Unit Test Report
Trần Nhật Huy (22127164)

1 Overview and Objectives of Unit Testing

Unit testing is a critical part of the software testing process, ensuring that each component (module, function, class) works as designed. The main objectives of unit testing are:

- **Ensure correctness of business logic:** Verify that each code unit behaves as expected and returns the correct results for given inputs.
- **Detect errors early:** Testing individual parts helps identify and fix errors during development, reducing bug-fixing costs and preventing error propagation.
- **Support maintainability and scalability:** A comprehensive unit test suite allows for safe refactoring or extending functionality, ensuring existing features remain unaffected.

In the Student Management application, unit tests focus on critical components to ensure that functionalities like adding, deleting, updating, and searching for students work correctly.

2 Understanding Unit Test Coverage and Best Practices

Unit testing is an essential tool for verifying the correctness of the smallest units of code, ensuring business logic behaves as expected, catching errors early, and supporting maintenance and scalability. Below is a summary of important aspects of code coverage and best practices in writing unit tests, along with illustrative examples.

2.1 Important Types of Code Coverage

- **Line Coverage:** Checks the percentage of code lines executed during tests. The goal is to ensure that most lines of code are executed.
- **Branch Coverage:** Ensures that all conditional branches (e.g., `if`, `else`, `switch`) are tested, which helps detect logical errors in decision-making.
- **Function Coverage:** Verifies that every function or method is called at least once during testing.
- **Path Coverage:** Tests different logical paths in the program, ensuring that special cases and boundary conditions are covered.

2.2 Illustrative Example

Consider a function that calculates a discount based on order amount and membership status:

```
def calculate_discount(order_amount, is_member):
    if order_amount > 100:            # Check branch when order_amount > 100
        discount = 10                 # Set initial discount
        if is_member:                 # Check branch for member
            discount += 5             # Increase discount for members
        else:                         # Branch for non-members
            discount += 2             # Increase discount for non-members
    else:
        discount = 0                  # No discount if order_amount <= 100
    return discount                   # Return the discount value
```

The test cases below ensure:

- **Line Coverage:** Every line is executed across different test scenarios.
- **Branch Coverage:** Both the `if order_amount > 100` branch and its nested branches are tested.
- **Function Coverage:** The `calculate_discount` function is invoked in every test.
- **Path Coverage:** Covers cases when `order <= 100`, `order > 100 with membership`, and `order > 100 without membership`.

Example unit tests:

```
import unittest

class TestCalculateDiscount(unittest.TestCase):
    def test_discount_for_member_with_high_order(self):
        result = calculate_discount(150, True)
        self.assertEqual(result, 15, "Discount␣for␣member␣with␣order␣>␣100␣
            ↪ should␣be␣15")

    def test_discount_for_non_member_with_high_order(self):
        result = calculate_discount(150, False)
        self.assertEqual(result, 12, "Discount␣for␣non-member␣with␣order␣>␣100␣
            ↪ should␣be␣12")

    def test_no_discount_for_low_order(self):
        self.assertEqual(calculate_discount(50, True), 0, "No␣discount␣expected
            ↪ ␣for␣order␣<=␣100")
        self.assertEqual(calculate_discount(50, False), 0, "No␣discount␣
            ↪ expected␣for␣order␣<=␣100")

    def test_edge_case_order_equal_to_100(self):
        self.assertEqual(calculate_discount(100, True), 0, "No␣discount␣
            ↪ expected␣for␣order␣=␣100")
        self.assertEqual(calculate_discount(100, False), 0, "No␣discount␣
            ↪ expected␣for␣order␣=␣100")

if __name__ == '__main__':
```

```
    unittest.main()
```

2.3 Minimum Coverage Requirements

- **Not Necessarily 100%:** The required coverage level does not need to be 100%, but it should be sufficient to test critical application logic.
- **Industry Practice:** Many systems achieve around 70-80% line coverage; more complex functionalities may require higher coverage.

2.4 Best Practices in Writing Unit Tests

1. **Write Maintainable Tests:**
   - Design tests to be independent, avoiding reliance on the database or persistent storage.
   - Use mocks or stubs to simulate external dependencies, which speeds up tests and simplifies error detection.
2. **Test Both Happy and Edge Cases:**
   - **Happy Case:** Verify that the system behaves as expected with valid input.
   - **Edge Case:** Test for unusual inputs, invalid data, exceptions, or boundary conditions to ensure robustness.
3. **Avoid Duplicate Tests and Dependence on Implementation Details:**
   - Focus on testing the outcome and behavior rather than internal implementation, reducing the need to modify tests when the code structure changes.
4. **Use Dependency Injection:**
   - Inject external dependencies (via constructor, setters, etc.) to facilitate mocking and reduce coupling between modules, increasing testability.

2.4.1 Example Using Mocks

Consider a function that retrieves a student's discount from the database:

```
def get_student_discount(student_id, database):
    student = database.get_student_by_id(student_id)
    return calculate_discount(student['order_amount'], student['is_member'])
```

Unit test using mocks:

```
import unittest
from unittest.mock import MagicMock

class TestGetStudentDiscount(unittest.TestCase):
    def test_member_discount(self):
```

```
        mock_db = MagicMock()
        mock_db.get_student_by_id.return_value = {'is_member': True, '
            ↪ order_amount': 200}
        result = get_student_discount(123, mock_db)
        self.assertEqual(result, 15, "Discount␣for␣member␣with␣order␣200␣should
            ↪ ␣be␣15")

    def test_non_member_discount(self):
        mock_db = MagicMock()
        mock_db.get_student_by_id.return_value = {'is_member': False, '
            ↪ order_amount': 200}
        result = get_student_discount(124, mock_db)
        self.assertEqual(result, 12, "Discount␣for␣non-member␣with␣order␣200␣
            ↪ should␣be␣12")

if __name__ == '__main__':
    unittest.main()
```

3 Writing Unit Tests for the Application

To ensure the application functions reliably and meets business requirements, the unit tests for the Student Management application should focus on core components and key functionalities.

3.1 Scope of Unit Testing

- **Focus on Critical Components:**
  - **Business Logic:** Functions or classes handling core operations such as calculations, data validation, and information processing.
  - **Key Functionalities:**
    - **Adding a New Student:** Validate input, verify data, and store information.
    - **Deleting a Student by ID:** Ensure that deleting by student ID correctly updates the dataset.
    - **Updating Student Information:** Verify that updates (e.g., address, email, phone number) are processed accurately.
    - **Searching for Students:** Test the search functionality based on name or ID, ensuring correct results even with invalid inputs or multiple matches.
- **Testing Objectives:**
  - **Accuracy:** Every function must be verified to produce the expected output.
  - **Early Error Detection:** Identify and resolve errors during the unit testing phase to minimize deployment risks.
  - **Maintainability:** Independent and well-structured unit tests simplify future maintenance and feature expansion.

3.2 Test Strategy

3.2.1 a. Separating Application Layers

To ensure clarity and independence in unit tests, the application layers should be separated as follows:

- **Controller:**
  - **Role:** Receive requests from users and return corresponding responses.
  - **Testing Strategy:**
    - Verify that the controller correctly formats requests and responses.
    - Minimize business logic within the controller to focus on routing and inter-layer communication.
- **Repository:**
  - **Role:** Handle data access operations (add, delete, update, query).
  - **Testing Strategy:**
    - Avoid direct dependency on the real database by using mocks for I/O operations.
    - Test data processing logic and exception handling during data access.
- **Service (if applicable):**
  - **Role:** Encapsulate the core business logic.
  - **Testing Strategy:**
    - Write independent unit tests for service classes to ensure correct business logic.
    - Test both successful scenarios (happy case) and abnormal situations (edge case).

3.2.2 b. Using a Mocking Framework

An essential aspect of effective unit testing is the ability to simulate (mock) external dependencies:

- **Objectives:**
  - Decouple dependencies such as databases, external APIs, or services not directly related to business logic.
  - Ensure that tests focus solely on the module under test without interference from external systems.
- **Implementation:**
  - Use a mocking library (e.g., `unittest.mock` in Python) to create dummy objects.
  - For example, when testing a discount calculation based on student information, use a mock to return sample data.

```
import unittest
from unittest.mock import MagicMock

def get_student_discount(student_id, database):
    student = database.get_student_by_id(student_id)
    return calculate_discount(student['order_amount'], student['is_member'])

class TestGetStudentDiscount(unittest.TestCase):
    def test_member_discount(self):
        mock_db = MagicMock()
        mock_db.get_student_by_id.return_value = {'is_member': True, '
            ↪ order_amount': 200}
        result = get_student_discount(123, mock_db)
        self.assertEqual(result, 15, "Discount␣for␣member␣with␣order␣200␣should
            ↪ ␣be␣15")

    def test_non_member_discount(self):
        mock_db = MagicMock()
        mock_db.get_student_by_id.return_value = {'is_member': False, '
            ↪ order_amount': 200}
        result = get_student_discount(124, mock_db)
        self.assertEqual(result, 12, "Discount␣for␣non-member␣with␣order␣200␣
            ↪ should␣be␣12")

if __name__ == '__main__':
    unittest.main()
```

4 Evaluating and Improving Design (If Needed)

During the development and testing of the Student Management application, certain design issues and dependencies can impact the effectiveness of unit tests. Below is an analysis of potential issues and proposed improvements to enhance testability and maintainability.

4.1 Issues to Evaluate

a. Overdependence on Database/Persistent Storage

Issue: When modules directly depend on the database or storage systems, testing becomes slow and difficult to control. Errors from the database layer may affect the outcomes of business logic tests.

Analysis:

- I/O operations can slow down tests, especially with large datasets or unstable testing environments.
- Setting up data for each test becomes complex and may lead to inconsistent results.

b. Modules with Excessive Responsibilities

Issue: A module that handles multiple functions and contains extensive business logic is hard to test since tests must cover too many aspects.

Analysis:

- For example, a controller that processes requests, handles business logic, and interacts with the database is difficult to test in isolation.
- Tests become dependent on implementation details, meaning any structural changes require test modifications.

c. Inability to Isolate Critical Business Logic

Issue: If the core business logic is entangled with other code (such as UI handling or data processing from the database), it becomes difficult to isolate and test the critical logic.

Analysis:

- Some modules lack a separate layer for business logic, resulting in tests that only assess the aggregated code rather than targeting specific algorithms or rules.
- This leads to gaps in detecting errors, particularly in edge cases or complex logic.

4.2 Proposed Improvements

a. Separate Business Logic

Solution: Refactor the application by moving business logic from complex modules (e.g., controllers) to dedicated service classes.

Benefits:

- Each class has a single responsibility, allowing independent testing of business logic.
- Promotes reusability and maintainability.

Example:

Instead of embedding logic directly in the controller:

```
def update_student(request):
    data = request.get_json()
    if validate_data(data):
        # Complex business logic
        student = StudentService.update_student(data)
        return jsonify(student), 200
    else:
        return jsonify({"error": "Invalid␣data"}), 400
```

Refactor to a dedicated service:

```
class StudentService:
    @staticmethod
    def update_student(data):
        # Separated business logic, easier to test
        if data['score'] < 0:
            raise ValueError("Score must be non-negative")
        return {"id": data["id"], "status": "updated"}
```

b. Apply Dependency Injection

Solution: Use Dependency Injection (DI) to provide external dependencies (e.g., repositories, database connections) to classes or functions rather than instantiating them internally.

Benefits:

- Reduces coupling between modules, facilitating the use of mocks or stubs during testing.
- Ensures tests focus solely on module logic without interference from external systems.

Example:

Before DI:

```
class StudentRepository:
    def __init__(self):
        self.db = DatabaseConnection()  # Hard to test due to external
            ↪ dependency
```

After applying DI:

```
class StudentRepository:
    def __init__(self, db):
        self.db = db  # Dependency injected from outside
```

Testing with a mock:

```
def test_get_student():
    mock_db = MagicMock()
    mock_db.get.return_value = {"id": 1, "name": "John Doe"}
    repo = StudentRepository(mock_db)
    student = repo.get_student(1)
    assert student["name"] == "John Doe"
```

c. Test for Edge Cases

Solution: Expand test coverage to include abnormal scenarios, invalid data, and exceptions.

Benefits:

- Ensures the application handles both typical and unexpected situations robustly.

Example:

```
def test_update_student_with_negative_score():
    data = {"id": 1, "score": -10}
    with self.assertRaises(ValueError):
        StudentService.update_student(data)
```

d. Reassess Code Coverage Levels

Solution: Use code coverage tools (e.g., Coverage.py for Python, JaCoCo for Java) to monitor test coverage.

Benefits:

- Identify critical code areas that lack adequate tests.
- Focus on adding tests for high-risk or complex logic.

Practice:

- Run coverage reports after tests and evaluate if the coverage meets requirements.
- Adjust the test suite based on coverage data, especially for low-coverage or complex sections.

5 Conclusion

Writing unit tests for the Student Management application not only ensures software quality but also facilitates future maintenance and scalability. In summary:

- **Code Coverage:** Ensure that key coverage metrics (line, branch, function, and path) are addressed.
- **Best Practices:** Write maintainable tests, cover both happy and edge cases, and avoid reliance on implementation details.
- **Design Improvements:** When tests are challenging, reassess architecture by separating business logic, applying dependency injection, and utilizing mocking tools.

This report provides an overview and improvement recommendations to guide effective development and testing, ensuring correctness and maintainability of the project.