Important topics

UNIT - 2

Cocomo Model :

- Constructive Cost Model
- Barry Boehm in 1981
- Key - Effort & Schedule(time required to complete)
- Types of Projects in Cocomo - Organic, Semi detached, Embedded
- Phases
  - Planning & Requirements
  - System design
  - Detailed Design
  - Code & test
  - Cost constructive model
- Basic - a (KLOC)^b & EFM , Tdev = c (E) ^d
- Intermediate -> 15 effort management factors
- Complete -> More detailed then both above

Software Projects	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

Aspect	Empirical Model	Putnam Model
Foundation	Based on historical data from past projects	Based on a mathematical equation derived from empirical data

Methodology	Uses regression analysis or data fitting to model the relationship between project characteristics (size, complexity) and effort	Uses a non-linear equation to relate size, time, and effort, with an S-curve model for project progress
Input Variables	Primarily project size (LOC, function points) and other historical factors	Size of the software, time available, and productivity factor
Focus	Cost, time, effort estimation based on past projects	Focuses on estimating the total effort required for a project given the size and time constraints
Time Management	Generally doesn't explicitly model the time aspect as comprehensively as the Putnam model	Explicitly models time along with effort, using an S-curve representation
Scalability	Well-suited for a variety of project sizes but may require adaptation for very large or small projects	Typically used for medium to large-sized projects, with an explicit focus on the relationship between time and effort
Complexity	May be easier to use for practitioners without a strong mathematical background	More complex, requires understanding of non-linear equations and S-curve dynamics

Comparison Summary

Aspect	White-box Testing	Black-box Testing	Grey-box Testing
Tester's Knowledge	Full access to internal code and structure	No access to internal code (only inputs/ outputs)	Partial access to internal code or design details
Focus	Internal workings, code logic, and structure	Functional behavior, user experience	Functional behavior with some internal knowledge
Type of Testing	Unit testing, integration testing, code coverage	Functional testing, system testing, acceptance testing	Integration testing, security testing, usability testing
Tester's Skill Requirements	Requires programming and technical knowledge	Does not require programming skills	Requires a mix of functional testing and technical knowledge
Test Coverage	Can achieve exhaustive test coverage (e.g., all code paths)	Focuses on testing user-visible behavior and system outputs	Balanced coverage, with efficiency in testing certain parts
Advantages	Thorough, early bug detection, optimization of code	User-centric, no need for coding knowledge	Efficient, combines functional testing with some insight into internals

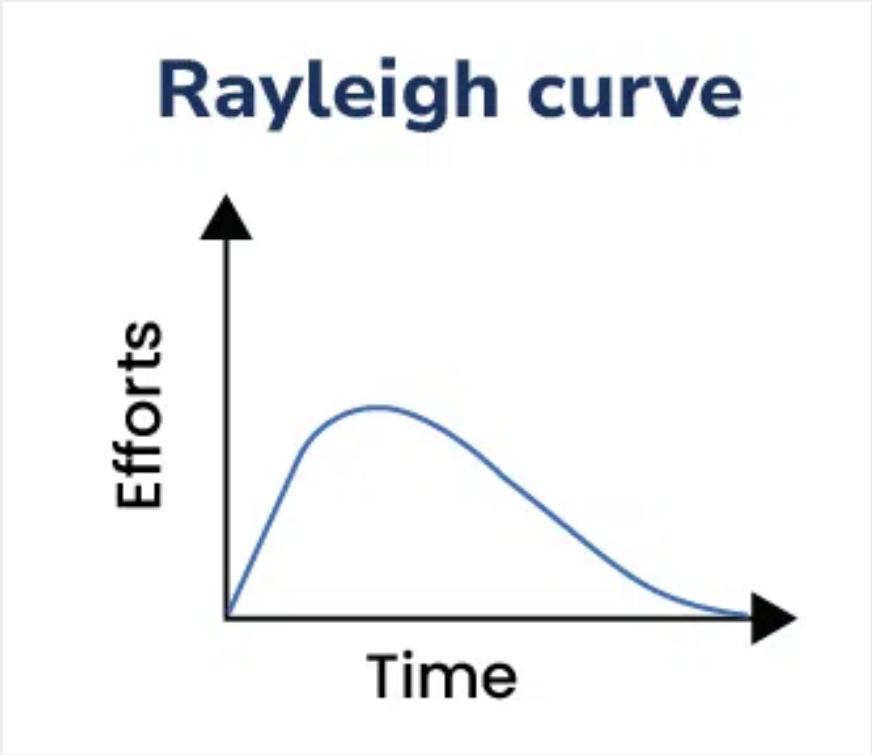
Disadvantages	Time-consuming, requires expertise, limited by code visibility	Limited coverage, hard to pinpoint root causes	Limited scope, requires expertise in both functional and technical aspects

Key Differences:

Aspect	Software Reliability	Software Safety
Focus	Ensuring software works as expected without failure.	Ensuring software failures do not lead to harmful consequences.
Concern	Performance and stability of the software.	Prevention of accidents, harm, or catastrophic failures.
Scope	General software behavior under normal conditions.	Software behavior under critical or hazardous conditions.
Measurement	Failure rate, uptime, MTBF, and defect rates.	Safety-critical failure modes and risk mitigation.

Putnam Resource Allocation Model explained with graph

It can be explained with regards to the Putnam Model for effort distribution over time in a software project using the Rayleigh curve. The graph below shows a typical Rayleigh curve:



Rayleigh Curv

Here, the X-axis takes time and the Y-axis takes effort in man hours for instance, months. Effort thus ranges upwards as the project advances and then may drop down as the project approaches its final stages, according to the curve. The shape of the curve implies the total work content necessary to complete the undertaking.

Putnam's Work Along with Equations

The basis for Putnam's research is Rayleigh curve and Software Life Cycle Model (SLIM) which he initiated. The SLIM model uses the following key equation, known as the Putnam-Norden-Rayleigh (PNR) equation, to estimate effort:The SLIM model uses the following key equation, known as the Putnam-Norden-Rayleigh (PNR) equation, to estimate effort:

$S=Ck1/3t4/3$

Where:

- S is the software size (in LOC, function points, etc.).
- C is the technology factor, reflecting the capabilities of the development environment and team.
- k is the effort coefficient, a constant depending on the project environment.
- t is the time to complete the project (in months or years).

Empirical Estimation Technique –

Empirical estimation is a technique or model in which empirically derived formulas are used for predicting the data that are a required and essential part of the software project planning step. These techniques are usually based on the data that is collected previously from a project and also based on some guesses, prior experience with the development of similar types of projects, and assumptions. It uses the size of the software to estimate the effort. In this technique, an educated guess of project parameters is made. Hence, these models are based on common sense. However, as there are many activities involved in empirical estimation techniques, this technique is formalized. For example Delphi technique and Expert Judgement technique.

Unit - 3

Comparison: Debugger vs Test Data Generator

Aspect	Debugger	Test Data Generator
Primary Purpose	Identify and fix bugs in the code during execution.	Generate input data for testing and validation.
Type of Tool	Debugging tool for real-time analysis.	Testing tool for automated test case generation.
Focus Area	Code behavior, logic, and variable inspection.	Input data variability, edge cases, and boundary testing.
Use Case	Used during the development phase to find logical errors and runtime issues.	Used in testing phases to validate software against diverse input data.
Interactivity	Interactive, allowing step-by-step execution.	Generally automated, with pre-defined configurations.

Output	Detailed error logs, variable values, and runtime information.	Large datasets, test cases, and input values for different scenarios.
User Knowledge	Requires in-depth understanding of the code.	Requires understanding of input domain and testing objectives.
Scope	Narrow scope (focusing on one or a few parts of the system at a time).	Broader scope (covering different input combinations for full system testing).

Techniques for Testing Source Code

1. **Unit Testing**:
    - Focuses on testing individual units (functions, methods, or classes) of the source code in isolation.
    - Ensures that each part of the code works as intended by testing it with a range of input values.
    - Unit tests are often automated and run frequently during development to catch issues early.
2. **Example**: Testing a calculateTotalPrice function to ensure it correctly computes the total cost of items in a shopping cart.
3. **Static Analysis**:
    - Analyzes the source code without executing it, often using automated tools.
    - Checks for coding standards, potential bugs, security vulnerabilities, and issues like memory leaks or unreachable code.
    - Can be used to enforce consistency and readability in the code.
4. **Example**: A static code analysis tool might flag a variable that is declared but never used, or a function that has an unreachable code block.
5. **Code Coverage Analysis**:
    - Measures how much of the code is being exercised by tests. The goal is to maximize code coverage to ensure

that the majority of the code paths are tested.
    - Common types of coverage metrics include statement coverage, branch coverage, and path coverage.
6. **Example**: A tool like **JaCoCo** or **Istanbul** can be used to check whether each statement or branch in the code has been executed by tests.
7. **Peer Code Review**:
    - Involves having other developers review the source code for logic errors, adherence to coding standards, and potential bugs.
    - Peer reviews often uncover issues that automated tools or the original developer might miss.
    - Code reviews can also improve code quality and promote knowledge sharing among the team.
8. **Example**: A developer submits their code for review, and other team members inspect the code for errors, optimization opportunities, or improvements.
9. **Dynamic Analysis**:
    - Involves executing the code and observing its behavior during runtime to identify issues like memory leaks, concurrency problems, or performance bottlenecks.
    - It is often used in conjunction with tools that track memory usage, CPU usage, and other system metrics during execution.
10. **Example**: A profiler might be used to identify slow functions or memory usage spikes during code execution.
11. **Integration Testing**:
    - Ensures that different modules or components of the system work together as expected.
    - After unit testing, integration testing verifies that the interactions between different parts of the code do not introduce defects.
12. **Example**: Testing the interaction between the payment gateway module and the user authentication module to ensure that a successful login results in a proper payment transaction.

Tools for Source Code Testing

- **JUnit** (for Java): A framework for writing and running unit tests.

- **PyTest** (for Python): A framework that allows writing simple test cases to ensure code behaves as expected.
- **SonarQube**: A static code analysis tool that helps detect code quality issues and vulnerabilities.
- **Coverity**: A static analysis tool for identifying security vulnerabilities and bugs in the source code.
- **FindBugs**: A tool for identifying potential errors in Java code by performing static analysis.
- **Checkmarx**: A tool that helps detect security vulnerabilities in the code during the development phase.

Merits and Demerits of Testing Source Code

Merits:

1. **Early Bug Detection**: By testing the source code during development, defects can be identified early, reducing the cost and effort of fixing them later in the process.
2. **Improved Code Quality**: Regular testing, especially with automated unit tests and code reviews, leads to cleaner, more maintainable code.
3. **Higher Confidence in Stability**: With thorough testing, developers can be more confident that the code behaves correctly, minimizing the risk of runtime errors.
4. **Documentation for Future Changes**: Well-tested code provides valuable documentation for other developers who may modify or extend the code in the future.

Demerits:

1. **Time-Consuming**: Writing and maintaining tests, especially for large systems, can be time-consuming.
2. **Overhead for Developers**: Implementing and running tests requires effort, which may be seen as overhead during the initial phases of development.
3. **Limited Scope**: Testing source code can only identify issues related to the logic, syntax, or structure of the code, but not always higher-level system issues like user experience or overall system integration.
4. **Complexity in Test Maintenance**: As the codebase evolves, tests might need constant updates, which can be challenging if the testing framework or coverage is not well designed.

Test Case Design

Test case design is the process of creating a set of test cases that will effectively verify the functionality and quality of the software under test. A **test case** is a detailed set of conditions and inputs that are executed to validate a particular aspect of the software's behavior. Well-designed test cases ensure thorough testing of the application and help detect potential defects early in the development process.

A **test case** typically contains the following components:

- **Test Case ID**: A unique identifier for the test case.
- **Test Case Description**: A brief description of what the test case is validating.
- **Preconditions**: Any setup or environment conditions that need to be met before executing the test case.
- **Test Steps**: The specific steps that need to be followed during the test.
- **Test Data**: The input values to be used during testing.
- **Expected Results**: The anticipated outcome or behavior of the system when the test is executed.
- **Actual Results**: The actual outcome when the test is executed (filled out during testing).
- **Pass/Fail Criteria**: A description of the conditions under which the test is considered to pass or fail.

Unit - 4

Differences Between FTR and Structured Walkthrough

Aspect	Formal Technical Review (FTR)	Structured Walkthrough
Purpose	Detect defects, improve quality, and ensure compliance with standards.	Provide feedback and identify issues with understanding or design.

Formality	Very formal, with detailed documentation, roles, and responsibilities.	Less formal, often focused on education or understanding.
Participants	Includes the author, a moderator, reviewers, and possibly other stakeholders.	Typically includes the author and a group of peers or stakeholders.
Focus	Strictly focused on finding defects and ensuring product quality.	Focuses more on gaining consensus, learning, and sharing knowledge.
Documentation	Detailed documentation of findings, decisions, and action items.	Documentation is usually minimal, often informal.
Process	Follows a structured process with defined phases (preparation, review, follow-up).	More relaxed process, often based on discussion and feedback.

Here's the summary in **7 concise points**:

1. **Functional Suitability**: How well the software meets functional requirements and performs expected tasks.
2. **Performance Efficiency**: System responsiveness, resource usage, and scalability under varying loads.
3. **Reliability**: Software's ability to perform consistently without failure over time.
4. **Usability**: Ease of use, user-friendliness, and overall user experience.
5. **Security**: Protection against unauthorized access, data breaches, and vulnerabilities.
6. **Maintainability**: Ease of making changes, updates, and fixing issues over time.
7. **Portability & Compatibility**: Ability to run across different platforms and integrate with other systems.

Characteristics of Software Configuration Management

1. **Version Control**:
    - SCM tools like Git, Subversion (SVN), or Mercurial help manage different versions of software artifacts, ensuring that changes are tracked and can be reverted or merged appropriately.
2. **Change Control**:
    - SCM tracks changes made to the codebase, configurations, and documentation. It ensures that changes are made systematically and authorized, reducing the risk of accidental or unapproved modifications.
3. **Configuration Identification**:
    - This involves identifying and defining software components (source code, documentation, libraries, etc.) in a way that ensures they can be tracked, managed, and modified consistently. It helps in establishing baselines for development and release.
4. **Build and Release Management**:
    - SCM ensures that builds are repeatable and consistent, and supports automated builds to ensure that software is built correctly with all dependencies. It also handles packaging and distributing software releases.
5. **Audit and Tracking**:
    - Every change made to the software is logged, creating an audit trail. This helps track the history of the software, identify who made each change, and understand why certain decisions were made.
6. **Collaboration**:
    - SCM supports team collaboration by allowing developers to work on different parts of the project simultaneously, merging changes when necessary, and resolving conflicts that might arise from simultaneous edits.
7. **Configuration Baselines**:
    - Baselines are stable versions of the software or configurations that are used as reference points for further development, testing, or releases. These baselines help ensure consistency in the development process and

are useful for release management.

Quality Management Issues and Standards in Software Development

In the field of software development, **quality management** refers to the practices, standards, and processes used to ensure that software products meet the desired quality criteria. It involves both the management of quality in the development process and the final quality of the software product itself.

Quality Management Issues in Software Development

Software quality management can face several challenges or issues that can negatively impact the software's quality, timelines, and costs. Here are some of the most prominent **quality management issues**:

1. Unclear or Changing Requirements

- **Issue**: One of the main causes of quality issues is the lack of clear, well-defined requirements or constant changes in requirements during the development phase. If the project's goals or specifications are not clearly understood or are altered frequently, it becomes difficult to build a quality product.
- **Impact on Quality**: The software may be developed based on incorrect or incomplete information, leading to defects, performance issues, or an inability to meet user expectations.

2. Poor Communication and Collaboration

- **Issue**: Quality problems can arise when teams (developers, testers, business analysts, project managers) do not communicate effectively, leading to misunderstandings, incomplete requirements, and errors in implementation.
- **Impact on Quality**: Without collaboration, teams may build software in silos, which can lead to integration issues, redundant work, and gaps in functionality.

3. Lack of Proper Testing

- **Issue**: Insufficient or improper testing practices, such as inadequate test coverage, poor test case design, or skipping testing phases (e.g., regression testing), can result in defects going unnoticed until after deployment.
- **Impact on Quality**: This often leads to software failures, customer dissatisfaction, and costly fixes post-release.

4. Tight Timelines and Budget Constraints

- **Issue**: When projects are rushed or constrained by budget and time limitations, quality can be sacrificed. Teams may cut corners, neglect proper design, skip reviews, or rush through testing.
- **Impact on Quality**: This results in software that may have defects, performance issues, or insufficient features that ultimately affect user experience and system stability.

5. Inefficient Processes and Tools

- **Issue**: Using outdated or inappropriate tools, methodologies, or processes can hinder quality. For example, manual testing in a large application might be inefficient, or a lack of automated testing tools could result in missed defects.
- **Impact on Quality**: Inefficient processes can delay development, increase costs, and lead to an error-prone product.

6. Inconsistent Documentation

- **Issue**: Lack of proper documentation (e.g., requirements, design documents, test cases) can lead to misunderstandings and make it difficult to track decisions or changes throughout the development lifecycle.
- **Impact on Quality**: Without accurate documentation, teams may not be aligned on requirements, leading to poor decision-making, misalignment between development and testing, and incomplete software.

7. Inadequate Risk Management

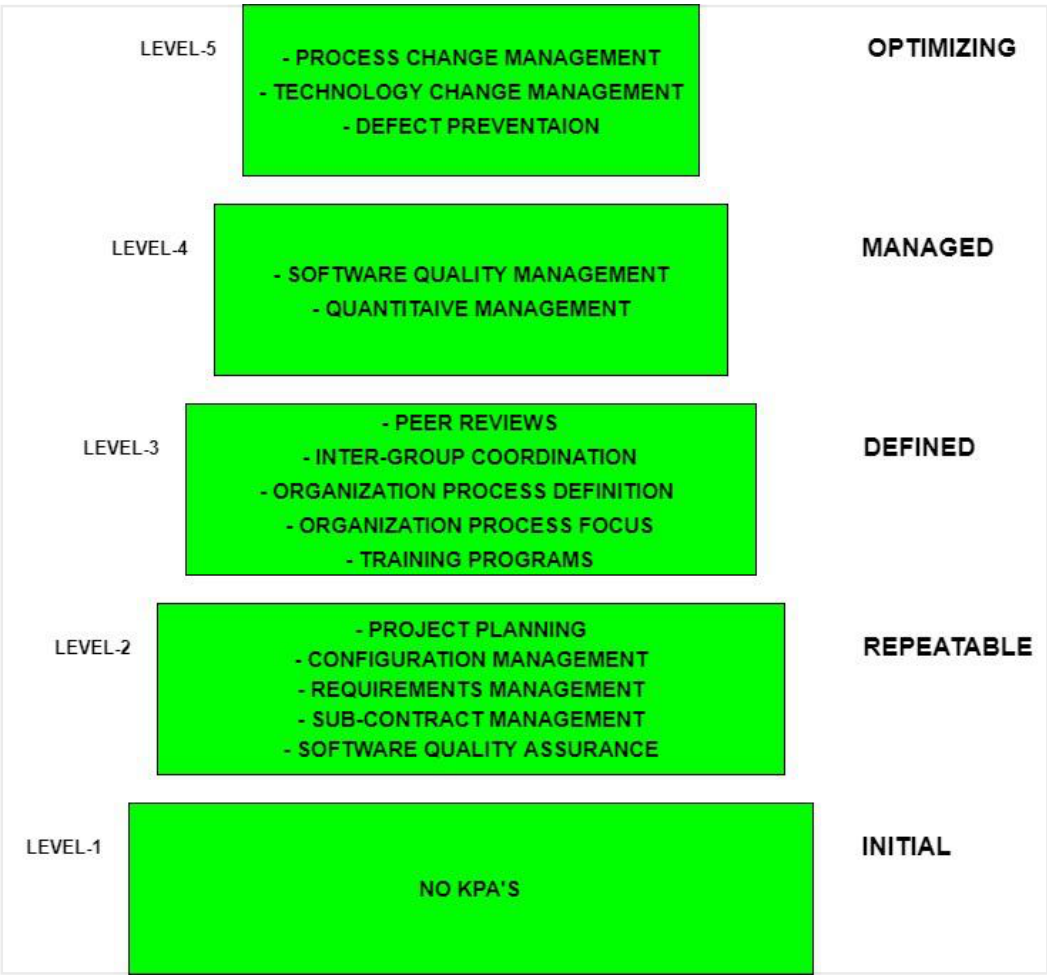
- **Issue**: Failing to identify, assess, and mitigate risks during the project can lead to unforeseen problems and quality issues. Risks related to technology, resources, or customer requirements need to be addressed proactively.
- **Impact on Quality**: Risks that are not managed can lead to technical debt, scope creep, or missed deadlines, all of which affect the final quality of the software.

8. Lack of Stakeholder Involvement

- **Issue**: If key stakeholders (such as end-users, business analysts, or customers) are not engaged throughout the development process, their needs and expectations may not be properly captured, leading to a mismatch between the software and user requirements.
- **Impact on Quality**: The product may fail to meet the needs of

its intended users, resulting in dissatisfaction and a lack of adoption.

CMM (Capability Maturity Model) vs CMMI (Capability Maturity Model Integration)



Aspects	Capability Maturity Model (CMM)	Capability Maturity Model Integration (CMMI)
Scope	Primarily focused on software engineering processes.	Expands to various disciplines like systems engineering, hardware development, etc.

Maturity Levels	Had a five-level maturity model (Level 1 to Level 5).	Initially had a staged representation; it introduced continuous representation later.
Flexibility	More rigid structure with predefined practices.	Offers flexibility to tailor process areas to organizational needs.
Adoption and Popularity	Gained popularity in the software development industry.	Gained wider adoption across industries due to broader applicability.

List – 1	List – 2
(a) Initial	(i) Processes are improved quantitatively and continually.
(b) Repeatable	(ii) The plan for a project comes from a template for plans.
(c) Defined	(ii) The plan for a project comes from a template for plans.
(d) Managed	(iv) There may not exist a plan or it may be abandoned.
(e) Optimizing	(v) There’s a plan and people stick to it.