CISC 327 Assignment 3

Radmehr Vafadar | 20421691 | 2025, November 9th

2) Stubbing vs Mocking Explanation

In this project I treated stubbing and mocking as two different kinds of test doubles. A stub is a fake implementation that simply returns hard coded data so I can drive a specific code path, without caring how the function was called. A mock is a fake object that not only returns values but also lets me verify interactions, such as how many times a method was called and with which arguments.

For the late fee logic I mostly used stubs. I stubbed get_book_by_id and get_borrow_record so they returned deterministic “database” rows and let me focus on the rules inside calculate_late_fee_for_book, such as different overdue windows and the 15 dollar cap. I also stubbed calculate_late_fee_for_book itself when testing higher level functions like pay_late_fees, so those tests did not depend on the exact fee formula.

For the payment layer I used mocks. The PaymentGateway represents an external service, so I created Mock(spec=PaymentGateway) instances and asserted that process_payment and refund_payment were called or not called with the right parameters, and that exceptions were handled correctly. These interaction checks are what make them mocks rather than simple stubs. I complemented those tests with a separate suite that runs the real PaymentGateway methods against many valid and invalid inputs to increase coverage and exercise boundary cases like amount limits and ID formats.

3) Test Execution Instructions

1. Open PowerShell/Terminal.
2. (Optional) Go to your preferred directory using cd.
3. Run:
   git clone https://github.com/RadmehrVafadar/cisc327-library-management-20421691.git
4. Go to the repo:
   cd ./cisc327-library-management-20421691
5. Install coverage plugin:
   pip install pytest-cov
6. Run tests with coverage:
   pytest --cov=services --cov-report=html --cov-report=term tests/

7. View the HTML Coverege report:
   start htmlcov/index.html

4) Test Cases Summary for the New Tests

File: test_payment_service_coverage.py

Test Function Name	Purpose	Stubs Used	Mocks Used	Verification Done
test_payment_gateway_initialization	Check default PaymentGateway is created with base config.	None	None	Assert default fields (API key, limits, timeouts) match expected defaults.
test_payment_gateway_initialization_custom_key	Check a custom API key is accepted and stored correctly.	None	None	Assert that the stored API key matches the custom key.
		process_payment()		
test_process_payment_successful	Normal successful payment with valid patron id and amount.	None	None	Assert success=True, transaction_id is non-empty, and message indicates success.
test_process_payment_invalid_amount_zero	Ensure zero amount is rejected.	None	None	Assert success=False and error message about invalid or zero amount.
test_process_payment_invalid_amount_negative	Ensure negative amounts are rejected.	None	None	Assert success=False and error message about negative amount.
test_process_payment_amount_exceeds_limit	Reject payments that exceed the maximum allowed amount.	None	None	Assert success=False and message mentioning the configured limit.
test_process_payment_invalid_patron_id_too_short	Validate that a too short patron ID is rejected.	None	None	Assert success=False and message about patron ID being too short or invalid.
test_process_payment_invalid_patron_id_too_long	Validate that a too long patron ID is rejected.	None	None	Assert success=False and message about patron ID being too long or invalid.
test_process_payment_with_description	Check that the optional description is handled with the payment.	None	None	Assert success=True and that the description is reflected in the result or message.
test_process_payment_transaction_id_uniqueness	Make sure multiple payments get different transaction ids.	None	None	Call process_payment several times and assert all transaction_ids are unique.
test_process_payment_boundary_amount_one_cent	Lower boundary test for the minimum positive amount (e.g. 0.01).	None	None	Assert success=True and message includes the formatted amount such as $0.01.
test_process_payment_boundary_amount_max_allowed	Upper boundary test at the maximum allowed amount.	None	None	Assert success=True and message confirms payment at the max limit.
		refund_payment()		

test_refund_payment_successful	Normal successful refund for a valid transaction and amount.	None	None	Assert success=True and confirmation message for the refund.
test_refund_payment_invalid_transaction_id_format	Reject refunds when the transaction id format is invalid.	None	None	Assert success=False and message about invalid transaction ID format.
test_refund_payment_empty_transaction_id	Ensure an empty transaction id is rejected.	None	None	Assert success=False and message about missing or empty transaction ID.
test_refund_payment_invalid_amount_zero	Reject a refund with amount equal to zero.	None	None	Assert success=False and message about invalid or zero refund amount.
test_refund_payment_invalid_amount_negative	Reject negative refund amounts.	None	None	Assert success=False and message about negative refund amount.
test_refund_payment_valid_transaction_id_format	Check that a correctly formatted transaction id passes basic validation.	None	None	Assert that the result indicates the ID format is acceptable (no format error).
test_refund_payment_boundary_amount	Boundary test for the minimum positive refund amount (e.g. 0.01).	None	None	Assert success=True and that the amount is shown with correct formatting.
		verify_payment_status()		
test_verify_payment_status_valid_transaction	Check status lookup for a valid transaction id.	None	None	Assert result shows a completed (or similar) status and the transaction ID matches.
test_verify_payment_status_invalid_transaction_id_format	Ensure invalid transaction id format is rejected for status.	None	None	Assert failure result and message about invalid transaction ID format.
test_verify_payment_status_empty_transaction_id	Ensure empty transaction id is handled as invalid for status.	None	None	Assert failure result and message about missing transaction ID.

File: test_services_mocking_stubbing.py

Test Function Name	**Purpose**	**Stubs Used**	**Mocks Used**	**Verification Done**
		TestCalculateLateFeeWithStubs		
test_calculate_late_fee_invalid_patron_id	Reject empty patron ID.	get_book_by_id, get_borrow_record	None	Assert error status and that stubs are not called.
test_calculate_late_fee_invalid_patron_id_too_short	Reject too short patron ID.	None	None	Assert error status without calling any helper functions.
test_calculate_late_fee_invalid_patron_id_non_numeric	Reject non-numeric patron ID.	None	None	Assert error status without calling any helper functions.
test_calculate_late_fee_book_not_found	Handle book-not-found case.	get_book_by_id	None	Assert status "Book not found." and fee_amount=0.
test_calculate_late_fee_book_not_borrowed_by_patron	Handle missing borrow record for patron.	get_book_by_id, get_borrow_record	None	Assert status indicates book not borrowed by patron and fee_amount=0.

test_calculate_late_fee_book_not_overdue	No fee if book is not overdue.	get_book_by_id, get_borrow_record	None	Assert fee_amount=0 and status says book is not overdue.
test_calculate_late_fee_3_days_overdue	Fee calculation for 3 days overdue.	get_book_by_id, get_borrow_record	None	Assert fee equals 3 days at the base rate and status is success.
test_calculate_late_fee_7_days_overdue	Boundary test at 7 days overdue.	get_book_by_id, get_borrow_record	None	Assert fee for 7 days at the base rate and status is success.
test_calculate_late_fee_10_days_overdue	Fee beyond 7 days at higher rate.	get_book_by_id, get_borrow_record	None	Assert fee uses the higher rate after day 7 and status is success.
test_calculate_late_fee_maximum_cap_at_15	Cap fee at $15 maximum.	get_book_by_id, get_borrow_record	None	Assert fee_amount is capped at 15 and status is success.
test_calculate_late_fee_1_day_overdue	Single-day overdue fee.	get_book_by_id, get_borrow_record	None	Assert fee for 1 day at the base rate and status is success.
		TestGetBookByIdWithStubs		
test_get_book_by_id_book_exists	Book is found in the stubbed DB.	get_book_by_id	None	Assert returned book matches stub data and stub called once with correct ID.
test_get_book_by_id_book_not_found	Book ID not found in stub.	get_book_by_id	None	Assert function returns None and stub called once with given ID.
test_get_book_by_id_with_zero_available_copies	Handle book with zero available copies.	get_book_by_id	None	Assert returned book has available_copies=0 and stub called as expected.

		TestPayLateFeesWithMocks		
test_pay_late_fees_successful_payment	Successful late fee payment flow.	get_book_by_id, get_borrow_record, calculate_late_fee_for_book	PaymentGateway.process_payment	Assert process_payment called once with correct patron, amount and description; result indicates success.
test_pay_late_fees_payment_declined	Handle declined late fee payment.	get_book_by_id, get_borrow_record, calculate_late_fee_for_book	PaymentGateway.process_payment	Assert process_payment called once and result indicates payment was declined.
test_pay_late_fees_invalid_patron_id_mock_not_called	Skip payment when patron ID is invalid.	None	PaymentGateway.process_payment	Assert_not_called on process_payment and error status returned.
test_pay_late_fees_zero_fees_mock_not_called	Skip payment when total fee is zero.	calculate_late_fee_for_book	PaymentGateway.process_payment	Assert_not_called on process_payment and message about no late fees due.
test_pay_late_fees_network_error_exception	Handle network exception from payment gateway.	get_book_by_id, get_borrow_record, calculate_late_fee_for_book	PaymentGateway.process_payment	Assert process_payment called once, exception is handled, and error status is returned.

test_pay_late_fees_book_not_found_mock_not_called	Skip payment when book is not found.	get_book_by_id	PaymentGateway.process_payment	Assert_not_called on process_payment and status "Book not found." returned.
test_pay_late_fees_with_multiple_patrons	Process late fees for multiple patrons.	get_book_by_id, get_borrow_record, calculate_late_fee_for_book	PaymentGateway.process_payment	Assert process_payment called for each patron with the correct patron IDs and amounts.

TestRefundLateFeesWithMocks

test_refund_successful	Normal refund success.	None	PaymentGateway.refund_payment	Assert refund_payment called once with correct transaction ID and amount; success status returned.
test_refund_invalid_transaction_id_format	Invalid transaction ID format, no refund call.	None	PaymentGateway.refund_payment	Assert_not_called on refund_payment and validation error status returned.
test_refund_invalid_transaction_id_empty	Empty transaction ID, no refund call.	None	PaymentGateway.refund_payment	Assert_not_called on refund_payment and error about missing transaction ID.
test_refund_negative_amount	Negative refund amount is rejected.	None	PaymentGateway.refund_payment	Assert_not_called on refund_payment and error status for invalid negative amount.
test_refund_zero_amount	Zero refund amount is rejected.	None	PaymentGateway.refund_payment	Assert_not_called on refund_payment and error status for invalid zero amount.
test_refund_exceeds_maximum_late_fee	Refund exceeding maximum late fee is rejected.	None	PaymentGateway.refund_payment	Assert_not_called on refund_payment and error about exceeding maximum refundable amount.
test_refund_exactly_maximum_amount	Max boundary refund success.	None	PaymentGateway.refund_payment	Assert refund_payment called once with maximum allowed amount and success status.
test_refund_gateway_failure	Gateway returns failure on refund.	None	PaymentGateway.refund_payment	Assert refund_payment called once and failure status/message propagated.
test_refund_exception_handling	Exception during refund is handled.	None	PaymentGateway.refund_payment	Assert refund_payment called once and error status returned when exception occurs.
test_refund_valid_small_amount	Small valid refund succeeds.	None	PaymentGateway.refund_payment	Assert refund_payment called once with small amount and success status.
test_refund_with_decimal_precision	Check decimal precision for refund amount.	None	PaymentGateway.refund_payment	Assert refund_payment called once and message shows correctly formatted decimal amount.

TestPaymentWorkflowWithStubsAndMocks				
test_complete_payment_workflow	End-to-end payment then refund workflow.	get_book_by_id, get_borrow_record, calculate_late_fee_for_book	PaymentGateway.process_payment, PaymentGateway.refund_payment	Assert both mocks called once in the correct order with expected patron ID, amount and transaction ID.
test_refund_after_incorrect_payment	Partial refund after incorrect payment.	calculate_late_fee_for_book	PaymentGateway.refund_payment	Assert refund_payment called once with adjusted refund amount matching the difference from the incorrect payment.

5) Coverage Analysis

Before adding new tests, the coverage report showed that our test suite only exercised 57% of the statements in the project. At the file level, services/library_service.py had 63% statement coverage (155 statements, 58 missing), while services/payment_service.py was much weaker at 27% statement coverage (30 statements, 22 missing). In practice this meant that almost all of the real PaymentGateway logic was uncovered: the constructor that sets the API key and base URL, the full validation and success flow in process_payment, the validation logic in refund_payment, and the success/error cases in verify_payment_status. In library_service.py, a large portion of the uncovered code was in error-handling branches (simulated database failures) and in the more complex late-fee and payment-integration paths. Many of the if conditions in these areas had one or both branches completely untested, so branch coverage was also quite low overall.

To improve coverage, I focused on two areas: fully exercising PaymentGateway and adding targeted tests around important library behaviours. I added a dedicated test module for payment_service that instantiates the real PaymentGateway and calls process_payment with a range of inputs: valid patron IDs and amounts (happy path), zero and negative amounts (hitting the “invalid amount” branch), amounts above the maximum limit (declined payments), and patron IDs that are too short or too long (invalid ID format). I also added tests that include a description and check that a transaction ID is generated, so the full success path runs. For refund_payment, I wrote tests that cover valid refunds as well as invalid transaction IDs (empty and wrong prefix) and zero/negative refund amounts, forcing both the success and failure branches to execute. Finally, for verify_payment_status, I added tests for valid transaction IDs and for empty/malformed ones so that both the “not_found” and “completed” paths are covered. On the library side, I added tests for more detailed patron status scenarios and late-fee boundaries, so that the main late-fee calculation logic and some of the previously untested branches in library_service.py are now exercised.

After adding these tests, statement coverage increased from 57% to 89% overall. services/payment_service.py rose from 27% to 100% statement coverage, and services/library_service.py improved from 63% to 86%. From a branch perspective, payment_service.py now has 100% branch coverage: every conditional in process_payment, refund_payment, and verify_payment_status has both its true and false branches exercised. In library_service.py, most of the core decision points (patron ID validation, borrowing and returning rules, and late-fee boundaries) are now covered on both branches, giving an estimated branch coverage of around 82–85% for that file and roughly 88% branch coverage overall.

There are still some uncovered lines remaining, almost all of them in library_service.py. These are primarily defensive or error-handling paths that would require artificially forcing lower-level database failures (for example, branches that return "Database error occurred…" when inserts or updates fail), plus a few rare edge cases in the payment-integration helper functions and the more complex late-fee aggregation code in the patron status report. The uncovered branches either duplicate logic that is already tested in more focused functions, or represent low-probability failure scenarios that would significantly increase test complexity. Given this, the remaining uncovered lines are low risk, while the newly added tests ensure that all critical payment and library workflows now have high statement and branch coverage.

Note: Please ignore __init__.py I don't know how to get rid of it in the report other than removing the file. But I don't think it has much of an affect.

Beginning:

Coverage report: 57%

Files Functions Classes

coverage.py v7.11.3, created at 2025-11-10 17:48 -0500

File ▲	statements	missing	excluded	coverage
services__init__.py	0	0	0	100%
services\library_service.py	155	58	0	63%
services\payment_service.py	30	22	0	27%
Total	**185**	**80**	**0**	**57%**

coverage.py v7.11.3, created at 2025-11-10 17:48 -0500

Final:

Coverage report: 89%

Files Functions Classes

coverage.py v7.11.3, created at 2025-11-10 13:45 -0500

File ▲	statements	missing	excluded	coverage
services__init__.py	0	0	0	100%
services\library_service.py	155	21	0	86%
services\payment_service.py	30	0	0	100%
Total	**185**	**21**	**0**	**89%**

coverage.py v7.11.3, created at 2025-11-10 13:45 -0500

6) Challenges and Solutions

I ran into several problems that forced me to really understand how mocking, stubbing, and coverage tools actually work in practice. Below I describe the main challenges, how I solved them, and what I learned about designing effective tests.

Challenge 1 – Understanding the Difference Between Stubs and Mocks
My first challenge was understanding when to use a stub and when to use a mock. At the beginning I treated them as the same thing. Over time I realized that stubs are meant to supply fixed data to the unit under test, while mocks both return data and verify that certain calls were made.

I applied this separation directly in my tests. For database-style functions such as get_book_by_id() and get_borrow_record(), I used pytest-mock's mocker.patch() to create simple stubs that returned fake book data and borrow records. I did not verify how many times they were called, because all I cared about was feeding deterministic data into calculate_late_fee_for_book() so I could test the late-fee business logic.

In contrast, for the PaymentGateway class I used real mocks with verification. When testing pay_late_fees() and refund_late_fee_payment(), it was important not only that the payment gateway was called, but that it was called with the correct patron ID, amount, and description. I used methods like assert_called_once_with() to confirm correct usage and assert_not_called() to ensure that invalid inputs (for example, empty patron IDs or zero fees) did not trigger unnecessary payment processing. This experience taught me that stubs are best for "give me data" dependencies, while mocks are best for "check that this interaction happened" dependencies.

Challenge 2 – Mock Specification and Type Safety
Another issue came from using plain mocks without a specification. If I created a mock with just Mock(), it would happily accept calls to methods that do not exist on the real object, which makes tests very fragile. I solved this by always using Mock(spec=PaymentGateway) for payment-related mocks. The spec argument forces the mock to only allow attributes and methods that exist on the real PaymentGateway class. If the code under test accidentally tried to call a non-existent method, the test would fail with an AttributeError, which is exactly what I want.

I also learned how to control mock behaviour precisely. For success cases I set mock_gateway.process_payment.return_value = (True, "txn_123", "Success"). For failure scenarios I used side_effect, for example raising an exception to simulate a network timeout. This gave me fine-grained control over success, failure, and exceptional paths without touching the real external service.

Challenge 3 – The Coverage Paradox: When Mocking Hides Uncovered Code
A big surprise arrived when I ran coverage after finishing the mocked tests in Task 2.1. Even though I had many tests around the payment flow, payment_service.py only showed 27% statement coverage and overall coverage was just 57%. The reason was that by mocking the PaymentGateway methods, I never actually executed their real implementations. The mocks intercepted the calls and returned predetermined values, so the real code – including validation branches such as if amount <= 0 and if amount > 1000 – remained completely untouched.

To fix this in Task 2.2, I wrote a separate set of tests that did not mock the payment gateway. Instead, I instantiated real PaymentGateway objects and called process_payment(), refund_payment(), and verify_payment_status() with different inputs. These tests exercised all branches (valid amounts, zero and negative amounts, large amounts above the limit, valid and invalid patron IDs, valid and invalid transaction IDs). As a result, payment_service.py increased from 27% to 100% statement coverage and contributed to raising total coverage from 57% to 89%.

This experience taught me that mocking is powerful for isolating units, but it also hides implementation code from coverage tools. A good test suite needs both interaction tests with mocks and direct implementation tests without them.

Challenge 4 – Import Path Confusion After Restructuring
When library_service.py was moved into a services package, many existing imports broke. Some tests still used invalid import forms like from services/library_service.py import add_book_to_catalog, which Python treats as a syntax error. The correct pattern is from services.library_service import add_book_to_catalog, and this only works if the services directory contains an __init__.py file.

My solution was to systematically update all imports in the test files and route files to use proper module notation, and to add an __init__.py file to mark services as a package. This fixed the import errors and reminded me how closely testing depends on a correct module structure.

Challenge 5 – Patching at the Right Location
Another subtle challenge was understanding where to patch functions when using mocker.patch(). The intuitive idea is to patch where a function is defined (for example, database.get_book_by_id). In reality, you have to patch the function where it is used. In library_service.py, the code does from database import get_book_by_id, so the function is stored in the services.library_service namespace. Patching database.get_book_by_id has no effect on the already imported reference. The correct patch is mocker.patch('services.library_service.get_book_by_id', ...), which targets the symbol that the code under test actually calls.

Learning to “patch where it is used, not where it is defined” was crucial for getting my stubs to work reliably and is a key lesson about Python imports and mocking.

Challenge 6 – Focusing on Branch Coverage, Not Just Line Coverage
At first I only looked at line coverage: if a line executed at least once, I considered it covered. The problem is that a line containing an if can report as covered even if only one side of the condition is executed. For example, if amount <= 0: is counted as covered when running with a positive amount, but the error path for zero or negative amounts is still untested.

Once I started paying attention to branch coverage and the HTML report’s partial-coverage highlighting, I added targeted tests for both sides of each important conditional. For process_payment(), this meant tests for amounts less than or equal to zero, amounts just above zero, amounts greater than 1000, and different patron ID lengths. I also added boundary tests (for example, amount equal to exactly 0.01 and 1000.00). This systematic approach gave me much more confidence that the validation logic behaves correctly in all relevant scenarios.

Challenge 7 – Deciding What Not to Cover
Even after the new tests, 21 lines in library_service.py remained uncovered. My first instinct was to push for 100% coverage, but that would have required complicated setups for low-value scenarios.

Most of the remaining lines fall into three categories: database error paths, duplicate late-fee logic, and defensive “should never happen” branches. Testing the database error paths would mean heavily mocking the database layer to force failures that are simple return statements with no real business logic. The late-fee calculations inside the patron status report duplicate logic that is already covered in a dedicated late-fee function. The defensive checks guard against impossible or extremely rare states and would require artificial, contrived setups.

I decided that leaving these lines uncovered, with clear documentation of why, was the more professional choice. This reinforced the idea that coverage is a tool, not a goal in itself. The aim is high confidence in critical behaviour, not blindly chasing 100%.

Challenge 8 – Slow Tests Caused by Sleep Statements
When I switched from mocks to real PaymentGateway calls, my test runtime jumped from under a second to more than 20 seconds. The reason was that the real methods contain time.sleep() calls to simulate network latency. With many tests calling these methods, the delays added up.

For this assignment I accepted the slower runtime because the goal of Task 2.2 was to run the real implementation for coverage. However, it made me think about how I would handle this in a real project: for example, by injecting a configurable delay or by mocking time-related functions in tests. It also showed the trade-off between fast mocked tests (good for quick feedback) and slower full-implementation tests (good for deeper confidence).

Challenge 9 – Balancing Assertions and Mock Verification
Early on I only asserted on return values, for example checking that success was True. That verified the output but said nothing about whether the payment gateway was used correctly. Over time I learned to always pair output assertions with mock interaction checks. For positive paths, I assert that process_payment or refund_payment was called exactly once with the expected arguments. For negative paths, such as invalid inputs, I assert that these methods were not called at all.

This helped me see mocks as having a dual role: they stand in for dependencies and also act as built-in probes that record how they were used. Ignoring either side leads to incomplete tests.

Reflections and Lessons Learned
Across these challenges I learned that stubbing, mocking, and coverage testing each solve different problems:

- Stubs are best for supplying predictable data so you can focus on business logic.
- Mocks are best for verifying that your code interacts with collaborators correctly, including both positive and negative cases.
- Coverage tests that avoid mocks are necessary to ensure that the actual implementations (not just the interaction patterns) are exercised, including all branches.

The key lesson is that a strong test suite uses all three techniques in combination. Mock-heavy tests give fast, isolated feedback; unmocked tests improve coverage and validate implementation details; and coverage reports help highlight blind spots and partially tested branches. I also learned to be intentional about what remains uncovered and to justify those decisions instead of chasing 100% for its own sake.

Overall, working through these issues gave me a much deeper, practical understanding of mocking and stubbing in Python, and how they fit into a broader testing strategy that balances isolation, realism, speed, and confidence.

7) Screenshots

```
PS C:\Users\radme\repo\cisc327-library-management-20421691> pytest --cov=services --cov-report=
html --cov-report=term tests/
==================================== test session starts =====================================
platform win32 -- Python 3.12.10, pytest-8.4.2, pluggy-1.6.0
rootdir: C:\Users\radme\repo\cisc327-library-management-20421691
plugins: anyio-4.7.0, cov-7.0.0, mock-3.15.1
collected 174 items

tests\test_add_book.py ...............                                                  [  8%]
tests\test_borrow_book.py ............                                                  [ 15%]
tests\test_integration.py .....                                                         [ 18%]
tests\test_late_fee.py .................                                                [ 28%]
tests\test_patron_status.py .....................                                       [ 40%]
tests\test_patron_status_web.py s........                                               [ 45%]
tests\test_payment_service_coverage.py ......................                           [ 58%]
tests\test_return_book.py ................                                              [ 67%]
tests\test_search.py ......................                                             [ 80%]
tests\test_services_mocking_stubbing.py ...................................             [100%]

======================================= tests coverage =======================================
______________________ coverage: platform win32, python 3.12.10-final-0 ______________________

Name                          Stmts   Miss  Cover
-------------------------------------------------
services\__init__.py              0      0   100%
services\library_service.py     155     21    86%
services\payment_service.py      30      0   100%
-------------------------------------------------
TOTAL                           185     21    89%
Coverage HTML written to dir htmlcov
============================== 173 passed, 1 skipped in 20.60s ===============================
```