UNIT III

Programming paradigms in software development: -

Programming paradigms are various approaches or styles used in software development to solve problems and design software systems. Each paradigm has its own set of principles and practices that guide how programmers' structure and write their code. Some common programming paradigms include:

1. Procedural Programming

Principles: Focuses on a sequence of instructions that are executed in order, typically using functions or procedures.

Languages: C, Pascal, Fortran, Python (can be used procedurally), JavaScript etc.

Key Concepts: Functions, control flow, modularization, and code reusability.

2. Object-Oriented Programming (OOP)

Principles: Based on the concept of "objects" that encapsulate both data and methods. Objects represent real-world entities, and interaction between objects happens through messages (method calls).

Languages: Java, C++, Python (supports OOP), C# etc.

Key Concepts: Classes, objects, inheritance, polymorphism, encapsulation, and abstraction.

3. Functional Programming

Principles: Treats computation as the evaluation of mathematical functions and avoids changing state or mutable data.

Languages: Lisp, JavaScript (supports functional style) etc.

Key Concepts: Pure functions, immutability, higher-order functions, recursion, and function composition.

4. Declarative Programming

Principles: Specifies what the program should accomplish rather than how to accomplish it, leaving the details of execution to the system.

Languages: SQL (for databases), HTML (for structuring content), CSS (for styling) etc.

Key Concepts: Focus on logic and rules, not explicit control flow.

5. Event-Driven Programming

Principles: Program behavior is determined by events such as user inputs (mouse clicks, keypresses) or messages from other programs.

Languages: JavaScript (widely used for web applications), C#, VB.NET etc.

Key Concepts: Event handlers, listeners, and callbacks.

6. Concurrent and Parallel Programming

Principles: Involves executing multiple computations simultaneously, either through parallel execution (multi-core processing) or concurrent threads (handling multiple tasks at once).

Languages: Java (with concurrency libraries), Python (via threading/multiprocessing), C++ etc.

Key Concepts: Threads, processes, locks, synchronization, and message passing.

7. Component-Based Programming

Principles: Emphasizes the design and use of reusable, interchangeable components (modular units of functionality) that can be composed to create a system.

Languages: C#, Java (with Enterprise JavaBeans (EJB)) etc.

Key Concepts: Components, interfaces, and reusability.

8. Data-Driven Programming

Principles: Data drives the control flow of the program, with transformations or computations directly tied to data changes or input.

Languages: R, Python (with Pandas, NumPy), SQL etc.

Key Concepts: Data pipelines, data transformations, and real-time updates.

9. Logic Programming

Principles: Based on formal logic where a program consists of a set of sentences in logical form, expressing facts and rules about problems.

Languages: Prolog etc.

Key Concepts: Facts, rules, and queries; reasoning and inference.

Coding standards: -

Coding standards in software engineering refer to a set of guidelines, conventions, and best practices that help developers write consistent, maintainable, and error-free code. Adopting and following coding standards is essential for ensuring code quality, improving collaboration among team members, and facilitating easier debugging and maintenance.

Common Coding Standards and Practices –

1. Naming Conventions

- Variables and Constants:
  - Use descriptive names that reflect the purpose of the variable (e.g., userCount instead of x).
  - Use camelCase for variables and functions (e.g., isLoggedIn)
  - PascalCase for class names (e.g., UserManager)
  - UPPER_CASE with underscores for constants (e.g., MAX_RETRY_COUNT).

- Classes and Methods:

Class names should be nouns (e.g., User, OrderManager), and method names should be verbs (e.g., calculateTotal, sendEmail).

2. Indentation and Spacing

- Use consistent indentation (commonly 2 or 4 spaces) for readability.
- Properly align braces {} and parentheses () to improve the structure and readability of code blocks.
- Ensure consistent use of whitespace around operators and after commas.

Example:

```
def calculate_sum(a, b):
  result = a + b
  return result
```

3. Code Comments and Documentation

Inline Comments: Brief comments that explain complex code logic or describe the purpose of a specific block of code.

Block Comments: Larger comments that explain the overall function or purpose of a module or method.

Documentation Comments: Use tools like Sphinx, MkDocs, Pdoc, Doxygen etc., to generate documentation from comments for functions, classes, and methods.

Documentation Tools for Python –

i. Sphinx – Sphinx is one of the most popular documentation generation tools in Python. It parses reStructuredText (reST) or Markdown-formatted docstrings and generates documentation in formats like HTML, PDF, or LaTeX.

ii. Pdoc – pdoc is a simple, fast tool to generate API documentation for Python libraries based on docstrings.

iii. MkDocs + mkdocstrings – MkDocs is a static site generator for documentation. Combined with the mkdocstrings plugin, it can extract docstrings from Python code and generate beautiful documentation websites.

iv. **Doxygen –** While originally intended for C++ and other languages, Doxygen can also be used with Python to generate documentation from docstrings.

Docstring Formats:

Python supports various formats for writing docstrings:

reStructuredText (reST): Used by Sphinx.

Google Style: Simple and popular for many projects.

NumPy/SciPy Style: Commonly used in scientific Python libraries.

Example of Docstrings in Python –

Python function with a Google-style docstring

Example:

```
def add_numbers(a, b):
    """
    Adds two numbers.

    Args:
        a (int or float): The first number.
        b (int or float): The second number.
```

```
Returns:
    int or float: The sum of a and b.
Example:
    >>> add_numbers(3, 4)
    7
    >>> add_numbers(10.5, 4.5)
    15.0
  """
  return a + b
```

Sphinx (reST format)

Docstring using reStructuredText (reST), which is the default format for Sphinx:

```
def add_numbers(a, b):
  """
  Adds two numbers.

  :param a: The first number (int or float).
  :param b: The second number (int or float).
  :return: The sum of a and b (int or float).
```

Generating Documentation with Tools –

1. Sphinx

Install Sphinx

pip install sphinx

Run sphinx-quickstart to set up a documentation project.

Write your Python code with docstrings and configure Sphinx to parse them.

2. Pdoc

Install pdoc

pip install pdoc

Generate HTML documentation with pdoc

pdoc --html my_module.py

3. MkDocs + mkdocstrings

Install MkDocs and the mkdocstrings plugin:

pip install mkdocs mkdocstrings[python]

Set up your mkdocs.yml file with:

```
plugins:
  - mkdocstrings:
      default_handler: python
```

Run “mkdocs serve” to generate and view the documentation.

4. Code Structure and Organization

Separation of Concerns: Divide code into distinct functions, classes, or modules, each with a single responsibility.

Modularity: Write small, reusable functions and modules to improve clarity and reduce redundancy.

Single Responsibility Principle (SRP): Each class or method should have one responsibility or purpose.

Consistent Ordering: Follow a consistent order of defining variables, methods, and classes in a file (e.g., public methods first, then private methods, or helper functions at the end).

5. Error Handling

Implement consistent error-handling practices.

Use exceptions and try/catch blocks appropriately to handle errors without causing program crashes.

Avoid using generic exceptions; always handle specific errors to improve debugging and troubleshooting.

Example:

```
try:
    result = divide(a, b)
except ZeroDivisionError:
    print("Error: Division by zero is not allowed.")
```

6. Code Readability

Avoid magic numbers: Use named constants instead of hard-coded values that lack meaning (e.g., use MAX_CONNECTIONS instead of 10).

Avoid deep nesting: Refactor nested conditions or loops to enhance readability.

Break long lines: Limit the length of code lines (typically 80–120 characters) to avoid horizontal scrolling and improve readability.

7. Version Control and Commit Messages

Write clear and descriptive commit messages that summarize the changes made (e.g., Fixed bug in payment processing).

Use version control systems (e.g., Git) to track changes and maintain a history of the codebase.

8. Code Reviews

Establish regular code reviews to ensure compliance with coding standards, improve code quality, and catch potential issues early.

Peer reviews help in maintaining consistent coding practices across the team.

9. Performance Optimization

Write efficient algorithms and use appropriate data structures.

Avoid premature optimization; focus on code clarity first, and optimize only when necessary.

Minimize unnecessary operations, such as redundant calculations inside loops.

10. Security Best Practices

Validate user inputs to prevent injection attacks.

Avoid hardcoding sensitive information (e.g., passwords, API keys) directly in the code. Use environment variables or configuration files.

Implement proper access control mechanisms and handle sensitive data securely.

11. Testing

Write unit tests, integration tests, and other automated tests to ensure code correctness.

Follow the Test-Driven Development (TDD) approach, where possible.

Use descriptive test names and ensure tests cover both expected and edge cases.

Conventions: -

In software engineering, conventions are established practices, rules, or guidelines followed to improve code readability, maintainability, consistency, and collaboration within a team or organization. They apply to various aspects of the software development process, including code structure, naming, documentation, testing etc.

Types of conventions in software engineering:

1. Coding Conventions

(1) Naming Conventions – Naming conventions define how variables, functions, classes, and other entities should be named. These conventions help improve the readability and clarity of the code.

Variables:

Use camelCase (e.g., userAge, isLoggedIn) for variables and functions.

Use PascalCase (e.g., UserManager, PaymentProcessor) for class names.

Use UPPER_CASE with underscores for constants (e.g., MAX_RETRY_COUNT, DEFAULT_TIMEOUT).

Functions and Methods:

Use descriptive verbs that reflect their actions (e.g., calculateTotal, sendEmail).

Methods should follow a pattern based on functionality, such as getName, setName, isAvailable.

Classes and Modules:

Class names should be nouns, describing the entity (e.g., Customer, Invoice).

Module names should reflect their purpose (e.g., authentication, databaseManager).

(2) Indentation – Proper use of spaces or tabs to structure code blocks, ensuring the code is easily readable.

(3) Commenting – Guidelines on when and how to write comments in the code, which explain complex logic or give context to specific parts of the code.

2. Version Control Conventions

Version control systems (VCS) like Git help teams collaborate and manage code changes. Version control conventions helps keep a project organized and maintainable.

Best practices for software development: -

Best practices in software development help teams produce high-quality software that is maintainable, scalable, and reliable. These practices apply to various stages of the development lifecycle and across different methodologies (Agile, Waterfall, etc.).

Some best practices:

1. Version Control

Use a Version Control System (VCS): Systems like Git help track changes, manage code versions, and enable collaboration.

Commit Early and Often: Regular, small commits make it easier to trace changes and debug.

Use Branches: Develop new features or fix bugs in separate branches to isolate work before merging.

Clear Commit Messages: Write descriptive messages to explain the purpose of each commit.

2. Coding Standards and Clean Code

Write Clean Code: Focus on clarity and simplicity, avoiding excessive complexity.

Meaningful Naming: Use clear, descriptive variable and function names.

Small Functions: Break large functions into smaller, single-purpose functions.

Avoid Code Duplication: It helps to avoid redundancy.

3. Code Reviews and Pair Programming

Code Reviews: Peer reviews help catch bugs, improve code quality, and ensure adherence to standards.

Pair Programming: Working in pairs can lead to better solutions and faster knowledge sharing.

4. Automated Testing

Unit Testing: Test individual components to ensure they behave correctly in isolation.

Integration Testing: Verify that different modules work together as expected.

Test-Driven Development (TDD): Write tests before the actual code to guide the design.

5. Continuous Integration and Continuous Deployment (CI/CD)

Continuous Integration: Regularly integrate code into a shared repository. Use automated tests to validate the integration.

Continuous Deployment: Automate the deployment process to quickly release updates into production.

6. Documentation

Document the Code: Write comments and documentation to explain the purpose of functions, classes, and modules.

Maintain a README: Provide clear setup instructions, usage examples, and dependency information.

7. Agile Development and Iterative Processes

Adopt Agile Methodologies: Use Scrum or Kanban to manage tasks and improve team collaboration.

Frequent Iterations: Deliver small, incremental improvements with feedback loops.

Daily Standups: Keep team members aligned with daily status updates and discussions on blockers.

8. Refactoring

Regularly Refactor: Continuously improve code without changing its behavior to make it more readable and maintainable.

Unit Testing: -

Testing –

Testing is a set of activities that can be planned in advance and conducted systematically.

For this reason, a template for software testing — a set of steps into which we can place specific test-case design techniques and testing methods should be defined for the software process.

Template for testing, provides following generic characteristics:

- To perform effective testing, you should conduct technical reviews. By doing this, many errors will be eliminated before testing commences.
- Testing begins at the component level and works “outward” toward the integration of the entire computer-based system.
- Different testing techniques are appropriate for different software engineering approaches and at different points in time.
- Testing is conducted by the developer of the software and for large projects an independent test group.
- Testing and debugging are different activities, but debugging must be accommodated in any testing strategy.

Software testing –

A strategy for software testing accommodates the low-level tests necessary to verify that a small source code segment has been correctly implemented as well as high-level tests that validate major system functions against customer requirements.

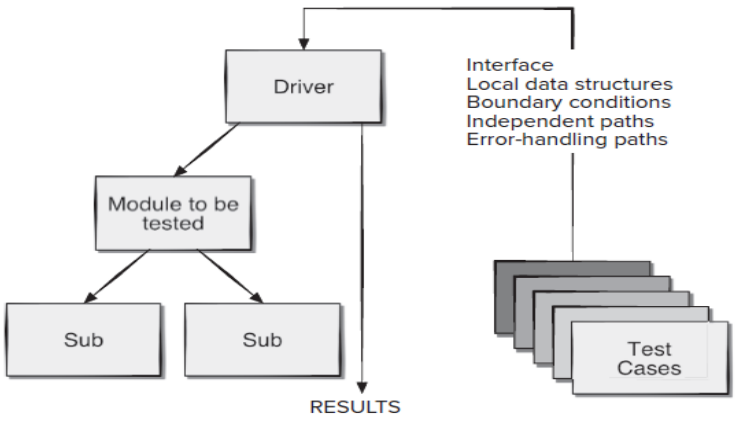
Unit testing –

Unit testing focuses verification effort on the smallest unit of software design—the software component or module.

The unit test focuses on the internal processing logic and data structures within the boundaries of a component. This type of testing can be conducted in parallel for multiple components.

Component testing is normally considered as an adjunct to the coding step. The design of unit tests can occur before coding begins or after source code has been generated.

Unit-test environment



A component is not a stand-alone program, some type of scaffolding (i.e., drivers and stubs) is required to create a testing framework.

For each unit test driver and/or stub software must often be developed.

Driver: In most applications a driver is a “main program” that accepts test-case data, passes such data to the component (to be tested), and prints relevant results.

Stubs: It serve to replace modules that are subordinate (invoked by) the component to be tested. A stub or “dummy subprogram” uses the subordinate module’s interface, may do minimal data manipulation, prints verification of entry, and returns control to the module undergoing testing.

Drivers and stubs represent testing “overhead.” That is, both are software that must be coded but that is not delivered with the final software product.

If drivers and stubs are kept simple, actual overhead is relatively low. Unfortunately, many components cannot be unit-tested with “simple” scaffolding software. In such cases, complete testing can be postponed until the integration test step (where drivers or stubs are also used).

Unit tests –

The module interface is tested to ensure that information properly flows into and out of the program unit under test.

Local data structures are examined to ensure that data stored temporarily maintains its integrity during all steps in an algorithm’s execution.

All independent paths through the control structure are exercised to ensure that all statements in a module have been executed at least once.

Boundary conditions are tested to ensure that the module operates properly at boundaries established to limit or restrict processing. And finally, all error-handling paths are tested.

When object-oriented software is considered, Class is usually the focus of unit testing. However, operations (methods) within the class are the smallest testable units. Because a class can contain a number of different operations, and a particular operation may exist as part of a number of different classes, the tactics applied to unit testing must change

Cost-Effective Testing –

Exhaustive testing requires every possible combination of input values and test-case orderings be processed by the component being tested.

Some techniques for minimizing the number of test cases required to do a good job testing.

Integration testing: -

Integration testing is a systematic technique for constructing the software architecture while at the same time conducting tests to uncover errors associated with interfacing. The objective is to take unit-tested components and build a program structure that has been dictated by design.

(A) “big bang” approach (non-incremental integration):

There is a tendency to attempt non-incremental integration, that is, to construct the program using a “big bang” approach. In the big bang approach, all components are combined in advance and the entire program is tested as a whole. Where, errors are encountered, but correction is difficult because isolation of causes is complicated by the vast expanse of the entire program.

(B)Incremental integration:

Incremental integration is the antithesis of the big bang approach. The program is constructed and tested in small increments, where errors are easier to isolate and correct; interfaces are more likely to be tested completely; and a systematic test approach may be applied.

Several common incremental integration testing strategies are –

1. Top-Down Integration

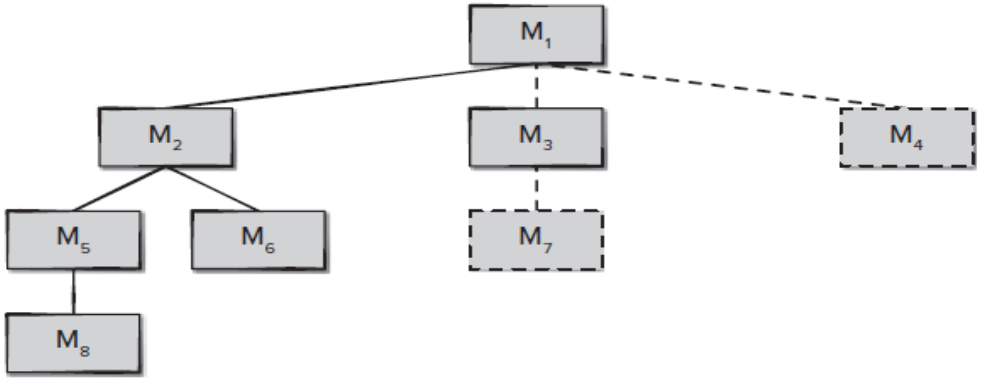
Top-down integration testing is an incremental approach to construction of the software architecture. Modules (also referred to as components) are integrated by moving downward through the control hierarchy, beginning with the main control module (main program). Modules subordinate to the main control module are incorporated into the structure in either a depth-first or breadth-first manner.

(a) Depth-first integration:

Depth-first integration integrates all components on a major control path of the program structure. Selection of a major path is somewhat arbitrary(random) and depends on application-specific characteristics (e.g., components needed to implement one use case).

For example, selecting the left-hand path, components M1, M2, M5 would be integrated first. Next, M8 or (if necessary for proper functioning of M2) M6 would be integrated. Then, the central and right-hand control paths are built.

Depth-first integration



(b) Breadth-first integration:

Breadth-first integration incorporates all components directly subordinate at each level, moving across the structure horizontally. Where from the above figure, components M2, M3, and M4 would be integrated first. The next control level, M5, M6, and so on, follows.

The integration process is performed in a series of five steps:

1. The main control module is used as a test driver, and stubs are substituted for all components directly subordinate to the main control module.

2. Depending on the integration approach selected (i.e., depth or breadth first), subordinate stubs are replaced one at a time with actual components.

3. Tests are conducted as each component is integrated.

4. On completion of each set of tests, another stub is replaced with the real component.

5. Regression testing may be conducted to ensure that new errors have not been introduced.

The process continues from step 2 until the entire program structure is built.

The top-down integration strategy verifies major control or decision points early in the test process.

2. Bottom-Up Integration

Bottom-up integration testing, as its name implies, begins construction and testing with atomic modules (i.e., components at the lowest levels in the program structure).

Bottom-up integration eliminates the need for complex stubs. Because components are integrated from the bottom up, the functionality provided by components subordinate to a given level is always available and the need for stubs is eliminated.

A bottom-up integration strategy may be implemented with the following steps:

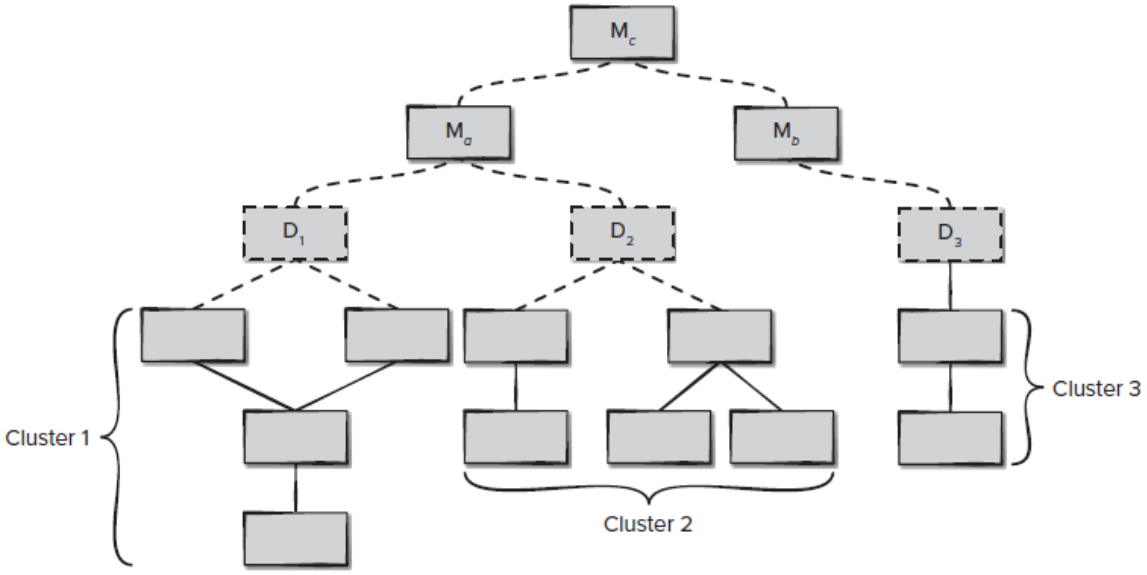
1. Low-level components are combined into clusters (sometimes called builds) that perform a specific software subfunction.

2. A driver (a control program for testing) is written to coordinate test-case input and output.

3. The cluster is tested.

4. Drivers are removed and clusters are combined, moving upward in the program structure.

Bottom-Up Integration



Components are combined to form clusters 1, 2, and 3. Each of the clusters is tested using a driver (shown as a dashed block). Components in clusters 1 and 2 are subordinate to Ma. Drivers D1 and D2 are removed, and the clusters are interfaced directly to Ma. Similarly, driver D3 for cluster 3 is removed prior to integration with module Mb. Both Ma and Mb will ultimately be integrated with component Mc, and so forth.

Verification and Validation: -

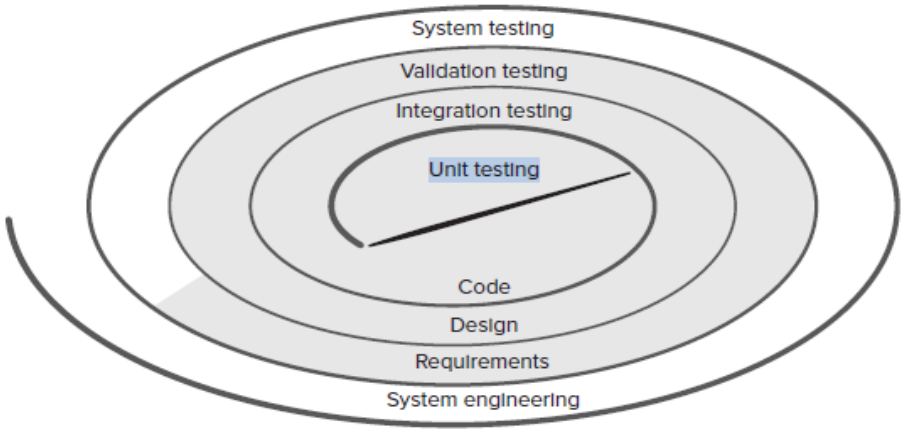
Verification refers to the set of tasks that ensure that software correctly implements a specific function.

Validation refers to a different set of tasks that ensure that the software that has been built is traceable to customer requirements.

Verification: “Are we building the product, right?”

Validation: “Are we building the right product?”

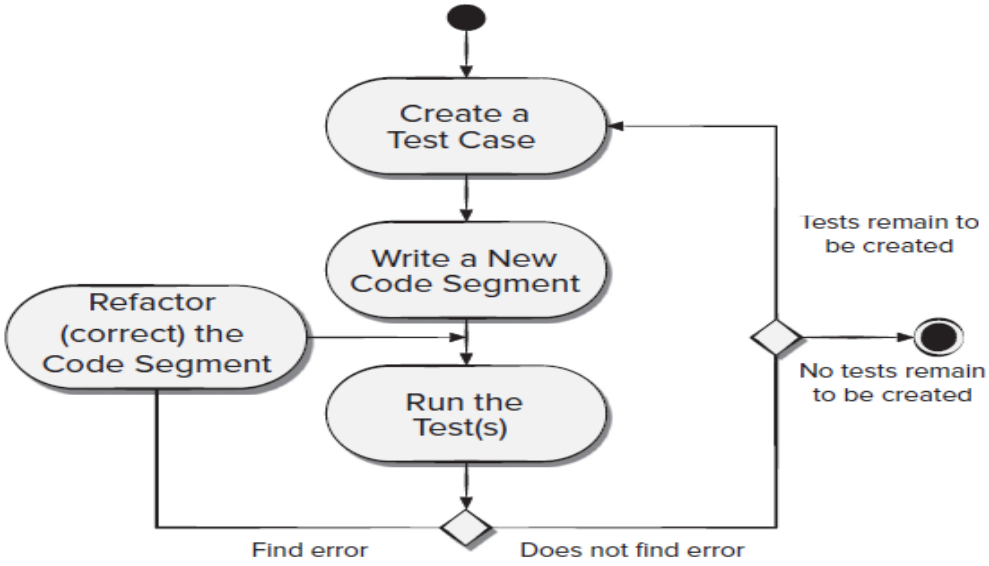
Testing Strategy



Test-Driven Development: -

- In test-driven development (TDD), requirements for a software component serve as the basis for the creation of a series of test cases that exercise the interface and attempt to find errors in the data structures and functionality delivered by the component.
- TDD is not a new technology but rather a trend that emphasizes the design of test cases before the creation of source code.
- The TDD process follows the simple procedural flow before the small segment of code is created, a software engineer creates a test to exercise the code (to try to make the code fail).

Test-driven development process flow



In the TDD process, a software engineer creates a test to exercise the code (to try to make the code fail). The code is then written to satisfy the test. If it passes, a new test is created for the next segment of code to be developed.

The process continues until the component is fully coded and all tests execute without error. However, if any test succeeds in finding an error, the existing code is refactored(corrected) and all tests created to that point are executed again. This iterative flow continues until there are no tests left to be created, implying that the component meets all requirements defined for it.

During TDD, code is developed in very small increments (one subfunction at a time), and no code is written until a test exists to exercise it.

Each iteration results in one or more new tests that are added to a regression test suite that is run with every change. This is done to ensure that the new code has not generated side effects that cause errors in the older code.

Regression refers to a bug or issue that occurs when a software change breaks existing functionality. This can happen after changes to the software's source code, or to the environment in which the software is running.

Behavior-Driven Development (BDD): -

Behavior-Driven Development (BDD) is a software development methodology that helps teams design and document applications around the user's expected experience. BDD is based on the idea of specification by example, and uses a language that's focused on the business and user perspective.

Some key aspects of BDD –

Collaboration: BDD is a collaborative process that involves developers, managers, and customers. It creates a shared understanding of requirements between the business and the development team.

Documentation: BDD creates documentation that is easy to maintain and can be used by all team members.

Automated testing: BDD tests can be automated to ensure that the system continues to meet the specified behavior as it evolves.

Focus on user experience: BDD helps developers focus on the requested behaviors of the app, avoiding unnecessary features or excessive code.

Language: BDD uses a language like Gherkin to describe application behavior. The most common format for examples is Given-When-Then, which describes the initial context, the action, and the expected outcome.

Benefits: BDD helps to improve collaboration, understanding of requirements, and early defect detection.

System Testing: -

System testing in software engineering is a critical phase of software testing that involves testing the entire system as a whole to ensure it meets the specified requirements. It occurs after integration testing and is the final stage before the software is deployed or delivered to the customer. The primary goal is to validate that the complete system functions correctly, performs as expected, and satisfies the requirements outlined in the software's specifications.

Key Aspects of System Testing –

End-to-End Testing: System testing verifies the functionality of the system from end to end, checking all interactions between components, subsystems, and external interfaces.

Functional Testing: This includes verifying that the software's functionalities align with the requirements. Each feature is tested to ensure it behaves as expected.

Non-Functional Testing: This includes performance testing, security testing, usability testing, and compatibility testing. Non-functional testing checks the software's behavior under specific conditions, such as load or stress.

Environment Testing: System testing is performed in an environment that closely resembles the production environment, helping to identify environment-specific issues.

Test Cases: Test cases for system testing are based on functional and non-functional requirements. They cover the entire system and its interactions with external systems.

Types of System Testing –

Performance Testing: Verifying how the system performs under various conditions (e.g., load testing, stress testing).

Security Testing: Ensuring the system is secure against vulnerabilities and threats.

Usability Testing: Evaluating the ease of use and user experience.

Recovery Testing: Testing the system's ability to recover from failures, such as crashes or system outages.

Compatibility Testing: Ensuring the system works across different devices, browsers, or platforms.

Automation: Some aspects of system testing, especially regression testing, are often automated to save time and ensure consistency.

Defect Detection: Since system testing focuses on the whole system, it can uncover integration issues, interface mismatches, performance bottlenecks, and functional defects that may have been missed in earlier stages.

Importance of System Testing –

Comprehensive Validation: Ensures that the software system works as expected in real-world conditions.

Risk Mitigation: Identifies and addresses major defects before the system is deployed to users.

Quality Assurance: Confirms that both functional and non-functional requirements are satisfied, improving software quality and reliability.

Here’s a step-by-step guide to setting up a basic CI pipeline using GitHub Actions: -

Step 1: Create a GitHub Repository

If you don’t have a repository, create one on GitHub by following these steps:

- Log in to GitHub.
- Create a new repository by clicking on the New button.
- Provide a repository name and set visibility (public or private).
- Initialize the repository with a README file (optional).

Step 2: Create a ***.github/workflows*** Directory

Inside your repository, create the directory where the workflow files will be stored.

1. Clone your repository locally:

In command prompt –

- C:\Users\Basanti>git clone https://github.com/<your-username>/ <your-repo>.git
- cd <your-repo>

2. Inside the root of your project, create a directory for workflows

- mkdir -p .github/workflows

Step 3: Create a Workflow YAML File

Now, you'll create a GitHub Actions workflow YAML file

1. Create a file named ***ci.yml*** in the ***.github/workflows/*** directory:

- touch .github/workflows/ci.yml

Or

- echo. > .github/workflows/ci.yml

2. Open the file and define the workflow. Below is an example:

```
name: Python CI on Windows


# Trigger the workflow on every push or pull request to the main branch
on:
  push:
    branches:
      - main
  pull_request:
    branches:
      - main
jobs:
```

```
  build:
    runs-on: windows-latest  # Use the latest Windows runner
    strategy:
      matrix:
        python-version: ['3.7', '3.8', '3.9', '3.11.4']  # Test on multiple Python versions
steps:
      # Step 1: Check out the repository code
      - name: Checkout code
        uses: actions/checkout@v2
      # Step 2: Set up Python (Choose the desired version)
      - name: Set up Python
        uses: actions/setup-python@v2
        with:
          python-version: ${{ matrix.python-version }} # '3.9' for Specify Python version
      # Step 3: Install dependencies from requirements.txt (if it exists)
      - name: Install dependencies
        run: |
          python -m pip install --upgrade pip  # Upgrade pip
          if (Test-Path requirements.txt) { pip install -r requirements.txt }
      # Step 4: Run tests
      - name: Run tests
        run: pytest
```

Step 4: Push the Workflow File to GitHub

After setting up the YAML file, commit the changes and push them to your repository.

- git add .github/workflows/ci.yml
- git commit -m "Add CI pipeline using GitHub Actions“
- git push origin main

error: src refspec main does not match any

- C:\Users\Basanti\my_pipeline>dir .git\refs\heads

  Output –

  Volume in drive C has no label.

  Volume Serial Number is 1E80-7F7B

  Directory of C:\Users\Basanti\my_pipeline\.git\refs\heads

  10/03/2024 02:36 AM <DIR> .

  10/03/2024 02:36 AM <DIR> ..

  10/03/2024 02:09 AM 41 .invalid

  1 File(s) 41 bytes

  2 Dir(s) 175,088,762,880 bytes free

- C:\Users\Basanti\my_pipeline>del .git\refs\heads*.lock

  Output –

  Could Not Find C:\Users\Basanti\my_pipeline\.git\refs\heads*.lock

- C:\Users\Basanti\my_pipeline>del .git\refs\heads\.invalid
- C:\Users\Basanti\my_pipeline>git add .
- C:\Users\Basanti\my_pipeline>git commit -m "Add Python CI pipeline using GitHub Actions"

  Output –

  fatal: unable to resolve HEAD after creating commit

- C:\Users\Basanti\my_pipeline>rmdir .git /s /q
- C:\Users\Basanti\my_pipeline>git init

  Output –

  Initialized empty Git repository in C:/Users/Basanti/my_pipeline/.git/

- C:\Users\Basanti\my_pipeline>git add .

- C:\Users\Basanti\my_pipeline>git commit -m "Initial commit"

  Output –

  [master (root-commit) e797012] Initial commit

  1 file changed, 34 insertions(+)

  create mode 100644 .github/workflows/ci.yml

- C:\Users\Basanti\my_pipeline>git push origin master

in above command in place of master take main or branch If not working