University Library Management System
OOP Final Project Report

MOD004883

Student Name: Oriol Morros Vilaseca

Student ID: 2270056

Academic Year: 2025/26

Trimester: 1

Contents

1. Introduction

This report covers the design, development, and evaluation of a Java console-based University Library Management System developed as part of the Object-Oriented Programming module. The main objective of the project was to demonstrate a clear and well-reasoned application of object-oriented principles including abstraction, encapsulation, inheritance, and polymorphism, as outlined in the module learning outcomes.

The system enables the management of library users, products, and loan transactions, supporting multiple user types (Adult, Child or Student) and media formats (Books, Audiobooks, CDs and DVDs). Borrowing behaviour is different depending on the user role and is governed through policy rules, making sure that borrowing limits, loan duration and renewal permissions are enforced consistently in the application.

A structured object-oriented methodology was adopted throughout the development. The project started with a modelling phase, where different UML diagrams were designed to define system scope, object responsibilities and the flow of interactions before getting implemented. These models provided the starting architectural decisions and were refined as the system evolved. The implementation phase consisted of translating the designed model into a clean and extensible Java code. To validate that the final application, functional and unit testing were carried out to validate and identify edge cases.

In addition, this report also reflects on feedback received in previous modules to demonstrate improvements during the code development, class design and documentation and comments. The report concludes with an evaluation of the system and the potential areas of improvement and limitations.

2. Model Development

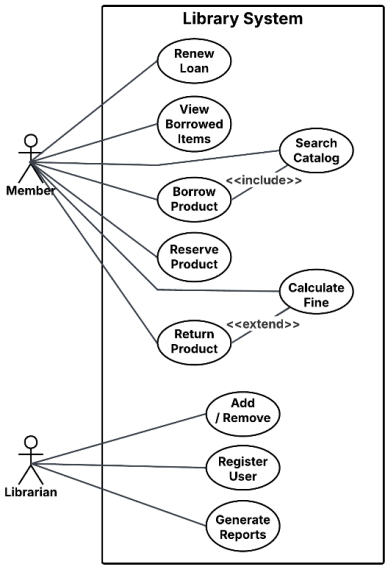
The development of the University Library Management System began with a modelling phase to establish a clear and structured architecture before any implementation or work started. This approach helped informing early architecture decisions and ensured that object-oriented principles were applied reducing coupling and improving maintainability as the system evolved

At the modelling stage, the system was scooped as a console-based application, in line with the assignment requirements. To focus only on object-oriented design a database system and a graphical user interface were not implemented in this project. User authentication was also an omitted feature and instead representative user instances were created to demonstrate polymorphic behaviour across the different user roles.

All the library media was externalised into different structured CSV files, allowing products to be added dynamically. This decision separated data concerns from business logic and avoided hard-coding product information within the application. Borrowing behaviour was designed to be different according to the user, with additional safety requiring the child users to be associated with an adult before borrowing products.

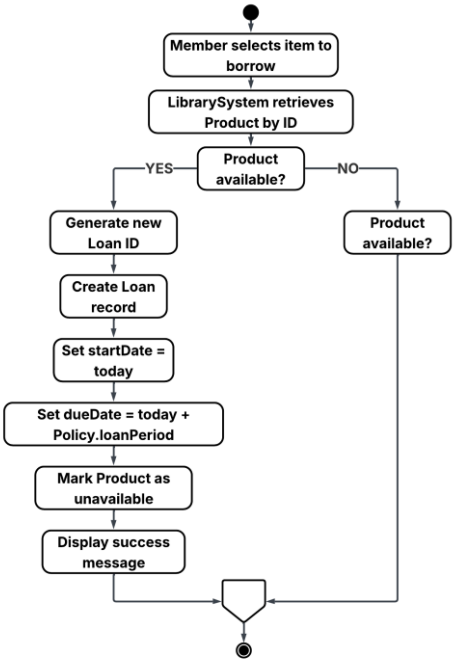
2.1 Use Case Modelling

The use case diagram was used to describe the functionalities of the system and identify the responsibilities of each actor. Two primary actors were modelled: Member and Librarian. Members can browse the catalogue, borrow and return products, view loans and renew products, on the other hand, Librarians are responsible for administrative tasks like registering new users, managing products, and generating reports.



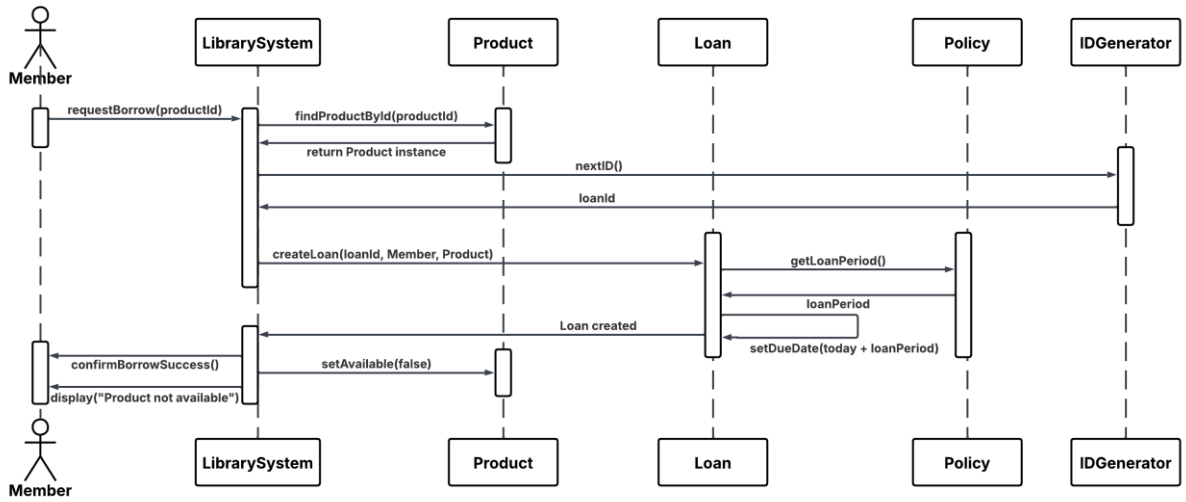
2.2 Activity Modelling

An activity diagram was produced to model the borrowing workflow from product selection to loan creation or rejection. This diagram was useful to identify decision points such as product availability checks, borrowing limits, and policy of the application. By visualizing the borrowing process this way, it was possible to validate that all conditions such as unavailable products or exceeded borrowing limits were handled correctly preventing system issues.



2.3 Sequence Modelling

The sequence diagram below illustrates the interaction between core components during a borrow request. The diagram confirms that the LibrarySystem class coordinates all the borrowing operations between users, products, loan records, and policy rules. This design ensures that no single object assumes multiple responsibilities. The diagram also demonstrates that unique identifiers are generated independently via IDGenerator, and that the loan duration is delegated to a Policy object instead of directly within the user or product classes. This separation improves clarity and supports reuse without modifying any existing logic.

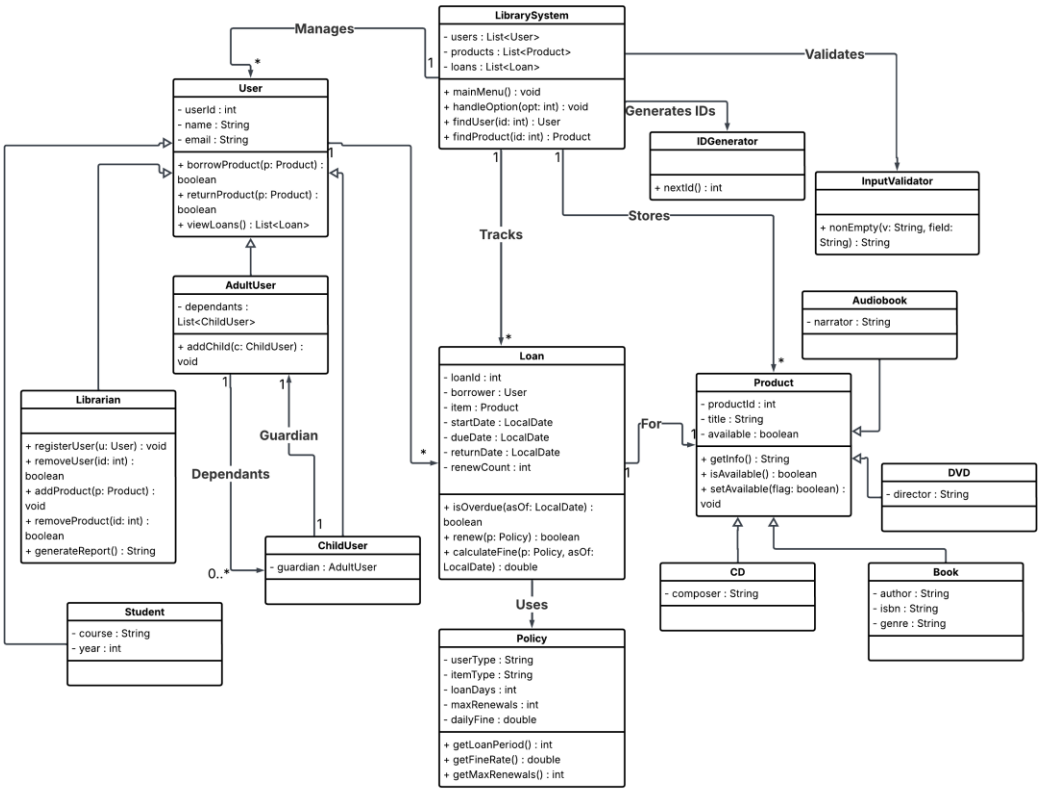


2.4 Class Design and Object Relationships

The class diagram formed the foundation of the system architecture. Abstract classes like User and Product define shared attributes and behaviour, while concrete subclasses specialise borrowing rules and all the product details. This enables polymorphic behaviour, allowing the system to treat user and product types differently.

Borrowable behaviour is defined through the Borrowable interface, ensuring all the product types expose a consistent contract for availability and information retrieval. Composition is used to model the relationship between adult and child users. Aggregation is used where objects are related but not dependant on each other's lifecycle, like the users managed by the library system.

The library system class was designed as a central controller, providing a simplified interface for operations such as borrowing, returning, searching, and reporting. This design also supports future extensibility. While this initial class diagram established the core object hierarchy and relationships, several changes were introduced during the implementation phase. These changes are reflected in the Development section with the final UML class diagram to discuss how the design evolved and the improvements done.



2.5 Application of Object-Oriented Principles and Design Patterns

Object oriented principles were applied consistently during the design phase. Abstraction was achieved through interfaces and abstract base classes, encapsulation through controlled access to internal state, inheritance through the user roles and the different products, and polymorphism through overridden borrowing and information methods.

A centralised identifier generation mechanism was applied through the IDGenerator class to have unique identifiers across the system. This implementation avoids duplication of logic across classes. The overall architecture follows a lightweight MVC-style separation with the classes forming the model, LibrarySystem acting as the controller and the Menu class providing the control-based view.

2.6 Model Validation

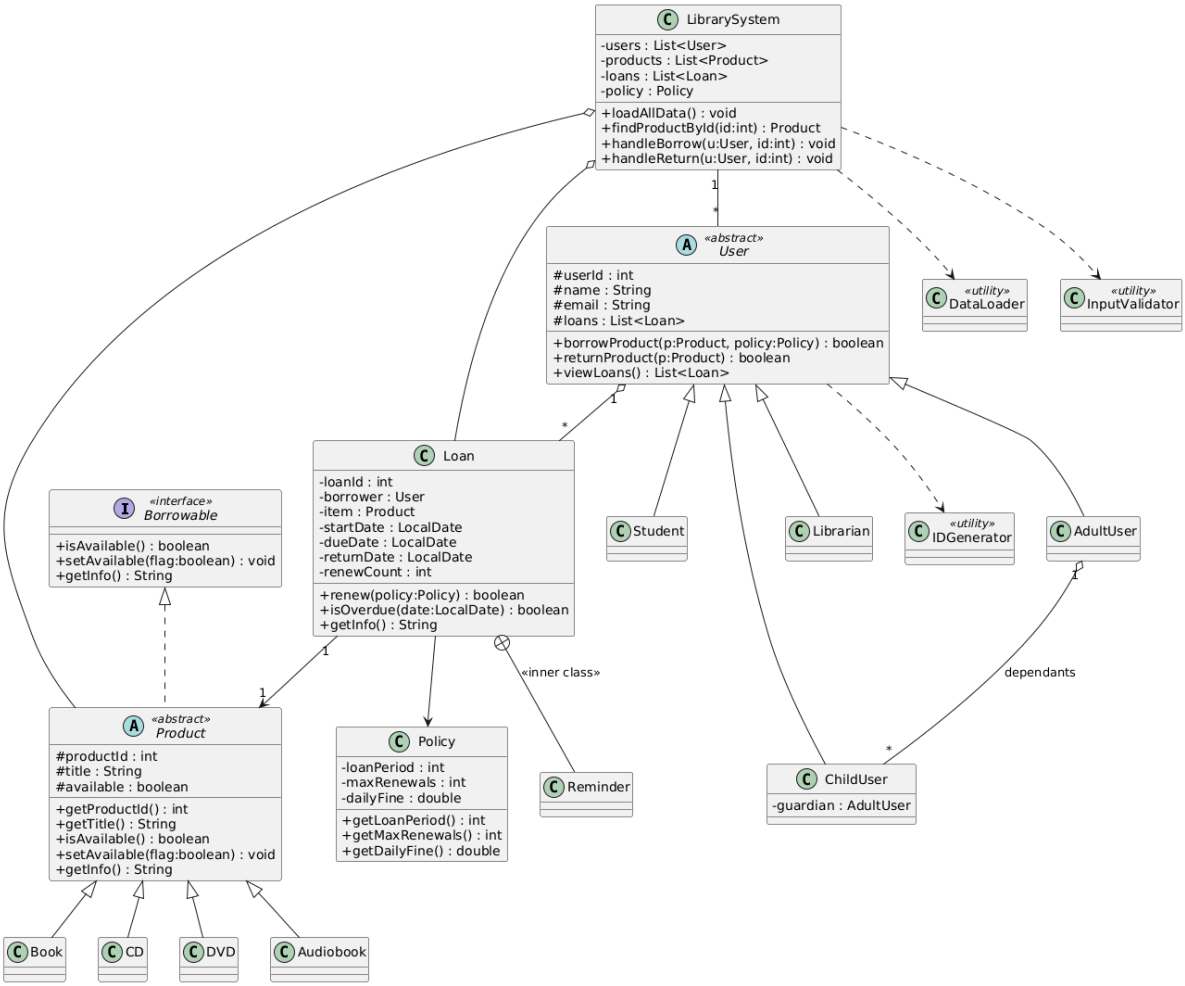
The modelling phase produced a clear object-oriented design that defined the system, object responsibilities, and the interaction flows prior to implementation. Architectural decisions like borrowing behaviour, policy abstraction and system control were fixed at this stage to minimize duplication. While the overall structure remained stable, some implementation details were refined during development. These refinements are reflected in the final UML class diagram presented in the Development section, where the evolution of the design is also discussed. This validated model provided a stable foundation for implementation while still allowing controlled refinement during the process.

Apart, alternative design approaches were also considered during the phase. One option was to centralise all the borrowing logic in the LibrarySystem controller. However, this approach was not implemented because it would have increased the coupling and reduced encapsulation. Another of the alternatives was to implement the borrowing rules using

conditional logic based on each of the user's type. This would have resulted in a less maintainable and extensive code and was discarded.

3. Development

The development phase focused on translating the initial object-oriented model into a working Java application while maintaining at the same time alignment with the previously designed architecture. The codebase is organized in different clear packages separating model classes that represent domain entities, utility classes that provide supporting functionalities and the user interface that is implemented through a console-based system.



The core domain logic is encapsulated within the model layer. Abstract classes like User and Product define shared attributes and behaviours, while subclasses implement borrowing rules depending on the roles, and product specific information using method overriding. This structure enables polymorphism, allowing the system to interact with users and products through interfaces while keeping specialized behaviour.

The LibrarySystem class acts as the central controller of the application. It coordinates borrowing and returning operations, manages the different collections of users, products, and loans. This design prevents excessive logic within the user interface and ensures that the rules remain encapsulated within the model layer.

During the implementation of the system, there were several refinements made to improve consistency and encapsulation. One of the key changes was the relocation of loan creation logic from the controller into the User class. This ensured that borrowing behaviour remained consistent across console interactions, direct method calls, and unit testing, while at the same time reducing the responsibility of the controller. The final class diagram reflects all these improvements and accurately represents the implemented code structure.

Overall, the implemented system closely follows the original model while incorporating targeted improvements that were identified during the development stage. The final design presented maintains strong separation of concerns, supports future extensibility, and demonstrates effective application of object-oriented programming principles learn during the module.

4. Testing

In this section the full testing process that was taken for the University Library system is documented. Covering all the functional testing through the console interface, unit testing for the individual classes and methods, and several targeted retests that had to be done after debugging issues that appeared during the development of the system. To ensure that all features behaved as planned, the tests covered different normal scenarios, edge cases, and error conditions. This process was crucial to ensure that the library management system works as intended and operates perfectly under a variety of conditions.

4.1 Functional Testing

Functional testing was carried out through the console interface to make sure that each of the user actions behaved as specified in the requirements. The main goal was to ensure that every menu option and system interaction worked correctly end to end. The tests mainly covered 3 areas: the login behaviour, the product visibility and all the borrowing rules, and error handling.

The full test set is presented in the functional testing table that follows. Each entry includes the test number and case, the test data that was used, the expected behaviour, if it passed or failed and any corrected action taken to debug. This approach ensures that every pathway was validated and working properly, including several edge cases such as invalid user input choices or attempts to borrow unavailable products and return operations without matching the loan record of the user.

Test No.	Test Case	Test Data	Expected Result	Pass/ Fail	Action	Re-Test No.	Date
1	AdultUser login works correctly	Login option: 1	System prints *"Logged in as AdultUser"*	Pass	N/A	N/A	02/12/25
2	ChildUser login assigns guardian	Login option: 2	ChildUser instance created and guardian set to "Parent"	Pass	N/A	N/A	02/12/25
3	Student login works correctly	Login option: 3	System prints *"Logged in as Student"*	Pass	N/A	N/A	02/12/25

4	Invalid login input defaults to AdultUser	Login option: 99	Should print: "Invalid choice, logged in as default AdultUser."	Fail	Added default case message	4.1	02/12/25
4.1	Re-test invalid input handling	Login option: 99	Correct error message displayed	Pass	N/A	N/A	02/12/25
5	AdultUser sees all product categories	View Products	Books, CDs, DVDs, Audiobooks visible	Pass	N/A	N/A	02/12/25
6	Student access restriction works	View Products	Only Books + Audiobooks visible	Pass	N/A	N/A	02/12/25
7	ChildUser restricted to Books only	View Products	Only Books shown	Fail	Corrected switch logic for category visibility	7.1	02/12/25
7.1	Re-test Child category restriction	View Products	Only Books shown	Pass	N/A	N/A	02/12/25
8	Invalid category option displays error	Option = 99	System prints: "Invalid option."	Fail	Added default-case error message	8.1	02/12/25
8.1	Re-test invalid category option	Option = 99	System prints: "Invalid option."	Pass	N/A	N/A	02/12/25
9	Adult borrows an available Book	AdultUser, Product ID: 1	Borrow succeeds products set unavailable + Loan created	Pass	N/A	N/A	02/12/25
10	Adult cannot exceed 10 borrowed products	AdultUser with 10 loans attempts Product ID: 3	Borrow denied, message: "Borrowing limit reached (10 products max)."	Fail	Fixed logic in AdultUser.borrow Product()	10.1	02/12/25
10.1	Re-test Adult borrowing limit	10 existing loans	Borrow denied correctly	Pass	N/A	N/A	02/12/25
11	ChildUser cannot exceed 3 borrowed products	ChildUser with 3 loans attempts Product ID: 5	Borrow denied, message: "Borrowing limit reached (3 products max)."	Fail	Updated borrowing condition in ChildUser	11.1	02/12/25
11.1	Re-test ChildUser borrowing limit	3 existing loans	Borrow denied correctly	Pass	N/A	N/A	02/12/25
12	Student should NOT access DVD category	Student selects DVD option	DVDs must NOT appear; system prints "Invalid option"	Fail	Updated borrow menu to restrict Student DVD access	12.1	02/12/25
12.1	Re-test Student DVD restriction	Student selects DVD option	"Invalid option" shown; no DVDs displayed	Pass	N/A	N/A	02/12/25
13	Invalid Product ID during borrow	Product ID: 999	System prints: "Product not found."	Pass	N/A	N/A	03/12/25
14	Borrowing an unavailable	Product availability = false	System prints: "Product is currently checked out."	Fail	Added explicit availability check	14.1	03/12/25

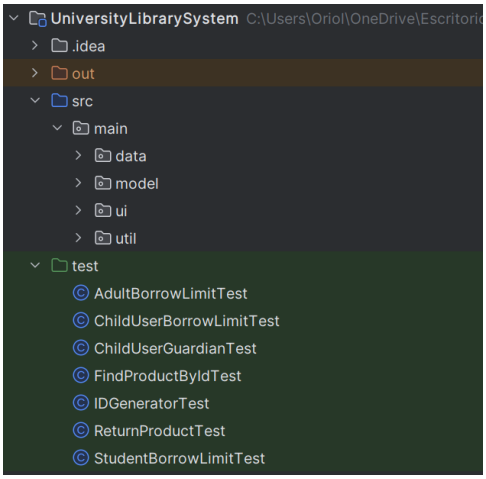
	product is rejected						
14.1	Re-test unavailable product borrows	Product unavailable	Correct error message shown	Pass	N/A	N/A	03/12/25
15	Student borrows Audiobook correctly	Student user, product ID: 4	Borrow succeeds and Loan created using Student policy	Pass	N/A	N/A	03/12/25
16	User returns a product they borrowed	AdultUser borrows + returns Product 2	Loan removed, availability restored, printed "Returned"	Pass	N/A	N/A	03/12/25
17	Returning non-borrowed product is rejected	Student returns Product 3 (never borrowed)	System prints: "Loan not found."	Fail	Added explicit missing-loan message in returnProduct()	17.1	03/12/25
17.1	Re-test non-borrowed product return	Student returns Product 3 (never borrowed)	"Loan not found" displayed	Pass	N/A	N/A	03/12/25

Functional testing confirmed that the system behaves correctly end to end from a user perspective and that the role restrictions are enforced consistently through the console interface. Several defects were identified during the process, especially in the input validation and borrowing rules, all were resolved through targeted code changes followed by a re-testing stage. This ensured that all the user interactions behave as specified under normal conditions.

The functional test set was designed to prioritize user paths and rule-based behaviour rather than exhaustive input permutations. This approach ensured that all constraints, borrowing policies and failure scenarios were validated from an end-to-end user perspective avoiding redundant testing already covered by unit testing at the class level.

4.2 Unit Testing

Unit testing was conducted using JUnit to validate the core business logic of the university library system independently of the console-based interface. All the unit tests done were placed in a separate test folder, clearly separated from the main production code, ensuring a distinction between the system logic and the testing.



The main goal of the unit testing was to verify that individual classes and methods behaved correctly on their own. Especially the ones responsible for enforcing borrowing limits, managing loan records, restoring product availability, and generating identifiers. This approach complemented the previous functional testing by validating the internal object behaviour without relying on the console menu navigation or the user input.

4.2.1 Unit Test Coverage

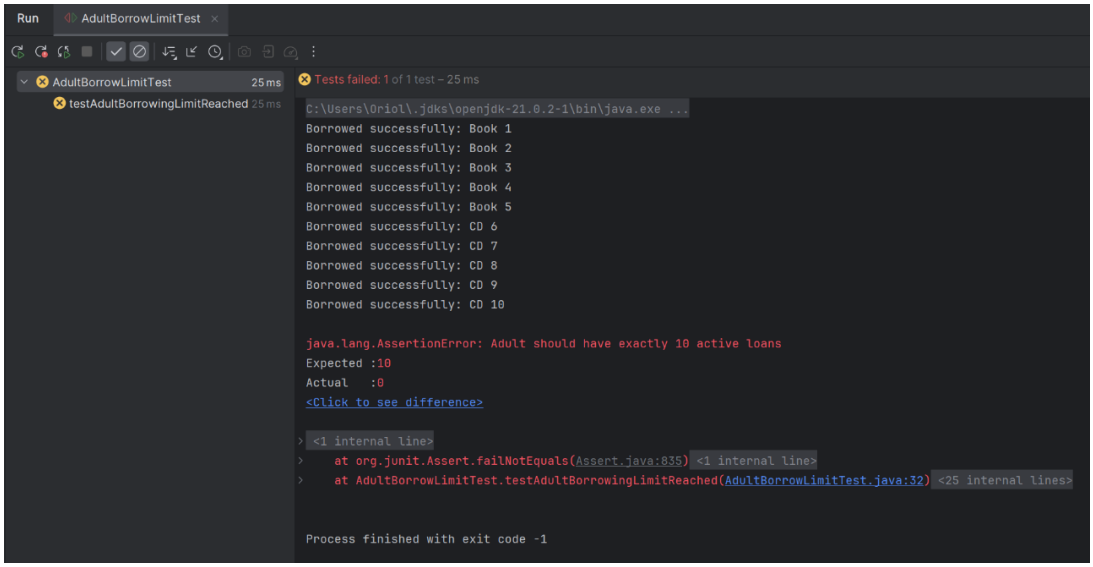
The following unit tests were implemented to validate the most critical system constraints:

AdultBorrowLimitTest	Ensures adult users can borrow up to 10 products and any additional attempt is rejected.
StudentBorrowLimitTest	Ensures student users are restricted to a maximum of 5 active loans.
ChildUserBorrowLimitTest	Validates that child users can't exceed 3 borrowed products.
ChildUserGuardianTest	Confirms that a child user can't borrow any products without an adult guardian assigned.
ReturnProductTest	Checks that returning a borrowed product restores its availability and removes the loan
FindProductByIdTest	Verifies that products can be correctly retrieved from the system using unique identifiers.
IDGeneratorTest	Ensures that identifier generator produces unique values.

All together these tests help validate the borrowing rules, product state changes, loan consistency, and the identifier management.

4.2.2 Bugs Identified and Fixed Through Unit Testing

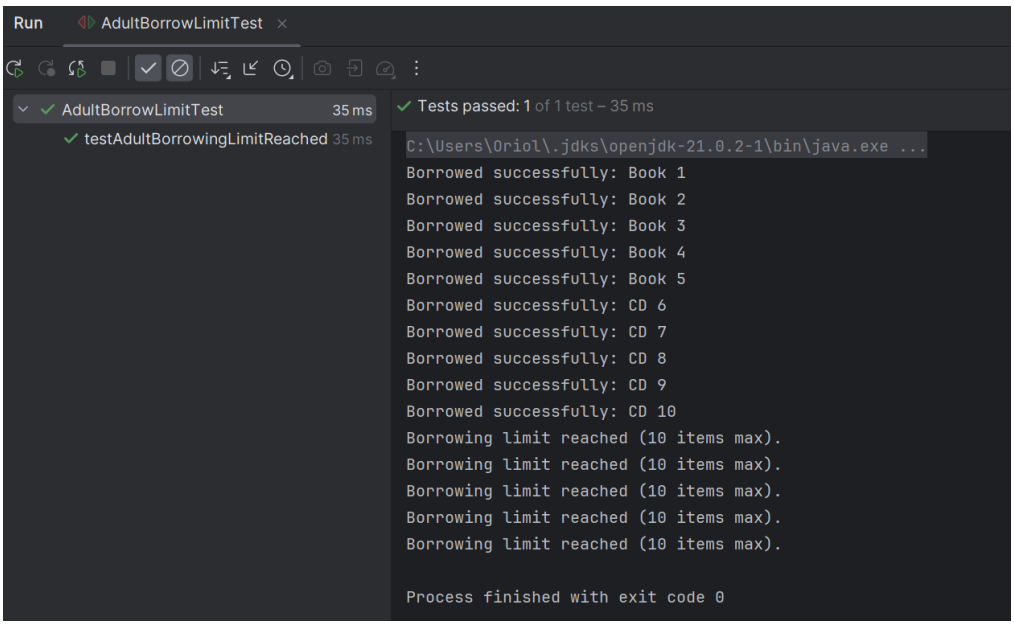
Unit testing revealed a critical design inconsistency during the execution of AdultBorrowLimitTest. Even though the borrowing operations appeared to be successful, the user internal loan list remained empty causing the borrowing limits to be ignored. This test resulted in failure because 10 loans were expected and none of them were recorded.



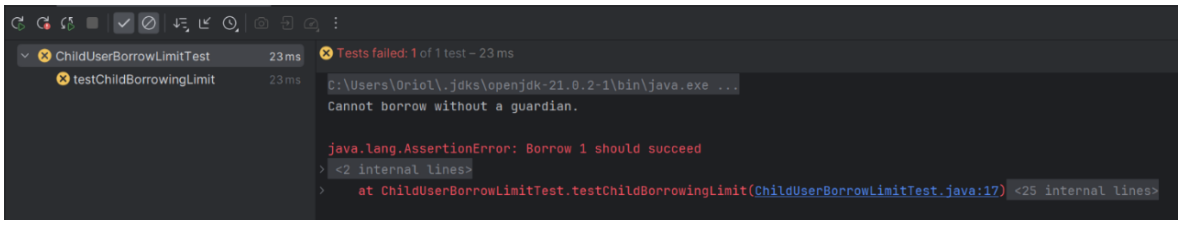
The root cause was that loan creation logic was implemented inside **LibrarySystem.handleBorrow()** rather than within **User.borrowProduct()**. As a result, borrowing initiated directly through method calls and did not behave consistently with console-based interactions.

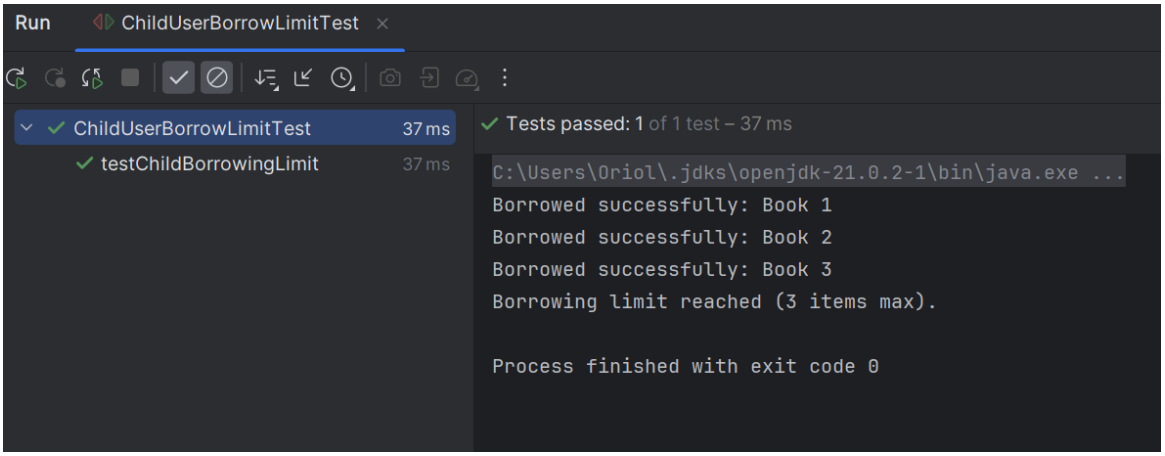
To solve this issue, first the loan creation logic was moved into **User.borrowProduct()**. Also, subclasses were updated to correctly follow the superclass borrowing behaviour and duplicate loan creation issue in **LibrarySystem.handleBorrow()** was completely removed.

After applying these changes, borrowing limits were enforced correctly and all related unit tests passed successfully. This change improved consistency, robustness and alignment of object-oriented principles.



A further issue was also observed during the **ChildUserBorrowLimitTest**, where borrowing failed due to the absence of an assigned guardian. This behaviour was corrected according to the system requirements. The test was updated to assign a guardian correctly using **AdultUser.addChild()**, after the borrowing and limit behaved in the expected way.





The unit tests focus on policy behaviour and state transitions that are critical to system correctness, such as borrowing limits, product availability updates, and loan consistency. Console input handling and menu navigation were excluded from the unit testing because they were already validated through the functional testing and don't represent domain logic.

5. Implemented Learning from Previous Modules

Feedback received during the first year in Software Principles module highlighted several weaknesses, particularly in the application of object-oriented design principles. One key issue identified was the presence of excessive logic within the main application class, resulting in a poor separation of the responsibilities and a limited encapsulation of the core system behaviour. Apart from that, feedback also indicated that domain logic was not always placed in the most appropriate classes, reducing the clarity of the code and extensibility of the design.

This project demonstrates a clear improvement in response to that feedback. The current library management system follows a more structured object-oriented programming approach, where responsibilities are distributed across classes such as User, Loan, Product, and LibrarySystem. The main application class is limited to coordinating user interaction rather than containing the business logic. This separation improves maintainability and aligns more closely with established software design principles.

In contrast to previous work, inheritance and polymorphism are applied more effectively. Different user types encapsulate their borrowing rules through overridden methods, allowing behaviour to vary without reliance on conditional logic. This results in cleaner, more extensible code and stronger alignment object-oriented programming principles discussed in earlier modules.

Apart, feedback received from previous modules highlighted insufficient and inconsistent documentation. In response to that, this project applies a better commenting approach, with better and consistent in-line comments with JavaDoc comments describing their purpose, parameters, and behaviour, improving code reliability and maintainability compared to earlier assignments.

6. Conclusion and future improvements

This project aimed to design and implement a console-based University Library Management System focusing on demonstrating the core object-oriented programming principles learned during the module. Starting with a modelling-first approach and then refining the system through iterative development and a testing process. The final solution accomplished demonstrates effective use of abstraction, encapsulation, inheritance, and polymorphism.

One of the main learning outcomes of this project was understanding how important it is to align class responsibilities with real world behaviour. During the implementation section several early design choices were revisited, the loan creation logic was moved into the User class to guarantee consistent behaviour regardless of how borrowing actions are triggered. This change improved encapsulation, removed duplicate logic, and showed the importance of testing. Functional and Unit testing played a key role at the time of validating behaviour and influence improvements to the system structure.

The project also shows clear progress compared to feedback received in previous modules, especially on how different logic is distributed across classes and how instead of having complex conditional statements polymorphism is used. Compared to earlier coursework, this implementation demonstrates a stronger modular design, class responsibilities, and a better and more consistent documentation combining in line comments and JavaDoc documentation.

Despite its strengths, the system has some intentional limits. A user authentication system and database storage were excluded to maintain more focus on the object-oriented design principles and the consol based user interaction, in line with the assessment requirements. Future improvements could include the integration of a database to support persistent storage or introducing authentication for the users. These improvements would give more realism and scalability to the system while preserving object-oriented design.

Overall, this project demonstrates a solid and practical understanding of object-oriented design in Java. The final system has a stable design and extensible base showing learning progression and the effective application of OOP principles in line with the module requirements.