COMP 4004 A/B – Fall 2025

Assignment 2 — BDD Acceptance Testing

Due: November 14, 2025, at 23:59 on Brightspace

Overview

Assignment 2 builds directly on **Assignment 1**, using the same GitHub repository. This assignment reuses the codebase from Assignment 1. Do not rewrite your system; extend it by adding Cucumber feature and step-definition files that call the same business-logic methods you tested with JUnit.

It has two learning objectives:

1. Learn to use **Cucumber** to implement and execute acceptance tests.
2. Understand the challenge of **test-data design** and choosing data that makes an acceptance test executable and verifiable.

Important Testing Clarification

Key Requirement

Your acceptance tests must call actual system methods directly, not test through the UI layer.

Language Clarification

The scenario descriptions use terms like "logs in", "logs out", and "selects Return Book" to describe the business flow in familiar terms. In your implementation:

- "logs in/logs out" = create/end user session in your system
- "selects Return Book" = invoke the return operation
- "sees" or "system shows" = your test verifies the business state
- "notified" = notification exists in system state (not UI display)

Your step definitions should invoke your library system's business logic methods directly and verify state changes, not parse console output or simulate menu navigation.

Example Approach
Instead of (as a Gherkin):
Then the system displays "Maximum borrowing limit reached"

Write (as a Gherkin):
Then the borrow operation fails due to borrowing limit
And "alice" still has exactly 3 books borrowed

Your assertions should verify that the business rule was enforced, not what text appeared on screen.

System Method Calls Consistency

Your Cucumber step definitions should invoke the same public methods that your Assignment 1 tests exercised (for example, borrowBook(), placeHold(), or returnBook()).
This ensures your BDD scenarios validate business logic directly, rather than testing UI strings or console text output.

Before You Begin

Do not attempt A2 until your A1 library system works correctly for all core use cases referenced below. In other words, you will need to complete your A1 library code. No starter code will be provided.

What You Must Do

1. Learn Cucumber

- Watch the tutorial posted by a former TA and try a small example of your own. https://www.youtube.com/playlist?list=PLSeNqF8cC2igPVa7BzUR9cmN5oh7q7wFN
- Cucumber.io website has examples, tutorials and documentation.
- Make sure you can run multiple scenarios successfully before starting the assignment tests.

2. Design Test Data

- Study each acceptance test below.
- Decide which users, books, and borrowing dates to use for each scenario so that the test can run and pass.

- Use parameterized values (e.g., <user>, <book>, <password>) in your Gherkin steps to make tests reusable and easier to maintain.
- **Important**: You **<u>must</u>** use the provided users and books, but choose different combinations from your classmates to ensure unique test cases.

3. Implement Step Definitions

- Code all Cucumber step definitions to call actual library system methods directly.
- Each **THEN** step must include appropriate assertions for the business state.
- Keep your step definitions modular, reusable, and parameterized.

4. Structure of Scenarios

- A single Cucumber **scenario** may contain multiple **`Given`, When, and Then** steps representing a sequence of interactions.
- Each step describes a specific system action or expected result as the state evolves.
- For example, one scenario might include several actions by the same user, with outcomes verified after each, such as borrowing additional items one at a time and checking due dates.
  Design your steps to clearly express intent and follow the **Given – When – Then** pattern repeatedly where appropriate, not just once.

Tip: Think of a scenario as a coherent story of user behaviour testing specific business rules.

5. Run Your Suite

- Place all scenarios in **one feature file**.
- Use **one step-definition file**.
- Run the entire suite in a single execution to confirm all scenarios pass.

Initial System Setup

Initialize your system with the following **baseline data**. It will serve as the source for your parameterized test values.

Users

Username	Password
alice	pass123
bob	pass456
charlie	pass789

Books (20 total)

The Great Gatsby, *To Kill a Mockingbird*, *1984*, *Pride and Prejudice*, *The Hobbit*, *Harry Potter*, *The Catcher in the Rye*, *Animal Farm*, *Lord of the Flies*, *Jane Eyre*, *Wuthering Heights*, *Moby Dick*, *The Odyssey*, *Hamlet*, *War and Peace*, *The Divine Comedy*, *Crime and Punishment*, *Don Quixote*, *The Iliad*, *Ulysses*.

All acceptance tests must draw users and books from this baseline set, but choose different combinations or sequences for each test to make your work unique.

General Guidelines

- Acceptance tests must **call actual system methods**, not simulate console I/O.
- Menu navigation can be **inferred,** and you do not need to script the menu prompts.
- Use **Cucumber parameters, Data Tables, and Scenario Outlines** to keep steps concise and reusable.
- Assert on business state, not UI presentation.

Acceptance Tests to Develop

1. A1_scenario (10 %)

Scenario 1 corresponds roughly to A1's A-TEST-01 but must be rewritten as a human-readable feature.

Test the basic borrow-return cycle with two users and one book, demonstrating that:

- A borrowed book becomes unavailable to other users
- A returned book becomes available again

- Only one user can have a book at a time

Assert: Availability changes and borrow attempts behave correctly.

---

2. multiple_holds_queue_processing (40 %)

Test the hold queue system with three users competing for the same book, demonstrating that:

- Users can place holds on unavailable books
- Hold queue follows FIFO ordering
- Notifications are sent to the correct user when book becomes available
- Only the notified user can borrow the reserved book
- Queue advances properly when reserved books are borrowed or returned

Assert: Hold-queue order and notifications are correct.

---

3. borrowing_limit_and_hold_interactions (40 %)

Test the interaction between borrowing limits and holds, demonstrating that:

- Users cannot exceed the 3-book borrowing limit
- Users can place holds even when at the borrowing limit (i.e., when they already have 3 books borrowed)
- When a user at the borrowing limit returns a book, they drop below the limit
- If that user is next in the hold queue for a book, they receive a notification that the book is now available for them to borrow

Assert: Users can't exceed 3 books, can place holds at the limit, gain borrowing capacity after returns, and receive notifications when their held books are returned by others (regardless of their current borrowing capacity).

---

4. no_books_borrowed_scenario (10 %)

When no books are borrowed, the system should display an informative message consistent with UC-03 Extension 1a (“no books currently borrowed”).

Test system behaviour when users have no borrowed books, demonstrating that:

- System correctly handles return operations with an empty borrowed list
- All books show as available when none are borrowed
- Multiple users with no books borrowed are handled correctly

Assert: System correctly reports when users have no borrowed books.

Testing Approach Summary

For Each Scenario, Your Tests Should:

1. Setup (Given):
    - Initialize the system with users and books
    - Create user sessions/authentication states
2. Actions (When):
    - Call library system methods directly (i.e. borrowBook, returnBook, placeHold)
    - Store operation results for verification
3. Verification (Then):
    - Assert on business state using system query methods
    - Verify: book status, user limits, queue positions, notification states
    - Check operation success/failure based on business rules

---

What Your Assertions Should Check:

Note that these method calls are examples.
Book State:

- book.isCheckedOut() - Is the book currently borrowed?
- book.getCurrentBorrower() - Who has the book?
- book.getDueDate() - When is it due? Remember, only a string representation.
- book.isReserved() - Is it reserved for someone?

User State:

- getBorrowedBooksCount(user) - How many books borrowed?
- isAtBorrowingLimit(user) - At 3-book limit?
- canBorrow(user, book) - Can this user borrow this book?

System State:

- getHoldQueue(book) - Queue ordering
- hasNotification(user, book) - Notification exists? (Return text that would be sent to console)
- getAvailableBooks() - Which books can be borrowed?

Required Tools and Technologies

- **IDE:** IntelliJ IDEA Community Edition 2025.2
- **Java:** Oracle OpenJDK 22
- **Build Tool:** Maven (version that IntelliJ provides)
- **Testing Framework:** JUnit 5 (Jupiter), version 5.13.4
- **Version Control:** Git with GitHub
- **Cucumber** (for Java) 7.23.0

Grading Breakdown

Test	Weight
A1_scenario	10 %
multiple_holds_queue_processing	40 %
borrowing_limit_and_hold_interactions	40 %
no_books_borrowed_scenario	10 %
Total	**100 %**

Deliverables

You must submit:

A2 work must continue in the same repository, in a new branch named A2-BDD-<LastName>

1. **Feature File and Step Definitions**

    - One complete **feature file** and **step definition file** in your A1 repository, but in a new branch. Branch Name: A2-BDD-<LastName>
    - Note that all the code, the feature, and the step definition files are to be submitted with the A1 original code.

2. **Demo Video Requirements**
   Your video must:

   - Your video plays correctly and begins with you **stating your last name, first name, and student number**.
   - Show you **downloading or cloning your repository**.
   - Indicate clearly **where your feature file** is located.
   - Indicate clearly **where your step-definition files** are located.
   - Show you **running your Cucumber test suite** with the **four required scenarios** in a single execution.
   - Clearly display the **test results showing that all scenarios pass**.
   - Ensure the terminal or IDE output is visible throughout.
   - Keep the video concise (about **10 minutes**) and easy to follow.

3. **Submission Grid**

   - Complete the provided submission grid (details to be posted).

4. **Commit Naming Policy for A2**

   - You may commit your entire assignment in a single commit or in multiple smaller commits. Use whichever approach you prefer.
   - There is **no required GitHub commit-naming scheme** for this assignment, unlike Assignment 1.
   - You may organize and name your commits in any clear, professional manner (for example: *“Add hold-queue scenario steps”* or *“Fix borrow-limit logic”*).
   - All work must be committed to your **A2-BDD-<LastName>** branch.