Software-Testing-Series-1-Important-Topics

For more notes visit

https://rtpnotes.vercel.app

1. Mutation testing

- Mutation testing is a technique used in software testing to check how good your test cases are at finding bugs.
- It works by making small changes (mutations) to your program and then running your test cases to see if they catch the changes. If they do, your tests are strong. If they don't, your tests need improvement.

How It Works

1. **Create Mutants** – Slightly modify the code by changing, deleting, or replacing statements.
2. **Run Tests** – Execute the test cases against the modified code.
3. **Check Results**
   - If the test fails, the mutant is **killed** (good!).

- If the test passes, the mutant **survives** (bad – means the test didn't detect the change).

The goal is to have a **high mutation score**, meaning most mutants are killed.

Types of Mutations

- **Statement Mutations** – Deleting or replacing a statement in the code.
- **Mutation Operators (Mutators)** – Rules that define what kind of changes should be made to create mutant versions.

Mutation Score Formula

$$MutationScore = \left(\frac{\text{Killed Mutants}}{\text{Total Mutants}} \right) \times 100$$

If the score is **100%**, the test cases are considered **mutation-adequate**, meaning they can catch all mutants.

Advantages of Mutation Testing

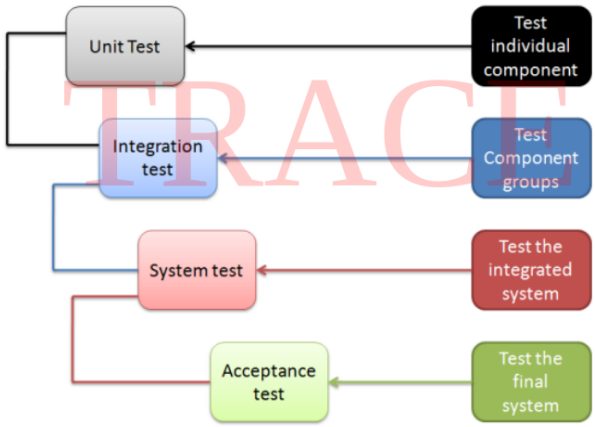
- Helps in achieving high code coverage.
- Effectively tests the robustness of your test cases.
- Detects hidden faults in the program.
- Leads to a more reliable and stable software.

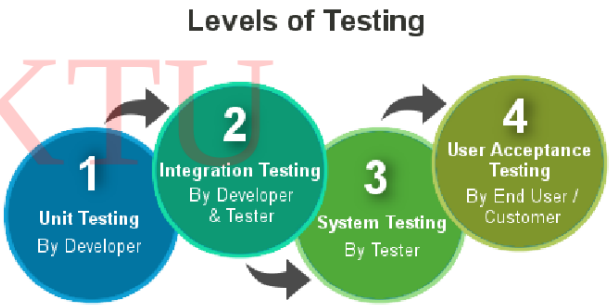
Disadvantages of Mutation Testing

- **Expensive & Time-Consuming** – Many mutant versions need to be tested.
- **Needs Automation** – Doing it manually is impractical.
- **Not Suitable for Black Box Testing** – Since it requires modifying the source code, it can't be applied when testing without looking at the code.

Mutation testing is a great way to measure how effective your test cases are. If your tests catch most mutants, they are strong. However, due to the high cost and time requirements, it's mainly used in critical systems where quality is a top priority.

2. Levels of Testing





1. Unit Testing

- Tests individual components such as functions or methods.
- Ensures each unit works as expected.
- Performed by developers using White Box Testing techniques.
- Example: Checking if a function correctly calculates the sum of two numbers.

Benefits:

- Catches bugs early, making them cheaper to fix.
- Improves code quality and reusability.
- Helps developers make changes confidently.

2. Integration Testing

- Tests how different modules of a system work together.
- Ensures smooth interaction between components.
- Performed after unit testing.

Approaches:

1. **Big Bang Integration**
   - All modules are combined and tested together.
   - Hard to find which module caused a failure.
   - Works well for small systems.
2. **Incremental Integration**
   - Modules are tested step by step.

- Uses Stubs and Drivers (dummy programs to simulate missing modules).
- Easier to pinpoint errors.
- **Bottom-Up Testing** uses Drivers.
- **Top-Down Testing** uses Stubs.

Benefits:

- Ensures smooth communication between modules.
- Helps detect integration issues early.

3. System Testing

- Tests the entire system as a whole.
- Ensures the system meets requirements.
- Uses Black Box Testing techniques.
- Examples: Checking if a shopping cart correctly calculates the total amount or verifying that a login system works with correct and incorrect passwords.

Benefits:

- Ensures all functionalities work as expected.
- Tests system performance, security, and usability.

4. Acceptance Testing

- Final level of testing before releasing the software.
- Ensures the system meets business requirements.
- Performed by end users or clients.

Types of Acceptance Testing:

- **User Acceptance Testing (UAT):** Checks if the software is useful for users.
- **Operational Acceptance Testing (OAT):** Ensures smooth system operation.
- **Contract Acceptance Testing:** Verifies compliance with a contract.
- **Compliance Acceptance Testing:** Ensures the software follows legal regulations.

Benefits:

- Confirms the product is ready for release.

- Reduces risk of failure after launch.

Other Types of Testing:

- **Performance Testing** – Checks speed, scalability, and stability.
- **Stress Testing** – Tests system behavior under extreme load.
- **Usability Testing** – Ensures the software is user-friendly.

3. Black, White and Gray Box Testing

1. Black Box Testing

- The tester does not need to know the internal structure or logic of the system.
- Focuses only on inputs and expected outputs.
- Performed using functional test cases.
- Used to verify system behavior based on specifications.

Example:
Testing a login page where the tester only enters valid and invalid credentials to check if access is granted or denied without looking at the backend code.

Advantages:

- No need for programming knowledge.
- Detects functionality issues from the user’s perspective.
- Effective for large applications.

Disadvantages:

- Does not reveal internal code issues.
- May miss edge cases without detailed internal knowledge.

2. White Box Testing

- The tester has full access to the internal code, logic, and structure.
- Ensures code quality by testing functions, loops, and conditions.
- Used by developers for unit and integration testing.

Example:
Testing a function that calculates discounts by checking all possible conditions within the code.

Advantages:

- Detects logical errors and security vulnerabilities.
- Ensures complete code coverage.
- Helps optimize performance.

Disadvantages:

- Requires programming knowledge.
- Can be time-consuming and complex.

3. Grey Box Testing

- A mix of Black Box and White Box testing.
- The tester has partial knowledge of internal workings but focuses on user experience.
- Useful for integration and security testing.

Example:
Testing an e-commerce website where the tester knows how the database handles transactions but primarily tests user-facing functions like order placement and payment processing.

Advantages:

- Balances functional and structural testing.
- More effective in finding security loopholes.
- Detects issues related to data flow and backend interactions.

Disadvantages:

- Limited access to full system internals.

- Not as thorough as White Box testing for code-level errors.

4. Static and Dynamic

Unit testing is a software testing approach where individual components of a system are tested in isolation. It is broadly classified into **Static Unit Testing** and **Dynamic Unit Testing**, each serving a different purpose in ensuring code quality and correctness.

1. Static Unit Testing

Static unit testing involves inspecting and analyzing code **without executing it**. The focus is on **identifying issues early** through manual or automated code reviews, walkthroughs, and inspections.

Key Aspects:

- Code is **reviewed at each milestone** of the software lifecycle.
- Common techniques:
  - **Inspection**: A structured, step-by-step peer review process to ensure code follows predefined criteria.
  - **Walkthrough**: The developer leads a discussion where the team manually simulates the code's execution.
- This process helps identify syntax errors, coding standards violations, and logical inconsistencies **before actual execution**.

Example:

Before running a newly developed function, the development team **reviews the logic, naming conventions, and potential pitfalls** in a peer review session.

Advantages:

- Detects issues **before execution**, reducing debugging effort.
- Ensures adherence to **coding standards and best practices**.
- Saves time and resources compared to debugging after execution.

Disadvantages:

- Cannot detect **runtime errors** (e.g., memory leaks, incorrect calculations).
- Requires manual effort, which may be time-consuming.

2. Dynamic Unit Testing

Dynamic unit testing involves **executing the code** and checking whether it produces the expected output. It helps verify functional correctness by running test cases in an **isolated environment**.

Key Aspects:

- The unit is executed **outside its normal environment** using a test harness.
- **Additional code (scaffolding)** is written to simulate real-world interactions.
- The outcome is collected using logs, direct observations, or software instrumentation.
- The result is compared to the **expected output**.

Example:

A function that calculates **tax on a purchase** is tested by providing different inputs and verifying that the returned values match the expected results.

Comparison of Static and Dynamic Unit Testing

Feature	Static Unit Testing	Dynamic Unit Testing
Execution	**No execution**, only code analysis	**Code is executed** with test inputs
Focus	**Code structure, syntax, logic**	**Functional correctness, runtime behavior**
Detects Errors	Syntax issues, coding standards violations	Runtime errors, incorrect calculations, unexpected behaviors
Tools	Code review tools, linters, checklists	Unit testing frameworks (JUnit, pytest, Google Test)
Effort Required	Manual or automated review	Requires writing test cases and scaffolding

- **Static Unit Testing** is essential for early defect detection and maintaining code quality.
- **Dynamic Unit Testing** validates functionality by executing code in a controlled environment.

- Combining both ensures a **robust and reliable** software development process.

5. Test driver and Test stubs

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

Let's put everything 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.