as expected.
  - Designed to **break** the system or expose hidden bugs.
- Testing Process Goals
  - Validation
    - Confirm software meets user needs.
    - A good test = correct output.
  - Defect testing
    - Identify faults
- Verification vs Validation
  - Verification
    - Focus on meeting specifications.
  - Validation
    - Focus on meeting **user needs**.
- V&V Confidence
  - Build **confidence** that the system is fit for its job.
- Inspections vs Testing
  - Review documents/code without running them (**static**).
  - Run the program with test data (**dynamic**).
- Software Inspections
  - People manually check requirements, designs, etc. for issues.
  - Useful before actual coding starts.
  - Effective for **early error detection**.
- Stages of Testing
  - **Development Testing**
    - Testing during development to discover bugs.
  - **Release Testing**
    - Independent team tests complete system version before release.
  - **User Testing**
    - Users test in their environment (e.g., beta testing).

- Development Testing Types
  - **Unit Testing:** Test individual classes/functions.
  - **Component Testing:** Test combined units.
  - **System Testing:** Test the complete integrated system.
- Unit Testing
  - Tests components in isolation.
  - Defect focused
  - Can be:
    - Single functions/methods
    - Object classes
    - Composite components
- Object Class Testing
  - Involves:
    - Testing all operations
    - Setting/interrogating attributes
    - Exercising in all possible states
- Automated Testing
  - Automate unit testing with frameworks like **JUnit**
- Components of Automated Tests
  - **Setup Part** – Initialize inputs and expected output
  - **Call Part** – Call the method
  - **Assertion Part** – Compare actual vs expected result
- Choosing Unit Test Cases
  - Two types:
    - Normal operation tests
    - Tests for abnormal input (edge cases)
- Testing Strategies
  - Partition Testing:
    - Input divided into classes
    - Select test from each class
  - Guideline-Based Testing:
    - Based on common programmer mistakes
- Partition Testing Concepts
  - Use **equivalence partitions**
  - Each partition is a domain of similar behavior
  - Choose test cases from every partition
- Testing Guidelines (Sequences)
  - **Single-value sequence:** Test with a one-element sequence to check behavior on minimal input.
  - **Varying sizes:** Use sequences of different lengths to ensure scalability and boundary behavior.
  - **Positional access:** Ensure the **first**, **middle**, and **last** elements are tested for correct indexing and boundary errors.
  - **Empty sequence:** Always test with zero-length inputs to check for null or empty array exceptions.
- General Testing Guidelines
  - Trigger **all possible error messages**.

    - Include **repeated inputs** to test for memory leaks or unintended side effects.Force the program to produce **invalid outputs** or incorrect computation results (too large/small).
- Component Testing
    - Components are often composed of **multiple interacting objects**.
    - Testing should focus on verifying the **public interface** of the component, **not internal details**.
    - **Assumes unit testing** of internal objects is already done.
- Interface Testing
    - Catch faults due to **incorrect usage of interfaces**.
- Interface Errors
    - **Misuse**: Wrong parameter ordering or number.
    - **Misunderstanding**: Wrong assumptions about interface behavior.
    - **Timing**: Data read/written at incorrect times due to async processes.
- Interface Testing Guidelines
    - Test **edge-case parameter values**.
    - Include tests with **null pointers**.
    - Apply **stress tests** (especially for message passing).
- System Testing
    - Integrate components and test them as a **whole system**.
    - Ensure **interactions between components** are correct.
    - Verify correct data transfer and **compatibility**.
    - Checks the system's **emergent behavior**.
- System and Component Testing
    - Often done by a **dedicated testing team**, separate from developers.
- Use-Case Testing
    - Use **realistic user scenarios** as the basis for tests.
    - **Sequence diagrams** help visualize and derive relevant test cases.
- Test-Driven Development (TDD)
    - TDD is a software development approach where tests are written before the code itself.
    - Testing and coding are interleaved, meaning each increment of functionality includes a test.
- TDD Process Activities
    - Identify a small, specific functionality.
    - Write a corresponding automated test.
    - Run the test — it should fail (functionality is not yet implemented).
    - Implement the functionality.
    - Re-run the test — it should now pass.
    - Move to the next increment only after passing.
- Regression Testing
    - Ensure new changes don't break previously working functionality.
    - **Manual vs. Automated**:
        - Manual regression is time-consuming and costly.
        - Automated regression is efficient and can run on every update.
- Release Testing
    - Tests the *release version* of the system for external use.

    - Usually **black-box testing**, based on system specification, not internal design.
- Requirements-Based Testing
    - Create tests for each requirement.
- Performance Testing
    - Assesses system's **speed, responsiveness, and reliability**.
    - Simulates realistic usage patterns.
- User Testing
    - Occurs **after system & release testing**.
    - Done with **real users** to capture real-world usability and environmental effects.
- Acceptance Testing Process
    - Define acceptance criteria.
    - Plan tests.
    - Derive test cases.
    - Run tests.
    - Negotiate outcomes.
    - Accept or reject the system.
- Agile & Acceptance Testing
    - No separate acceptance testing phase.
    - Tests are:
        - Defined by the user.
        - Run automatically with every update.

Chapter 9: Evolution

- What is Software Evolution?
    - Software needs to change over time due to:
        - New user requirements
        - Business or hardware environment changes
        - Bugs or performance improvements
    - Managing these changes is essential for organizations.
- Why is Evolution Important?
    - Software is a **critical asset**.
    - Most software budgets are used to **maintain and update**, not build new software.
- Stages in Evolution
    - **Evolution**: New features added during operational use.
    - **Servicing**: Only small changes like bug fixes.
    - **Phase-out**: No further changes; system may still be used.
- Evolution Processes
    - Driven by **change proposals**.
    - Depend on:
        - Type of software
        - Dev process used
        - Skill of developers
    - Change happens throughout the system’s life.
- Change Implementation

    - First step: **Understand the code**
    - Then: Design, implement, and test changes.
- Agile & Evolution
    - Agile = Continuous development & evolution.
    - Frequent releases + automated tests.
    - Changes = new user stories.
- Agile Handover Issues
    - Agile team → Plan-based team = Documentation gap.
    - Plan-based → Agile team = No refactoring/tests in place.
- What are Legacy Systems?
    - Old systems using outdated tech.
    - Includes software, hardware, processes, policies.
    - Still used because replacement is **risky and expensive**.
- System Metrics
    - Collect data like:
        - Change requests
        - Interface complexity
        - Volume of data
        - More requests = lower quality.
- Software Maintenance
    - After deployment, mostly:
        - Bug fixes
        - Adapt to new environments
        - Add/change features
- Predicting Maintenance Costs
    - Depends on:
        - Number of changes
        - System complexity
        - Maintainability
- Reengineering
    - Revamp old systems without changing what they do.
    - Benefits:
        - Lower risk
        - Lower cost
    - Involves:
        - Code translation
        - Reverse engineering
        - Structure/data improvement
- Refactoring
    - Clean and simplify code to avoid future issues.
    - Doesn't add features—**just makes code better**.
- Code "Bad Smells"
    - Duplicate code → Use reusable methods
    - Long methods → Split into smaller ones
    - Switch statements → Use polymorphism
    - Data clumps → Make classes
    - Over-general code → Remove if unused

BORING ASF COURSE KILL ME

Agile Methodologies

- Core Ideas of Agile
  - Agile = a **project management framework**
  - Project split into **phases (sprints)**
  - Agile is:
    - Iterative (repeats in cycles)
    - Reflective (team reviews itself)
    - Adjustable (improves over time)
- Types of Agile Methodologies
  - Kanban
  - Scrum
  - XP (Extreme Programming)
  - FDD (Feature-Driven Development)
  - APF (Adaptive Project Framework)
  - ASD (Adaptive Software Development)
  - XPM (Extreme Project Management)
  - DSDM (Dynamic Systems Development Method)
- Kanban (what we did on Trello)
  - Visual board-based system
  - Tasks = cards
  - Stages = columns (e.g. To Do → Doing → Done)
  - Move cards to track task progress
  - Helps **see roadblocks** and **manage workload**
  - Best for **continuous delivery**

  - Eg Trello
- Scrum
  - Most **popular** Agile method
  - Ideal for **small teams**

    - Work done in **sprints** (1–4 weeks)
    - Key roles & events:
        - Scrum Master = team guide (The person who helps the team follow Agile rules and removes any blockers.)
        - Daily Standups (Quick daily team meetings to update progress and share issues.)
        - Backlog (A list of all the work (tasks) that needs to be done.)
        - Sprint Planning (A meeting where the team decides which tasks to complete in the next sprint.)
        - Sprint Retrospective (A meeting after the sprint to discuss what went well and what can be improved.)
- Extreme Programming (XP)
    - Frequent **code reviews**
    - Short **development cycles**
    - Less documentation
- Feature Driven Development (FDD)
    - Focused on **specific features** of a product
    - Strong **customer input**
    - Works in **iterations**
    - Allows **quick updates and fixes**
    - Blends different Agile practices

API calls with Postman

- HTTP Request Methods
    - **GET:** Ask for data from the server
    - **POST:** Send new data to the server
    - **PUT:** Update existing data on the server
    - **DELETE:** Remove data from the server
- HTTP Headers
    - **Accept:** What type of response you want (e.g., JSON, image)
    - **Authorization:** Credentials (like tokens or keys)
    - **Content-Type:** Format of data being sent (e.g., JSON, HTML)
    - **Content-Length:** Size of the data sent
    - **Accept-Encoding / Content-Encoding:** Compression info (e.g., gzip)
    - **Cache-Control:** Tells browser how long to store the response
- API Endpoint Example
    - Base URL for testing:

      `https://cacb277107c12f42fb40.free.beeceptor.com/api/users/`

- **Actions you can do:**
    - List users
    - Add user
    - Edit user
    - View one user
    - Delete user

API Integration Issues

- Data Consistency
  - Data between backend and frontend must match.
  - Errors happen if formats or rules are different.
- API Versioning
  - New backend changes can break the frontend.
  - Versioning helps keep old and new systems working together.
- Authentication & Authorization
  - Secure login and access control is important.
  - Mismatches can block access or expose data.
- Performance Optimization
  - Too much data moving between backend and frontend slows things down.
  - Use caching, lazy loading, and fewer API calls to improve speed.
- Error Handling & Logging
  - Good error messages and logs help fix problems fast.
  - Without them, debugging is hard and slow.
- Concurrency & Race Conditions
  - Multiple users at once can cause conflicts.
  - Use synchronization to control request order and avoid bugs.
- Testing & Debugging
  - Test backend and frontend together to catch bugs early.
  - Use automated and integration tests to save time.
- Documentation & Communication
  - Clear docs and teamwork help avoid confusion.
  - Everyone should understand how APIs work.

API

- What is an API?
  - **API** = Application Programming Interface.
  - It helps two software programs talk to each other.
- Types of APIs
  - **SOAP API**: Uses XML to share data. Old and less flexible.
  - **RPC API**: Client calls a function on the server, and gets the result.
  - **WebSocket API**: Two-way communication using JSON. Server can send messages anytime.
  - **REST API**: Most common. Uses HTTP and supports actions like GET, POST, DELETE.
- REST APIs
  - **REST** = Representational State Transfer.
  - Uses HTTP methods (GET, POST, PUT, DELETE).
  - **Stateless**: Server doesn't remember previous client requests.
  - Like browsing a website – you send a URL (request), get data (response).
- API Endpoints

    - **Endpoints** are the final destinations for API calls (e.g., URLs, services, ports).
- Microservices
    - A way of building apps as **small, separate parts** (services).
    - Each service does one task and talks to others using HTTP or messages.

Estimation

- What is Estimation?
    - Estimation = Predicting how much **time, effort, or money** is needed to build or maintain software.
    - Estimates help in **project planning, budgeting, pricing, and resource allocation**.
- How to Estimate Cost and Effort
    - Know your **Software Development Life Cycle**.
    - Understand **what your project needs**.
    - Break the project into **smaller tasks**.
    - Use a **reliable estimation method**.
- Why Estimation Matters
    - Helps allocate **resources wisely**.
    - Keeps the **team happy** by avoiding overwork.
    - Sets **realistic expectations** for clients.
    - Maintains **quality** and avoids technical debt.
- Estimation Techniques
    - **Method 1) Experience-based:** Use past experience.
        - Based on your team's past projects.
        - List deliverables and estimate time for each.
        - Group discussion makes estimates more accurate.
        - Problems:
            - Doesn't work well for **new or unfamiliar technologies**.
            - Your past experience may not apply.
    - **Method 2) Algorithmic model:** Use formulas based on project data (like size or team skill).
    -
        - Uses a **formula**:

$$\text{Effort} = A \times (\text{Size})^B \times M$$

        - **A**: Company-specific constant
        - **B**: Size-based adjustment
        - **M**: Factors like people/process quality
        - Needs product attributes (like **code size**).

- Estimation Accuracy
  - True size of software is only clear **after completion**.
  - Factors like reused code and language impact size.
  - Estimates get better as the project progresses.
- Popular Estimation Techniques

PERT: Uses Optimistic, Likely, and Pessimistic times:

$$\text{Estimate} = \frac{O + 4M + P}{6}$$

- Best for **complex projects with risk**.

Analogous Estimation:

- Compare with **similar past projects**.

Agile Estimation:

- Use **points**, not time.
- Involve whole team.
- Faster and simpler.

- Method 5) Agile Estimation: Story Points
  - Story Points = Unit to estimate task difficulty.
  - Consider:
    - Complexity
    - Risk/Unknowns
    - Familiarity
    - Use numeric scales like Fibonacci (1, 2, 3, 5, 8...)
- Agile Estimation Process
  - Identify user stories.
  - Discuss requirements.
  - Use a numeric scale.
  - Pick a technique.
  - Plan sprint.
  - Make sure team is consistent in estimates.
- Sprint Velocity
  - How many story points a team completes in one sprint.
  - Helps **predict timeline** using past data.
- Method 6) Three-Point Method
  - Use **3 time estimates** (best, worst, most likely).
  - Average them for final estimate.
  - Best for **uncertain projects**.
- Method 7) Planning Poker
  - Team members **vote** using cards.
  - Discuss and agree on a final estimate.

    - Best for **small/medium teams**.
- Method 8) Bucket System
    - Group tasks into **"buckets"** based on effort.
    - Similar to T-shirt sizing, but still **subjective**.
- Method 9) LSU: Large Small Uncertain
    - Categorize tasks as:
        - Large
        - Small
        - Uncertain
    - Estimate based on these groups.
    - Needs team agreement on categories.
- Estimating Cost
    - Total story points = 1000
    - Velocity = 100/story points per sprint
    - Sprints = 10
    - Cost/sprint = $900
    - Total Cost = $9000
- Future of Estimation
    - **Humans are biased** and limited.
    - **AI tools** can give better, faster, more accurate estimates.
    - AI is the future of project estimation.

UI Design

- UI design is done in steps with users involved.
- Three main steps:
    - User analysis – Understand what users want.
    - Prototyping – Make rough versions of the design.
    - Evaluation – Test these with real users.
- Interface Design Models
    - There are 4 types of design views:
        - Design model – What developers build.
        - User model – What we know about users.
        - User's model – How users think the system works.
        - System image – What users actually see.
- GUI (Graphical User Interface)
    - Pros:
        - Easy to use and learn.
        - Fast screen interactions.
        - Good for multitasking.
    - Cons:
        - Bad designs cause errors.
        - Can make people avoid using the software.
- Direct Manipulation Interfaces
    - Pros:
        - Feels like you control the system.

        - Fast learning.
        - Immediate feedback.
    - Cons:
        - Hard to design.
        - Tough to use in big systems.
- Command Interfaces
    - You type commands to control the system.
    - Pros:
        - Good for fast, expert use.
        - Can combine commands.
    - Cons:
        - Hard for beginners
        - Must memorize commands.
        - Needs good error handling.
- Usability Testing
    - Test users doing sample tasks.
    - Record results via video or code.
    - Ask for feedback.
    - Test with real users and goals in mind.
- UI vs. UX
    - **UI (User Interface)**: The look (buttons, colors, layout).
    - **UX (User Experience)**: The whole feeling and journey using the product.
- UX Designers
    - Focus on:
        - How users feel and behave.
        - Solving problems.
        - Making usable and enjoyable experiences.
- UI Designers
    - Focus on:
        - Visual design.
        - Page layouts, colors, fonts, buttons.
        - Making it look and feel great.
- Wireframes
    - Basic sketch of how a page will look.
    - Shows layout and features, without styling or color.
- Wireframe Benefits
    - Helps design team plan layout.
    - Shows user navigation.
    - Makes design intentions clear.
- Types of Wireframes
    - **Low-fidelity**: Quick, rough sketches.
    - **High-fidelity**: Detailed and closer to final look.

Deployments Branching CI/CD

- Deployments are becoming more frequent (e.g., Tesla, WhatsApp).
- Benefits: Faster product delivery, quick feedback, happier developers.
- Drawbacks: May cause reliability issues.
- Apps now use cloud and microservices.
- They must handle scale, failure, and abstract infrastructure layers.
- What is DevOps?
    - Combines development + operations.
    - Goals: Fast delivery, automation, reliability, collaboration, and security.
- Popular Deployment Strategies
    - **Big Bang**: Deploy all at once.
        - Full update in one go.
        - Risky, hard to roll back.
        - Usually long and requires user updates.
    - **Rolling**: Gradual updates.
        - Updates are rolled out slowly.
        - Safer, easier rollback.
        - Used in apps like Office 365.
    - **Blue-Green**: Two environments, switch after testing.
        - Two environments (blue = old, green = new).
        - Test new version in green, then shift traffic if successful.
    - **Canary**: Small user group first, then expand.
        - Deploy new code to a small set of users.
        - Monitor before full rollout.
        - Safer, gradual rollout.
    - Feature Toggle Deployment
        - New features are hidden.
        - Can enable/disable with a switch.
        - Helps with A/B testing and risk control.
    - A/B Testing Deployment
        - Old and new versions run side-by-side.
        - Small group uses new version.
        - Choose best based on performance.
- Best Practices
    - Use checklists.
    - Automate builds and rollbacks.
    - Notify teams via communication tools.
- Post-Deployment Monitoring
    - Use tools like Rollbar for error logs.
    - Monitor app performance.
    - Share logs with the team for quick fixes.
- Version Control Tools
    - Git (most popular), SVN, TFS, VSS.
    - Used for tracking code changes and backup.
- Git Basics
    - Functions: Clone, Pull, Commit, Push, Merge.
    - Helps collaboration and version tracking.

- Main / Branch / Tag
    - **Main**: Stable, working code.
    - **Branches**: For features or experiments.
    - **Tags**: Snapshots for releases.
- What is a Branch?
    - A separate copy of code for changes.
    - Shares history with main code.
    - Allows safe changes and testing.
- Merge
    - Combine changes from one branch into another.
    - May lead to conflicts that need fixing.
- Branching Strategies
    - **Trunk-Based**: Everyone commits to main.
    - **Feature Branching**: Separate branches for features.
    - **Git Flow**: Includes branches for releases and fixes
- Advanced Git Topics
    - Rebase, Stash, Reset, Revert, Cherry Pick, etc.
    - Used for advanced version control tasks.
- What is CI/CD?
    - CI: Code is tested and built automatically.
    - CD: Code is deployed automatically.
    - Part of DevOps automation.
- Continuous Integration (CI)
    - Regularly merge and test code.
    - Find bugs early, validate code quality.
    - Uses automated tools.
- Continuous Delivery (CD)
    - Automates release to any environment.
    - Works with CI to make deployment easier.
    - Ensures deployment is always possible.
- CI/CD Fundamentals (1)
    - Keep all code and setup in one repo.
    - Use main branch for frequent, small merges.
- CI/CD Fundamentals (2)
    - Automate builds and tests.
    - Only pass code that meets quality checks.
- CI/CD Fundamentals (3)
    - Make small changes often.
    - Test in real-like environments.
    - Easy rollback if needed.
- CI/CD Fundamentals (4)
    - Share visibility with all team members.
    - Keep deployments regular and low-risk.
- Benefits of CI/CD
    - Faster delivery, fewer bugs.
    - Happier team and users.
    - Easier recoveries and better productivity.