

Gideon Clottey

(202289192)

ENGI 9839 Assignment 2

Q1a)

In order to assess a fully integrated software system against the established criteria, system testing is a crucial validation step in the Software Development Life Cycle (SDLC). After unit and integration testing, it is carried out at the top of the testing pyramid to make sure that all modules and subsystems work properly together in a production-like setting.

Main Objectives of System Testing are as follows:

- **Validation of End-to-End System Functionality**
Verifying that the system functions as intended when all of its parts are integrated is the main goal. This involves confirming that user interactions through interfaces (such as GUIs or APIs) cause the appropriate internal processes to be triggered and produce the appropriate outputs.
- **Verification Against Functional and Non-Functional Requirements**
Testing the system guarantees that it satisfies both functional (like checking out a book or logging in) and non-functional (like system dependability, security, and performance under load) requirements. Usually specified at the start of the project, these requirements serve as the foundation for creating system test cases.
- **Detection of Interface and Integration Defects**
System tests evaluate the entire system as a black box through its public interfaces in order to detect interaction flaws, such as timing problems, misuse of the interface, or misinterpretation of data contracts, whereas unit tests validate specific classes or methods in isolation.
- **Support for Acceptance and Deployment Readiness**
System testing assists in verifying that the program is reliable and functional enough to move forward with production deployment or user acceptance testing (UAT). Prior to the system being made available to end users, this stage serves as the last quality check.

Three Distinct Types of System Tests with Examples

The following lists the three main types of system testing and provides examples relevant to web-based library management systems (LMS):

1. Functional Testing

Verifying that the system satisfies its functional requirements and carries out the planned tasks as delineated during the design phase is the main goal of this kind of testing. "Does the system work as promised?" is addressed. Business logic, user interactions, input/output processing, and data flows are frequently examined during functional testing.

Example: Testing the “Search Book” feature to ensure a user can search by title, author, or ISBN and receive accurate results.

SearchBook()

Concrete Test Case 1 (Successful Search)

- **Test Input:**

- ✓ User ID: U00123 (logged in)
- ✓ Search Term: "Harry Potter"
- ✓ Book Catalog: Includes "Harry Potter and the Philosopher’s Stone", "Harry Potter and the Goblet of Fire"

- **Expected Outcome:**

- ✓ System displays: “2 results found for ‘Harry Potter’”
- ✓ Book list
- ✓ shows matching titles with availability
- ✓ API Response Status: 200 OK
- ✓ Response Body:

```
{
  "results": [
    { "title": "Harry Potter and the Philosopher’s Stone", "available": true },
    { "title": "Harry Potter and the Goblet of Fire", "available": false }
  ]
}
```
- ✓ Database: No change (read-only operation)

2. **Performance Testing:**

Assesses how the library system responds to different load scenarios, such as when several users are working at once. Important parameters like throughput, response time, and server resource usage are tracked. For example, we could use Apache JMeter to simulate 1000 concurrent visitors trying to check out a book. All queries should be answered by the system in less than two seconds, and CPU and memory use should be kept within reasonable bounds. Testing for load, stress, and soak would assist guarantee the system's scalability and resilience to stress?

Example: Checking if the system can handle 500 concurrent users searching for books or borrowing books simultaneously.

LoginAllUsers()

Concrete Test Case (Simulated High Load Login)

- **Test Input:**

- ✓ 500 concurrent login requests at 10:00 AM (peak time)
- ✓ Each user provides correct credentials
- ✓ Server baseline: 2.0 GHz CPU, 8 GB RAM

- **Expected Outcome:**

- ✓ System handles all requests within 3 seconds
- ✓ Login success rate $\geq 98\%$
- ✓ No server crash or timeout
- ✓ Load Balancer metrics: CPU usage $< 80\%$, RAM usage $< 90\%$
- ✓ API Response:
 - i. Status: 200 OK for each user
 - ii. Response Time (median): $< 2.5s$
- ✓ Logs: No unexpected errors in authentication module

3. Security Testing:

The goal of security testing is to confirm that a system's built-in defenses against intrusions, breaches, and other vulnerabilities are effective. It seeks to detect any dangers and guarantee the security of user information and sensitive data.

Example: Testing that a user cannot access admin features like deleting books without admin credentials.

AccessAdminDashboard()

Concrete Test Case (Unauthorized Access Attempt)

- **Test Input:**

- ✓ User ID: U00987 (role: "Student", not an admin)
- ✓ URL Access Attempt: <https://librarysystem.ca/admin/dashboard>

- **Expected Outcome:**

- ✓ System redirects to login or access denied page
- ✓ Message: "Access Denied. You are not authorized to view this page."
- ✓ API Response Status: 403 Forbidden
- ✓ Response Body: {

```
"errorCode": "UNAUTHORIZED_ACCESS",  
"message": "Admin privileges required."  
}
```

- ✓ Security Log Entry:
User: U00987
Action: Blocked unauthorized admin panel access
Timestamp recorded
- ✓ Database: No admin data exposure, no changes made

Q1b)

System Under Test (SUT)

Web based Library Management System specifically the "**Book Borrowing**" feature accessible by authenticated users (readers).

High Level Function under Test

BorrowBook () a complete, observable user action involving:

- Selecting a book
- Confirming the borrow action
- System updates records, limits, and due dates

Identify Choices for BorrowBook ()

1. **Book Availability:** (Is the book available or already borrowed?)
2. **User Authentication:** (Is the user logged in and registered?)
3. **Borrow Limit:** (Has the user exceeded the borrowing limit?)
4. **Membership Status:** (Is the membership active or expired?)
5. **System Connectivity:** (Is the server online and database reachable?)
6. **Server Load:** (Is the system under heavy load?)

Identify Representative Values (Partitions)

1. **Choice: Book Availability**
 - Book_Available (normal scenario)
 - Book_Already_Borrowed (submission should fail)
 - Book_Does_Not_Exist (invalid book ID)

2. **Choice: User Authentication**
 - Authenticated_User (normal case)
 - Not_Logged_In (error condition)
3. **Choice: Borrow Limit**
 - Below_Limit (allowed)
 - At_Limit (boundary condition)
 - Above_Limit (error case)

Generate Test Case Specifications

- **Specification 1 (Normal, Successful Borrowing)**
 - Choices & Values:
 - Book Availability: Book_Available
 - User Authentication: Authenticated_User
 - Borrow Limit: Below_Limit
 - (Implicit: Membership Active, Server Online)
 - Expected Outcome: Book borrowed successfully, due date set, confirmation displayed.
- **Specification 2 (Failed Borrowing - Book Already Borrowed)**
 - Choices & Values:
 - Book Availability: Book_Already_Borrowed
 - User Authentication: Authenticated_User
 - Borrow Limit: Below_Limit
 - (Implicit: Membership Active, Server Online)
 - Expected Outcome: Submission rejected, message: "Book already borrowed," no record updated.

Generate Concrete Test Cases

- **Concrete Test Case 1 (Specification 1: Successful Borrow)**
 - Test Input:
 - User ID: U1001 (logged in, membership active)
 - Book ID: B123 (status: available)
 - Current Borrow Count: 2 (Limit: 5)
 - Action Time: July 21, 2025, 14:00 NDT
 - Expected Outcome:
 - Message: "Book successfully borrowed. Due date: August 4, 2025."
 - API Response Status: 200 OK

- Response Body: {"userId": "U1001", "bookId": "B123", "status": "Borrowed", "dueDate": "2025-08-04"}
 - Database: New entry in BorrowedBooks table for user U1001 and book B123
 - Admin View: Book B123 marked as "On Loan"
- **Concrete Test Case 2 (Specification 2: Failed Borrow - Book Already Borrowed)**
 - Test Input:
 - User ID: U1020 (logged in, membership active)
 - Book ID: B456 (status: already borrowed by another user)
 - Current Borrow Count: 1
 - Action Time: July 21, 2025, 14:15 NDT
 - Expected Outcome:
 - Message: "Error: Book is currently borrowed by another user."
 - API Response Status: 409 Conflict
 - Response Body: {"errorCode": "BOOK_UNAVAILABLE", "message": "Book is not available for borrowing."}
 - Database: No new borrow record for U1020
 - Admin View: Book B456 still shows borrower as another user

Q1c)

By comparing the behavior and performance of the integrated system to these predetermined standards, system testing plays a crucial role in confirming that software satisfies both functional and non-functional requirements.

Functional Requirements:

These describe the "what" of the system the precise functions, features, and behaviors that it must have in order to satisfy user needs. They specify how the system ought to react to different inputs in various scenarios.

Example (Library Management System):

Requirement	Testing Approach
User Login – Users should be able to log in with valid credentials.	Create test cases with correct and incorrect credentials. Confirm successful login redirects to the dashboard, and failed login shows an error like "Invalid password."
Book Search – Users should be able to search for books by title, author, or ISBN.	Search for different keywords and verify if the correct results are displayed. Check edge cases like partial matches and no results.

Borrow Book – Registered users should be able to borrow available books.	Simulate the borrowing process. Validate that the book status updates to "borrowed" and appears in the user's loan history.
Add New Book (Librarian Only) – Admins should be able to add new books.	Log in as a librarian, access the "Add Book" form, submit valid data, and verify that the book appears in the catalog. Also test that non-admins cannot access this feature.

Non-Functional Requirements:

These specify the features, limitations, and characteristics of the system, emphasizing "how" the system functions, including scalability, performance, security, dependability, and usability. Even if they have nothing to do with a particular function, they still affect the user experience.

Example (Library Management System):

Sn	Requirement Type	Requirement	Testing Approach
1	Performance	The system must return search results within 2 seconds for up to 100 concurrent users .	Use tools like JMeter or Locust to simulate user load. Measure response time and ensure it stays under the threshold.
2	Security	All login credentials must be encrypted during transmission and storage.	Use penetration testing or packet sniffers (e.g., Wireshark) to verify encrypted transmission. Inspect database to ensure passwords are hashed (e.g., using SHA-256).
3	Usability	The book borrowing interface must be intuitive for users with basic computer skills .	Conduct usability testing with novice users. Observe their interaction flow, collect feedback, and analyze metrics like click count and task completion time.
4	Reliability	The system should have zero crashes during extended use over 24 hours .	Run soak tests where the system is used continuously, simulating user behavior to check for memory leaks or failures.

Q2a)

Given code

```
tests / ... testing / ...  
1  def divide(a, b):  
2      return a / b  
3
```

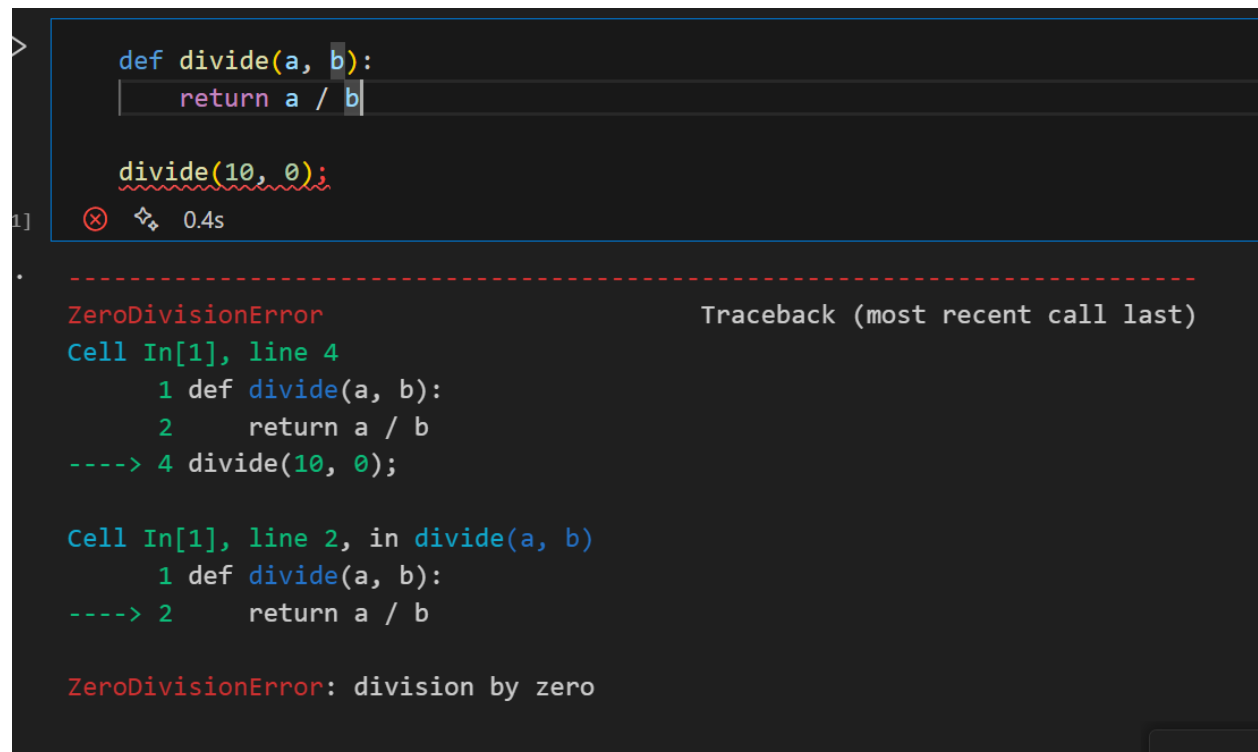
Identified Issues:

1. Division by Zero

Any number divided by zero in Python results in a `ZeroDivisionError`. The application may crash or behave strangely during runtime if the system is unable to handle this issue.

Example Error:

`divide(10, 0) → ZeroDivisionError: division by zero`



```
>  
def divide(a, b):  
    return a / b  
  
divide(10, 0);  
1] [X] 0.4s  
  
-----  
ZeroDivisionError                                Traceback (most recent call last)  
Cell In[1], line 4  
      1 def divide(a, b):  
      2     return a / b  
----> 4 divide(10, 0);  
  
Cell In[1], line 2, in divide(a, b)  
      1 def divide(a, b):  
----> 2     return a / b  
  
ZeroDivisionError: division by zero
```

Unit Testing Benefit: Writing a test case for this scenario helps identify the crash point early:


```
def test_divide_by_zero():
    try:
        divide(10, 0)
        assert False # should not reach this point
    except ZeroDivisionError:
        assert True
```

✓ 0.0s

2. Non-Numeric Input Types

Explanation: If either a or b is a non-numeric value like a string, the function raises a `TypeError`.

```
divide("10", 2)
```

✗ 0.0s

```
-----
TypeError                                Traceback (most recent call last)
Cell In[3], line 1
----> 1 divide("10", 2)

Cell In[1], line 2, in divide(a, b)
      1 def divide(a, b):
----> 2     return a / b

TypeError: unsupported operand type(s) for /: 'str' and 'int'
```

Unit Testing Benefit: A test case for type validation ensures robustness:

```
def test_divide_string_input():
    try:
        divide("10", 2)
        assert False
    except TypeError:
        assert True
```

✓ 0.0s

Improved Version of the Code:

```
def divide(a, b):
    if not isinstance(a, (int, float)) or not isinstance(b, (int, float)):
        return "Error: Both inputs must be numeric."
    if b == 0:
        return "Error: Division by zero is not allowed."
    return a / b
divide("10", 2)
27] ✓ 0.0s
.. 'Error: Both inputs must be numeric.'

>
28] ✓ 0.0s
.. 'Error: Division by zero is not allowed.'
```

Why These Changes Matter:

- Type Checking: Prevents improper data types that could crash the function.
- Zero Check: Prevents division errors before they occur.
- User Feedback: Returns friendly error messages, improving usability and debugging.
- Improved Robustness: Function is now more resilient and suitable for use in larger applications or APIs.

Q2b)

SubmitPayment()

Based on your answer, perform the System testing on that functionality.

SubmitPayment() represents a complete, observable action performed by the user (e.g., clicking a 'Pay Now' button). It involves multiple underlying components (card validation, transaction execution, confirmation receipt, and record update) working together. Other payment-related processes like card verification or balance check are internal units.

Identify Choices for SubmitPayment()

1. **Payment Info Validity:** (Are the card details correct and complete?)
2. **Funds Availability:** (Are there enough funds in the account?)

3. **User Authentication:** (Is the user logged in?)
4. **Network Connectivity:** (Is the system connected to the payment gateway?)

Identify Representative Values (Partitions)

1. **Choice: Payment Info Validity**
 - Valid_Info (normal case)
 - Invalid_Card_Number (error case)
2. **Choice: Funds Availability**
 - Sufficient_Funds (success)
 - Insufficient_Funds (failure)
3. **Choice: User Authentication**
 - Logged_In (normal)
 - Not_Logged_In (error case)

Generate Test Case Specifications

- **Specification 1 (Successful Transaction)**
 - Choices & Values:
 - Payment Info: Valid_Info
 - Funds Availability: Sufficient_Funds
 - User Authentication: Logged_In
 - (Implicit: Network Available)
 - Expected Outcome: Payment processed successfully, confirmation message shown, and receipt stored in database.
- **Specification 2 (Failed Transaction - Invalid Card Number)**
 - Choices & Values:
 - Payment Info: Invalid_Card_Number
 - Funds Availability: Sufficient_Funds
 - User Authentication: Logged_In
 - (Implicit: Network Available)
 - Expected Outcome: Payment rejected, message shown to user: "Invalid card details", and no transaction recorded.

Generate Concrete Test Cases

- **Concrete Test Case 1 (for Specification 1: Successful Transaction)**
 - **Test Input:**
 - User ID: U202501 (logged in)
 - Payment Amount: \$25.00

- Card Number: 4111 1111 1111 4652 (Visa test card)
 - Expiry: 12/26, CVV: 123
 - Balance: \$100.00
 - Time: July 21, 2025, 14:00 NDT
 - **Expected Outcome:**
 - System displays: "Payment successful. Receipt sent to your email."
 - API Response Status: 200 OK
 - Response Body: {"transactionId": "TX-98765", "status": "Success", "amount": 25.00}
 - Database: Transaction entry created for U202501 with status = "Completed"
 - Admin View: User U202501 shows payment received for \$25.00
-
- **Concrete Test Case 2 (for Specification 2: Failed Transaction - Invalid Card Number)**
 - **Test Input:**
 - User ID: U202502 (logged in)
 - Payment Amount: \$15.00
 - Card Number: 1234 5678 9012 3456 (invalid format)
 - Expiry: 11/25, CVV: 456
 - Balance: \$80.00
 - Time: July 21, 2025, 14:10 NDT
 - **Expected Outcome:**
 - System displays: "Error: Invalid card number. Please try again."
 - API Response Status: 400 Bad Request
 - Response Body: {"errorCode": "INVALID_CARD", "message": "Card number is not valid."}
 - Database: No transaction recorded for U202502
 - Admin View: No payment reflected for U202502