UNIT-1

INTRODUCTION TO SOFTWARE ENGINEERING

Software Engineering is a **systematic, disciplined, and quantifiable** approach to software development. It focuses on applying engineering principles (planning, designing, building, testing, measuring) to software to ensure **high quality, reliability, cost-effectiveness, and ethical responsibility**.

The main purpose is to ensure that the software we build is:

- Correct
- Reliable
- Maintainable
- Secure
- Delivered on time
- Ethically responsible
- Does not harm the users, business, or society

SOFTWARE DEVELOPMENT LIFE CYCLE (SDLC)

Definition

SDLC is a **structured process** that defines how software is planned, developed, tested, deployed, and maintained. It ensures that the project is completed **systematically** rather than randomly.

It also helps in:

- Efficient resource utilisation.
- Reducing software cost
- Managing complex projects
- Ensuring quality
- Increasing customer satisfaction
- Managing risks

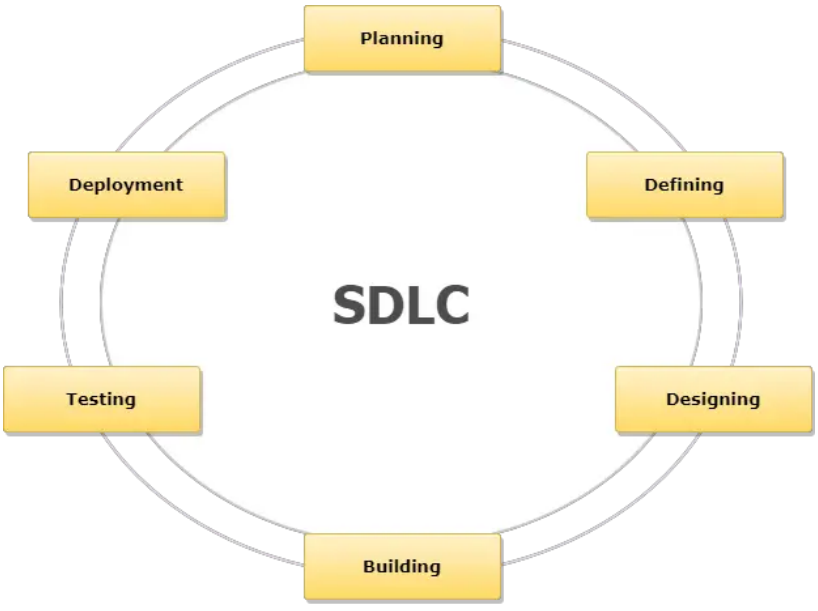


Figure: Software Development Life Cycle Stages

1. Requirement Analysis

- This is the **foundation** of the entire process. Understanding **what the customer needs**, not what the developers assume.
- The Planning phase identifies the purpose of the software, evaluates feasibility, and prepares an overall project plan with scope, timeline, cost, and required resources.
- During Planning, risks are assessed, stakeholders are identified, and a clear direction is set so the team understands what the project aims to achieve.
- The Requirement Analysis phase focuses on interacting with users and stakeholders to understand exactly what the software should do.
- In this phase, both functional needs (features) and non-functional needs (performance, security, usability) are collected and clarified.
- All requirements are documented in a Software Requirement Specification (SRS), which becomes a guiding document for designers, developers, and testers.
- This phase ensures that everyone has a common and complete understanding of the system before development begins, preventing confusion later in the project.

2. System Design

- Once the requirements are known, engineers create a technical blueprint that describes how the software will be structured internally.
- This includes database design, architecture selection, algorithms, interfaces, and workflow diagrams. Proper design reduces complexity, helping developers write better, more consistent code.

3. Implementation (Coding)

- This is the phase where programmers begin writing actual code based on the design.
- The code must follow standards, best practices, ethical coding principles, and proper documentation so that it is easy to understand, maintain, and test.

4. Testing

- After coding, the software is tested thoroughly to identify hidden bugs, logical errors, security vulnerabilities, and performance issues.
- Testers verify whether the product works according to the requirements gathered earlier.
- Ethical testing means not ignoring known bugs, not hiding product failures, and ensuring that the software does not harm users.

5. Deployment

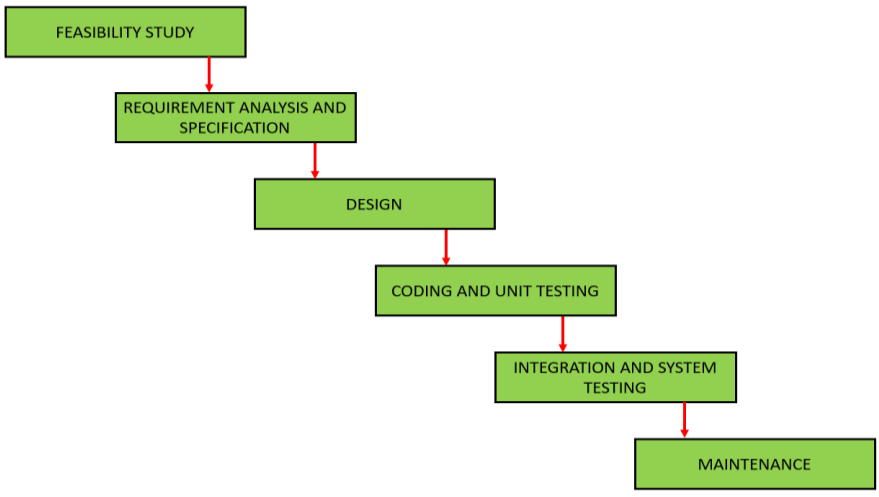
- After successful testing, the software is installed and made available for users. Deployment must be carefully managed so that the transition from old to new systems occurs smoothly, without disrupting user activities.

6. Maintenance

- This phase continues after deployment and includes fixing errors reported by users, upgrading features, improving performance, and adapting the software to new laws, ethical guidelines, or hardware changes.
- Maintenance often lasts longer than development, which makes ethical and sustainable design extremely important.

WATERFALL MODEL

- The Waterfall Model is a traditional and linear approach where each SDLC phase flows downward like a waterfall from one stage to the next without going back.
- It works best when requirements are clearly understood from the beginning, but it becomes difficult to use when changes occur later, because revisiting earlier phases is not encouraged.



Characteristics:

- Each phase must be fully completed before moving to the next
- No backward movement
- Documentation-heavy

Advantages:

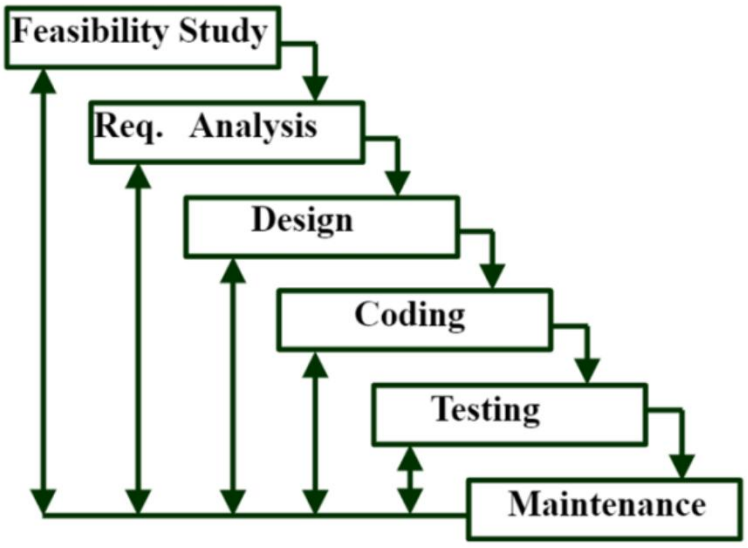
- Very easy to manage
- Works well for small & simple projects
- Clear deliverables for each phase

Limitations:

- **No feedback or flexibility**
- Requirements MUST be stable; otherwise, project fails
- Not suitable for large or complex projects

ITERATIVE MODEL

- The Iterative Model is a software development process where the project is developed in small cycles (iterations) instead of building the entire system at once. Each iteration includes planning, design, coding, and testing. After every cycle, the team reviews the output, gathers feedback, and improves it for the next iteration.
- This model focuses on continuous improvement, allowing developers to refine the software gradually until the complete system is ready.



Advantages

- Early working prototype is available in the initial stages.
- Improves flexibility, since changes can be made after each iteration.

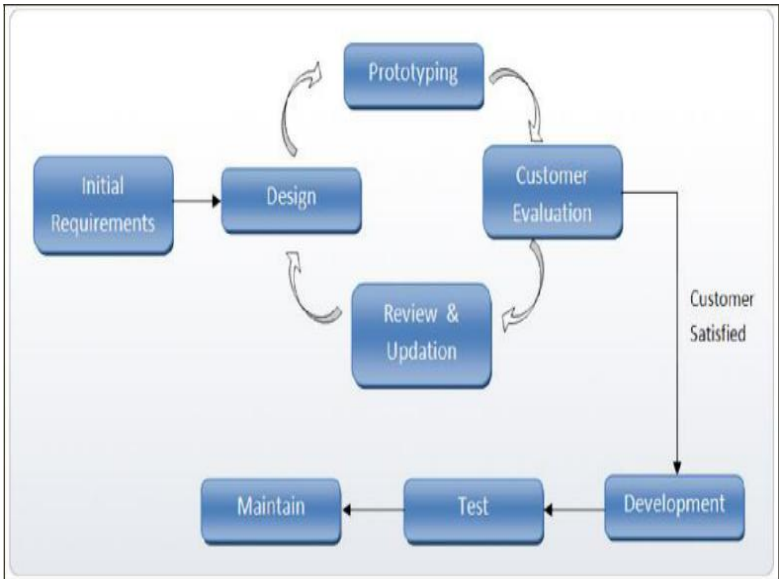
- Continuous feedback from users helps increase the quality of the final product.
- Risk is reduced because issues are identified early in small cycles.
- Better requirement understanding, as requirements become clearer over time.

Disadvantages

- Not suitable for small or resource-limited projects because it requires continuous evaluation.
- More time-consuming, as repeated iterations may extend project timelines.
- Requires active user involvement, which may be difficult to maintain.
- Complex to manage, since multiple cycles need planning, tracking, and coordination.
- Cost may increase due to repeated iterations and redesign.

PROTOTYPING MODEL

- The Prototype Model is a software development approach where a sample version (prototype) of the system is built early to understand user requirements more clearly.
- This prototype is not the final product, but rather a preliminary model that demonstrates how the system will appear and function.
  Users interact with the prototype, give feedback, request changes, and based on that feedback, the development team improves the design and builds the final system.



- It is most useful when requirements are unclear, incomplete, or frequently changing.

Why We Use the Prototype Model

- To understand user requirements clearly when they are not well-defined.

- To bridge the communication gap between users and developers.
- To get early feedback on the system's look, feel, and functionality.
- To reduce misunderstandings and avoid major rework later.
- To visualise the system early, helping with planning and design.

Advantages

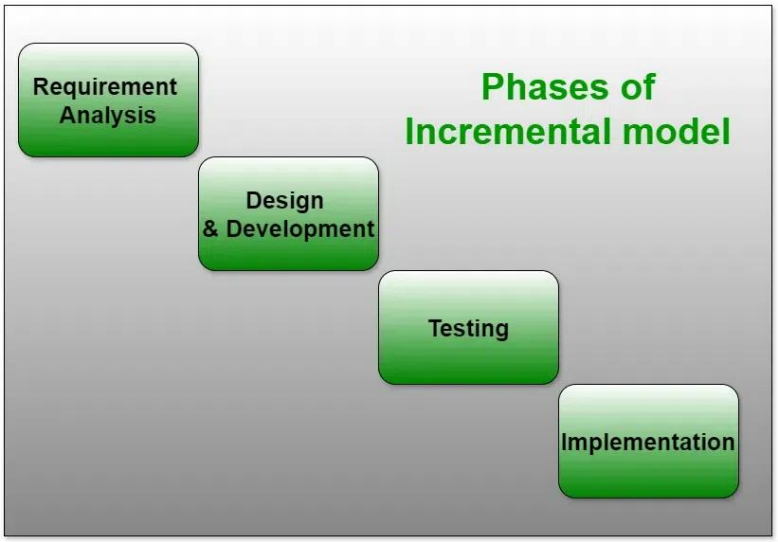
- Improves requirement understanding, especially when the client is unsure of what they want.
- Early feedback helps developers correct mistakes at the beginning.
- Reduces risk by identifying gaps and issues early.
- Better user involvement, leading to higher user satisfaction.
- Improves system usability, because users test the interface before final development.
- Saves time and effort later, as major design errors are caught early.

Disadvantages

- Prototype may create false expectations, as users may think it is the final system.
- Repeated changes can increase the development time and cost.
- Not suitable for large, complex systems, because building prototypes for everything becomes difficult.
- Poorly built prototypes may mislead developers and users.
- Requires constant user involvement, which may not always be available.

INCREMENTAL MODEL

- The Incremental Model divides the project into smaller parts called increments. Each increment delivers a working portion of the software, which is improved step by step.
- This model allows users to see partially working software early and provide feedback, which can guide future increments.



Advantages:

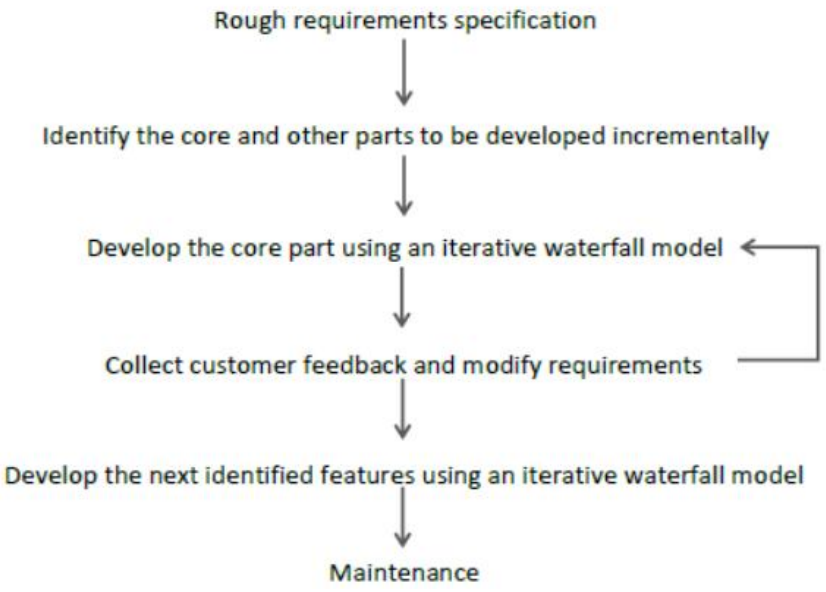
- Errors are easy to recognise.
- Easier to test and debug
- More flexible.
- Simple to manage risk because it is handled during its iteration.
- The Client gets important functionality early.

Disadvantages:

- Need for good planning
- Total Cost is high.
- Well-defined module interfaces are needed.

EVOLUTIONARY MODEL

- Evolutionary model is a combination of the Iterative and incremental model of software.
- Incremental model first implemented a few basic features and delivered to the customer. Then build the next part and deliver it again, and repeat this step until the desired system is fully completed.
- In the Evolutionary model, the software requirement is first broken down into several modules.
- Such models are applied because the requirements often change. So, the end product will be unrealistic, for a complete version is impossible due to tight market deadlines.



Application of Evolutionary Model

- It is used in large projects where you can easily find modules for incremental implementation. The evolutionary model is commonly used

when the customer wants to start using the core features instead of waiting for the full software.
- Evolutionary model is also used in object-oriented software development because the system can be easily partitioned into units in terms of objects.

Necessary Conditions for Implementing this Model

- Customer needs are clear and have been explained in deep to the developer team.
- There might be small changes required in separate parts, but not a major change.
- As it requires time, there must be some time left for the market constraints. Risk is high, and continuous targets to achieve and report to the customer repeatedly.
- It is used when working on a technology that is new and requires time to learn.

Advantages:

- Risk analysis is better.
- It supports a changing environment.
- Initial operating time is less.
- Better suited for large mission-critical projects.
- During the life cycle, software is produced early, which facilitates customer evaluation and feedback.

Disadvantages:

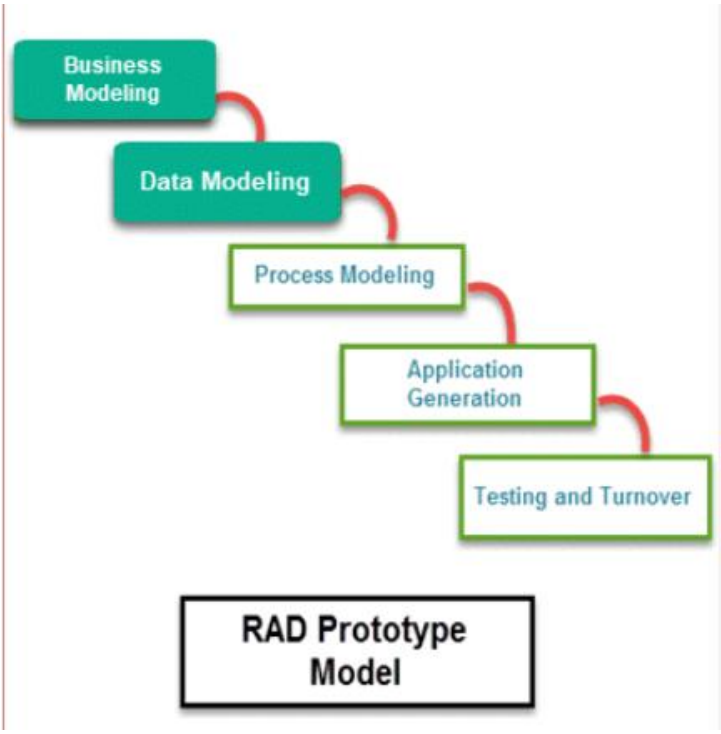
- Management complexity is higher.
- Not suitable for smaller projects.
- Can be costly to use.
- Highly skilled resources are required for risk analysis.

RAPID APPLICATION DEVELOPMENT (RAD)

- RAD is an adaptive software development model based on prototyping and quick feedback with less emphasis on specific planning.
- The RAD approach prioritises development and building a prototype, rather than planning.
- RAD involves parallel development.

RAD model phases:

- Communication phase (requirement gathering)
- Planning phase
- Modelling phase
- Construction phase
- Deployment phase



Focuses on:

- Fast development
- Continuous user involvement
- Heavy use of prototypes

Advantages:

- This model is flexible for change.
- In this model, changes are adoptable.
- Each phase in RAD brings highest priority functionality to the customer.
- It reduced development time.
- It increases the reusability of features.

Disadvantages:

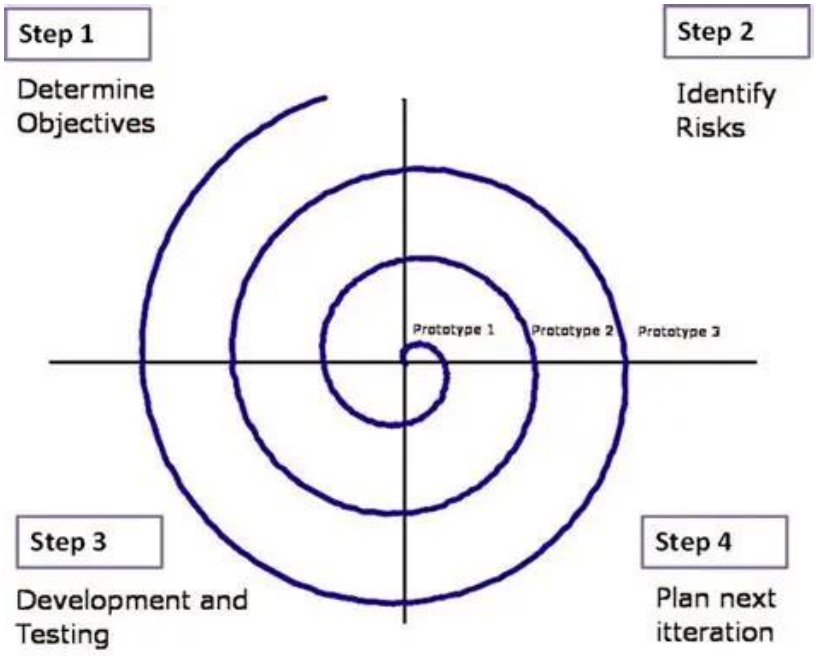
- It required highly skilled designers.
- All application is not compatible with RAD.
- For smaller projects, we cannot use the RAD model.
- On the high technical risk, it's not suitable.
- Required user involvement.

SPIRAL MODEL

- The Spiral Model combines design and prototyping in repeated cycles, called spirals. Every spiral includes risk analysis, which makes this model suitable for large and high-risk projects.
- It allows flexibility and continuous refinement, but needs experienced risk analysts.

Each loop consists of:

1. Objective identification



- Identify objectives of the phase,
- Examine the risks associated with these objectives.
    - Risk: any adverse circumstance that might hamper the successful completion of a software project.
- Find alternative solutions.

2. Risk analysis

- For each identified project risk,
    - a detailed analysis is carried out.
- Steps are taken to reduce the risk.
- For example, if there is a risk that the requirements are inappropriate:
    - A prototype system may be developed**.**

3. Development

- develop (prototype) and validate the next level of the product.

4. Testing

- Review the results achieved so far with the customer and plan the next iteration around the spiral.
- With each iteration around the spiral, a progressively more complete version of the software gets built

Advantages:

- High amount of risk analysis
- Useful for large and mission-critical projects.
- Customers can see the development of the product at the early phase of the software development
- The Spiral Model provides for regular evaluations and reviews, which can improve communication between the customer and the development team.

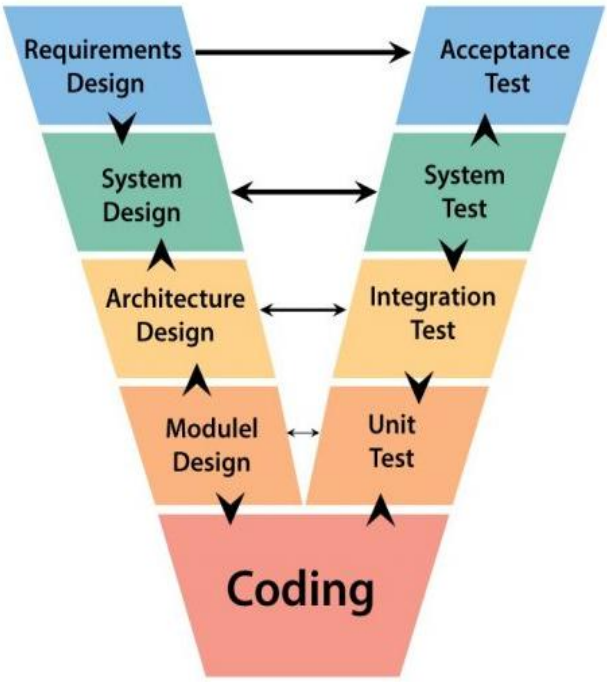
- The Spiral Model allows for multiple iterations of the software development process, which can result in improved software quality and reliability.

Disadvantages:

- Can be a costly model to use.
- Risk analysis needed highly particular expertise.
- Doesn't work well for smaller projects.
- As the number of phases is unknown at the start of the project, time estimation is very difficult.
- The Spiral Model can be complex, as it involves multiple iterations of the software development process.

V-MODEL

- The V-Model is an extension of the Waterfall model that emphasises testing at every stage of development.
- Each development phase has a corresponding testing phase, forming a V-shaped structure. This ensures early detection of defects and strong quality control.



V-Model Verification Phases:

- The verification phase refers to the practice of evaluating the product development process to ensure the team meets the specified requirements.
- The verification phase includes several steps:
- In the **requirement analysis** step, the team comes to understand the product requirements as laid out by the customer.
- In the **system analysis** step, the system engineers analyse and interpret the business requirements of the proposed system by studying the user requirements document.
- In the **software architecture design** stage, the team selects the software architecture based on the list of modules, the brief functionality of each

module, the interface relationships, dependencies, database tables, architecture diagrams, technology details and more. The integration testing model is developed in this phase.
- In the **module design** stage, the development team breaks down the system into small modules and specifies the detailed design of each module, which we call low-level design.
- Then, we begin **coding**. The development team selects a suitable programming language based on the design and product requirements.

V-Model-Validation-Phases:

The validation phase involves dynamic analysis methods and testing to ensure the software product meets the customer's requirements and expectations. This phase includes several stages:

- During the **unit testing** stage, the team develops and executes unit test plans to identify errors at the code or unit level. This testing happens on the smallest entities, such as program modules, to ensure they function correctly when isolated from the rest of the code.
- The **integration testing** stage involves executing integration test plans developed during the architectural design step to verify that groups created and tested independently can coexist and communicate with each other.
- The **system testing** stage involves executing system test plans developed during the system design step, which are composed by the client's business team. System testing ensures the team meets the application developer's expectations.
- The **acceptance testing** step is related to the business requirement analysis part of the V-model and involves testing the software product in the user environment to identify compatibility issues with the different systems available within the user environment. Acceptance testing also identifies non-functional issues like load and performance defects in the real user environment.

Advantages:

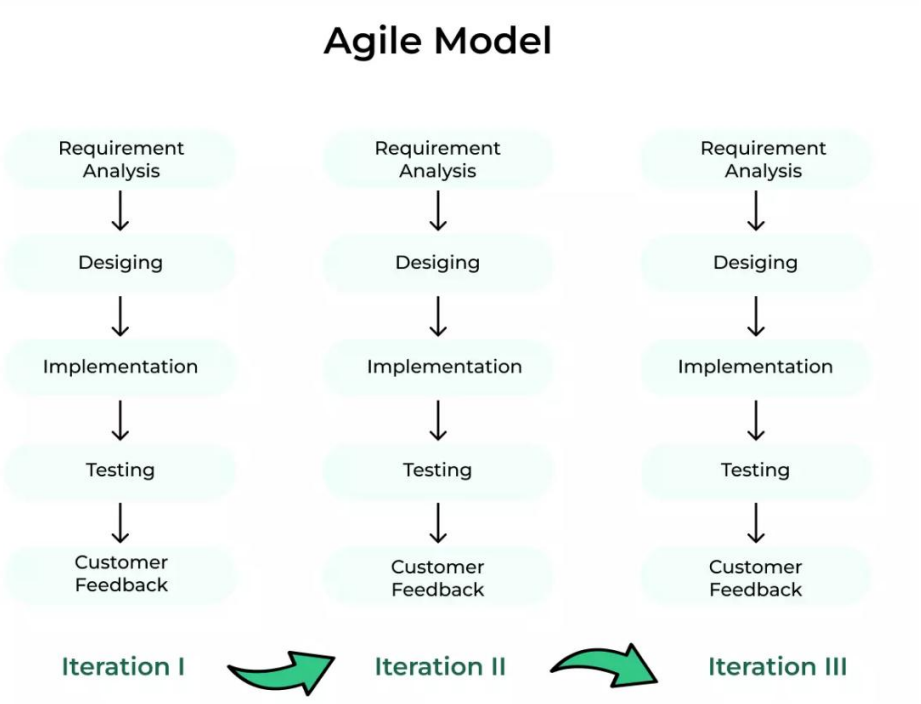
- This is a highly disciplined model, and Phases are completed one at a time.
- V-Model is used for small projects where project requirements are clear.
- Simple and easy to understand and use.

- This model focuses on verification and validation activities early in the life cycle, thereby enhancing the probability of building an error-free and high-quality product.
- It enables project management to track progress accurately.

Disadvantages:

- High risk and uncertainty. It is not good for complex and object-oriented projects.
- It is not suitable for projects where requirements are not clear and contain a high risk of change.
- This model does not support iteration of phases. It does not easily handle concurrent events.
- Inflexibility: The V-Model is a linear and sequential model, which can make it difficult to adapt to changing requirements or unexpected events.
- Time-Consuming: The V-Model can be time-consuming, as it requires a lot of documentation and testing.

AGILE MODEL



- The Agile Model is an iterative and incremental approach to software development that is designed to be flexible, adaptive, and customer-focused. The Agile Model emphasises collaboration, continuous improvement, and rapid prototyping to deliver high-quality software products.
- Agile focuses on flexibility, continuous collaboration, and customer feedback. Instead of creating the whole product at once, Agile teams

work in short cycles called **sprints**, delivering small but functional software pieces quickly.
- Agile encourages communication, adaptability, and rapid response to changing requirements.

Advantages

- Easy to adapt to changing requirements.
- Continuous feedback improves satisfaction.
- Working software delivered after every sprint.
- Problems are detected early.
- Daily meetings improve coordination.
- Continuous testing in every sprint.
- Self-organised teams increase productivity.

Disadvantages

- Not suitable for beginners.
- Requirements keep changing.
- Can cause issues during maintenance.
- Continuous involvement needed.
- Scope creep may occur.
- Frequent changes increase expenses.
- No fixed plan or timeline.

SCRUM MODEL

- Scrum is a lightweight, agile framework used to develop and deliver complex products, especially in software development. It works through iterative and incremental cycles called sprints, usually lasting two weeks to one month.
  A Scrum Team (up to 10 members) plans work in small goals and reviews progress through daily 15-minute meetings called Daily Scrums. At the end of each sprint, the team conducts a Sprint Review to show completed work and a Sprint Retrospective to improve processes.
- Scrum supports self-organising teams, close collaboration, and quick adaptation to changing requirements. It is effective because it accepts

that customer needs change and uncertainty is common, making traditional predictive models less suitable.

Scrum Methodology & Process:

- Scrum is executed in temporary blocks that are short and periodic, called Sprints, which usually range from 2 to 4 weeks. This is the term for feedback and reflection.
- The process has as a starting point a list of objectives/ requirements that make up the project plan.
- **Sprint** - A Sprint is a time box of one month or less. A new Sprint starts immediately after the completion of the previous Sprint.
- **Release:** When the product is completed, it goes to the Release stage.
- **Sprint Review:** If the product still has some non-achievable features, it will be checked in this stage and then passed to the Sprint Retrospective stage.
- **Sprint Retrospective:** In this stage, the quality or status of the product is checked.
- **Product Backlog:** A product backlog is a registered list of work for the development team. The roadmap and its requirements drive it. According to the prioritised features, the product is organised. The essential task is represented at the top of the product backlog so that the team member knows what to deliver first.
- **Sprint Backlog:** The Sprint Backlog is divided into two parts: Product assigned features to the sprint and the Sprint planning meeting.

Advantages

- Product increments are delivered every 2–4 weeks.
- Customer reviews each sprint and gives feedback.
- Daily Scrum improves teamwork and transparency.
- Requirements can be added or modified anytime.
- Frequent testing and continuous review improve software quality.
- Self-organising teams feel ownership and responsibility.
- Issues are identified quickly in daily meetings.
- Short sprints make risk management easier.

Disadvantages

- Requires skilled & disciplined teams.
- Customer must be available for sprint reviews.
- Difficult to estimate the complete cost and time.
- Continuous change may expand the project unnecessarily.
- Requires strict time and attention from all members.
- Works best with small (5–9 members) teams.
- Focus is more on working software than documents.

COCOMO (COST ESTIMATION MODEL)

- The COCOMO (Constructive Cost Model) is a mathematical model used to estimate the cost, time, and effort required to develop software.
- It predicts the total person-months needed based on the size of the project measured in lines of code.
- COCOMO helps managers plan resources, estimate budgets, and avoid unrealistic deadlines.
- By providing quantitative predictions, it supports ethical project management where engineers are not forced into unreasonable time pressures that could lead to poor code quality or unsafe practices.
- **Efforts calculation:** Efforts can be calculated by the number of persons required to complete the task successfully. It is calculated in person-month units.
- **Development time:** the time that is required to complete the task. It is calculated in units of time, such as months, weeks, and days. It depends on the effort calculation; if the number of persons is greater, then definitely the development time is lower.
- Various types of models of Cocomo have been proposed to check the correctness of the software products and to calculate the cost estimations at different levels.

Line Of Code (LOC/SLOC):

- A **line of code (LOC)** is any line of text in a code that is not a comment or blank line, and also header lines, in any case of the number of statements or fragments of statements on the line.
- LOC clearly consists of all lines containing the declaration of any variable, and executable and non-executable statements.

- As Lines of Code (LOC) only counts the volume of code, you can only use it to compare or estimate projects that use the same language and are coded using the same coding standards.

  - Software projects under the COCOMO model strategies are classified into 3 categories,
    1.Organic
    2.Semi-detached
    3.Embedded

  Organic:

- A software project is said to be an organic type if-
- Project is small and simple.
- The project team is small with prior experience.
- The problem is well understood and has been solved in the past.
- Requirements of projects are not rigid, such as a mode example is a payroll processing system.
- Examples of this type of project are simple business systems, simple inventory management systems, and data processing systems.

Semi-Detached Mode:

A software project is said to be a Semi-Detached type if-

• The project has complexity.

• Project team requires more experience, better guidance and creativity.

• The project has an intermediate size and has mixed rigid requirements, such as a mode example is a transaction processing system, which has fixed requirements.

• It also includes the elements of organic mode and embedded mode.

• Few such projects are- Database Management System(DBMS), a new, unknown operating system, and a difficult inventory management system.

Embedded Mode:

A software project is said to be an Embedded mode type if-

• A software project has fixed requirements for resources.

• Product is developed within very tight constraints.

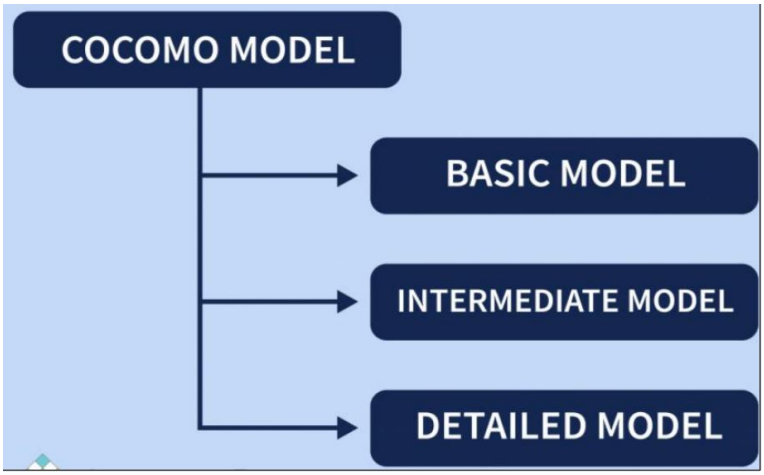
• A software project requiring the highest level of complexity, creativity, and experience falls under this category.

• Such a mode of software requires a larger team size than the other two models.

• For Example: ATM, Air Traffic Control.

Types of COCOMO Models :

- The constant values a,b,c, and d for the Basic Model for the different categories of the system.



Basic COCOMO Model :

• The first level, Basic COCOMO, can be used for quick and slightly rough calculations of Software Costs.

• It requires calculating the efforts which are required to develop in three modes of development: that are organic mode, the semi-detached mode, and the embedded mode.

The basic COCOMO estimation model is given by the following expressions:

$E = a \times (KLOC)b$

$D = c \times (Effort)d$

$P = effort/time$

Where E is the development effort applied in person-months.

D is development time (Duration) in months.

P (team size) is the total no. of persons required to accomplish the project.

- The constant values a,b,c, and d for the Basic Model for the different categories of the system.

Software Project	A	B	C	D
Organic	2.4	1.05	2.5	0.38
Semi-Detached	3.0	1.12	2.5	0.35
Embedded	3.6	1.20	2.5	0.32

QUESTION:

Q1. Consider a software project using semi-detached mode with 300 Kloc. Find out effort estimation, development time, and person estimation.

Answer:-

1.Effort Estimation(Person-Months)

$$\text{Effort}(E) = a \times (\text{KLOC})^{b}$$

2. Development Time Estimation(Months)

$$\text{Development Time } (T) = c \times (E)^{d}$$

3. Person Estimation (Team Size)

$$\text{Number of Persons} = \frac{E}{T}$$

Q2. Consider a software project using semi-detached mode with 30,000 lines of code. Find out effort estimation, development time, and person estimation.

1. Effort Estimation(Person – Months)
   $E = a \times (\text{KLOC})^{b}$
2. Development time Estimation(Months)
   $T = c \times (E)^{d}$
3. Person Estimation (Team Size)

$$\text{Number of Persons} = \frac{E}{T}$$

Intermediate COCOMO Model:

- The Intermediate COCOMO model is an extension of the Basic COCOMO model, which includes a set of cost drivers to enhance the accuracy of the cost estimation model as a result.
- The estimation model makes use of a set of "cost driver attributes" to compute the cost of the software.
- The relationship gives the estimated effort and scheduled time:

Effort (E) = a*(KLOC)b *EAF (PM)

D = c x (Effort)d

The values of a and b in the case of the intermediate model are as follows:

Software Project	A	B	C	D
Organic	3.2	1.05	2.5	0.38
Semi-Detached	3.0	1.12	2.5	0.35
Embedded	2.8	1.20	2.5	0.32

Where E = Total effort required for the project in Person-Months (PM). KLOC = The size of the code for the project in Kilo lines of code.

a, b = The constant parameters for the software project.

EAF = An EAF is a number used to correct a calculation or adjust terms in agreement to reflect changes in conditions or ensure accuracy. It is an Effort Adjustment Factor, which is calculated by multiplying the parameter values of different cost driver parameters. For ideal, the value is 1.

Classification of Cost Drivers and their attributes:

The cost drivers are divided into four categories

Product attributes

- Required software reliability extent

- Size of the application database
- The complexity of the product

Hardware attributes

- Run-time performance constraints
- Memory constraints
- The volatility of the virtual machine environment
- Required turnabout time

Personal attributes

- Analyst capability
- Software engineering capability
- Application experience
- Virtual machine experience
- Programming language experience

Project attributes

- Use of software tools
- Application of software engineering methods
- Required development schedule

EXAMPLE: Consider a software project with 30,000 lines of code, which is an embedded software with an area, hence reliability is high. Find out effort estimation, development time, and person estimation.

Given

- **Mode**: Semi-detached
- **Size**: 30,000 lines of code = **30 KLOC**

COCOMO constants for Semi-detached mode

- a=3.0a
- b=1.12b
- c=2.5
- d=0.35

1. Effort Estimation (Person-Months)

$$E = a \text{ X } (KLOC)^b$$

2. Development Time Estimation (Months)

$$T = c \text{ X } (E)^d$$

3. Person Estimation (Team Size)

$$\text{Number of persons} = \frac{E}{T}$$

Detailed/Advanced COCOMO model:

- This model combines the features of both Basic COCOMO and Intermediate COCOMO approaches across all software engineering activities.
- It considers the impact of each development phase (such as analysis, design, etc.) by applying phase-wise effort multipliers, making it a more complex model.
- The model incorporates a greater number of influencing factors, resulting in more precise effort estimation. The software system is divided into multiple modules, and COCOMO is applied separately to each module. The total effort is obtained by summing the individual estimates. For example, Detailed COCOMO performs cost estimation at every phase of the Waterfall model.
- The six phases of detailed COCOMO are:
    1. Planning and requirements
    2. System design
    3. Detailed design
    4. Module code and test
    5. Integration and test
    6. Cost Constructive model

  In the Detailed COCOMO Model, the cost of each subsystem is estimated separately. This approach reduces the margin of error in the final estimate.

Advantages

- COCOMO is a transparent estimation model, allowing users to clearly understand its working mechanism, unlike models such as SLIM.
- The cost drivers assist estimators in analysing how various factors influence overall project cost.
- The model offers insights based on data from previous and historical software projects.
- COCOMO makes it easier to estimate the overall development cost of a software project.
- The cost drivers are useful in identifying and understanding the effects of different factors that may lead to project risks or cost overruns.

Introduction to Ethics:

- Ethics in software engineering plays a crucial role in ensuring that technology is designed and applied in a manner that benefits society. Ethical guidelines help software engineers make decisions that are morally appropriate, legally compliant, and professionally accountable.
- Software engineering ethics encompasses the moral values and professional norms that influence the conduct and decision-making of software developers. It is essential for safeguarding public welfare, protecting data privacy, promoting fairness, and ensuring accountability in software systems and applications.

SOFTWARE PROFESSIONAL RESPONSIBILITIES

- **Confidentiality:** The obligation of software professionals to protect sensitive information and prevent unauthorised disclosure.
- **Competence:** The responsibility of engineers to work within their areas of expertise and maintain adequate professional skills and knowledge.
- **Intellectual Property Rights:** Respecting legal ownership of software assets, including copyrights, patents, and proprietary materials.
- **Computer Misuse:** Unethical or illegal use of computer systems, such as unauthorised access, data manipulation, or system abuse.

Morality in Software Engineering

Morality refers to the principles that distinguish right from wrong and good from unethical behaviour. In the context of software engineering, moral concerns arise when professionals must decide between what can be technically implemented and what is ethically acceptable.

Examples:

- Developing software that violates user privacy.
- Creating addictive features that deliberately manipulate user behaviour.
- Designing algorithms that introduce bias or unfair discrimination.

Common moral questions faced by software engineers:

- Whether to disclose a security vulnerability that has been identified.
- Whether it is ethical to collect or monitor user data without clear consent.
- Whether to release software despite knowing it contains serious defects.

Laws in Software Engineering

Laws in software engineering are officially enacted rules defined by governments and regulatory authorities that software professionals are required to comply with. These laws are designed to prevent harm, protect user privacy, and safeguard intellectual property rights in software systems.

Major legal domains include:

- Data protection and privacy regulations (such as GDPR and HIPAA).
- Intellectual property laws, including copyrights and patents.
- Cybersecurity and information security regulations.
- Accessibility laws to ensure inclusive software design.

Unlike morality, which is personal and subjective, laws are external, mandatory, and enforceable. A particular action may comply with legal requirements but still be ethically questionable, highlighting the need for software engineers to consider both legal and ethical responsibilities.

Professional Codes of Ethics

Professional codes of ethics are defined by recognized professional bodies to guide ethical behavior and decision-making within the software engineering profession.

Major organizations issuing ethical codes include:

- Association for Computing Machinery (ACM)
- Institute of Electrical and Electronics Engineers (IEEE)
- Other bodies such as ISACA and BCS

Core ethical principles commonly emphasized are:

- **Public Interest:** Giving highest priority to the safety and well-being of society.
- **Integrity:** Acting honestly, transparently, and ethically.
- **Confidentiality:** Protecting private information and ensuring data security.
- **Competence:** Undertaking professional work only within one's area of expertise.
- **Accountability:** Taking responsibility for professional actions and decisions.

Ethics in software engineering extends beyond the prevention of harm to the proactive promotion of social good. By understanding moral principles, complying with legal requirements, and adhering to professional ethical standards, software engineers can develop systems that are secure, fair, and socially accountable. As technology increasingly shapes modern society, ethical judgment continues to be a fundamental element of responsible software development.

ACM/IEEE Software Engineering Code of Ethics and Professional Practice

The Association for Computing Machinery (ACM) and the IEEE Computer Society (IEEE-CS) jointly formulated the Software Engineering Code of Ethics to encourage ethical conduct and professional responsibility among software engineers. The code highlights obligations toward the public, clients and employers, and fellow professionals, while emphasising integrity, fairness, and respect for human rights.

Purpose of the Code

The code serves as a guideline for ethical decision-making in professional practice. It assists software engineers in:

- Acting in the best interest of the public
- Maintaining trust and credibility within the profession
- Addressing and resolving ethical dilemmas encountered during software development

Structure of the ACM/IEEE Code of Ethics

The Code is organised into eight core principles that software engineers are expected to follow.

- **Public** – Act in accordance with the public interest
  Software engineers must prioritise the safety, privacy, and well-being of users and society above all else. They should prevent harm to individuals, the environment, and the public, and communicate potential risks associated with software systems responsibly.
  Example: Reporting software faults that may cause serious harm, even when facing organisational pressure to remain silent.
- **Client and Employer** – Serve the best interests of clients and employers
  Engineers should provide truthful and accurate technical information, avoid conflicts of interest, and respect confidentiality obligations.
  Example: Refraining from sharing proprietary or confidential information with competitors.
- **Product Principle** – Ensure high professional standards in software products
  Software engineers should deliver reliable, well-tested, and maintainable systems. They must clearly identify system limitations and risks, and properly document requirements and design details.
  Example: Avoiding the release of software that has not been adequately tested.
- **Judgment Principle** – Exercise integrity and independence in professional decisions
  Engineers should make unbiased decisions based on technical facts and professional

expertise, without being influenced by external pressure or personal interests. *Example:* Refusing requests to falsify data or manipulate outcomes for personal benefit.

- **Management Principle** – Promote ethical practices in software development and maintenance Leaders should manage teams ethically, ensure compliance with ethical standards, and support professional growth through clear objectives and fair practices. *Example:* Assigning responsibilities based on skills and competence rather than favouritism.
- **Profession Principle** – Enhance the dignity and reputation of the profession Software engineers should contribute to professional development, promote ethical awareness, and guide others responsibly. *Example:* Reporting unethical behaviour when necessary to protect professional integrity.
- **Colleagues Principle** – Treat colleagues with fairness and respect Engineers should support ethical conduct, recognise contributions appropriately, and avoid any form of discrimination or exploitation. *Example:* Giving proper credit to all contributors in a collaborative project.
- **Self Principle** – Commit to continuous ethical and professional improvement Professionals should engage in lifelong learning, accept responsibility for their actions, and learn from past mistakes. *Example:* Taking training programs to improve knowledge of secure and ethical coding practices.

Responsibility of Software Engineers: Stakeholders, Public Interest, and Harm Mitigation

Software engineers function not only as technical experts but also as ethical decision-makers whose work can significantly impact individuals and society. Therefore, they carry important responsibilities toward all parties affected by their software systems. These responsibilities primarily involve recognising stakeholder interests, prioritising the public good, and taking measures to prevent or reduce potential harm.

1. Responsibility to Stakeholders

Stakeholders include all individuals and groups who are directly or indirectly affected by the development, deployment, and use of a software system. Their involvement in a project may vary, but their interests must be considered throughout the software lifecycle.

Types of Stakeholders:

- **Primary stakeholders:** Clients, employers, users, developers, and project team members.
- **Secondary stakeholders:** End users, the general public, regulatory authorities, and future generations.

Key Responsibilities:

- Identify and address the needs, expectations, and rights of all stakeholders.
- Clearly communicate system risks, limitations, and constraints to clients and users.
- Avoid conflicts of interest and disclose any possible biases.
- Preserve confidentiality and protect sensitive or proprietary information.

Example: Transparently informing clients about technical limitations that may impact system performance or user privacy, even if the information is unfavorable.

2. Responsibility to the Public Interest

Software engineers have an obligation to society as a whole to ensure that their work promotes public welfare and does not cause harm. Their decisions should prioritise safety, dignity, and societal well-being.

Key Responsibilities:

- Design and develop software that enhances human safety, welfare, and dignity.
- Evaluate the social, environmental, and cultural consequences of software systems.
- Ensure transparency in systems that influence public rights, such as legal, financial, or healthcare applications.
- Maintain honesty and integrity, even when facing organisational or commercial pressure.

Example: Declining to participate in the development of facial recognition systems that could enable mass surveillance or violate human rights.

3. Harm Mitigation

Even software created with good intentions can result in unintended negative outcomes. Software engineers are responsible for anticipating, reducing, and managing potential harm associated with software systems.

Potential Forms of Harm:

- Data breaches and identity theft
- Algorithmic bias and discriminatory outcomes
- Physical injury caused by faulty embedded software (e.g., in medical devices or vehicles)
- Psychological harm due to addictive or manipulative software features

Key Responsibilities:

- Perform continuous risk assessments throughout the software development lifecycle.

- Apply thorough testing, strong security measures, and effective error-handling techniques.
- Regularly update and patch systems to address newly identified vulnerabilities.

Example: Issuing an urgent patch or recall for a defect in medical software that could lead to incorrect patient treatment.

Case Study 1: Therac-25 Radiation Therapy Machine

Overview

- Therac-25: A computer-controlled medical machine used in the 1980s to deliver radiation therapy to cancer patients.
- AECL: The system was designed and manufactured by Atomic Energy of Canada Limited.
- Software-only control: Radiation delivery relied entirely on software, with no hardware safety interlocks.

What Went Wrong

- Race condition bug: A timing error in software caused uncontrolled radiation overdoses.
- Cryptic error messages: System warnings were unclear and failed to alert operators to real dangers.
- No independent testing: Critical safety software was not independently reviewed or rigorously tested.
- Removed hardware safeguards: Earlier hardware safety mechanisms were eliminated to reduce cost and size.

Consequences

- Radiation overdoses: At least six severe radiation overdose incidents occurred between 1985 and 1987.
- Patient harm: Patients suffered serious injuries and some deaths due to excessive radiation.
- Delayed response: AECL responded slowly and initially denied that software faults were responsible.

Ethical Lessons

- Poor testing practices: Inadequate testing of safety-critical software can lead to catastrophic outcomes.
- Overreliance on software: Trusting software alone without hardware backups increases system risk.

- Lack of transparency: Failure to act openly and promptly worsened the impact of reported harms.
- Public safety neglect: Engineers failed in their ethical duty to prioritize human safety.

Case Study 2: Boeing 737 MAX

Background

- 737 MAX: An upgraded version of Boeing's popular 737 aircraft series.
- MCAS system: Software introduced to automatically adjust the aircraft's nose for flight stability.
- Single-sensor dependency: MCAS relied on data from only one angle-of-attack sensor.

What Went Wrong

- Sensor failure risk: Faulty sensor data caused MCAS to repeatedly force the plane's nose downward.
- Lack of pilot training: Pilots were not properly informed or trained to handle MCAS behaviour.
- Ignored warnings: Internal safety concerns raised by engineers were reportedly overlooked.
- Business pressure: Competition and cost pressures led to rushed development and certification.

Consequences (Boeing 737 MAX)

- Two major crashes: The 737 MAX was involved in two fatal accidents caused by MCAS-related failures.
- Lion Air Flight 610 (2018): The aircraft crashed into the Java Sea shortly after takeoff due to faulty MCAS activation.
- Ethiopian Airlines Flight 302 (2019): A similar MCAS malfunction led to the crash minutes after departure.
- 346 people died: A total of 346 passengers and crew lost their lives in the two accidents.
- Reputation and financial loss: Boeing suffered severe reputational damage and billions of dollars in recalls, lawsuits, and penalties.

Ethical Lessons

- Lack of transparency: Boeing failed to clearly inform regulators and pilots about MCAS risks and behaviour.

- Profit over safety: Business and competitive pressures were prioritised above passenger safety.
- Poor software validation: Inadequate testing and human-factor analysis led to unsafe system behaviour.
- Delayed accountability: Boeing was slow to accept responsibility and admit software-related faults.