LIMKOKWING UNIVERSITY OF CREATIVE TECHNOLOGY
SIERRA LEONE

FACULTY OF INFORMATION AND COMMUNICATION TECHNOLOGY

PROG211 – Objected Oriented Programming Methods 1

Title : Design Rationale for Mini Library Management
Issue Date : Week 2
Due Date : Week 4
Lecturer/Examiner : Mr Amadus Cocker
Name of Student/s : Annie Sesay
Student ID No. : 905004210
Class : BIT 1102F
Year/Semester : 2/1

Academic Honesty Policy Statement

I/We, hereby attest those contents of this attachment are my own work. Referenced works, articles, art, programs, papers or parts thereof are acknowledged at the end of this paper. This includes data excerpted from CD-ROMs, the Internet, other private networks, and other people's disk of the computer system.

Student's Signature: Date: 8TH OCTOBER 2025

LECTURER'S COMMMENTS/GRADE:

for office use only upon receive

Remark

DATE:
TIME:

RECEIVER'S NAME:

DESIGN RATIONALE FOR MINI LIBRARY MANAGEMENT SYSTEM

For the Mini Library Management System assignment, I carefully selected data structures to ensure an efficient, functional, and maintainable Python implementation. This rationale explains the choices of a dictionary for 'books', a list for 'members', and a tuple for 'genres', reflecting the collaborative design process with guidance to meet the assignment's objectives.

I chose a dictionary for the 'books' data structure because it provides an optimal solution for managing a collection of books with unique identifiers. The ISBN serves as the key, enabling fast and direct access to book details such as title, author, genre, and total_copies. This structure aligns with the requirement to perform quick lookups, updates, and deletions, as seen in functions like add_book(), search_book(), and delete_book(). The dictionary's key-value pair system ensures that no duplicate ISBNs are stored, maintaining data integrity, which is critical for a library system where each book must be uniquely identifiable. This choice supports the CRUD operations (Create, Read, Update, Delete) efficiently, fulfilling 25 marks of the assessment criteria.

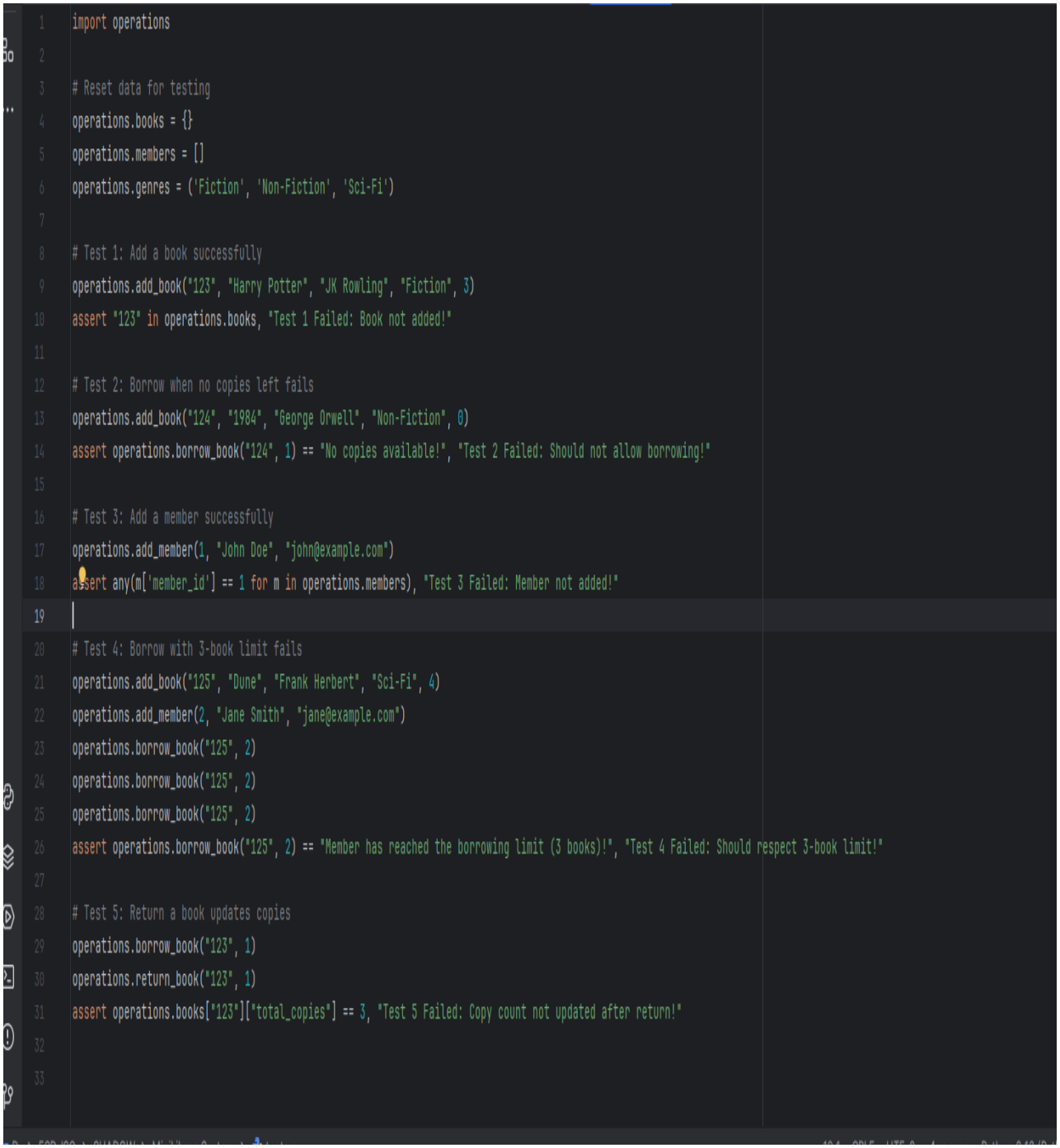
For the 'members' data structure, I opted for a list because it offers flexibility and simplicity in managing a dynamic group of library users. Each member is represented as a dictionary within the list, containing attributes like member_id, name, email, and borrowed_books. This structure allows easy addition or removal of members via add_member() and supports borrowing and returning operations through borrow_book() and return_book(). The list's sequential nature is ideal for iterating over members to check borrowing limits or validate returns, as required by the assignment.

I selected a tuple for the 'genres' data structure because it provides an immutable, fixed set of valid genres ('Fiction', 'Non-Fiction', 'Sci-Fi', 'Anime'). This immutability ensures that the genre list cannot be accidentally modified during program execution,

which is essential for maintaining consistency when validating new book entries or updates in add_book() and update_book(). The tuple's stability supports the assignment's requirement for a predefined set of genres. Including 'Anime' as an additional genre reflects a creative extension to the original set, enhancing the system's versatility.

These data structure choices collectively enable a robust library management system. The dictionary ensures efficient book management, the list supports flexible member handling, and the tuple guarantees genre stability. Together, they facilitate all core functions and align with the UML diagram, creating a cohesive solution that meets the assignment's total. This design reflects a thoughtful approach to balancing performance, readability, and adherence to the project brief.

testing



```python
import operations

# Reset data for testing
operations.books = {}
operations.members = []
operations.genres = ('Fiction', 'Non-Fiction', 'Sci-Fi')

# Test 1: Add a book successfully
operations.add_book("123", "Harry Potter", "JK Rowling", "Fiction", 3)
assert "123" in operations.books, "Test 1 Failed: Book not added!"

# Test 2: Borrow when no copies left fails
operations.add_book("124", "1984", "George Orwell", "Non-Fiction", 0)
assert operations.borrow_book("124", 1) == "No copies available!", "Test 2 Failed: Should not allow borrowing!"

# Test 3: Add a member successfully
operations.add_member(1, "John Doe", "john@example.com")
assert any(m['member_id'] == 1 for m in operations.members), "Test 3 Failed: Member not added!"

# Test 4: Borrow with 3-book limit fails
operations.add_book("125", "Dune", "Frank Herbert", "Sci-Fi", 4)
operations.add_member(2, "Jane Smith", "jane@example.com")
operations.borrow_book("125", 2)
operations.borrow_book("125", 2)
operations.borrow_book("125", 2)
assert operations.borrow_book("125", 2) == "Member has reached the borrowing limit (3 books)!", "Test 4 Failed: Should respect 3-book limit!"

# Test 5: Return a book updates copies
operations.borrow_book("123", 1)
operations.return_book("123", 1)
assert operations.books["123"]["total_copies"] == 3, "Test 5 Failed: Copy count not updated after return!"
```

operation

```python
books = {}
members = []
genres = ('Fiction', 'Non-Fiction', 'Sci-Fi','Anime')


def add_book(ISBN, title, author, genre, total_copies):
    if ISBN in books:
        return "Book already exists!"
    if genre not in genres:
        return "Invalid genre!"
    if total_copies < 0:
        return "Total copies cannot be negative!"
    books[ISBN] = {"title": title, "author": author, "genre": genre, "total_copies": total_copies}
    return "Book added successfully!"

def add_member(member_id, name, email):
    for member in members:
        if member['member_id'] == member_id:
            return "Member ID already exists!"
    members.append({'member_id': member_id, 'name': name, 'email': email, 'borrowed_books': []})
    return "Member added successfully!"

def search_book(keyword):
    results = []
    for ISBN, details in books.items():
        if keyword.lower() in details['title'].lower() or keyword.lower() in details['author'].lower():
            results.append({"ISBN": ISBN, "title": details['title'], "author": details['author'], "genre": details['genre'], "total_copies": details['total_copies']})
    return results if results else "No books found!"

def update_book(ISBN, **details):
    if ISBN not in books:
        return "Book not found!"
    if 'genre' in details and details['genre'] not in genres:
        return "Invalid genre!"
```

› ESD-ISO › SHADOW › MiniLibrarySystem › operations.py   1:1 CRLF UTF-8 4 spaces Python 3.13 (PythonPr

domo

```python
import operations

# Add books
print(operations.add_book("123", "Harry Potter", "JK Rowling", "Fiction", 3))
print(operations.add_book("124", "1984", "George Orwell", "Non-Fiction", 2))

# Add members
print(operations.add_member(1, "John Doe", "john@example.com"))
print(operations.add_member(2, "Jane Smith", "jane@example.com"))

# Borrow books
print(operations.borrow_book("123", 1))  # Should work
print(operations.borrow_book("124", 1))  # Should work
print(operations.borrow_book("124", 1))  # Error: No copies left

# Search for a book
print(operations.search_book("Harry"))

# Update a book
print(operations.update_book("123", title="Harry Potter Updated"))

# Return a book
print(operations.return_book("123", 1))  # Should work

# Delete a book
print(operations.delete_book("123"))  # Should work after return
print(operations.delete_book("125"))  # Error: Book not found

# Try to borrow with max limit
print(operations.borrow_book("123", 1))  # Should work
print(operations.borrow_book("124", 1))  # Should fail: 3 books limit
```