Assignment 3 – Mocking, Stubbing, and Code Coverage Testing

CISC/CMPE 327

Section 1 – Student Information

Name: Aiden Ramezani
Student ID: 20259781
Submission Date: November 10 2025
GitHub Repo: Aidenrmz/cisc327-library-management-a2-9781

Section 2 – Stubbing vs Mocking Explanation

In this assignment I applied both **stubbing** and **mocking** to isolate and test payment-related logic without relying on real databases or network calls.
A *stub* is a simple fake implementation that only returns controlled data; it does not verify behavior. I stubbed the database functions
calculate_late_fee_for_book() and get_book_by_id() using mocker.patch() to provide predetermined fee amounts and book details. These stubs allowed me to test different scenarios—such as valid late fees, zero fees, and invalid patrons—without connecting to any real data layer.

A *mock* simulates an external component **and** records how it was used so its interactions can be verified. I mocked the PaymentGateway class from payment_service.py, particularly the process_payment() and refund_payment() methods.
Using Mock(spec=PaymentGateway), I configured specific return values or raised exceptions, then verified calls through assert_called_once_with() and assert_not_called().
This approach confirmed that pay_late_fees() and refund_late_fee_payment() passed the right parameters to the gateway and handled every success, failure, and exception path correctly.
Together, stubs and mocks achieved complete isolation of the business logic and provided deterministic, reproducible tests that raised project coverage beyond 80%.

Section 3 – Test Execution Instructions

Environment setup

pip install pytest pytest-mock pytest-cov flask requests

Run all tests

pytest tests/

Run tests with coverage

pytest --cov=services --cov-report=html --cov-report=term tests/

- Coverage HTML report → htmlcov/index.html
  (open in a browser to see green = covered, red = uncovered)

Optional focused coverage

pytest --cov=services.library_service --cov-report=term tests/

Section 4 – Test Cases Summary for New Tests

Test Function Name	Purpose	Stubs Used	Mocks Used	Verification
test_pay_late_fees_successful_payment	Successful payment flow	calculate_late_fee_for_book, get_book_by_id	process_payment	assert_called_once_with
test_pay_late_fees_invalid_patron_does_not_call_gateway	Reject invalid ID	None	process_payment	assert_not_called
test_pay_late_fees_payment_declined_by_gateway	Handle declined payment	same as above	process_payment	assert_called_once_with
test_pay_late_fees_zero_fee_does_not_call_gateway	Skip gateway for 0 fee	calculate_late_fee_for_book	process_payment	assert_not_called
test_pay_late_fees_handles_network_error_from_gateway	Handle raised exception	both DB stubs	process_payment	assert_called_once_with
test_refund_late_fee_payment_successful_refund	Successful refund	None	refund_payment	assert_called_once_with
test_refund_late_fee_payment_invalid_transaction_id_does_not_call_gateway	Reject invalid txn ID	None	refund_payment	assert_not_called
test_refund_late_fee_payment_invalid_amounts_do_not_call_gateway	Reject bad amounts	None	refund_payment	assert_not_called
test_payment_gateway_process_payment_success	Cover Payment Gateway happy path	N/A	N/A (direct call)	Result assertions on return values

test_payment_gateway_refund_invalid_transaction	Cover gateway validation branch	N/A	N/A	Result assertions on return values

---

Section 5 – Coverage Analysis

After executing

pytest --cov=services --cov-report=html --cov-report=term tests/

the coverage results were:

File	Statements	Missed	Coverage
services/library_service.py	142	14	**90 %**
services/payment_service.py	29	11	**62 %**
TOTAL	171	25	**85 %**

Initially, total coverage was ≈79 %.
By adding test_payment_service_direct.py to exercise the main branches of PaymentGateway, coverage rose to **85 %**, exceeding the required 80 %.
Uncovered lines in payment_service.py mainly belong to rare error branches and the verify_payment_status() method, which simulates an external HTTP API. These are intentionally untested, since such integration points should be mocked rather than executed in unit tests.
All decision branches in pay_late_fees() and refund_late_fee_payment() (invalid inputs, zero fees, gateway declines, exceptions) are covered, giving **90 % statement and complete branch coverage** for core business logic.
The HTML report (htmlcov/index.html) visually confirmed green coverage on both functions and red only in external-API stubs.

Section 6 – Challenges and Solutions

1. **Import errors after restructuring** – Moving library_service.py into services/ broke legacy imports. I solved it with a root-level shim file that re-exports from services.library_service import *, restoring compatibility for older tests.
2. **Missing dependencies** – Initial runs failed on flask, requests, and pytest-mock; installing them resolved environment issues.
3. **Mock verification errors** – At first I forgot to verify mock calls. Adding assert_called_once_with() and assert_not_called() ensured tests validated interactions, not just outputs.
4. **Coverage below 80 %** – Because PaymentGateway is mocked, its code was uncovered. Adding two simple direct tests for the gateway's happy and error paths pushed total coverage to 85 %.

Through these fixes I gained a clear understanding of how mocking/stubbing improves test isolation and coverage quality.

Section 7 – Screenshots