CISC/CMPE 327, Fall 2025

Assignment 2: Library Management System - Extended Unit Testing & CI/CD

Name: Lore Chiepe

Student ID: 20343672

GitHub Repository: [https://github.com/Lchiepe/cisc327-library-management-a2-3672]

1. Complete Function Implementation

1.1 Implementation Status

Based on my Assignment 1 report, I have successfully completed all remaining function implementations. The implementation experience was both challenging and educational, particularly in understanding how to integrate database operations with business logic while maintaining proper error handling and validation.

1.2 Functions Implemented

The following four functions were implemented in library_service.py to complete the Library Management System:

R4: Book Return Processing - return_book_by_patron() with full validation and database updates, including patron verification and availability management

R5: Late Fee Calculation API - calculate_late_fee_for_book() implementing the specified fee structure: $0.50/day for first 7 days overdue, $1.00/day for each additional day, with a maximum cap of $15.00 per book

R6: Book Search Functionality - search_books_in_catalog() supporting comprehensive search by title, author, and ISBN with case-insensitive matching

R7: Patron Status Report - get_patron_status_report() providing comprehensive borrowing information including current loans, overdue status, and fee calculations

1.3 Implementation Experience

The implementation required careful coordination between the business logic layer and database operations, ensuring data consistency and proper error handling throughout. All functions were tested locally and successfully integrated with the existing Flask web interface and API routes. The system now provides complete library management functionality as specified in the requirements.

2. Comprehensive Test Suite Development

2.1 Test Strategy

According to my previous implementation status, R1 to R3 were fully functional and tested in Assignment 1. Following the implementation of the remaining functions (R4-R7), I created comprehensive test cases to ensure code quality and functionality.

2.2 Additional Test Cases Added

For R4 - Book Return Processing:

test_return_book_success() - Tests successful book return scenario with proper database updates

test_return_book_invalid_patron() - Tests return process with invalid patron ID format

test_return_book_not_found() - Tests return of non-existent book with appropriate error handling

For R5 - Late Fee Calculation:

test_late_fee_basic() - Tests basic function structure and return format validation

For R6 - Book Search Functionality:

test_search_by_title() - Tests title-based search with partial matching

test_search_by_author() - Tests author-based search functionality

test_search_no_results() - Tests search behavior with no matching results

test_search_empty_term() - Tests search with empty input handling

For R7 - Patron Status Report:

test_patron_status_basic() - Tests basic report structure and required fields

test_patron_status_invalid_id() - Tests report generation with invalid patron ID

2.3 Test Coverage

Total Test Coverage: 21 test cases covering all major functionality with both positive and negative test scenarios, achieving comprehensive validation of the Library Management System.

3. AI-Assisted Test Generation

3.1 AI Tool Selection

Tool Used: ChatGPT

Rationale: Selected for its comprehensive code generation capabilities and ability to understand complex testing requirements for Python applications.

3.2 Prompt Engineering

Initial Prompt:

“Generate comprehensive pytest unit tests for a Library Management System with the following Python functions:

1. return_book_by_patron(patron_id, book_id) - Processes book returns with validation

2. calculate_late_fee_for_book(patron_id, book_id) - Calculates overdue fees with structure: $0.50/day for first 7 days, $1.00/day after, maximum $15.00

3. search_books_in_catalog(search_term, search_type) - Searches books by title/author/ISBN

4. get_patron_status_report(patron_id) - Generates patron borrowing report

Please create pytest test cases covering normal scenarios, edge cases, error conditions, and boundary conditions. Include both valid and invalid inputs for patron IDs, book IDs, and search terms. Use pytest monkeypatch for mocking database calls.”

Follow-up Prompts:

"Can you add more edge cases for the late fee calculation, specifically testing the fee structure boundaries?"

"Generate negative test cases for invalid ISBNs and patron IDs"

"Create integration tests that test multiple functions working together"

"Add test cases for database error scenarios"

3.3 AI-Generated Test Cases

python

```python
# AI generated test for late fee calculation

def test_late_fee_calculation_boundaries(monkeypatch):

    """Test late fee calculation at boundary points."""
    # Test exactly 7 days overdue
    mock_borrowed_books = [{
        'book_id': 1,
        'due_date': datetime.now() - timedelta(days=7)
    }]
    monkeypatch.setattr("database.get_patron_borrowed_books", lambda pid: mock_borrowed_books)

    result = calculate_late_fee_for_book("123456", 1)
    assert result['fee_amount'] == 3.50  # 7 * 0.50

    # Test 8 days overdue (crossing the boundary)
    mock_borrowed_books[0]['due_date'] = datetime.now() - timedelta(days=8)
    result = calculate_late_fee_for_book("123456", 1)
    assert result['fee_amount'] == 4.50  # 7*0.50 + 1*1.00

# AI generated integration test
def test_patron_borrow_return_workflow(monkeypatch):
    """Test complete borrow-return workflow for a patron."""
    # Mock initial state
    monkeypatch.setattr("database.get_book_by_id", lambda bid: {"id": 1, "title": "Test Book", 
"available_copies": 3})
    monkeypatch.setattr("database.get_patron_borrow_count", lambda pid: 0)
    monkeypatch.setattr("database.insert_borrow_record", lambda *args: True)
    monkeypatch.setattr("database.update_book_availability", lambda *args: True)

    # Borrow a book
    success, message = borrow_book_by_patron("123456", 1)
    assert success is True
    # Now return it
    monkeypatch.setattr("database.get_patron_borrowed_books", lambda pid: [{"book_id": 1}])
    monkeypatch.setattr("database.update_borrow_record_return_date", lambda *args: True)

    success, message = return_book_by_patron("123456", 1)
    assert success is True
```

4. Test-Case Comparison & Analysis

4.1 Comparative Analysis

Looking at both testing approaches, I noticed some clear differences. My own tests focused mainly on making sure the basic functions worked correctly. I tested obvious cases like valid inputs, simple errors, and whether functions returned the right format.

The AI-generated tests were more thorough in some ways. They caught edge cases I hadn't thought about, like testing exactly when late fees change from $0.50 to $1.00 per day. The AI also created integration tests that checked how multiple functions work together, which was smarter than my separate unit tests.

However, the AI tests weren't perfect. Some were too complicated with lots of mocking, making them hard to understand. Others didn't match our actual database structure, so I had to adjust them to work properly.

Overall, both approaches had value. My tests were practical and worked right away, while the AI tests helped me see gaps in my testing strategy. The best approach seems to be using AI for ideas but applying human judgment to make tests useful.

4.3 Conclusion

While my human-written tests were more practical and immediately usable within the current system architecture, the AI-generated tests provided valuable insights into test design philosophy and revealed significant coverage gaps. The ideal approach is a hybrid methodology - leveraging AI for test idea generation, boundary case identification, and coverage analysis, while relying on human judgment for practical implementation, domain-specific adaptations, and integration with existing systems.

5. CI/CD Pipeline Setup

5.1 Implementation Notes

The CI/CD pipeline successfully validates that the Library Management System functions correctly in a clean environment, providing assurance that the implementation is portable and not dependent on local development environment configurations.

```
name: Run Python Tests

on: [push, pull_request]

jobs:
  test:
    runs-on: ubuntu-latest

    steps:
    - name: Checkout code
      uses: actions/checkout@v4

    - name: Set up Python
      uses: actions/setup-python@v4
      with:
        python-version: '3.12'

    - name: Install dependencies
      run: |
        pip install flask pytest
        pip install -r requirements.txt

    - name: Run tests
      run: |
        python -m pytest test/ -v
```

6. Learning Outcomes & Reflection

This assignment provided comprehensive experience in several critical areas of modern software development:

Complete Software Development Lifecycle: Progressing from requirements analysis to implementation, testing, and deployment

Test-Driven Development Principles: Creating and maintaining comprehensive test suites for both existing and new functionality

AI-Assisted Development: Effectively leveraging LLMs for test generation while critically evaluating their output

CI/CD Pipeline Configuration: Setting up automated testing with GitHub Actions for continuous integration

Quality Assurance: Understanding the importance of test coverage, automated verification, and code quality metrics

The integration of AI tools for test generation was particularly insightful, demonstrating both the remarkable capabilities and current limitations of AI systems in software testing. While AI can generate comprehensive test ideas and identify coverage gaps, human oversight remains crucial for practical implementation, domain-specific adaptations, and integration with existing systems.

The project successfully demonstrates a complete Library Management System with robust testing practices and continuous integration capabilities, meeting all specified requirements and providing a solid foundation for future enhancements.