Software-Testing-Module-2-Important-Topics-PYQs

- 4. Outline 2 types of static unit testing strategies
  - 1. Code Reviews
  - 2. Static Analysis Tools
- 5. Write a short note on defect prevention
  - How defect prevention is done:
  - Why defect prevention is important:
- 6. Describe the roles of presenter and moderator in a review team
  - Presenter
  - Moderator
- 7. Explain Junit framework for unit testing
  - What is JUnit?
  - Why is JUnit important?
  - Important JUnit Concepts
  - Example — Simple JUnit Test (Failing Example)
- 8. What is dynamic unit testing and control flow testing
  - Example
  - Control Flow Testingx
    - Example
- 9. Discuss the concept of mutation testing with testing process
  - What is Mutation Testing?
  - Steps in mutation
  - Important Terms
  - Types of mutation
  - Advantages of Mutation Testing
  - Disadvantages of Mutation Testing
- 10. Explain seven types of mutation operators with examples
  - 1. Arithmetic Operator Replacement (AOR)
    - Example:
  - 2. Relational Operator Replacement (ROR)
    - Example:
  - 3. Logical Operator Replacement (LOR)
    - Example:
  - 4. Constant Replacement (CR)

        - Example:
    - 5. Statement Deletion (SDL)
        - Example:
    - 6. Unary Operator Insertion/Deletion (UOI)
        - Example:
    - 7. Conditional Operator Replacement (COR)
        - Example:
- 11. Explain Dynamic unit test environment with a neat sketch.
    - 1. Test Driver
    - 2. Stubs
- 12. Differentiate between control flow testing and Data flow testing.
- 13. Summarize the pros and cons of static unit testing.
    - Advantages (Pros):
    - Disadvantages (Cons):
- 14. With the help of a diagram, explain the steps in the code review process
    - 1. Criteria
    - 2. Readiness
    - 3. Preparation
    - 4. Examination
    - 5. Change Requests
    - 6. Rework
    - 7. Validation
    - 8. Exit
    - 9. Report

1. Suppose that the C programming language is chosen in project. Recommend a detailed code review checklist to the review team.

1. Code Correctness

- Does the code meet the functional requirements?
- Are all functions implemented completely?
- Are edge cases handled properly (e.g., division by zero, null pointers)?

- Is error handling present and appropriate (e.g., checking return values of functions)?
- Are magic numbers avoided and replaced with constants or macros?
- Are all conditions and loops terminating correctly?

2. Code Structure and Organization

- Is the code modular (split into small, focused functions)?
- Are functions short and doing only one job (Single Responsibility)?
- Is the code organized logically (e.g., headers, source files, folders)?
- Are header files clean and free from unnecessary includes?

3. Coding Standards (Style)

- Is naming consistent and meaningful for variables, functions, and constants?
- Are `#define` macros and `typedef`s used properly?
- Is indentation and spacing consistent?
- Are block braces `{}` used even for single-line `if`, `else`, `while`, `for` statements?
- Are comments meaningful, necessary, and not excessive?

Example:

```
// Good comment:
int max = findMax(array, size); // Find the maximum element
```

4. Memory Management

- Is every `malloc`/`calloc` properly followed by a `free`?
- Are memory leaks avoided?
- Are null pointer checks done after dynamic memory allocation?
- Are arrays and pointers managed carefully (no buffer overflow)?

5. Error Handling

- Are all system/library calls (like `fopen`, `malloc`, etc.) checked for errors?
- Is proper fallback or error handling logic provided?
- Are exit points (`return`, `exit`) handled gracefully?

6. Security Checks

- Are all user inputs validated?
- Are dangerous functions like `gets()` avoided?

7. Performance

- Is there any unnecessary use of heavy loops or recursions?
- Are operations optimized for time and memory where needed?
- Are data structures used appropriately for the given problem?

8. Testing and Debugging

- Are there unit tests for all major functions?

2. Describe the special role of a recordkeeper

In software projects (especially in **code reviews**, **meetings**, or **testing processes**), a **recordkeeper** has a very important role.

Who is a Recordkeeper?
A **recordkeeper** is the person responsible for **writing down** and **maintaining accurate records** of what happens during important activities.
Their job is to **capture important information** so that nothing is forgotten and everyone stays on the same page.

Responsibility	Explanation
Take Notes	Write down important points during reviews, meetings, and discussions (e.g., decisions made, action items, bugs found, responsibilities assigned).
Organize Information	Arrange the recorded information neatly and clearly so that it’s easy to understand later.
Share Records	Send the notes, summaries, or reports to all the team members after the activity.
Maintain History	Keep a history of discussions and decisions, so the team can refer back if there are doubts later.

Responsibility	Explanation
Track Progress	Help the team keep track of which tasks or corrections are completed and which are pending.

Why is the Recordkeeper Important?

- Helps avoid **misunderstandings**.
- Makes sure **everyone knows their tasks**.
- Saves time when **reviewing decisions** later.
- Makes the project more **organized** and **professional**.
- Very helpful for **audits**, **project reports**, and **future planning**.

3. Explain the use of stubs and test drivers

When building software, different parts (or modules) of the system depend on each other. But what if some parts are not ready yet? How do we test a module without waiting for everything else to be built?

That is where **Test Drivers** and **Test Stubs** come in.

What is a Test Driver?

A **Test Driver** is a temporary helper program that tests a module when the main system is not ready.

Example:

Imagine you are testing a calculator application that adds two numbers, but the user interface is not finished yet.

- The **Test Driver** will act like the user interface and send numbers to the calculator function.
- It will check whether the calculator gives the correct sum.

How It Works:

1. The **Test Driver** gives inputs to the module under test.
2. The **Module Under Test** performs its task, such as addition.

3. The **Test Driver** checks if the output is correct.

What is a Test Stub?

A **Test Stub** is a temporary replacement for a missing piece of the system. It helps the module under test continue working even if a dependent part is not ready.

Example:

Imagine a shopping application where the "Checkout" function needs to get product prices from a pricing database.

- But what if the database is not ready yet?
- A **Test Stub** can be used to return fake product prices so that testing can continue.

How It Works:

1. The **Module Under Test** asks for data from another module.
2. Since the real module is not ready, the **Test Stub** provides fake data.
3. The **Module Under Test** continues running as if the real data was there.

How Test Drivers and Test Stubs Work Together

- The **Test Driver** starts the test by calling the **Module Under Test**.
- The **Module Under Test** needs some data from another module.
- Since that module is not built yet, a **Test Stub** provides fake data.
- The **Module Under Test** processes the data and gives the result back to the **Test Driver**.
- The **Test Driver** checks if the result is correct.

Simple Diagram

```
    Test Driver → Module Under Test → Test Stub
```

- **Test Driver**: Starts the test and gives input.
- **Module Under Test**: The actual function being tested.
- **Test Stub**: Provides fake responses when needed.

Why Use Them?

- Allows testing even when the full system is not ready
- Saves time by testing modules separately
- Helps find errors early

Some points

- Use **Test Drivers** when the caller (higher-level module) is missing.
- Use **Test Stubs** when the called (lower-level module) is missing.
- They help in isolated testing, making sure each part of the system works before everything is put together.

4. Outline 2 types of static unit testing strategies

In **static testing**, we check the code **without running it**. We only *read*, *analyze*, or *review* the code to find mistakes early.
Here are **two important static unit testing strategies**:

1. Code Reviews

What it is:

- Developers carefully **read through the code** together (or individually) to find errors, style problems, or areas for improvement.

How it works:

- A team of developers (sometimes including testers) **examines the source code manually**.
- They **discuss problems** like logic errors, coding standard violations, incomplete functionality, etc.

Example:

- A senior developer reviews your function and points out that a loop may cause an infinite loop if not handled properly.

Why it's useful:

- Finds bugs early before running the program.
- Improves code quality and team knowledge.

2. Static Analysis Tools

What it is:

- Using **automated tools** that **scan the source code** to detect common errors, security issues, or bad practices.

How it works:

- They highlight things like unused variables, memory leaks, dangerous code patterns, etc.

Example:

- A tool warns that a variable is declared but never used, which could be a sign of a missing operation.

Why it's useful:

- Saves time compared to manual checking.
- Can scan **thousands of lines** of code quickly.
- Reduces human error during checking.

5. Write a short note on defect prevention

Defect prevention means **finding ways to stop mistakes (defects) from happening** during the software development process.

Instead of just fixing bugs after they are found, defect prevention focuses on **avoiding** bugs in the first place.
It is a **proactive** approach — doing the right things early so that problems don't occur later.

How defect prevention is done:

- **Proper requirements gathering:** Make sure requirements are clear and complete.
- **Good design practices:** Design software carefully before coding.
- **Code reviews:** Review code to catch mistakes early.
- **Training developers:** Teach programmers good coding standards and practices.
- **Using testing early:** Test small parts early (unit tests) to find issues quickly.

- **Using tools:** Use static analysis tools to automatically find errors before running the program.

Why defect prevention is important:

- Saves **time and money** (fixing bugs later is more expensive).
- Improves **software quality** and **user satisfaction**.
- Reduces **stress** for the development team.

6. Describe the roles of presenter and moderator in a review team

Presenter

- **Who they are:**
  The person who **created** or **wrote** the work being reviewed (for example, code, design, or document).
- **Main role:**
  - **Present** the work to the review team.
  - **Explain** the purpose, logic, and important points of the work.
  - **Answer questions** from reviewers during the review.
  - **Stay open-minded** to feedback and suggestions.

Moderator

- **Who they are:**
  A **neutral** person who **manages** the review meeting (not the author or reviewer).
- **Main role:**
  - **Plan and organize** the review session.
  - **Ensure everyone stays focused** and sticks to the agenda.
  - **Control discussions** — if people argue too much, the moderator brings them back on track.
  - **Encourage participation** from all team members.
  - **Summarize** major findings and **ensure action items** are recorded.

7. Explain Junit framework for unit testing

What is JUnit?

- **JUnit** is a **free**, **open-source** testing framework for **Java** developers.
- It helps developers **write and run tests easily and repeatedly**.
- Mainly used for **unit testing** — testing **small parts** (units) of your program like functions or classes.
- **Test-Driven Development (TDD)** uses JUnit — first you write tests, then the code.

Why is JUnit important?

- **Finds bugs early** during development.
- Makes code **more reliable and readable**.
- Helps in **regression testing** (making sure new changes don't break old features).
- **Automates** testing — no need to manually check everything every time.
- Builds **developer confidence** while coding.

Important JUnit Concepts

- **Fixtures**:
  - A fixed environment for running tests.
  - Methods like `@Before` (setUp) and `@After` (tearDown) prepare and clean after tests.

```java
import junit.framework.*;

public class JavaTest extends TestCase {
   protected int value1, value2;

   // assigning the values
   protected void setUp(){
      value1 = 3;
      value2 = 3;
   }

   // test method to add two values
   public void testAdd(){
      double result = value1 + value2;
      assertTrue(result == 6);
   }
}
```

- **Test Suites**:
  - Group multiple tests and run them together using `@Suite` and `@RunWith`.

```java
import org.junit.runner.RunWith;
import org.junit.runners.Suite;

//JUnit Suite Test
@RunWith(Suite.class)

@Suite.SuiteClasses({
   TestJunit1.class ,TestJunit2.class
})

public class JunitTestSuite {
}
```

- **Test Runner**:
  - Runs your tests and collects results.

```
import org.junit.runner.JUnitCore;
import org.junit.runner.Result;
import org.junit.runner.notification.Failure;

public class TestRunner {
   public static void main(String[] args) {
      Result result = JUnitCore.runClasses(TestJunit.class);

      for (Failure failure : result.getFailures()) {
         System.out.println(failure.toString());
      }

      System.out.println(result.wasSuccessful());
   }
}
```

- **JUnit Classes**:
  - `Assert` — contains assertion methods (like `assertTrue()`, `assertEquals()`).
  - `TestCase` — a base class for tests (older JUnit versions).
  - `TestResult` — collects test results.

Example — Simple JUnit Test (Failing Example)

```
import static org.junit.Assert.*;
import org.junit.Test;

public class HelloWorldJUnit {
    @Test
    public void testHelloWorld() {
        String str = "Hello world";
        assertEquals("hello world", str); // Will fail because of case
sensitivity
    }
}
```

This test will **fail** because `"Hello world"` ≠ `"hello world"` (case difference).

8. What is dynamic unit testing and control flow testing

- **Dynamic unit testing** means **running** small parts of the program (units) to **see if they work properly**.
- In this, we **actually execute** the code.

Important Points:

- The unit (a small piece of the program) is **taken out** of the original program and tested separately.
- We create a **testing environment** by writing extra code:
  - **Test Driver**: A small program that **calls** the unit to test it.
  - **Stubs**: Dummy programs that **replace** the other units the tested unit depends on.

Example

```
// Unit under test
public int add(int a, int b) {
    return a + b;
}

// Test Driver
public static void main(String[] args) {
    int result = add(2, 3);
    System.out.println(result); // Should print 5
}
```

How it works:

- The driver calls `add(2, 3)`, and we check if the output is 5.
- If it is, the test passes

Control Flow Testingx

- **Control flow testing** is a special type of dynamic testing where we **check all the different paths** the program can follow.
- It is **white-box testing**, meaning **we know the code**.
- **Important Points:**
  - We draw a **Control Flow Graph (CFG)** showing different paths.
  - Then, we **create test cases** to check each path.

    - This helps find logic errors early.
- **Steps in Control Flow Testing:**
    1. **Create a control flow graph** from the program.
    2. **Decide what parts to test** (nodes, edges, paths).
    3. **Write test cases** to cover all these parts.
    4. **Run the tests**.
    5. **Analyze the results** to find bugs.

Example

```
public void checkNumber(int n) {
    if (n > 0) {
        System.out.println("Positive");
    } else {
        System.out.println("Not Positive");
    }
}
```

- Two paths:
    - n > 0
    - n <= 0
- So, we create two tests:
    - Test with n = 5 (Positive)
    - Test with n = -3 (Not Positive)

9. Discuss the concept of mutation testing with testing process

What is Mutation Testing?

- **Mutation Testing** is a software testing method where **small changes** (called **mutations**) are made to the program's code.
- Then we **run the test cases** to see if they can **catch** these changes (errors).
- The goal is to **check how strong** and **effective** the test cases are.

Main Idea:
If the test cases are good, they should **fail** when they find the mistakes in the changed (mutated) code.

Steps in mutation

Step	What Happens
Step 1	**Make small changes** (mutations) to the source code to create many **mutants**.
Step 2	**Run test cases** on both the **original** and **mutated** programs.
Step 3	**Compare outputs** of the original and mutant programs.
Step 4	If outputs are **different**, the mutant is **killed**. (Good test case!)
Step 5	If outputs are **same**, the mutant is **alive**. (Need better test cases.)

Important Terms

- **Mutants**:
  The **mutated versions** of the original program with small changes.
- **Killed Mutants**:
  Mutants that the test cases **caught** (found errors).
- **Survived (Alive) Mutants**:
  Mutants that the test cases **missed** (errors not detected).
- **Equivalent Mutants**:
  Mutants that are **different in code** but **behave the same** as the original (so they cannot be caught).
- **Mutators (Mutation Operators)**:
  The **rules** that tell **how** to change the code. Example: changing a `+` to `-`, or changing a value from `5` to `10`.

Types of mutation

Type	Meaning
Statement Mutation	Add, remove, or change a statement.
Value Mutation	Change a constant or value. (e.g., 5 → 10)
Decision Mutation	Change logical conditions. (e.g., `>` to `<`, `==` to `!=`)

Advantages of Mutation Testing

- Helps find **hidden bugs** early.
- Improves the **quality of test cases**.
- Helps create a **more reliable and stable** software.
- Tests are very **thorough**.

Disadvantages of Mutation Testing

- **Very time-consuming** (many mutants need to be tested).
- **Expensive** because a lot of testing effort is needed.
- **Needs automation tools** — cannot easily do it manually.
- **Not useful for black-box testing** (because it needs code access).

10. Explain seven types of mutation operators with examples

Mutation operators are **rules** that tell **how** to change (mutate) the program's code to create a **mutant**.

1. Arithmetic Operator Replacement (AOR)

- **Change one arithmetic operator to another** like `+`, `-`, `*`, `/`, `%`.

Example:

Original Code:

```
c = a + b;
```

Mutation:

```
c = a - b;
```

Here, `+` is replaced with `-`.

2. Relational Operator Replacement (ROR)

- **Change relational operators** like `>`, `<`, `>=`, `<=`, `==`, `!=`.

Example:

Original Code:

```
if (a > b)
```

Mutation:

```
if (a < b)
```

`>` is replaced with `<`.

3. Logical Operator Replacement (LOR)

- **Change logical operators** like `&&`, `||`, `!`.

Example:

Original Code:

```
if (a > 0 && b > 0)
```

Mutation:

```
if (a > 0 || b > 0)
```

`&&` is replaced with `||`.

4. Constant Replacement (CR)

- **Change a constant value** to another constant.

Example:

Original Code:

```
int size = 5;
```

Mutation:

```
int size = 10;
```

Constant `5` is replaced with `10`.

5. Statement Deletion (SDL)

- **Delete a statement** from the code.

Example:

Original Code:

```
sum = a + b;
print(sum);
```

Mutation:

```
// sum = a + b; (Deleted)
print(sum);
```

The statement `sum = a + b;` is **deleted**.

6. Unary Operator Insertion/Deletion (UOI)

- **Add or remove a unary operator** like `++`, `--`, `-`, `+`, `!`.

Example:

Original Code:

```
a = b;
```

Mutation:

```
a = -b;
```

Here, a **minus (-)** operator is **added**.

7. Conditional Operator Replacement (COR)

- **Change a conditional expression** like `if (condition)` to always `true` or `false`, or modify the condition.

Example:

Original Code:

```
if (a > b)
```

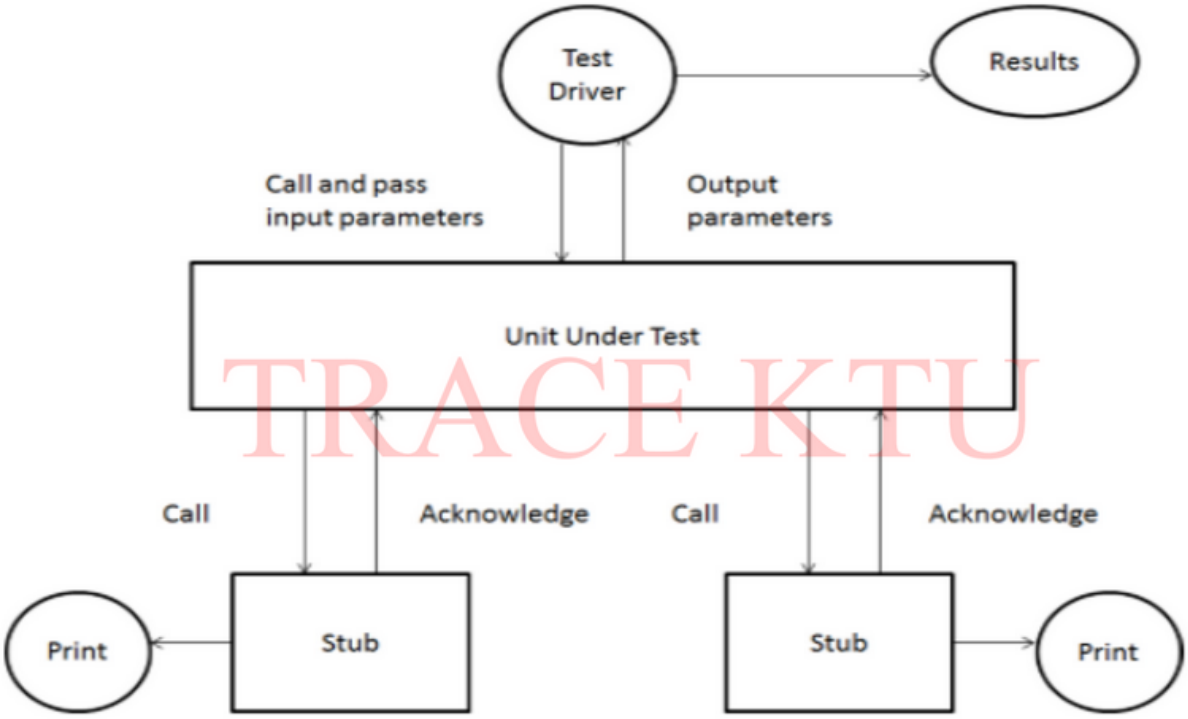
Mutation:

```
if (true)
```

The condition `a > b` is replaced with **true**.

Operator	What it Changes	Example Change
Arithmetic Operator Replacement (AOR)	+, -, *, /, %	`a + b` → `a - b`
Relational Operator Replacement (ROR)	>, <, >=, <=, ==, !=	`a > b` → `a < b`
Logical Operator Replacement (LOR)	&&,	
Constant Replacement (CR)	Numeric constants	`5` → `10`
Statement Deletion (SDL)	Remove a line of code	Delete `sum = a + b;`
Unary Operator Insertion/Deletion (UOI)	Add/remove ++, --, -	`a = b;` → `a = -b;`
Conditional Operator Replacement (COR)	Modify `if` conditions	`if (a > b)` → `if (true)`

11. Explain Dynamic unit test environment with a neat sketch.



Dynamic Unit Testing means **actually running** the code (executing the program) to check for errors.

When you want to dynamically test a **unit** (like a function or a module), you need a **special environment** because the unit usually:

- **Takes inputs from** another function
- **Calls** other functions during its execution

So, to create this environment, we prepare two things:

1. Test Driver

- A **test driver** is a small program written to **call** (invoke) the unit under test.
- It **sends inputs** to the unit.
- It **collects outputs** from the unit.
- It **compares** the output with the **expected result**.

Think of the driver as the "main function" that *starts* the testing.

2. Stubs

- A **stub** is a **dummy function** that **pretends to be** a real function that the unit under test calls.
- If your unit calls other modules which are **not ready yet** or are **complicated**, we use stubs.
- **Stubs show** that they were called (maybe by printing a message).
- **Stubs return fake (predefined) outputs** to allow the unit to continue working.

12. Differentiate between control flow testing and Data flow testing.

Aspect	Control Flow Testing	Data Flow Testing
Definition	Tests the flow of the program's control (execution paths) based on logic and decisions.	Tests the flow of **data** by checking how variables are **defined, used, and destroyed**.
Focus	Focuses on **program structure**, like conditions, loops, and branches.	Focuses on **variables' life cycle** — definition, usage, and killing (deletion).
Purpose	To check **logical errors** in the flow of the program.	To detect **incorrect use of variables** like using uninitialized variables or unused variables.
Model Used	Uses a **Control Flow Graph (CFG)** showing nodes (statements) and edges (transfers of control).	Also uses **Control Flow Graph (CFG)** but emphasizes **data definition and usage points**.
Steps	1. Create Control Flow Graph 2. Define coverage targets (nodes, edges, paths) 3. Create test cases 4. Execute test cases 5. Analyze results	1. Create Data Flow Graph 2. Select test criteria 3. Identify paths based on variable definition and usage 4. Develop test cases
Main Errors Found	- Missing conditions - Incorrect loops - Skipped code blocks - Wrong branching	- Variables used without being defined - Variables defined but never used - Redefinition without use - Deleting variables before use
Type of Testing	Structural (White Box) Testing	Structural (White Box) Testing
Testing Style	Mainly **dynamic** (requires program execution).	Can be **static** (code inspection without execution) or **dynamic**

Aspect	Control Flow Testing	Data Flow Testing
		(executing code).
Advantages	Ensures all code logic and branches are tested.	Catches hidden bugs related to **variables**, improving code reliability.
Disadvantages	May not catch variable misuse easily.	Time-consuming and needs deep programming knowledge.

13. Summarize the pros and cons of static unit testing.

Advantages (Pros):

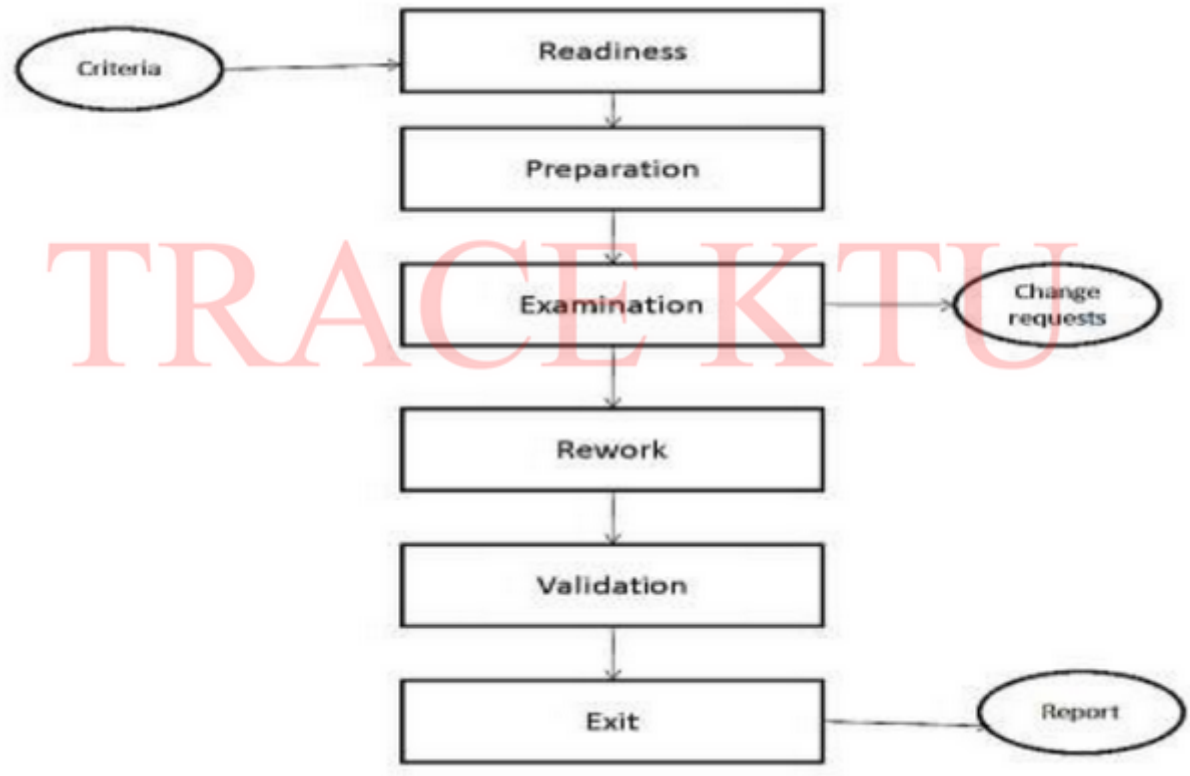
- **Early Detection of Errors:**
  Errors like syntax mistakes, missing variables, and incorrect logic are found **before code execution**.
- **Cost-Effective:**
  Fixing errors early is **cheaper** than fixing them after execution or deployment.
- **Improves Code Quality:**
  Helps ensure the code follows coding standards, improves readability, and reduces complexity.
- **Saves Time Later:**
  By catching mistakes early, it **reduces the time** spent on later debugging and testing.
- **Helps Beginners:**
  Especially helpful for new developers to learn good coding practices through early feedback.
- **No Need for Executable Code:**
  Code can be reviewed **even if it can't be compiled or run yet**.

Disadvantages (Cons):

- **Cannot Find All Errors:**
  Static testing **cannot detect runtime errors** like memory leaks or incorrect program behavior.
- **Requires Expertise:**
  Reviewers must have good programming knowledge to identify deep or logical issues.

- **Time-Consuming for Big Projects:**
  Manually reviewing large codebases can be slow without automation tools.
- **Limited Coverage:**
  Some issues related to performance, integration, or actual output **can only be found through dynamic testing**.
- **False Sense of Security:**
  Developers might think their code is fully correct just because it passed static testing — but dynamic errors can still exist.

14. With the help of a diagram, explain the steps in the code review process



1. Criteria

- Set clear **rules and guidelines** for what to check during code review.
- Examples: coding standards, naming conventions, security practices, etc.
- The goal is to make sure everyone knows **what "good code" looks like**.

2. Readiness

- Before starting the review, **check if the code is ready**.
- Code should be fully written, compiled (if needed), and **meet basic quality** standards.
- If not ready, the review **should be delayed**.

3. Preparation

- Reviewers **read and understand** the code independently.
- They **prepare questions, notes, and observations** about possible issues.
- No discussions yet — just **personal study** of the code.

4. Examination

- The team **meets to review the code together** (physically or online).
- They **discuss issues**, suggest improvements, and clarify doubts.
- Focus is on **finding bugs, bad practices, and design problems** — not criticizing the coder personally.

5. Change Requests

- If problems are found, reviewers **formally request changes**.
- Each issue is **documented** clearly, so the developer knows exactly what to fix.

6. Rework

- The **author of the code fixes** the problems pointed out.
- They **update the code** based on the change requests.

7. Validation

- After rework, **reviewers check** if the corrections are made properly.
- They **re-validate** the updated code to ensure all issues are resolved.

8. Exit

- If all issues are fixed and the code meets the criteria, the review is **officially closed**.
- If not, it may **go through another review cycle**.

9. Report

- A **report is generated** summarizing:
  - What issues were found,
  - What changes were made,
  - Who participated,
  - The final result of the review.
- Helps maintain **records** and **improve future code quality**.