

SOFTWARE:

“SET OF instructions (computer programs) that when executed provide desired features, function, and performance”

Nature of Software Characteristics of software

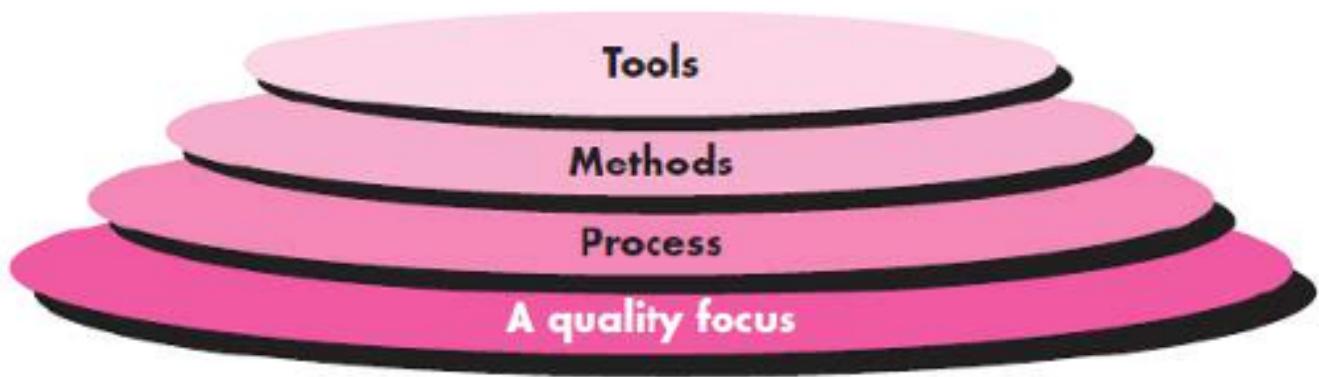
- Software is developed or engineered; it is not manufactured
- Software doesn't “wear out.”

Software Application Domain

- **System software**—a collection of programs written to service other programs. Some system software (e.g., compilers, editors, and file management utilities)
- **Application software**—stand-alone programs that solve a specific business need e.g., point-of-sale transaction processing, real-time manufacturing process control
- **Engineering/scientific software**—has been characterized by “number crunching” algorithms. use to perform complex calculation this is used by engineer and scientist.
- **Web applications**—called “WebApps”- this network-centric software category spans a wide array of applications.
- **Artificial intelligence software**—makes use of nonnumerical algorithms to solve complex problems .

Software Engineering

Def: “The application of a systematic, disciplined, quantifiable approach to the development, operate, and maintenance of software; that is, the software engineering”



Software engineering is a layered technology. Referring to Figure any engineering approach (including software engineering) must rest on an organizational commitment to quality.

- Total quality management, Six Sigma, effective approaches to software engineering.
- The foundation for software engineering is the process layer. The software engineering process is the glue that holds the technology layers together and enables rational and timely development of computer software. Process defines a framework
- Software engineering methods provide the technical how-to's for building software. Methods encompass a broad array of tasks that include communication, requirements analysis, design modeling, program construction, testing, and support.

- Software engineering tools provide automated or semiautomated support. When tools are integrated so that information created by one tool can be used by another

Software Process:

A generic process framework for software engineering encompasses five activities:

- **Communication.** Before any technical work can commence, it is critically important to communicate and collaborate with the customer (and other stakeholders¹¹) the intent is to understand stakeholders' objectives for the project and to gather requirements that help define software features and functions.
- **Planning.** Any complicated journey can be simplified if a map exists. A software project is a complicated journey, and the planning activity “map” that helps guide the team as it makes the journey. The map—called a software project plan—defines the software engineering work by describing the technical tasks to be conducted, the risks that are likely, the resources that will be required, the work products to be produced, and a work schedule.
- **Modeling.** Whether you're a landscaper, a bridge builder, an aeronautical engineer, a carpenter, or an architect, you work with models every day. You create a “sketch” of the thing so that you'll understand the big picture—what it will look like architecturally, how the constituent parts fit together, and many other characteristics. If required, you refine the sketch into greater and greater detail in an effort to better

understand the problem and how you're going to solve it. A software engineer does the same thing by creating models to better understand software requirements and the design that will achieve those requirements.

- **Construction.** This activity combines code generation (either manual or automated) and the testing that is required to uncover errors in the code.
- **Deployment.** The software (as a complete entity or as a partially completed increment) is delivered to the customer who evaluates the delivered product and provides feedback based on the evaluation.

Software Engineering Practice

Essence of Practice

Understand the problem. (communication and analysis).

- Identify stakeholders
- Solve data, functions, and features are required
- Represent in smaller problems
- Represent problem graphically

Plan the solution. (modeling and software design).

- Understand patterns
- Has a similar problem been solved so can be reusable
- Define the subproblems

Carry out the plan. (code generation).

- Design Source code according to model
- Prepare solution in part wise and correct

Examine the result

- Test each component part of the solution
- Solution produce results that conform to the data, functions, and features that are required

Core Principles of Software Engineering

1. The First Principle: The Reason It All Exists

A software system exists **to provide value to its users**.

If the software does not solve a real problem, it holds no purpose.

2. The Second Principle: Keep It Simple, Stupid (KISS)

Software design should always be kept **as simple as possible**.

Avoid unnecessary complexity—simple designs are easier to understand, maintain, and extend.

3. The Third Principle: Maintain the Vision

A successful project needs a **clear and consistent vision**.

The entire team must share the same understanding of goals and design direction.

4. The Fourth Principle: What You Produce, Others Will Consume

Write code as if someone else will read, maintain, or extend it. Clear, readable, and well-structured work reduces confusion and improves collaboration.

5. The Fifth Principle: Be Open to the Future

Design systems that can adapt to **future needs and changes**. Flexible and extensible systems hold greater long-term value.

6. The Sixth Principle: Plan Ahead for Reuse

Reusability saves time and effort.

Design components so they can be reused in different parts of the project or even across projects.

7. The Seventh Principle: Think!

Before writing code, **think through the problem clearly**.

Good planning and reasoning lead to better decisions and fewer mistakes.

Software Crisis

The term **Software Crisis** refers to the major problems faced during the development of software. The word “crisis” is used because software projects often reached a point where it was uncertain whether they would succeed or fail.

Main Reasons for the Software Crisis:

1. Software projects were **not completed on time**.
2. The **cost became higher** than the planned budget.
3. The **quality of the software was poor**, with many bugs and errors.

4. It was difficult to **manage and develop large, complex software.**
5. **Maintaining existing software** became harder as systems grew.
6. The **demand for new software increased rapidly**, but development methods were not advanced enough.

The Software Crisis occurred because the old methods of developing software were not suitable for large and complex systems. Better techniques, tools, and proper planning were needed to improve quality and reduce failures.

Software Myths

Management myths.

Managers with software responsibility, most disciplines, are often under pressure to maintain budgets, keep schedules from slipping, and improve quality.

- **Myth:** We already have a book that's full of standards and procedures for building software. Won't that provide my people with everything they need to know?

Reality: The book of standards may very well exist, but is it used? Are software practitioners aware of its existence? Does it reflect modern software engineering practice? Is it complete? Is it adaptable? Is it streamlined to improve time-to-delivery while still maintaining a focus on quality? In many cases, the answer to all of these questions is "no."

- **Myth:** If we get behind schedule, we can add more programmers and catch up.

Reality: Software development is not a mechanistic process like manufacturing. “adding people to a late software project makes it later.” However, as new people are added, people who we’re working must spend time educating the newcomers

- **Myth:** If I decide to outsource the software project to a third party, I can just relax and let that firm build it.

Reality: If an organization does not understand how to manage and control software projects internally, it will invariably struggle when it outsources software projects.

Customer myths.

A customer who requests computer software

- **Myth:** A general statement of objectives is sufficient to begin writing programs—we can fill in the details later.

Reality: Although a comprehensive and stable statement of requirements is not always possible. Unambiguous requirements are developed only through effective and continuous communication between customer and developer.

- **Myth:** Software requirements continually change, but change can be easily accommodated because software is flexible.

Reality: It is true that software requirements change, but the impact of change varies with the time changes are requested early the cost impact is relatively small than the cost impact of change at commitment level

Practitioner’s myths.

Myths that are still believed by software practitioners

- **Myth:** Once we write the program and get it to work, our job is done.

Reality: Someone once said that “the sooner you begin ‘writing code,’ the longer it’ll take you to get done.” Industry data indicate 60 and 80% of all effort expended on software will be expended after it is delivered to the customer for the first time.

- **Myth:** Until I get the program “running” I have no way of assessing its quality.

Reality: the technical review. Software reviews are a “quality filter” that have been found to be more

- **Myth:** The only deliverable work product for a successful project is the working program.

Reality: A working program is only one part of a software configuration includes many elements. A variety of work products provide a foundation for successful engineering

- **Myth:** Software engineering will make us create voluminous and unnecessary documentation and will invariably slow us down.

Reality: Software engineering is not about creating documents. It is about creating a quality product. Better quality leads to reduced rework.

Agility

Agility in software development means the ability to **respond quickly and effectively to changes**.

Agile methods focus on small, incremental development, continuous feedback, and close communication with customers.

Key points of Agility:

- Software is developed in **short iterations**.
- **Requirements can change** at any time.
- Teams work closely with customers for continuous improvement.
- Working software is delivered **frequently**, not all at once.
- Focus on **simplicity, flexibility, and quick delivery**.
- Helps reduce waste, improve quality, and speed up development.

In short: Agility means building software in a way that allows **fast adaptation** to new requirements or changes.

Cost of Change

The **Cost of Change** refers to how much time, effort, and money it takes to fix or modify something in a software project.

Traditional View (Waterfall)

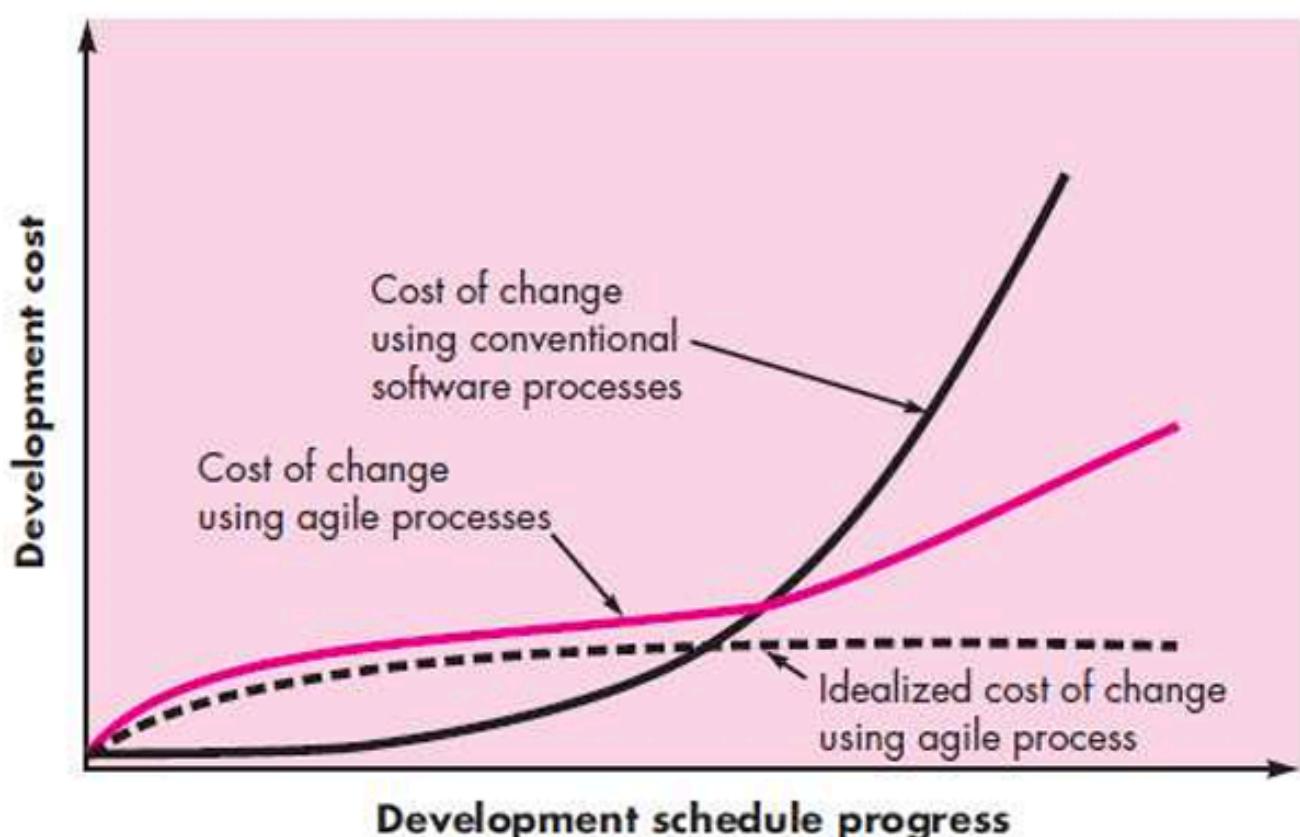
- In traditional models, the cost of change **increases heavily** as the project progresses.
- Example:
 - Changing a requirement during testing or after release becomes very expensive.

- A small mistake in early stages can become a big cost later.

Agile View

- Agile methods try to **reduce the cost of change** by:
 - Frequent testing
 - Continuous integration
 - Iterative development
 - Early feedback from users
- Because changes are handled in every iteration, cost does not rise sharply.

The cost of change is higher in rigid processes but **much lower in Agile**, because Agile supports continuous updates and early detection of issues.



Agile Principles (Short Notes)

1. **Customer satisfaction** through early and continuous delivery.
2. **Welcome changing requirements**, even late in development.
3. Deliver working software **frequently**.
4. **Close collaboration** between business people and developers.
5. Build projects around **motivated individuals**.
6. Use **face-to-face communication** for effectiveness.
7. Working software is the **primary measure of progress**.
8. Maintain a **sustainable development pace**.
9. Focus on **technical excellence** and good design.
10. Keep things **simple**.
11. Teams **self-organize** for better results.
12. Regularly reflect and **improve** (retrospectives).

4 P's of Software Project

1. People

People are the most important part of a software project. This includes developers, testers, managers, customers, and stakeholders. Their skills, communication, and teamwork directly affect project success.

2. Product

Product refers to the software being developed. It includes understanding the product objectives, features, functions,

constraints, and quality requirements. Clear requirements lead to better design and development.

3. Process

Process defines the methods and frameworks used to build the software (e.g., Waterfall, Agile, Spiral). A good process helps maintain quality, reduce risks, and ensure proper planning, testing, and delivery.

4. Project

Project includes planning, scheduling, monitoring, risk management, and resource management. It ensures that the software is delivered **on time, within budget, and with expected quality**.

SRS(Software Requirements Specification)

- Is a document which is used as a communication medium between the customer and the supplier.
- When the software requirement specification is completed and is accepted by all parties, the end of the requirements engineering phase has been reached.
- After the acceptance phase, any of the requirements cannot be changed, but the changes must be tightly controlled.
- The software requirement specification should be edited by both the customer and the supplier,

Need of Software Requirement Specification

- company publishing a software requirement specification to companies for competitive tendering,
- company writing their own software requirement specification in response to a user requirement document.
- SRS allows a number of different suppliers to propose solutions,
- Use to identify any constraints which must be applied.
- SRS is used to capture the users requirements and
- SRS highlights inconsistencies and conflicting requirements and define system and acceptance testing activities.

Characteristics of a Good Software Requirements Specification (SRS)

1. Complete

An SRS is complete when it contains **all functional and non-functional requirements**.

Nothing important should be missing—every input, process, output, and constraint must be clearly specified.

2. Consistent

The SRS must not have **conflicting requirements**.

All statements should agree with each other. No two requirements should describe opposite or incompatible behaviors.

3. Unambiguous

An SRS should be **clear and exact**, with only one possible meaning for each requirement.

No vague terms like “fast”, “user-friendly”, “good performance”.

Every requirement must be interpreted the same way by all developers and testers.

4. Trackable (Traceable)

Each requirement must be **uniquely identifiable**, so it can be traced throughout the development process.

Traceability helps map requirements to design, code, test cases, and maintenance activities.

5. Verifiable

A requirement is verifiable if it can be **checked or tested**.

There must be a clear way to confirm that the software meets the requirement—through testing, review, or inspection.

Components of SRS

1. Functional Requirements

These describe **what the system should do**—the functions, features, and operations the software must perform.

2. Non-Functional Requirements

These specify **how the system should behave**, including reliability, usability, security, and maintainability.

3. Performance Requirements

These define the system's **speed, response time, capacity, throughput, and efficiency** expectations.

4. Design Constraints

These are **limitations or restrictions** on design choices, such as hardware limits, programming language, standards, or legal rules.

5. External Interface Requirements

These describe **how the system interacts with external systems**, including user interfaces, hardware interfaces, software interfaces, and communication interfaces.

Process Metrics (5 Marks Answer)

Process metrics are measurements used to evaluate and improve the **software development process**. They help organizations build better-quality software by analyzing how effective and efficient their development practices are.

Key Points:

1. Process as a Controllable Factor

The software process is one of the major controllable factors that influence **software quality** and **organizational performance**. It forms a triangle with **product** and **project** factors that together affect quality.

2. Indirect Measurement

We measure the **efficacy (effectiveness)** of a software process **indirectly**. This is done by observing outcomes

such as defect rates, productivity, and time taken, rather than measuring the process directly.

3. Guidance for Improvement

Process metrics help teams understand **how processes should be defined** and **how to measure their quality and productivity**. This gives a clear direction for improving the development process.

4. Private and Team-Level Metrics

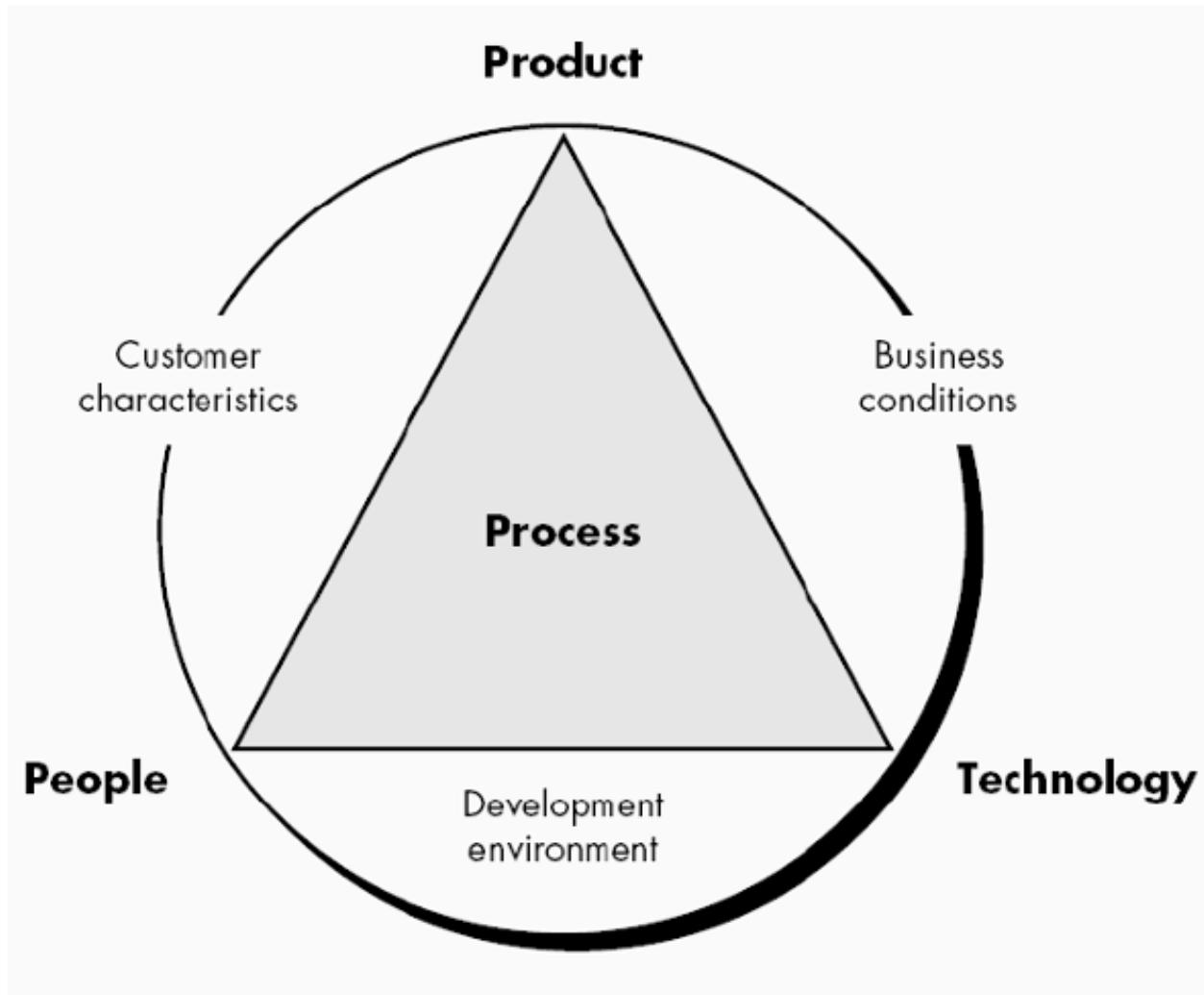
Some metrics, like personal productivity data, are **private to individual engineers**, helping them self-improve.

Other metrics, like team defect rates, effort, and schedules, are **shared within the project team** but are never used to blame individuals.

5. Organizational Benefits

By collecting and analyzing project-level data, organizations can identify weaknesses, improve their **overall process performance**, and reduce failures.

Process metrics also support **failure analysis**, helping teams understand why problems occurred and how to prevent them in the future.



Project Metrics – 5 Marks Answer (Exam English)

Project metrics are measurements used to monitor, control, and improve the progress and performance of a software project. They help project managers take corrective actions, avoid delays, and improve product quality.

1. Used to Adapt Workflow:

Project metrics and indicators help managers adjust

project workflow, tasks, and technical activities whenever needed.

2. Used During Estimation:

Metrics collected from past projects are used to estimate **effort**, **cost**, and **time** for new projects. This improves the accuracy of planning.

3. Monitoring & Control:

As the project progresses, actual effort and calendar time are compared to the original estimates. This helps track progress and take corrective action.

4. Measures Project Productivity:

Production rates such as pages of documentation, review hours, function points, and source lines of code are measured to determine productivity.

5. Improves Schedule, Quality, and Cost:

Project metrics help reduce development time by identifying delays early. They also help improve product quality and reduce the overall cost of the project.

6. Types of Measurements:

Every project should measure:

- **Inputs:** resources used (people, tools, time).
- **Outputs:** deliverables or work products produced.
- **Results:** effectiveness of the outputs (quality, defects removed, etc.).

Thus, project metrics help in planning, monitoring, controlling, improving quality, and reducing the total cost of the software project.

Exam-Ready Answer (5 Marks – English)

Software organizations should **measure and use software metrics** to improve their development process, product quality, and delivery time. Measurements should be **simple, customized to the organization**, and must provide **useful information**.

The main guideline for software measurement is “**Keep it simple**”. Instead of collecting too many metrics, organizations should focus on **results** that help improve performance.

A small organization can collect the following **simple and useful measures** for change requests:

1. **Elapsed time** from receiving a change request to completing its evaluation.
2. **Effort (person-hours)** required to evaluate the change.
3. **Elapsed time** from evaluation completion to assignment of the change to personnel.
4. **Effort (person-hours)** needed to implement the change.
5. **Time required** to complete the change implementation.
6. **Errors found during change implementation.**
7. **Defects discovered after release** to customers.

After collecting these measures for multiple change requests, the organization can calculate the **total time taken from change request to implementation**, helping to improve efficiency and product quality.

Definitions (Exam-Ready)

Cohesion

Cohesion refers to the **degree to which the elements within a single module are related to each other.**

High cohesion is desirable because it makes the module **easy to understand, maintain, and reuse.**

Coupling

Coupling refers to the **degree of interdependence between different modules** in a software system.

Low coupling is preferred because it reduces the impact of changes and improves system flexibility.

Software Project Estimation (4 Marks – Short Answer)

Software project estimation is difficult because many factors –**human skills, technology, environment, and organizational issues**—affect the final cost and effort.

To get more reliable estimates, several approaches are used:

1. **Use past project data** – Estimates are based on similar completed projects.
2. **Decomposition techniques** – The project is divided into smaller functions and activities, and effort is estimated step-by-step.
3. **Empirical models** – Mathematical models (like COCOMO) are used to predict cost and effort.
4. **Estimate refinement over time** – Estimates become more accurate as the project progresses and more information becomes available.

Using multiple estimation techniques together provides cross-checking and improves accuracy.

However, without historical data, estimating becomes weak and uncertain.