A2 - Matt Dobaj - 20350312 - TA Group 1

Github repo: https://github.com/dobaj/cisc327-library-management-a2-0312

Previous A1 Content

function name	implementation status (complete/partial)	what is missing
R1 Add Book	partial	Does not check that the ISBN is 13 digits, only that it is 13 characters long. ISBN-13s cannot contain anything other than numbers.
R2 Book Catalog Display	complete	nothing
R3 Book Borrowing Interface	partial	The borrow limit in the app is 6, not 5 as described in the requirements.
R4 Book Return Processing	partial	Does not allow user to return books. Does not update available copies or record return date. Does not calculate or display any late fees.
R5 Late Fee Calculation API	partial	Does not calculate late fees based on decription's parameters. Returns nothing.
R6 Book Search Functionality	partial	Does not search through book catalog. Does not return books in the same format as the catalog display.
R7 Patron Status Report	partial	Does not display patron status with borrowed books, late fees, number of books borrowed, or borrowing history. There is also no menu option for showing the patron status in the main interface.

Test Summary

Requirement	Test Overview	Failed Tests
R1 Add Book	Tests for a variety of input handling cases involving no title, a title that is too long, no author, an invalid isbn number, and an invalid number of book copies.	Fails case where an ISBN containing letters is used. This fails the requirement where the ISBN should be 13 digits, not characters.
R2 Book Catalog Display	Tests that the database reflects a recently added book, that all books are returned sorted, that all book fields are populated, and that borrowing a book changes the available copies.	Fails no cases.
R3 Book Borrowing Interface	Tests borrowing a book with valid input, invalid patron ids, exceeding the borrow limit, a book that is unavailable, and a book with an invalid id.	Fails the borrow limit test as the logic allows the user to borrow 6 books, not 5 as per the requirements.
R4 Book Return Processing	Tests that a book can be returned with valid input and input handling cases involving a book that is not borrowed, a patron id that is too long, and a book with an invalid id.	Fails all tests as this requirement is not implemented.

Requirement	Test Overview	Failed Tests
R5 Late Fee Calculation API	Tests that a late fee is not calculated for a book that is not late. Tests input handling for invalid patron ids, book ids and for when a book is not borrowed. Tests when a book is considered late and that the late fee is capped to the maximum fee.	Fails all tests as this requirement is not implemented.
R6 Book Search Functionality	Tests for a search with an exact match for the title or isbn of a book, and the case insensitive partial match of the title or author of a book. Also tests the search for an isbn that no book is associated with.	Fails all tests as this requirement is not implemented.
R7 Patron Status Report	Tests that the number of borrowed books is valid, that no books nor history are returned when no books have been borrowed, that the late fee for a borrowed book matches its expected value, and tests input handling for an invalid patron id.	Fails all tests as this requirement is not implemented.

New A2 Content

Github repo: https://github.com/dobaj/cisc327-library-management-a2-0312

Completed Function Implementations

To complete the R1 implementation, I simply added a check to ensure that the isbn contains only digits by seeing if it is parsable as an integer. No additional tests or modifications were required for this implementation change.

To complete the R3 implementation, I simply changed an if statement that checks if a user is over their borrow limit. Initially, it checked if the user has borrowed more than 5 books and I changed it to check if they have borrowed 5 or more books. No additional tests or modifications were required for this implementation change.

To completely implement R5, I had to add checks to see if the patron id was valid, that the given book is currently borrowed by the patron, and then calculate the fees if the book ends up being overdue. I decided to truncate the due date to only include the date portion as it would make more sense from a functional standpoint; most libraries don't have due times, just dates. I had to slightly modify my tests to better represent actual books in the database by making the due dates 14 days after the borrow date. I also needed to change the text in a test case in order to match a more clear error message I implemented. I added an extra test case as well to verify that the fee calculation would align with the requirements, testing the combination of the first week's per day cost and the cost per day afterwards.

Completely implementing R4 required implementing R5 first in order to calculate late fees. All of the error checking was able to be copied from the late fees calculation function, which meant I just needed to then get the late fees, update the borrow record with the return date, and update the book's availability. Like in R5, I needed to change the text in a test case to match a new, more clear, error message. I also added two new test cases. One ensures that the late fees are properly calculated when returning a book that is overdue. The other ensures that a user can't return a single borrowed book twice, which could lead to incorrect behaviour. I also removed the incomplete disclaimer from the frontend.

Completely implementing R6 required me to add two new functions to search the books database, one for titles and one for authors. I implemented partial case-insentive matching in these functions. The main function for R6 just returns results from either isbn, author, or title search based on the search_type. I also removed the incomplete disclaimer from the frontend.

To completely implement R7 I added a new database function to get a user's previously borrowed books. This was done by copying the get_patron_borrowed_books function and checking for entries that have a return date. This return date is then passed to the frontend. I then created a new page and routes for the patron status functionality. I also created a form to get the patron id using elements from the return books page and used table elements from the catalog page to show the patron's current and previously borrowed books. The tests required slight modifications as I changed the variable names in my api format. I also extended my testing by adding a new test to see if books properly move to the prev_books section after being returned and verifying that book ids match the results.

AI-Assisted Test Generation

I used the free tier of ChatGPT to generate test cases for the requirement specifications. I cleared the chat context with each requirement and followed a generalized prompt that had the following prompt

`generate 4 pytest test cases for the requirement {{ Snippet from requirements_specification.md }} which uses this function: {{ Relevant functions definitions }}`

Follow up prompts were limited to just giving the LLM other function definitions when neccessary.

R1:

```
import pytest
from unittest.mock import patch
from library_service import add_book_to_catalog, get_all_books  # Assuming these are the actual names

@pytest.fixture
def valid_book_data():
    return {
        'title': 'Test Book Title',
        'author': 'Test Author',
        'isbn': '1234567890123',
        'total_copies': 10
    }

def test_add_valid_book(valid_book_data):
    # Mock the get_all_books to simulate database state before the book is added
    with patch('library_service.get_all_books') as mock_get_all_books:
        mock_get_all_books.return_value = []

        # Call the function that adds the book to the catalog
        success, message = add_book_to_catalog(
            valid_book_data['title'],
            valid_book_data['author'],
            valid_book_data['isbn'],
            valid_book_data['total_copies']
        )

        # Check for success message
        assert success
        assert message == 'Book "Test Book Title" has been successfully added to the catalog.'

        # Verify the book was added
        books = get_all_books()
        # assert len(books) == 1
        assert books[0]['title'] == valid_book_data['title']
        assert books[0]['author'] == valid_book_data['author']
        assert books[0]['isbn'] == valid_book_data['isbn']
        assert books[0]['total_copies'] == valid_book_data['total_copies']

def test_invalid_isbn_format():
    invalid_data = {
        'title': 'Invalid ISBN Book',
        'author': 'Test Author',
        'isbn': '12345',  # Invalid ISBN, should be exactly 13 digits
        'total_copies': 5
    }

    success, message = add_book_to_catalog(
        invalid_data['title'],
        invalid_data['author'],
        invalid_data['isbn'],
```

```
        invalid_data['total_copies']
    )

    assert not success
    assert message == "ISBN must be exactly 13 digits."

def test_missing_title():
    invalid_data = {
        'title': '',  # Missing title
        'author': 'Test Author',
        'isbn': '1234567890123',
        'total_copies': 5
    }

    success, message = add_book_to_catalog(
        invalid_data['title'],
        invalid_data['author'],
        invalid_data['isbn'],
        invalid_data['total_copies']
    )

    assert not success
    assert message == "Title is required."

def test_invalid_total_copies():
    invalid_data = {
        'title': 'Invalid Total Copies Book',
        'author': 'Test Author',
        'isbn': '1234567890123',
        'total_copies': -3  # Invalid, should be a positive integer
    }

    success, message = add_book_to_catalog(
        invalid_data['title'],
        invalid_data['author'],
        invalid_data['isbn'],
        invalid_data['total_copies']
    )

    assert not success
    assert message == "Total copies must be a positive integer."
```

The output from ChatGPT required little processing; I just had to change the module import location to library_service and comment out some assertions. While the fact that it ran without little modification was impressive, the tests weren't extremely in depth and lacked edge case testing. They didn't test the bounds like whether the ISBN check considered letters or not, for example. It also didn't programatically generate new books to test with by either generating new ISBNs like I did or by creating a mock db, even when given the functions for the database. The messages are also static, which would cause the tests to fail if the error messages were updated in any way. This is in comparison to my test cases which simply look for keywords or small word pairs in the error messages, allowing the messages to vary a little bit.

R2:

```
import pytest
from your_module import get_all_books

@pytest.fixture
def setup_database():
    """Fixture to setup and teardown a test database."""
    setup_test_db()  # Initialize the test database with data
    yield
    teardown_test_db()  # Clean up the database after the test
```

```
def test_get_all_books(setup_database):
    # Fetch books from the database
    books = get_all_books()

    # Ensure the list contains books and is in the expected format
    assert len(books) > 0  # There should be at least one book in the catalog
    assert 'id' in books[0]
    assert 'title' in books[0]
    assert 'author' in books[0]
    assert 'isbn' in books[0]
    assert 'available_copies' in books[0]
    assert 'total_copies' in books[0]

    # Ensure books are sorted by title (as per the requirement)
    assert books == sorted(books, key=lambda x: x['title'])

def test_catalog_available_copies(setup_database):
    # Fetch all books from the catalog
    books = get_all_books()

    for book in books:
        # Ensure available_copies is less than or equal to total_copies
        assert book['available_copies'] <= book['total_copies']

def test_catalog_actions(setup_database):
    # Fetch all books from the catalog
    books = get_all_books()

    for book in books:
        # Check if the book has available copies
        if book['available_copies'] > 0:
            # Simulate a patron borrowing the book
            patron_id = '123456'  # Assume a valid patron ID
            book_id = book['id']

            success, _ = borrow_book_by_patron(patron_id, book_id)

            # Assert that the borrow function works, i.e., the book is available to borrow
            if success:
                # Verify that the number of available copies is reduced
                updated_book = get_book_by_id(book_id)
                assert updated_book['available_copies'] == book['available_copies'] - 1
        else:
            # For books with no available copies, there should be no "borrow" button
            assert book['available_copies'] == 0

def test_catalog_mixed_availability(setup_database):
    # Fetch all books from the catalog
    books = get_all_books()

    # Ensure some books have available copies and some do not
    available_books = [book for book in books if book['available_copies'] > 0]
    unavailable_books = [book for book in books if book['available_copies'] == 0]

    # There should be books with both available and unavailable copies
    assert len(available_books) > 0
    assert len(unavailable_books) > 0
```

For some reason ChatGPT performed significantly worse on this task. At first it kept trying to use packages that the project

doesn't use to create a mock database and eventually settled on using placeholder functions for the mock database. The tests, like in R1, lack the depth required to properly test the functionality of the get_all_books() function, with one test simply testing if some books are unavailable vs. available. As this uses non-existent functions, it does not run.

R3:

```
import pytest
from datetime import datetime, timedelta
from library_service import borrow_book_by_patron
from unittest.mock import patch

# Test case 1: Invalid patron ID (not exactly 6 digits)
def test_invalid_patron_id():
    # Patron ID should be a 6-digit number. Test with an invalid ID
    patron_id = "12345"  # 5 digits (invalid)
    book_id = 101  # Valid book ID (mocked)

    # Call the function
    success, message = borrow_book_by_patron(patron_id, book_id)

    # Assert failure due to invalid patron ID
    assert not success
    assert message == "Invalid patron ID. Must be exactly 6 digits."


# Test case 2: Book is unavailable (no copies available)
@patch('library_service.get_book_by_id')
def test_book_unavailable(mock_get_book_by_id):
    # Mocking a book with no available copies
    mock_get_book_by_id.return_value = {"book_id": 101, "title": "Test Book", "available_copies": 0}

    patron_id = "123456"  # Valid patron ID
    book_id = 101  # Book ID that is unavailable

    # Call the function
    success, message = borrow_book_by_patron(patron_id, book_id)

    # Assert failure due to book availability
    assert not success
    assert message == "This book is currently not available."


# Test case 3: Patron has reached the maximum borrowing limit (5 books)
@patch('library_service.get_patron_borrow_count')
def test_patron_max_limit(mock_get_patron_borrow_count):
    # Mocking a patron who has already borrowed 5 books
    mock_get_patron_borrow_count.return_value = 5  # Patron already has 5 books borrowed

    patron_id = "123456"  # Valid patron ID
    book_id = 101  # Valid book ID

    # Call the function
    success, message = borrow_book_by_patron(patron_id, book_id)

    # Assert failure due to borrowing limit
    assert not success
    assert message == "You have reached the maximum borrowing limit of 5 books."


# Test case 4: Successful book borrowing
```

```
@patch('library_service.get_book_by_id')
@patch('library_service.get_patron_borrow_count')
@patch('library_service.insert_borrow_record')
@patch('library_service.update_book_availability')
def test_successful_book_borrowing(mock_update_book_availability, mock_insert_borrow_record, mock_get_patron_b
    # Mocking book availability and patron borrow count
    mock_get_book_by_id.return_value = {"book_id": 101, "title": "Test Book", "available_copies": 5}
    mock_get_patron_borrow_count.return_value = 2  # Patron has borrowed 2 books

    # Mocking the success of insert and update functions
    mock_insert_borrow_record.return_value = True
    mock_update_book_availability.return_value = True

    patron_id = "123456"  # Valid patron ID
    book_id = 101  # Valid book ID

    # Call the function
    success, message = borrow_book_by_patron(patron_id, book_id)

    # Assert success
    assert success
    assert "Successfully borrowed" in message
    assert "Due date" in message  # Ensure due date is included in the success message
```

Overall ChatGPT performed well on this task, making use of (new to me) functions in pytest like mock return values. I felt that the tests in this instance contained the right amount of depth for this requirement. Because I only asked for 4 tests, it didn't fully cover the requirement's specifications, though. While it does use some hardcoded values, they aren't actually added to the database so as not to impact reusability. It doesn't appear to run perfectly, which may be on account of the mock values ChatGPT is using because the test to see if the patron is over their borrowing limit has a "Book not found" error. This implies that the function is trying to read from the actual database rather than ChatGPT's mock databases.

R4:

```
import pytest
from datetime import datetime, timedelta
from library_service import return_book_by_patron

# Mock functions
def mock_get_patron_borrowed_books(patron_id):
    return [
        {"book_id": 1, "borrow_date": datetime.now() - timedelta(days=10), "due_date": datetime.now() - timede
        {"book_id": 2, "borrow_date": datetime.now() - timedelta(days=5), "due_date": datetime.now() + timedel
    ]

def mock_update_borrow_record_return_date(patron_id, book_id, return_date):
    return True

def mock_update_book_availability(book_id, change):
    return True

def mock_calculate_late_fee_for_book(patron_id, book_id):
    if book_id == 1:
        return {'fee_amount': 3.0, 'days_overdue': 5, 'status': 'Book is overdue'}
    return {'fee_amount': 0.0, 'days_overdue': 0, 'status': 'Book is not overdue'}

# Assign mock functions to replace actual database calls
return_book_by_patron.get_patron_borrowed_books = mock_get_patron_borrowed_books
return_book_by_patron.update_borrow_record_return_date = mock_update_borrow_record_return_date
return_book_by_patron.update_book_availability = mock_update_book_availability
return_book_by_patron.calculate_late_fee_for_book = mock_calculate_late_fee_for_book
```

```
# Test cases
def test_return_book_success_with_late_fees():
    patron_id = "123456"
    book_id = 1
    result = return_book_by_patron(patron_id, book_id)
    assert result[0] == True
    assert "Book returned successfully" in result[1]
    assert "Late fees: 3.00" in result[1]
    assert "Days overdue: 5" in result[1]

def test_return_book_no_late_fees():
    patron_id = "123456"
    book_id = 2
    result = return_book_by_patron(patron_id, book_id)
    assert result[0] == True
    assert "Book returned successfully" in result[1]
    assert "Late fees: 0.00" in result[1]
    assert "Days overdue: 0" in result[1]

def test_return_book_not_borrowed_by_patron():
    patron_id = "123456"
    book_id = 3  # Not borrowed by patron
    result = return_book_by_patron(patron_id, book_id)
    assert result[0] == False
    assert result[1] == "A book with this id is not borrowed by this patron or the book id is invalid."

def test_return_book_invalid_patron_id():
    patron_id = "12345"  # Invalid patron ID
    book_id = 1
    result = return_book_by_patron(patron_id, book_id)
    assert result[0] == False
    assert result[1] == "Invalid patron id."
```

Here I found that ChatGPT did well conceptually, with good test coverage given the limited number of tests and good choices of scenarios to test. It does have some mistakes, however, like asserting incorrect late fees. Some mistakes are critical, like the fact the mock functions are not actually overriding the original functions as intended leading to incorrect test results. In this instancce, the return books function is always failing.

R5:

```
import pytest
from datetime import datetime, timedelta
from library_service import calculate_late_fee_for_book

@pytest.fixture
def mock_data_not_overdue():
    return {
        'patron_id': '123456',
        'book_id': 1,
        'borrowed_books': [
            {
                'book_id': 1,
                'due_date': (datetime.now() + timedelta(days=14)),  # Not overdue
                'is_overdue': False
            }
        ]
    }

def test_book_not_overdue(mock_data_not_overdue):
    result = calculate_late_fee_for_book(mock_data_not_overdue['patron_id'], mock_data_not_overdue['book_id'])
    assert result['fee_amount'] == 0.0
```

```
    assert result['days_overdue'] == 0
    assert result['status'] == 'Book is not overdue'

@pytest.fixture
def mock_data_overdue_5_days():
    return {
        'patron_id': '123456',
        'book_id': 1,
        'borrowed_books': [
            {
                'book_id': 1,
                'due_date': (datetime.now() - timedelta(days=5)),  # Overdue by 5 days
                'is_overdue': True
            }
        ]
    }

def test_book_overdue_5_days(mock_data_overdue_5_days):
    result = calculate_late_fee_for_book(mock_data_overdue_5_days['patron_id'], mock_data_overdue_5_days['book
    assert result['fee_amount'] == 2.5  # 5 days * $0.50/day
    assert result['days_overdue'] == 5
    assert result['status'] == 'Book is overdue'

@pytest.fixture
def mock_data_overdue_10_days():
    return {
        'patron_id': '123456',
        'book_id': 1,
        'borrowed_books': [
            {
                'book_id': 1,
                'due_date': (datetime.now() - timedelta(days=10)),  # Overdue by 10 days
                'is_overdue': True
            }
        ]
    }

def test_book_overdue_10_days(mock_data_overdue_10_days):
    result = calculate_late_fee_for_book(mock_data_overdue_10_days['patron_id'], mock_data_overdue_10_days['bo
    assert result['fee_amount'] == 5.0  # 7 days * $0.50 + 3 days * $1.00 = $5.00
    assert result['days_overdue'] == 10
    assert result['status'] == 'Book is overdue'

@pytest.fixture
def mock_data_overdue_20_days():
    return {
        'patron_id': '123456',
        'book_id': 1,
        'borrowed_books': [
            {
                'book_id': 1,
                'due_date': (datetime.now() - timedelta(days=20)),  # Overdue by 20 days
                'is_overdue': True
            }
        ]
    }

def test_book_overdue_20_days(mock_data_overdue_20_days):
    result = calculate_late_fee_for_book(mock_data_overdue_20_days['patron_id'], mock_data_overdue_20_days['bo
    assert result['fee_amount'] == 15.0  # Maximum fee cap
```

```
    assert result['days_overdue'] == 20
    assert result['status'] == 'Book is overdue'
```

I think on this requirement ChatGPT took a bit of a step backwards as none of the tests work in this snippet. Conceptually, it is only testing the fee amount calculation multiple times, which lacks the breadth of testing that the requirement should have. It doesn't test for input validation at all, which I find necessary in testing this function. There are also more issues with the mock functions not working as intended and the tests don't check if the results of the function are a success or not before trying to read the fee_amount. This is important as the fee_amount is not set unless the function is successful.

R6:

```
import pytest
from unittest.mock import patch
from library_service import search_books_in_catalog

# Sample book data for testing
book_data = [
    {"id": 1, "title": "The Catcher in the Rye", "author": "J.D. Salinger", "isbn": "1234567890123"},
    {"id": 2, "title": "To Kill a Mockingbird", "author": "Harper Lee", "isbn": "2345678901234"},
    {"id": 3, "title": "The Great Gatsby", "author": "F. Scott Fitzgerald", "isbn": "3456789012345"},
]

# Mock functions for database queries
def mock_get_book_by_isbn(isbn):
    return next((book for book in book_data if book['isbn'] == isbn), None)

def mock_get_book_by_title(title):
    return [book for book in book_data if title.lower() in book['title'].lower()]

def mock_get_book_by_author(author):
    return [book for book in book_data if author.lower() in book['author'].lower()]


# Test cases
@pytest.mark.parametrize("search_term, search_type, expected_result", [
    ("catcher", "title", [{"id": 1, "title": "The Catcher in the Rye", "author": "J.D. Salinger", "isbn": "123
    ("mockingbird", "title", [{"id": 2, "title": "To Kill a Mockingbird", "author": "Harper Lee", "isbn": "234
    ("fitzgerald", "author", [{"id": 3, "title": "The Great Gatsby", "author": "F. Scott Fitzgerald", "isbn":
    ("1234567890123", "isbn", [{"id": 1, "title": "The Catcher in the Rye", "author": "J.D. Salinger", "isbn":
    ("nonexistent", "title", []),
])
@patch("library_service.get_book_by_isbn", side_effect=mock_get_book_by_isbn)
@patch("library_service.get_book_by_title", side_effect=mock_get_book_by_title)
@patch("library_service.get_book_by_author", side_effect=mock_get_book_by_author)
def test_search_books_in_catalog(mock_get_by_isbn, mock_get_by_title, mock_get_by_author, search_term, search_
    result = search_books_in_catalog(search_term, search_type)
    assert result == expected_result
```

This seems to be another step backwards for ChatGPT. While the tests all run perfectly well, they don't test the database functions which serve the core functionality of this requirement. Even when ChatGPT was explicitly prompted to interface with the real database it would create its own mock functions. The mock functions completely skip over the requirement's implementation, rendering the tests useless. The tests also don't test for the edge case of imperfect isbn matches which I would find important to include.

R7:

```
import pytest
from unittest.mock import patch
from datetime import datetime

from library_service import get_patron_status_report
```

```
def test_get_patron_status_report_with_borrowed_books():
    patron_id = "123456"
    # Mock the responses from database functions
    with patch('database.get_patron_borrowed_books') as mock_borrowed_books, \
         patch('database.get_patron_prev_borrowed_books') as mock_prev_books, \
         patch('library_service.calculate_late_fee_for_book') as mock_calculate_late_fee:

        # Mock data for borrowed books
        mock_borrowed_books.return_value = [
            {"book_id": 1, "title": "Book 1", "borrow_date": datetime(2023, 10, 1), "due_date": datetime(2023,
            {"book_id": 2, "title": "Book 2", "borrow_date": datetime(2023, 9, 15), "due_date": datetime(2023,
        ]

        # Mock data for previous borrowed books
        mock_prev_books.return_value = [
            {"book_id": 3, "title": "Book 3", "borrow_date": datetime(2023, 8, 1), "return_date": datetime(202
        ]

        # Mock late fee calculation (no fees, book not overdue)
        mock_calculate_late_fee.return_value = {"fee_amount": 0.00, "days_overdue": 0, "status": "Book is not

        # Call the function
        result = get_patron_status_report(patron_id)

        # Assert the result contains the expected structure
        assert result["status"] == "Got books for patron"
        assert len(result["curr_books"]) == 2
        assert len(result["prev_books"]) == 1
        assert result["borrow_count"] == 2
        assert "fee_amount" in result["curr_books"][0]
        assert "fee_amount" in result["curr_books"][1]

def test_get_patron_status_report_invalid_patron_id():
    invalid_patron_id = "12345"  # Invalid ID (should be 6 digits)

    # Call the function with an invalid patron ID
    result = get_patron_status_report(invalid_patron_id)

    # Assert the result contains an error message
    assert result["status"] == "Invalid patron id."

def test_get_patron_status_report_no_borrowed_books():
    patron_id = "123456"

    # Mock the response to simulate no borrowed books
    with patch('database.get_patron_borrowed_books') as mock_borrowed_books, \
         patch('database.get_patron_prev_borrowed_books') as mock_prev_books:

        mock_borrowed_books.return_value = []  # No current borrowed books
        mock_prev_books.return_value = []  # No previous borrowed books

        # Call the function
        result = get_patron_status_report(patron_id)

        # Assert the result contains the expected structure
        assert result["status"] == "Got books for patron"
        assert len(result["curr_books"]) == 0
        assert len(result["prev_books"]) == 0
        assert result["borrow_count"] == 0
```

```
def test_get_patron_status_report_late_fee_calculation_error():
    patron_id = "123456"

    # Mock the response to simulate an error in late fee calculation
    with patch('database.get_patron_borrowed_books') as mock_borrowed_books, \
         patch('database.get_patron_prev_borrowed_books') as mock_prev_books, \
         patch('library_service.calculate_late_fee_for_book') as mock_calculate_late_fee:

        # Mock data for borrowed books
        mock_borrowed_books.return_value = [
            {"book_id": 1, "title": "Book 1", "borrow_date": datetime(2023, 10, 1), "due_date": datetime(2023,
        ]

        # Simulate a late fee calculation error
        mock_calculate_late_fee.return_value = {"status": "Error calculating late fee for book 1"}

        # Call the function
        result = get_patron_status_report(patron_id)

        # Assert the result contains an error message
        assert result["status"] == "Error calculating late fee for book 1"
```

Here I find ChatGPT didn't perform very well again. The mock functions it created don't appear to be working again, causing the function to fail in every test. I find that conceptually it seems to be testing sensible scenarios given the limited amount of tests. I would prefer to see it test the previously borrowed book functionality in some capacity, however, in order to get better coverage of the requirement's specifications. It does also hallucinate some error message text in the late_fee_calculation_error() test case.

Conclusions

Overall I find that ChatGPT's free tier has issues with consistency. It sometimes captures the breadth of a requirement's specifications perfectly, like in R3 and R4, but at other times it appears to be in complete disagreement with what I found important in my test cases, like in R5 and R6. Its tests also often checked for exact error messages where I would rather check for keywords or substrings in error messages to allow for some flexibility.

Functionally, all of ChatGPT's tests were flawed. Some requirements required only minor tweaks, like R1, however most tests did not perform correctly at all and would require extensive debugging to implement as their mock functions simply weren't working. Oftentimes this resulted in the tests running okay but the function they were testing would appear to be working incorrectly. This would cause the developer to incorrectly investigate the tested function to fix these errors instead of the tests.