

**MEMORIAL UNIVERSITY OF NEWFOUNDLAND**

Faculty of Engineering

ENGI 9839

Software Verification and Validation

Dr. Raja Manzar Abbas

Project Report

Submitted by:

Oluwaseun Akinfenwa

202382513

Anesu S Makuto

202481536

Master of Science in Computer Science

Date

July 20, 2025

| **Test** | **Oluwaseun Akinfenwa** | **Anesu S Makuto** | **Total** |
| --- | --- | --- | --- |
| *Unit tests* | 3 | 2 | 5 |
| *Integration tests* | 2 | 2 | 4 |
| *System tests* | 4 | 4 | 8 |
| *Exploratory tests* | 6 | 7 | 13 |
|  |  |  | **30** |

[1. Introduction](#_kmsuawy9k5o5)

[2. Verification and Validation Strategy](#_gu9lbl7i7f9g)

[3. Unit Testing](#_tnu4e71qeshj)

[4. Integration Testing](#_5bhkg7u1fawj)

[5. System Testing](#_jz5sltbstm80)

[6. Exploratory Testing](#_vbkjyoxyubkw)

[7. Challenges and Gaps](#_pwwe4wtr1y3e)

[8. Conclusion and Reflection](#_w0uywm8nm7y6)

[9. Appendices](#_q83qe7hwoq9m)

[10. References](#_2pfo9p9btss1)

# 1. Introduction

This report covers verification and validation of the Library Management System, a Python-based desktop application built with the Tkinter GUI framework. This system’s design also includes the use of JavaScript Object Notation(JSON) files for persistent data storage. All records are serialized into JSON format and stored locally; this approach ensures data portability, human readability, and eliminates the need for a database engine, making it a suitable deployment for the Software Verification and Validation course (ENGI 9839). The Library Management System provides an interface for managing core library operations from a Librarian or Library Administrator point of view; the operations include registering books, readers, and book rentals. The key features of the Library Management System include adding and deleting readers, registering book rentals and returns, and searching books by title, author, or ISBN. The system also tracks book availability and ensures that the lending and return processes update records appropriately.

Software Verification and validation are crucial in this Library Management System, to ensure functionality which is reliable and also intended or expected. Verification checks whether the software has been implemented correctly according to its specifications[1]. For this project, Verification checks whether the Library Management System has been implemented correctly according to its specifications, while validation confirms that the Library Management System meets the needs of the user and their expectations. The verification aspect will be covered in this project as we do unit and integration tests to confirm the correctness of the internal logic of the Library Management System. The internal logic includes tracking the book copies, matching the Identification of readers to the rightful rentals. Validation will be achieved through real-world user interaction with GUI simulations.

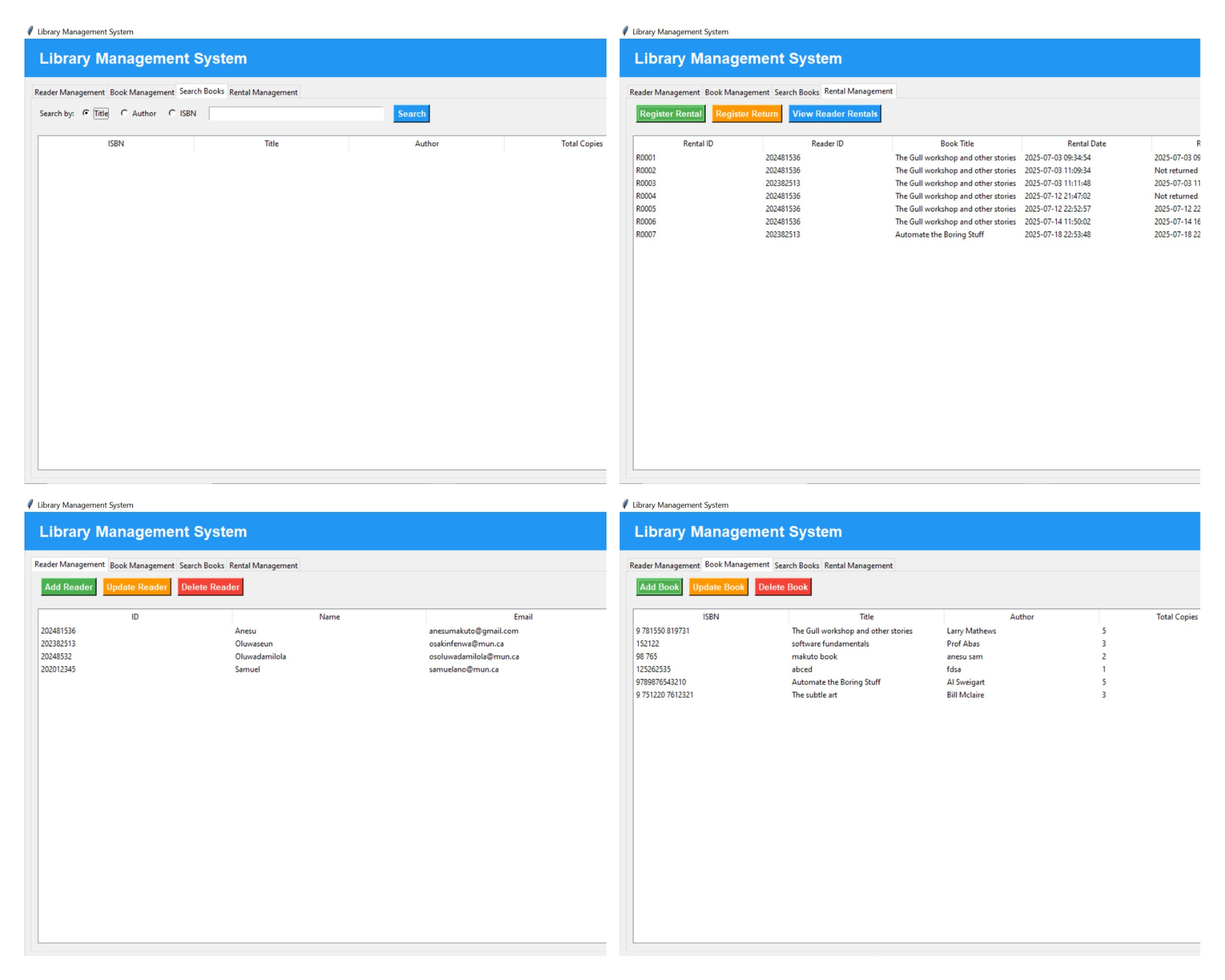
Applying unit, integration, system, and exploratory testing to the Library Management System project demonstrates the importance of Verification and Validation in building maintainable, scalable, and bug-resistant software.

The early application of testing helped detect design flaws of the Library Management System, which included tracking availability, which was inconsistent, and improper GUI element recognition. These issues were corrected by refining the logic of the Library Management System written in Python and refining automation scripts written using PyAutoGUI.

**Note:** Screenshots and Video walkthrough of the interface and tests are found on the link below in Google Drive**:**

<https://drive.google.com/drive/folders/18fvFmvACLKtwRTYeDIPmAz-o-itFzFID?usp=sharing>

This project has also been pushed to GitHub:



*Interface screenshots for the Library Management System tabs; more screenshots are on the Google Drive link*

**Steps to run the Library Management System Desktop Application and tests**

1. *Requirements*

Python 3.9+

Install dependencies

pip install pyautogui opencv-python pillow

pip install pytest

pip install unittest2

1. *Run the Library Management System.*

Navigate to the folder with the system

cd library\_management\_system

Launch the Library Management System

Python library\_app.py

1. *Running Unit and Integration Tests*

Go to the test folder and run test\_library2.py

Python test\_library2.py

1. *Run System Tests*

Ensure the desktop is open and in focus

Run any test script from the main folder

Example: python test\_view\_reader\_rental\_automation.py

# 2. Verification and Validation Strategy

We used layered testing principles for the Library Management System. This strategy is meant to verify internal correctness and validate user-visible behaviour using both manual and automated testing techniques. This layered approach ensures the Library Management System is robust, reliable, and meets its specified requirements.

**2.1 Overview of Test Layers**

The Library Management System adopted a three-layered testing strategy:

* Unit testing: This verifies individual components for correctness(Book, Reader, Rental)
* Integration testing: Ensures correct collaboration between components, such as rental and return operations.
* System testing: validates the Library Management System desktop application, simulating real interactions with the GUI via PyAutoGUI.

**2.2 Testing Environments and Dependencies**

Library Management System testing was performed in a local Python environment running Python 3.13.5 with the below listed libraries:

* Unittest: Built-in Python module for unit and integration test suites [2].
* Pyautogui: An External library used to script and automate GUI interactions, such as clicking buttons, entering text, and reading on-screen dialogues [3].

Supporting dependencies for GUI automation:

* Pillow: requirement of pyautogui, which is responsible for image handling [4].
* Pyscreeze, pygetwindow, and pymsgbox: these assist with screen element detection and window control [5] [6].

The Library Management System GUI was developed using Tkinter, Python’s standard GUI toolkit, which integrates seamlessly with both unittest and pyautogui.

**2.3 Unit and Integration Verification**

The first stage of verification involved defining a comprehensive suite of unit tests using unittest. These tests cover:

* Data serialization/deserialization via to\_dict () and .from\_dict()
* Internal object state correctness, such as book availability
* Boolean logic, such as rental return status

The second layer, integration tests, was used to confirm interactions between components. These integration tests ensure consistency between class boundaries and correctness, adding to the verification logic’s core.

An example of this:

* When a rental is created, it must link to both a book and a reader.
* Renting and returning a book should correctly adjust the book’s available copies.

**2.4 System Testing and Automation**

To validate the Librarian workflow end-to-end, system testing was implemented using Pyautogui. These system tests simulate a human user, specifically a librarian, performing real Library Administration interactions such as:

* Logging in to the Library Management System with the admin credentials
* Adding a reader under the Reader Management Tab
* Adding a book under the Book Management Tab
* Searching book by Title, Author, or ISBN under the Search Books Tab
* Registering a book rental under the Rental Management Tab
* Registering a book return under the Rental Management Tab
* View Reader Rentals under the Rental Management Tab

These tests are fully automated; they rely on pixel-based image recognition for the detection of buttons and fields.

Each test script used for system testing includes the following:

* A wait window, mostly 5 seconds, for the user to focus on the application
* Screenshot labelled, corresponding to UI elements
* Keyboard input and mouse clicks, for example, moving to the next text box in some instances involved automatically pressing the tab button to move to the next field and in some cases, mouse clicks to select the tab just like the Librarian would do.
* Assertion of success dialogs, for example, confirming a book has been added successfully or a rental has been added successfully.

This approach allowed us to repeatedly validate the key workflow of the Librarian, ensuring the Library Management System behaves correctly from a Librarian Administration perspective.

**2.5 JSON-Based Data Validation**

A core component of the Library Management System is its reliance on JSON files to persist data for books, readers, and rental transactions. Instead of using a relational database, LMS serializes in-memory data structures to JSON format. JSON is lightweight, human-readable, and well-suited for small-scale desktop deployment.

To ensure data integrity during this serialization process, the system employs custom to\_dict() and from\_dict() methods in all major model classes: Book, Reader, and Rental. These methods define the exact structure used to store and retrieve data from the JSON files. Verification of these conversions is handled through dedicated unit tests that assert the equivalence of original objects and their reconstructed versions after JSON round-trip conversion.[9]

For example, the test method test\_book\_to\_from\_dict() confirms that a Book instance is serialized using to\_dict() and deserialized back using from\_dict(), retaining the correct title and available copies.

Similar tests are written for Reader and Rental, ensuring that fields like email, rental\_date, and is\_returned are preserved accurately.

This verification strategy guarantees that any data stored in JSON will maintain its structural and semantic integrity upon reloading, which is crucial for consistent functionality across sessions. Additionally, integration tests such as test\_register\_and\_return\_flow() simulate changes to the available copies of a book during rental and return flows, validating not just in-memory logic but also the underlying assumptions that will be reflected in persisted JSON data.

Although explicit JSON file writing and reading are handled elsewhere in the system, these tests function as a reliable proxy by confirming that all data transformations are reversible and accurate, thus contributing to the system’s validation strategy.

This aligns well with early-stage, file-based system design and fulfills key verification principles by ensuring that the core data handling logic is robust, testable, and maintainable.

# 3. Unit Testing

The Library Management System implements unit testing as a foundational quality assurance practice to ensure correctness and reliability in its core logic. These unit tests were written using Python’s built-in unittest framework. Python’s built-in unittest is well-suited for validating isolated functionality without external dependencies such as the GUI [2] [7].

def test\_book\_to\_from\_dict(self):

book = Book("123", "Test Book", "Author A", 3)

book.available\_copies = 2

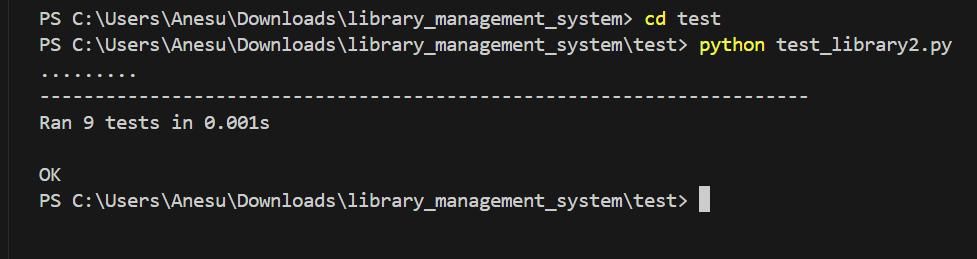
book\_dict = book.to\_dict()

book2 = Book.from\_dict(book\_dict)

self.assertEqual(book2.title, "Test Book")

self.assertEqual(book2.available\_copies, 2)

*Listing 1: Sample unit test case logic; The complete test suite is available via the appendix link to Google Drive.*

*Listing 2: The above terminal output confirms that all 9 unit and integration test cases executed without errors, indicating the correctness of the core system components*

**3.1 Scope and Purpose**

Unit tests in the Library Management System project focused on three core classes: Book, Reader, and Rental. These classes form the model layer of the Library Management System application, responsible for representing and managing domain-specific data. The purpose of the unit tests is to verify the behaviour of each class, that it performs as it is expected in isolation, particularly in terms of:

* Object creation and attribute assignment
* Serialization and deserialization using JSON-compatible dictionaries
* Library rules, such as book availability and return status

This ensures that critical methods like to\_dict() and from\_dict() consistently preserve the state of objects across sessions, which is crucial given the Library Management System’s use of JSON files for data persistence.

**3.2 Sample Test Cases and Logic**

The TestLibraryUnit test class includes the following targeted tests:

* test\_book\_to\_from\_dict() checks that a Book object retains its title and availability after being serialized to a dictionary and re-instantiated.
* test\_reader\_to\_from\_dict() ensures that Reader objects preserve personal details like email address.
* test\_rental\_to\_from\_dict() and test\_rental\_return\_status() validate that the Rental object tracks whether a book has been returned, based on whether a return date is present
* Test\_book\_initial\_availability confirms that a new book starts with the correct number of available copies.

**3.3 Effectiveness and Coverage**

The current unit tests provide coverage for the core object lifecycle and primary behaviour within the data model. These include :

* Class: Book

Method Tested: to\_dict(), from\_dict()

Assertion Goal: Preserve attributes, especially availability

* Class: Reader

Method Tested: to\_dict(), from\_dict()

Assertion Goal: Retain identity and contact details

* Class: Rental

Method Tested: to\_dict(), from\_dict()

Assertion Goal: Track rental-return status

A manual code review has been provided to confirm that class methods with high-level impact were covered since we did not make use of test coverage tools like [coverage.py](http://coverage.py).

**3.4 Strengths and Limitations of Unit Tests**

**Pros**

* These unit tests ensure fast feedback cycles
* The test logic is straightforward, repeatable, and does not depend on Librarian Admin’s input or GUI state.
* Unit tests offer base-level confidence, therefore complement integration tests.

**Cons**

* These unit tests do not test the interaction with the Tkinter interface.
* These unit tests assumed the LMS file I/O logic worked correctly based on the to\_dict() output, but they were not tested explicitly.
* External schema validation could strengthen the correctness guarantees.

# 4. Integration Testing

Integration testing in the Library Management System project was implemented using Python’s unittest framework and focuses on verifying the interactions between different components of the Library Management System, specifically the Book, Reader, and Rental classes.

Integration tests evaluate how classes work together during realistic workflows such as borrowing and returning books, whereas unit tests ensure that each class functions correctly in isolation.

Integration tests simulate application logic without involving the GUI, ensuring correctness of the core model before testing system-level behaviour.

def test\_register\_and\_return\_flow(self):

self.book.available\_copies -= 1

self.assertEqual(self.book.available\_copies, 1)

self.rental.return\_date = "2025-07-11 10:00:00"

self.rental.is\_returned = True

self.book.available\_copies += 1

self.assertEqual(self.book.available\_copies, 2)

*Listing 3: Sample integration test case logic for rental registration and return flow; The complete test suite is available via the appendix link to Google Drive*

**4.1 Purpose and Coverage**

The goal of integration tests is to confirm that the Library Management System behaves correctly when its components are combined, for instance, when a reader borrows a book, the availability of the book should decrease, and when the book is returned, the availability should be restored. These operations span multiple objects and represent typical user interactions.

**4.2 Integration Test Scenarios**

These tests are defined in the TestLibraryIntegration class. These tests verify :

* Test Case: test\_rent\_book\_decreases\_availability()

Description: Simulates borrowing a book by decrementing available\_copies

* Test Case: test\_return\_book\_increases\_availability()

Description: Simulates returning a book and increasing its availability count

* Test Case: test\_rental\_reader\_book\_linking()

Description: Validates that the Rental object correctly references the intended Reader and Book

* Test Case: test\_register\_and\_return\_flow()

Description: Simulates a full rental-return transaction, adjusting book availability accordingly.

**4.3 Key Differences from Unit Tests**

* Complexity: Integration tests involve multiple classes interacting together, while unit tests isolate a single class.
* Flow-based testing; Integration tests simulate real-world user operations, while unit tests verify specific method behaviour.
* Cross-class validation: Tests like test\_rental\_reader\_book\_linking() validate logical relationships between entities, not just their internal states.

**4.4 Tool and Setup**

The Integration tests were run using Python’s built-in unittest module. The setUp() method was used to initialize a Book, Reader, and Rental object for each test, ensuring consistent setup without repetition.

def setUp(self):

self.book = Book("456", "Integration Book", "Author B", 2)

self.reader = Reader("R2", "Bob", "bob@example.com", "987654")

self.rental = Rental("RENT003", self.reader.reader\_id, self.book.isbn, "2025-07-10 12:00:00")

This facilitates realistic test scenarios where shared objects interact through logical flows.

**4.5 Strengths and Weaknesses of Integration Testing using Unittest**

**Pros**

* Integration testing simulates realistic use cases by combining multiple components
* Integration testing helps uncover interface-level bugs
* It ensures coordinated logic across modules
* Integration tests use setup() to avoid repetition, improving maintainability
* Offers efficient verification of rules, like proper tracking of borrowed and returned books
* Integration testing lays the groundwork for full workflows needed in the Library Management System tests

**Cons**

* Integration tests do not capture full user interactions through the Library Mnaagement System GUI
* It might miss an isolated internal logic error that unit tests are better suited for
* Integration tests require a more complex setup and teardown of test objects.
* Code can become cluttered in integration tests if too many components are tested together without clear boundaries
* Integration tests do not validate persistence unless explicitly added.
* As integration complexity grows, tests may become harder to isolate and debug.

# 5. System Testing

System testing for the Library Management System was conducted using the pyautogui library, which automates GUI interactions by simulating real user behaviour through mouse movements, keyboard inputs, and image-based component recognition. This type of testing evaluates the application as a whole and ensures that the integrated components, from login interfaces to data submission forms, function correctly when operated as a complete system.

# Click "Search Books" tab

if not locate\_and\_click('search\_books\_tab.png', "Search Books tab"):

sys.exit()

# Click "Title" radio button

if not locate\_and\_click('search\_by\_title\_radio.png', "Title radio button"):

sys.exit()

# Click search input box

search\_box = locate\_and\_click('search\_input\_box.png', "Search input box")

if not search\_box:

sys.exit()

# Type book title

pyautogui.write("The Gull workshop and other stories", interval=0.05)

print("Entered book title.")

time.sleep(0.5)

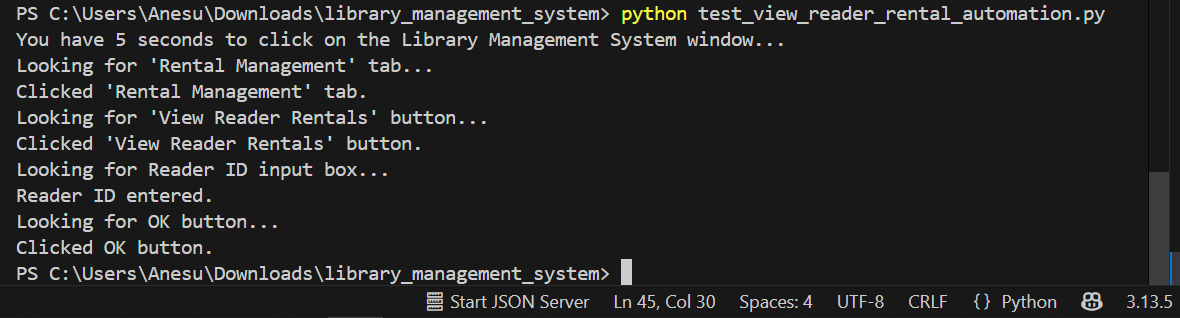
# Click Search button

if not locate\_and\_click('search\_book\_button.png', "Search button"):

sys.exit()

print("Search Book by Title test completed successfully.")

*Listing 4: Sample system test: Search book by title; The complete test suite is available via the appendix link to Google Drive*

**

*The script test\_view\_reader\_rental\_automation.py simulates a system-level interaction where the user navigates to the Rental Management tab, enters a Reader ID, and views the reader’s rental history. The successful output confirms that the GUI automation performed the intended operations correctly*

**Tools and Setup**

The system tests were implemented as standalone Python scripts and executed in the presence of the running desktop Library Management System. Each script starts with a delay (time.sleep(5)) to allow the tester to manually focus the Library Management System window, after which automation begins. The tests rely on pre-captured GUI elements, such as add\_reader\_button.png, book\_management\_tab.png, for precise interaction with UI components. These images must match the current state of the UI; therefore, they are stored in a local directory and referenced within each script using pyautogui.locateCenterOnScreen() with a confidence threshold of 0.8 and, in some cases, 0.72 to ensure that the recognition is robust.

**System Test Scenarios Covered**

1. Login and Logout Testing

The test\_login\_automation.py script verifies a successful user login using valid credentials(admin, password123). The login form is filled by simulating keystrokes and pressing Enter to log in. The test\_logout\_automation.py script confirms that the user can terminate the session by locating and clicking the “Logout” button. These scripts ensure that basic access control flows are working properly.

1. Reader Management

The test\_add\_reader\_automation.py script automates the addition of a new reader through the UI form. It locates and clicks the "Add Reader" button, enters test data (e.g., R100, John Doe), and confirms successful submission. The test\_delete\_reader.py script selects the newly added reader from a row identified by an image (john\_doe\_row.png) and proceeds to delete the reader by confirming dialogs. These two scripts test both (Create, Read, Update, and Delete) CRUD and modal dialog functionalities.

1. Book Management

The test\_add\_book\_automation.py script automates the addition of a new book. It logs in, navigates to the Book Management tab, clicks "Add Book", fills in book details (ISBN, title, author, copies), and then confirms the operation. The test concludes by verifying the success dialog. This test validates that the book module handles data entry correctly and reflects changes in the UI.

1. Rental Operations

The test\_register\_rental\_automation.py script simulates the process of renting a book by inputting the Reader ID and ISBN into the appropriate fields, then clicking the "Register" button. The test\_register\_return\_automation.py script automates the return of a previously registered rental. These scripts test core application features that directly affect the available book copies and rental tracking.

1. Viewing Rentals by Reader

The test\_view\_reader\_rentals.py script allows system testers to verify if the rental history retrieval is functioning. After navigating to the Rental Management section and selecting “View Reader Rentals”, it inputs the reader ID and confirms that the list is generated

**Pros of PyAutoGUI for System Testing**

* It is realistic, the simulation closely mirrors the Librarian's interaction with the Library Management System, testing what an end-user would experience.
* It gives a validation of the complete flow of actions from login, navigation, form input, and confirmation of actions.
* Failures during the test are visible in real time, which allows quick debugging or re-capturing of UI elements which might have been missed.
* Pyautogui gives a high level of flexibility, which means that scripts can be easily edited and reused for various test scenarios.

**Cons of PyAutoGUI for System Testing**

* During the testing and scripting, pyautogui’s dependency on images was a disadvantage because it relied on images and changes in resolution or scaling would break image matching, requiring recapturing because, in some cases, it would not move to the desired text box to click there due to not seeing the textbox.
* The second limitation was that pyautogui requires the GUI to be in focus and the system to be in focus, and the system to be idle during execution.
* The scripts written in pyautogui rely on print() statements rather than programmatic assertions, which reduces automated pass or fail clarity.

**Library Management System Testing using the Category Partition Method**

Target function: search book by title

*A. Identify Choices (Input Variables and Environmental Factors)*

1. Input Variable: UI Navigation

* Search Tab Visibility
  + Visible\_Tab
  + Missing\_Tab
* Search Mode
  + Title\_Selected
  + Other\_Selected
  + None\_Selected

2. Input Variable: Search Field

* Field Visibility
  + Field\_Found
  + Field\_Not\_Found
* Field Editability
  + Editable
  + Not\_Editable

**3**. Input Variable: Search Text

* + Text Validity
  + Valid\_Existing\_Title
  + Non\_Existent\_Title
  + Empty\_String
  + Invalid\_Format (e.g., too long, symbols)

4. Search Button State

* Button Visibility
  + Found\_Button
  + Not\_Found
* Button Functionality
  + Clickable
  + Unclickable

5. Environmental Factor: App/UI State

* UI Responsiveness
  + Responsive
  + Frozen
  + Crashing
* Book Database State
  + Book\_Exists
  + Book\_Not\_Exists

*B. Categories and Partitions*

Category: Search\_Tab

Partitions: Visible\_Tab, Missing\_Tab

Category: Search\_Mode

Partitions: Title\_Selected, Other\_Selected, None\_Selected

Category: Search\_Field

Partitions: Field\_Found, Field\_Not\_Found

Category: Field\_Editability

Partitions: Editable, Not\_Editable

Category: Search\_Text

Partitions: Valid\_Existing\_Title, Non\_Existent\_Title, Empty\_String, Invalid\_Format

Category: Search\_Button

Partitions: Found\_Button, Not\_Found

Category: Button\_Clickable

Partitions: Clickable, Unclickable

Category: UI\_State

Partitions: Responsive, Frozen, Crashed

Category: Book\_DB\_State

Partitions: Books\_Exists, Book\_Not\_Exists

*C. Impose constraints*

Error Constraints

* If Search\_Tab = Missing\_Tab, then all other actions are irrelevant system should fail immediately.
* If Search\_Mode ≠ Title\_Selected, the system must throw “Please select a valid search mode.”
* If Search\_Field = Field\_Not\_Found or Field\_Editability = Not\_Editable, the user cannot input; then terminate.
* If Search\_Button = Not\_Found or Button\_Clickable = Unclickable, search cannot proceed.
* If Search\_Text = Empty\_String, the system should validate and reject input early.
* If UI\_State ≠ Responsive, no guarantees on behaviour system should fail.
* If Search\_Text = Valid\_Existing\_Title but Book\_DB\_State = Book\_Not\_Exists, the system should return “Not Found.”

Single Constraints

* Test Search\_Text = Empty\_String at least once.
* Test UI\_State = Frozen or Crashed at least once.
* Test Search\_Mode = None\_Selected at least once.

# 6. Exploratory Testing

Exploratory testing was employed alongside structured automation to identify unpredictable behaviours and UI limitations within the Library Management System. Exploratory testing was unscripted and reactive; this helped uncover real-time interface glitches and usability concerns that automation alone could not fully capture.

**Note**: Exploratory testing sessions for various scenarios listed below were screen-recorded to demonstrate actual user interactions with the system. These recordings are available in the Google Drive folder linked below for reference and validation purposes.

Google Drive Link:

**6.1 Purpose and Approach**

The aim of the exploratory testing was to simulate user interaction to evaluate how the Library Management System behaves under varied conditions, such as timing variations, partial inputs, missing UI elements, or rendering delays. This was done primarily by running automation scripts, manually observing their failure points, and experimenting with solutions in real-time.

For each critical operation, exploratory testing was conducted after running automated PyAutoGUI scripts.

**Exploratory Test Session illustration:** Login Module  
 Tour Type: Business District Tour (essential usage path & basic input behaviour)

1. Mission and Charter

* Mission: Assess the Library Management system’s robustness and usability in handling incorrect login attempts and input validation.
* Charter:
  + Enter incomplete or incorrect credentials
  + Observe how errors are handled
  + Verify dialogs provide meaningful and secure feedback
  + Simulate both human error and malicious input attempts

2. Actions Performed and Observations

| Action | Expected Result | Observed Behavior |
| --- | --- | --- |
| Entered username admin and left password blank | Error message or validation prompt | Modaldialog:"Invalid credentials!" with OK button |
| Entered the correct password, but incorrect username | Error message, no access | Modaldialog:"Invalid credentials!" |
| Clicked OK on dialog | Dialog dismissed, login screen retained | Worked as expected |
| Attempted to log in again without refreshing | System remains responsive and resets the field focus | Ready for new input |

3. Key Exploratory Testing Insights

* Validation: Login form handles missing fields gracefully via modal alerts.
* Security: Generic error message "Invalid credentials" protects against credential enumeration (i.e., doesn’t reveal which field is wrong).
* Usability: The Modal is clear, but automation tests need to account for dismissing the modal before retrying.

4. Defects & Recommendations

* Pass: Proper modal display and rejection of incomplete credentials.
* Suggestion: Include inline field-level hints before submission (e.g., "Password required").

*Login error handling exploratory test*

An exploratory test was conducted to verify how the system handles incorrect login attempts. The tester entered the username admin but left the password field blank, then clicked the Login button. The system responded by displaying a modal error dialog with the message "Invalid credentials!" and an OK button.

This confirms that the application correctly detects invalid login input and prevents unauthorized access. While the generic message maintains security best practices, automation scripts should be configured to detect and dismiss the dialog before proceeding. The same test was also done, entering a correct password but an incorrect username, and it also gave an output that incorrect credentials.

*Search Books: Exploratory Testing*

Exploratory testing was carried out on the Search Books module to evaluate how effectively the system handles various search queries and edge cases. The feature allows users to search for books by Title, Author, or ISBN. From the exploratory tests, the following behaviours were observed;

*Successful Search by Title:*  
 Searching for the partial keyword "gull" under the Title option returned the correct book, “The Gull Workshop and Other Stories” by Larry Mathews. This indicates that the system supports partial text matches when using the Title field.

*Failed Partial Search by ISBN:*  
 Inputting "97" under ISBN returned no results, even though a book with an ISBN starting with "97" exists. This reveals a limitation: the ISBN field appears to require exact matches, with no support for partial input, which reduces flexibility.

*Failed Partial Title Match:*  
 Entering "makuto" under Title yielded no results, although the book “makuto book” is in the system. This discrepancy may suggest case sensitivity, unexpected whitespace, or an indexing issue that prevents accurate matching.

*Successful Full ISBN Match:*  
 When the full ISBN "98 765" was entered, the correct book “makuto book” was successfully retrieved. This confirms that full-match searches are functioning properly.

*Clear Feedback Dialogs:*  
 In each search case, successful or not, the system consistently displayed appropriate pop-up dialogs such as "Found 1 book(s)" or "No books found matching your search criteria". These dialogs enhance usability by confirming the system response.

The exploratory tests on the Search Books tab confirmed that the search logic works reliably under exact match conditions, but it highlighted limitations in partial matching, to be particular, for ISBNs and some titles, this may need refinement for a better Library Management System Admin experience.

*Add Reader Form Accessibility exploratory test*

The Add Reader dialog was triggered from the Reader Management section. The test assessed the form’s readiness for user interaction.

The form displayed all expected fields: Reader ID, Name, Email, and Phone, along with a visible Save button. Navigation between fields using the Tab key was fluid and responsive, indicating proper focus handling. No validation errors appeared when the fields were left blank, suggesting that field-level validation might be absent or deferred until submission.

*Book Management Operations exploratory testing*

We did an exploratory test on the Book Management module, focusing on adding, updating, and displaying book records. The goal was to explore how the system responded to typical user behaviour and edge cases in an unscripted manner.

Test scenarios explored included launching the Add Book form and testing blank field submission handling, updating an existing book’s title to test form pre-fill accuracy and record persistence. We also observed the dynamic update of the book list after add or update actions. The Library Management System successfully launched dialogs for both adding and updating books, the book list refresh was immediate after each action, confirming data binding worked correctly, lastly, no field-level validation warnings were observed during blank submissions, suggesting input validation can be improved.

*Rental Management exploratory testing*

We performed an exploratory test on the Rental Management features of the Library Management System, including the Register Rental, Register Return, and View Reader Rentals functionalities. The goal of these exploratory tests was to identify any usability issues, validation gaps, or unexpected behaviours without using formal test scripts.

*The key observations included:*

*Register Rental*

Attempting to rent a book using a partial or non-existent Reader ID triggered an appropriate error message (“Reader not found”), confirming input validation is in place. The system correctly prevents rental registration when invalid input is entered, reducing the risk of incorrect data entry.

*Register Return*  
The return process is initiated by entering a valid Rental ID. The field allows only existing rental IDs to proceed, ensuring proper linkage between rentals and returns.  
Input flexibility and error prevention were confirmed through manual testing with both valid and invalid IDs.

*View Reader Rentals*  
A prompt for Reader ID enables users to filter rental history by reader. When a correct ID is entered, relevant rentals are listed. This feature provides clear and efficient access to reader-specific activity, which is helpful for staff managing overdue returns.

# 7. Challenges and Gaps

During the development and testing of the Library Management System, several challenges and limitations were encountered. These issues highlight areas where the current implementation may be improved in future iterations, particularly in test robustness, maintainability, and automation tooling.

**7.1 GUI Automation Limitations**

*Screen Resolution Dependency*  
 PyAutoGUI relies heavily on image recognition to interact with GUI elements. This approach is inherently sensitive to screen resolution, scaling settings, and window positions. For example, a test that passes might fail if the pixel dimensions differ. This led to inconsistency in the system testing of the Library Management System.

*Lack of Headless Automation*  
PyAutoGUI requires the Library Management System to be in the foreground during execution and does not support headless mode. This limits the ability to run system tests in parallel or in the background, which is a key feature in modern CI/CD pipelines. During testing, we had to avoid interacting with the machine to prevent accidental interference with automated mouse and keyboard actions.

*Image Capture Maintenance Overhead*  
UI changes, such as repositioning buttons or updating icon colours, led to existing image-based scripts' failure, requiring new screenshots to be captured and stored. Duplicates can be found in the folder for the project.

**7.2 Testing Tooling and Coverage Gaps**

*No Formal Coverage Metrics*  
 Although unit and integration tests were written using unittest, no formal code coverage tools (such as coverage.py) were integrated to quantify test completeness. As a result, test quality was validated through manual code review rather than measurable statistics. This limited our ability to identify untested edge cases or logic paths.

*No Continuous Integration (CI) Pipeline*  
 The testing process was entirely local and manual. The project lacked automated pipelines (e.g., Jenkins). This gap in CI automation reduces the scalability of the test process and increases the likelihood of regression errors being introduced unknowingly.

*No System Test Assertions*  
PyAutoGUI-based tests relied primarily on print() statements for status updates and manual observation of success/failure. This limits automated pass/fail detection and makes it difficult to generate meaningful test reports. A more robust validation mechanism, such as screenshot comparison or state verification, would improve automation reliability.

**7.3 Validation and UX Limitations**

*Lack of Field-Level Validation*  
Exploratory testing revealed that many input forms, such as "Add Reader" and "Add Book", accepted blank submissions or invalid formats without showing warnings. This points to a gap in field-level validation, which could lead to inconsistent or incorrect data being saved into the system.

*Rigid Search Functionality*  
Tests on the Search Books module revealed that partial matches did not always return results, especially in the case of ISBNs. This reduces search flexibility for librarians and may lead to a poor user experience if exact inputs are not known. Enhancing partial and case-insensitive search would improve usability.

*Manual Data Setup for Tests*  
Both integration and system tests required test data to be manually seeded or removed between runs. This created repetition and made it difficult to test edge cases at scale. A future improvement would be to integrate test fixtures or a temporary sandbox data layer to simulate various scenarios programmatically.

# 8. Conclusion and Reflection

Through a layered testing methodology that included unit tests, integration tests, system tests using PyAutoGUI, and exploratory testing, the reliability and functionality of the Library Management System significantly improved. The methodology allowed us to achieve a high degree of confidence in the Library Management system’s correctness and usability, improving maintainability and reliability for future use. The development and testing of the Library Management System served as a comprehensive practical application of software verification and validation principles learned in Software Verification and Validation ( ENGI 9839). By systematically applying unit, integration, system, and exploratory testing, we verified the correctness of our core logic and validated real-world usability through GUI simulations.

The use of unittest provided structured and repeatable verification of object behaviour, while PyAutoGUI enabled realistic validation of librarian admin workflows. Exploratory testing uncovered critical gaps in field validation and search flexibility that scripted tests alone could not detect. These insights reinforced the value of combining automated and human-centred testing approaches.

This Library Management System project emphasized the importance of early and continuous testing in the software development lifecycle. It highlighted the strengths of lightweight tools in rapid prototyping environments while revealing the challenges of GUI automation and data integrity management.

Overall, this experience deepened our understanding of building testable, maintainable software systems and strengthened our skills in designing effective V&V strategies. It has prepared us to approach future software projects with a verification-first mindset and a practical appreciation for real-world system behaviour.

# 9. Appendices

*Appendix A – Video Demonstrations*

Walkthrough videos of system testing and GUI interactions can be accessed via:

<https://drive.google.com/drive/folders/1DocVQ40tvGgBRA4eMYDy3nnDCpuM5FYH?usp=sharing>

*Appendix B – Screenshots*

Screenshots of tests can be accessed via:

<https://drive.google.com/drive/folders/1FH9QyimOjf1-W9mYatl-3qpzRSRdRhG8?usp=sharing>

*Appendix C – Source Code Repository*

The complete source code for the Library Management System, including unit, integration, and system testing scripts, is available on:

GitHub

<https://github.com/AnesuMakuto1201/ENGI-9839-LMS-202382513-202481536->

Google-Drive: <https://drive.google.com/drive/folders/1BW5R32n3kyx_8Tkp0Er2kIqZOTBb_oll?usp=drive_link>

# 10. References

[1] R. M. Abbas, *Software Verification and Validation*, Lecture Notes, ENGI 9839, Faculty of Engineering, Memorial University of Newfoundland, St. John’s, NL, Canada, 2025.

[2] Python Software Foundation, “unittest — Unit Testing Framework.” [Online]. Available:<https://docs.python.org/3/library/unittest.html>. [Accessed: Jul. 14, 2025].

[3] AI. Sweigart, “PyAutoGUI Documentation.” [Online]. Available:<https://pyautogui.readthedocs.io>. [Accessed: Jul. 14, 2025].

[4] The Pillow Project Contributors, “Pillow (PIL Fork) Documentation.” [Online]. Available:<https://pillow.readthedocs.io/>. [Accessed: Jul. 19, 2025].

[5] AI. Sweigart, “PyScreeze Documentation.” [Online]. Available:<https://pypi.org/project/PyScreeze/>. [Accessed: Jul. 19, 2025].

[6] Python Software Foundation, “Python Documentation.” [Online]. Available:<https://docs.python.org/>. [Accessed: Jul. 14, 2025].

[7] AI. Sweigart, “PyGetWindow Documentation.” [Online]. Available:<https://pypi.org/project/PyGetWindow/>. [Accessed: Jul. 19, 2025].

[8] Python Software Foundation, “json — JSON encoder and decoder.” [Online]. Available:<https://docs.python.org/3/library/json.html>. [Accessed: Jul. 19, 2025].

[9] Python Software Foundation, “Tkinter GUI Programming.” [Online]. Available:<https://docs.python.org/3/library/tkinter.html>. [Accessed: Jul. 12, 2025].

[1] R. M. Abbas, *Software Verification and Validation*, Lecture Notes, ENGI 9839, Faculty of Engineering, Memorial University of Newfoundland, St. John’s, NL, Canada, 2025.

[2] Python Software Foundation, *unittest — Unit Testing Framework*. [Online]. Available:<https://docs.python.org/3/library/unittest.html>. [Accessed: Jul. 14, 2025].

[3] Al Sweigart, *PyAutoGUI Documentation*. [Online]. Available:<https://pyautogui.readthedocs.io>. [Accessed: Jul. 14, 2025].

[4] The Pillow Project Contributors, *Pillow (PIL Fork) Documentation*. [Online]. Available:<https://pillow.readthedocs.io/>. [Accessed: Jul. 19, 2025].

[5] Al Sweigart, *PyScreeze Documentation*. [Online]. Available:<https://pypi.org/project/PyScreeze/>. [Accessed: Jul. 19, 2025].

[6] Python Software Foundation, *Python Documentation*. [Online]. Available:<https://docs.python.org/>. [Accessed: Jul. 14, 2025].

[7] Al Sweigart, *PyGetWindow Documentation*. [Online]. Available:<https://pypi.org/project/PyGetWindow/>. [Accessed: Jul. 19, 2025].

[8] Python Software Foundation, *json — JSON encoder and decoder*. [Online]. Available:<https://docs.python.org/3/library/json.html>. [Accessed: Jul. 19, 2025].

[9] Python Software Foundation, *Tkinter GUI Programming*. [Online]. Available:<https://docs.python.org/3/library/tkinter.html>. [Accessed: Jul. 12, 2025].