MAD 1 WEEK 10

Lec1:Application Testing

Why?

Testing ensures that the application works as intended by checking:

- **Requirements**: Meets the specified requirements.
- **Input Handling**: Responds correctly to inputs.
- **Performance**: Operates within a reasonable time.
- **Environment**: Installs and works in the expected environment.
- **Usability & Correctness**: Is user-friendly and error-free.

What?

Application testing involves evaluating the behavior, functionality, and performance of software to detect and fix issues.

When?

- During **development**: To identify issues early.
- After **deployment**: To ensure continued functionality.
- During **updates**: To prevent regressions.

How?

1. Define test cases based on requirements.
2. Choose the type of testing (static, dynamic, etc.).
3. Execute tests and analyze results.
4. Fix detected issues and re-test.

Pytest

- A Python testing framework.
- **Why use Pytest?**
  - Easy to write and execute tests.
  - Supports detailed reporting and fixtures.
  - Great for both small and complex testing needs.

Static vs. Dynamic Testing

Static Testing

- Involves reviewing code without executing it.
- Examples: **Code reviews, correctness proofs.**

Dynamic Testing

- Involves executing the application with different inputs.
- Examples: **Functional tests, performance tests.**

White-box Testing

- **Internal structure known**: Tests are designed based on code.
- **Pros**:
  - Provides detailed insights.
  - Creates more thorough tests.
- **Cons**:
  - May lead to focusing on less important parts.
  - Can result in over-complication.
  - Does not encourage clean abstraction.

Black-box Testing

- **Code unknown**: Tests are based on expected functionality.
- **Pros**:
  - Mimics real-world usage.

    - Encourages clean interface design.
- **Cons**:
    - May miss edge cases.
    - Debugging is harder since code isn't visible.

Grey-box Testing

- **Combination** of white-box and black-box testing.
- Uses the internal structure for debugging but enforces interface testing.

Regressions

- **What?**: Loss of functionality due to code changes.
- **Why?**: Modifications in the code can break existing features.
- **Solution**:
    - Maintain a series of tests for all features.
    - Update tests and API versions as necessary.

Coverage

What is Coverage?

- **Definition**: How much of the code is tested.
- **Types**:
    - **Code coverage**: Every line executed at least once.
    - **Branch coverage**: All decision branches tested.
    - **Condition coverage**: Each condition within decisions tested.

Why is Coverage Important?

- Ensures thorough testing but doesn’t guarantee correctness in all cases.

Coverage Examples

Function Coverage

Goal: Ensure the function is invoked at least once.

```
def foo(x, y):
    z = 0
    if x > 0 and y > 0:
        z = x
    return z
```

- Example: `foo(1, 1)`

Statement Coverage

Goal: Ensure all statements are executed.

- Example: `foo(1, 1)` ensures all lines are run.

Branch Coverage

Goal: Test all branches of decision points.

- Example:
  - `foo(1, 1)` (branch taken)
  - `foo(1, 0)` (branch not taken)

Condition Coverage

Goal: Test all conditions within decisions.

- Example:
  - `foo(0, 1)` (first condition fails, second succeeds)
  - `foo(1, 0)` (first condition succeeds, second fails)

Lec 2:Levels of Testing

Initial Requirements Gathering

Stakeholders

- **Students**: Log in and view courses.
- **Admins**: Manage student information.
- **Teachers**: Update and manage course materials.

Requirements

1. **Functional**:
    - Address the unique needs of each stakeholder group.
2. **Non-functional**:
    - Design elements like page colors, fonts, and logos.

Example: Student Page

Functional Requirements

- Display latest updates.
- Allow students to register exam hall preferences.
- Enable hall ticket downloads.
- Update course registration.
- Show completed courses.

Non-Functional Requirements

- Consistent header and footer colors.
- Display copyright information.
- Use appropriate logos and fonts.

Requirements Gathering

- Conduct extensive discussions with end-users.
- Avoid ambiguity in language and specifications.
- Capture detailed use cases and examples.
- Start drafting test cases to validate requirements effectively.

Units of Implementation

- Break functional requirements into smaller, manageable units.
- Examples of units:
  - View course list.
  - Edit course status.
  - Modify exam preferences.
  - Download completion certificates.

Controller Design

- Each unit may correspond to an individual controller.
- Some units can be combined into a single controller for efficiency.

Unit Testing

- Test individual units of functionality in isolation.
- Clearly define inputs and expected outputs.
- Create artificial datasets for testing purposes.

Example: Unit Test

- **Scenario**: A student registers for a course.
  1. Create a dummy database:
     - Include one student and one course.
  2. Test:
     - Controller functionality to add the course.
     - Displaying the form correctly.
     - Handling invalid inputs (e.g., student ID or course ID).
     - Preventing duplicate registrations.

Integration Testing

- Focus on interactions between modules:
  - **Modules**: Student management, course management, payment systems, admin interface.
  - **Examples**:
    - Student + Payment Gateway.
    - Student + Course + Admin.

Challenges

- Modules may function independently but fail when integrated.
- Dependency issues might require server redesign.

Continuous Integration (CI)

- Combine with version control systems.
- Automatically trigger integration tests on commits to the main branch.
- Run multiple tests daily for consistent validation.

System-Level Testing

- Validate the entire application, including servers and environment.
- Primarily black-box testing to simulate real-world use.

Example: Online Degree Dashboard

- Deploy the application in its final environment (e.g., Google App Engine).
- Test domain connections and overall behavior.
- **Non-functional Tests**:
  - Performance under load.
  - Scaling capabilities.
  - Cost efficiency.

System Testing Automation

- Simulate user interactions through browser automation.
- Use tools like **Selenium** for automated testing.
- Ensure testing covers databases, persistent connections, and full system workflows.

User Acceptance Testing (UAT)

- Deploy the final system for testing by a limited group of users.
- Conduct "Beta" testing to identify any pre-production issues.
    - **Beta software**: A pre-release version tested by real users in a controlled environment.

Test Generation

1. **API-Based Testing**
    - Use API definitions (e.g., OpenAPI, Swagger) to generate test cases.
    - Focus on endpoint-specific scenarios, data validation, and potential problem cases.
2. **Model-Based Testing**
    - Define system states and transitions (e.g., user login/logout).
    - Generate tests for each state transition using abstract models.
3. **GUI Testing**
    - Validate the presence of elements, navigation links, and random clicks.
    - Browser automation (e.g., Selenium) handles scenarios requiring user interaction.
4. **Security Testing**
    - Employ techniques like fuzz testing to inject invalid or random inputs.
    - Test server robustness against overloads, injections, and crashes.

Lec 3 & 4 :Testing

Pytest Framework

1. **Features and Setup**
    - Simplifies Python testing with setup/teardown environments and fixtures.
    - Supports exception handling, temporary directories, and more.

Basic Example

```
def func(x):
    return x + 1

def test_answer():

    assert func(3) == 5
```

2. **Fixtures**
   - Reusable components to initialize and clean up test environments.

```
@pytest.fixture

def setup_list():

    return ["apple", "banana"]
```

3. **Conventions**
   - Test discovery rules:
     - Files: `test_*.py` or `*_test.py`
     - Functions: Prefix `test_`
     - Classes: Prefix `Test` without `__init__`

Testing Flask Applications

1. **Fixture Setup**
   - Use Flask's test client and a temporary database.

```
@pytest.fixture

def client():

    app = create_app({'TESTING': True})

    with app.test_client() as client:

        yield client
```

Example Test

```
def test_empty_db(client):

    rv = client.get('/')

    assert b'No entries here so far' in rv.data
```

Login/Logout

```
def test_login_logout(client):

    rv = login(client, "username", "password")

    assert b'You were logged in' in rv.data
```

Evaluation

- Test file existence (e.g., HTML files for a web project).

Example:

```
def test_public_case1(self, student_assignment_folder):

    file_path = student_assignment_folder + "contact.html"

         assert os.path.isfile(file_path) == True
```
