Technical report

This is the technical report, which describes the development and design of an application oriented to the online sale of books, implemented following the principles of object-oriented programming and using design patterns to ensure the scalability and maintainability of the system. This virtual book store allows users, whether buyers or administrators, to interact with the system efficiently to manage the acquisition and management of books. The main objective of this application is to offer a simple and intuitive shopping experience for customers, who can search for books, add them to a shopping cart and place orders. On the other hand, administrators have the possibility to keep the book catalog updated, adding and modifying the details of the available products.

Business Model:

This business model is characterized by offering users (buyers) a wide range of products (books) that they can browse, search and buy efficiently. The purchasing process is carried out through a shopping cart, where the customer can manage the books they wish to purchase and proceed to place an order. In parallel, the application has an administrative functionality that allows sellers or administrators to manage the book catalog. Through this interface, administrators can add new books, modify existing details (such as price, stock or description), and keep the product offering updated, ensuring that the store remains relevant and competitive.

The user stories for the Web Book Store:

- As a customer, I want to see a list of available books, so I can choose which one to buy.
- As a customer, I want to search for a book by some characteristic (author, ISBN, etc.), to quickly find the one I'm looking for.
- As a customer, I want to be able to see the details of a book, to know its description, price, and author before deciding whether to buy it.
- As a customer, I want to be able to add books to a shopping cart, so I can buy them all at once.
- As a customer, I want to be able to remove books from the shopping cart, to adjust my order before checkout.
- As a customer, I want to be able to review my cart before purchasing, to make sure I have the right books.
- As an administrator, I want to be able to add new books to the catalog, to keep the store up to date.
- As an administrator, I want to be able to modify the details of a book (price, stock, description), to adjust the product information.

CRC Cards:

Book:

Book	
Responsibilities	**Collaborators**
Save information about the book	Catalog
Show details about the book to the user	Shopping Cart

Catalog:

Catalog	
Responsibilities	**Collaborators**
Manage book collection	Buyer
Allow book search	Book
Allow filtering by category, price, author, etc.	

Order:

Order	
Responsibilities	**Collaborators**
Process the order placed by the buyer	Shopping Cart
Record the order details	Buyer
Save the transaction	Book

Seller:

Seller	
Responsibilities	**Collaborators**
Add, modify and delete books in the catalog	Book
Manage inventory	Catalog

User:

User	
Responsibilities	**Collaborators**
Save user information	Buyer
Manage user authentication and profil	Seller

Functional Requirements:

Book Management:

Data Loading: The system must allow loading book data from a file.
Book Listing: The system must allow listing all the books available in the catalog.
Book Search: The system must allow searching for books by different criteria (author, ISBN, title).

Shopping Cart:

Cart Creation: The system must allow creating a shopping cart for a user.
Add Items: The system must allow adding items to the shopping cart.
Delete Items: The system must allow removing items from the shopping cart.
Show Cart: The system must allow displaying the contents of the shopping cart.

Catalog:

Catalog Initialization: The system must allow initializing the book catalog.
Add Books to Catalog: The system must allow adding books to the catalog.

Catalog Search: The system must allow searching for books in the catalog using different search strategies.

Non-Functional Requirements:

Performance:

- The system should be able to handle a large number of books and users without degrading performance.
- Searches should be fast and efficient.

Scalability:

- The system should be scalable to handle an increase in the number of users and books.

Maintainability:

- The code should be modular and follow design principles that make it easy to maintain and extend.
- Documentation should be clear and complete.

Portability:

- The system should be portable and run in different environments without problems. Docker is used to ensure portability.

Technical Considerations:

Architecture:

Modularity: The system is divided into clear modules (repositories, services, controllers), which facilitates maintainability and scalability.
Design Patterns: Design patterns such as Factory Method, Strategy, and Singleton are used to solve specific problems efficiently.

Technologies Used:

FastAPI: Used to create the system API, providing endpoints for different functionalities.
Spring: Used to create the system API, providing endpoints for different functionalities.
Docker: Used to ensure portability and consistency of the runtime environment.

SOLID Principles:

Single Responsibility Principle: Each class has a single, clear responsibility.
Open/Closed Principle: Classes are open for extension but closed for modification.
Liskov Substitution Principle: Subclasses can be used in place of their base classes without altering the expected behavior.
Dependency Inversion Principle: Classes depend on abstractions rather than concrete implementations.

Dependency Management:

Poetry:It is used to manage backend dependencies in python, ensuring that all necessary libraries are installed.
Maven: It is used to manage backend dependencies in Java, ensuring that all necessary libraries are installed.

SOLID Principles Analysis in Architecture

Single Responsibility Principle (SRP):

BookDAO: This class has a single responsibility, which is to represent the data of a book.
BookFactory: Its only responsibility is to create BookDAO instances.
BookRepository: It is responsible for loading and providing book data.
Catalog: Manages the book catalog and search strategies.
SearchStrategy and its subclasses: Each has the responsibility of implementing a specific search strategy.

Open/Closed Principle (OCP):

SearchStrategy: The SearchStrategy base class is open for extension (you can add new search strategies) but closed for modification (you do not need to change the base class to add new strategies).
Catalog: You can add new search strategies without modifying the Catalog class.

Liskov Substitution Principle (LSP):

SearchStrategy subclasses (SearchByAuthor, SearchByISBN, SearchByTitle) can be used in place of the SearchStrategy base class without altering the expected behavior of the program.

Interface Segregation Principle (ISP):

Not directly applicable in this case as there are no interfaces that are implemented by multiple classes. However, classes are designed to have clear and specific interfaces.

Dependency Inversion Principle (DIP):

Catalog depends on the SearchStrategy abstraction instead of depending on concrete implementations. This allows changing search strategies without modifying the Catalog class.

Analysis and Detail of the Implementation of Design Patterns

Factory Method:

BookFactory: Implements the Factory Method pattern to create BookDAO instances. This allows centralizing the object creation logic and makes it easier to modify the way objects are created without changing the code that uses them.

Strategy:

SearchStrategy and its subclasses (SearchByAuthor, SearchByISBN, SearchByTitle): Implement the Strategy pattern to allow different search strategies in the catalog. This makes it easy to add new search strategies without modifying existing classes

Singleton:

Catalog: Implements the Singleton pattern to ensure that there is only one instance of Catalog in the entire application. This is useful for managing a single book catalog shared across the entire application.

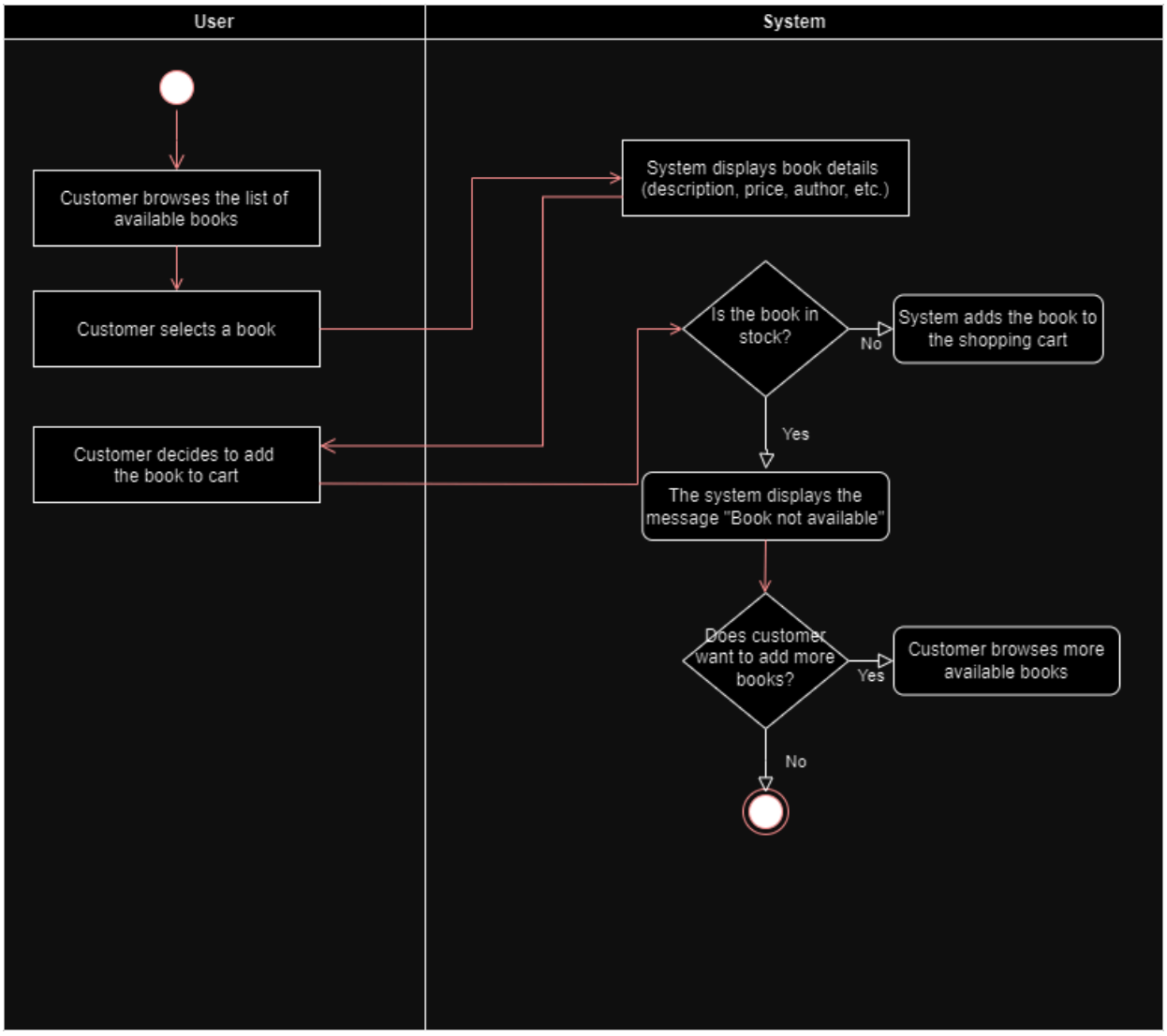
Best Practices and Anti patterns:

- **Modularity:** The code is well organized into modules and classes, making it easy to understand and maintain.
- **Documentation:** Each class and method has a clear description of its purpose and function.

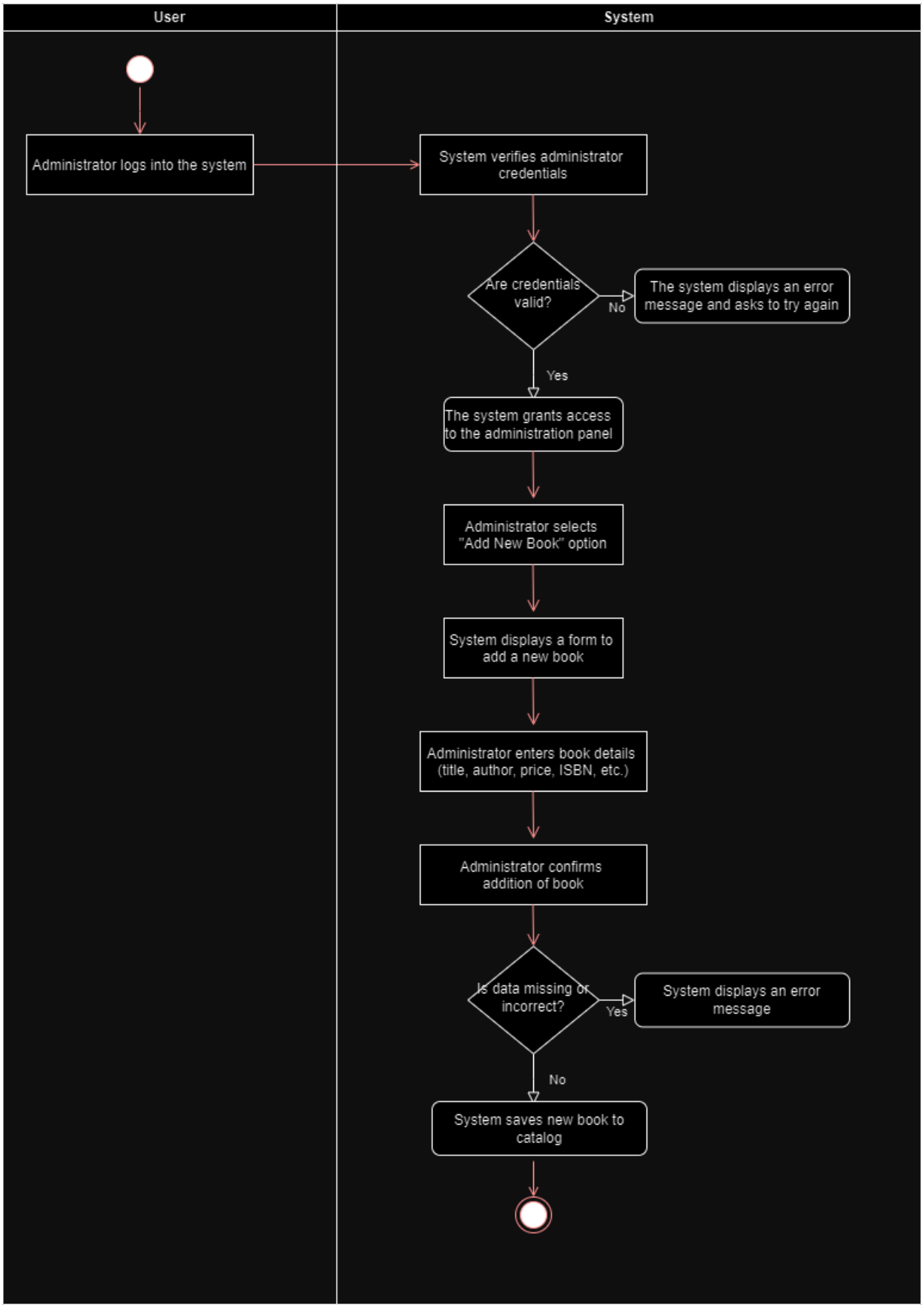
- **Use of Types:** Type annotations are used to improve readability and facilitate error detection.
- **SOLID Principles:** The architecture follows SOLID principles, which improves the flexibility and maintainability of the code.
- **God Object**: There are no objects that do too many things. Each class has a clear responsibility.
- **Hardcoding**: There are no values hard-coded into the code. Data is loaded from external files.
- **Spaghetti Code:** The code is well structured and modularized, avoiding tangled and difficult-to-follow dependencies.

Activities Diagrams:

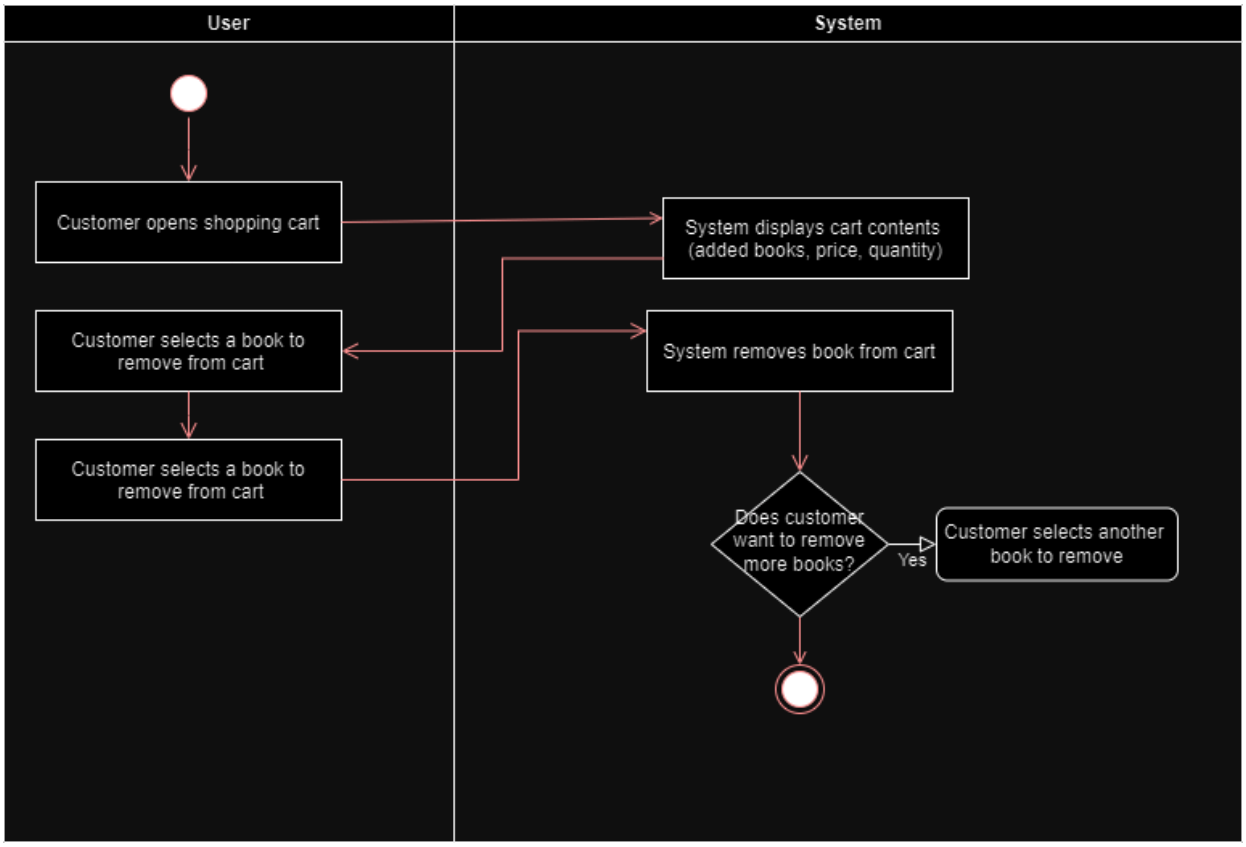
Add Books Cart:



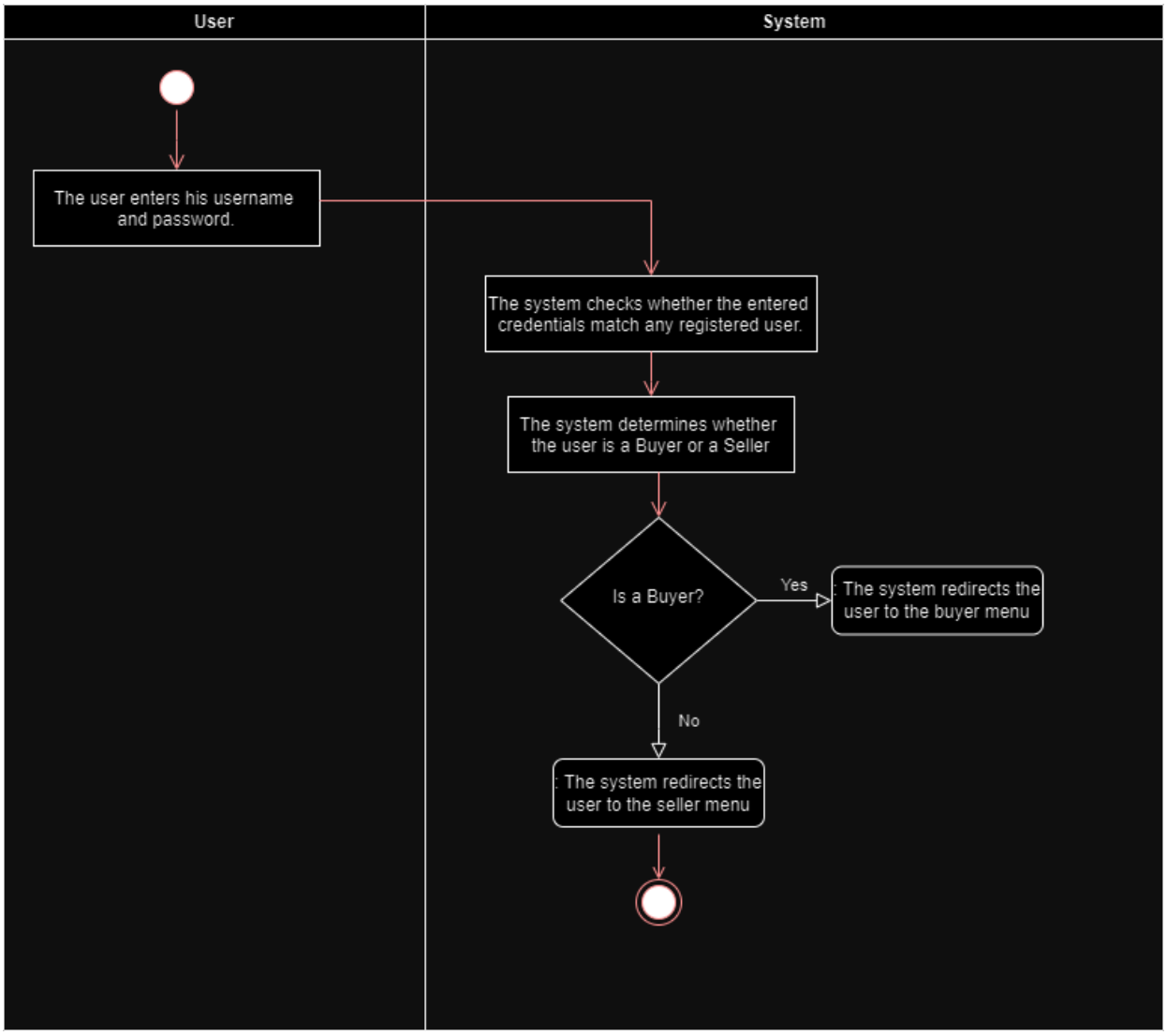
Add new Books:


User
System
Administrator logs into the system
System verifies administrator credentials
Are credentials valid?
No
The system displays an error message and asks to try again
Yes
The system grants access to the administration panel
Administrator selects "Add New Book" option
System displays a form to add a new book
Administrator enters book details (title, author, price, ISBN, etc.)
Administrator confirms addition of book
Is data missing or incorrect?
Yes
System displays an error message
No
System saves new book to catalog

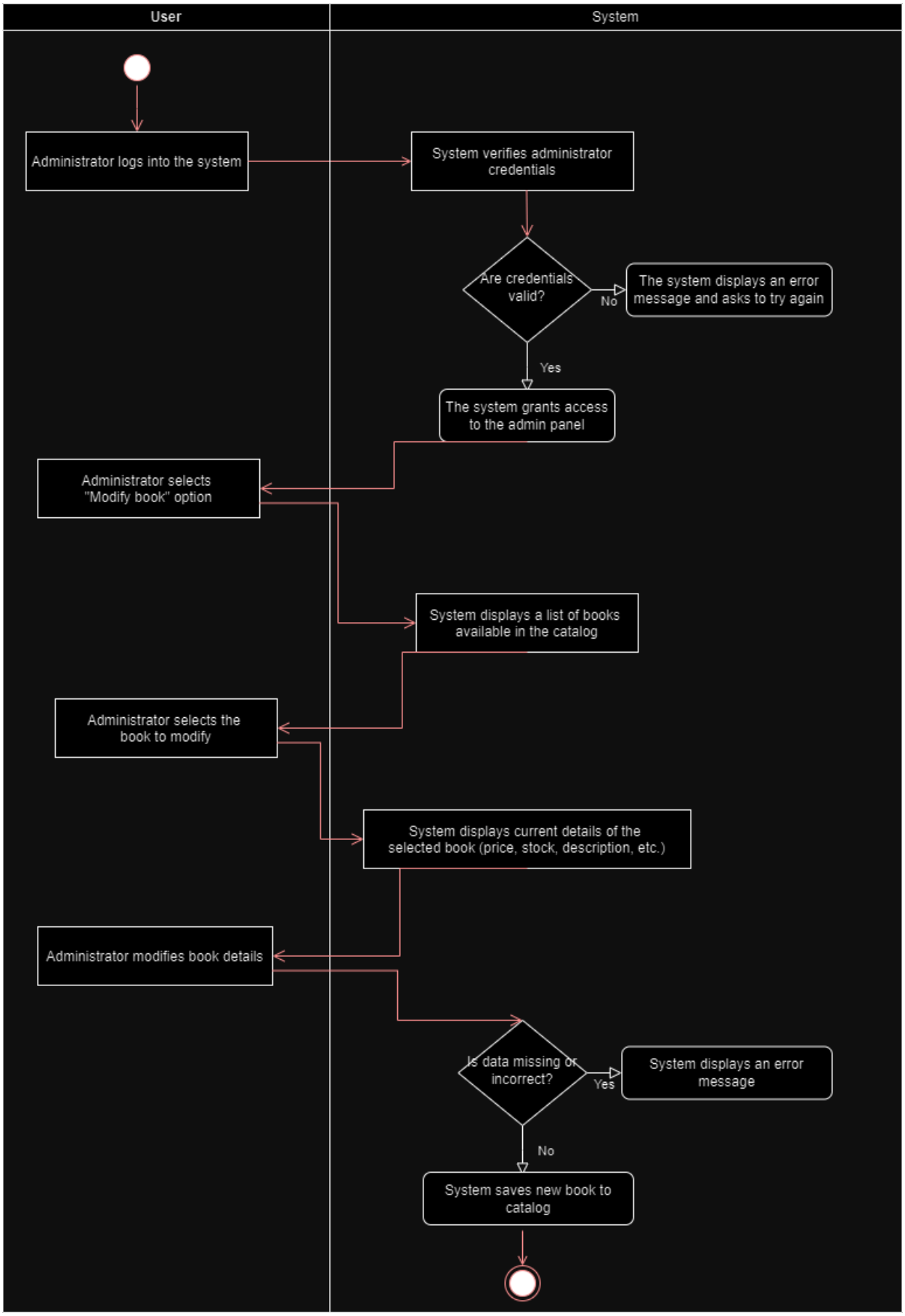
Delete Books Cart:



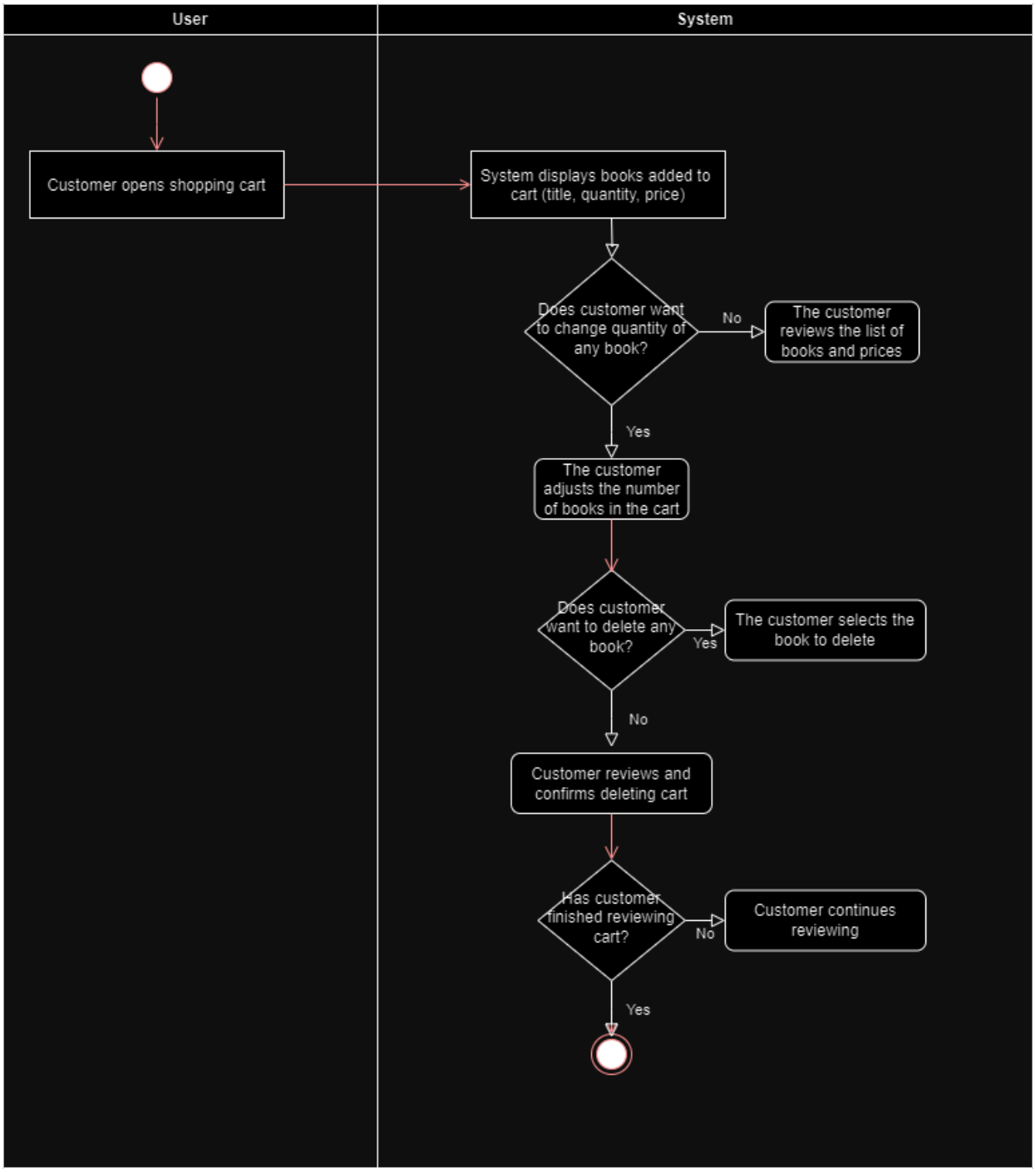
Login:



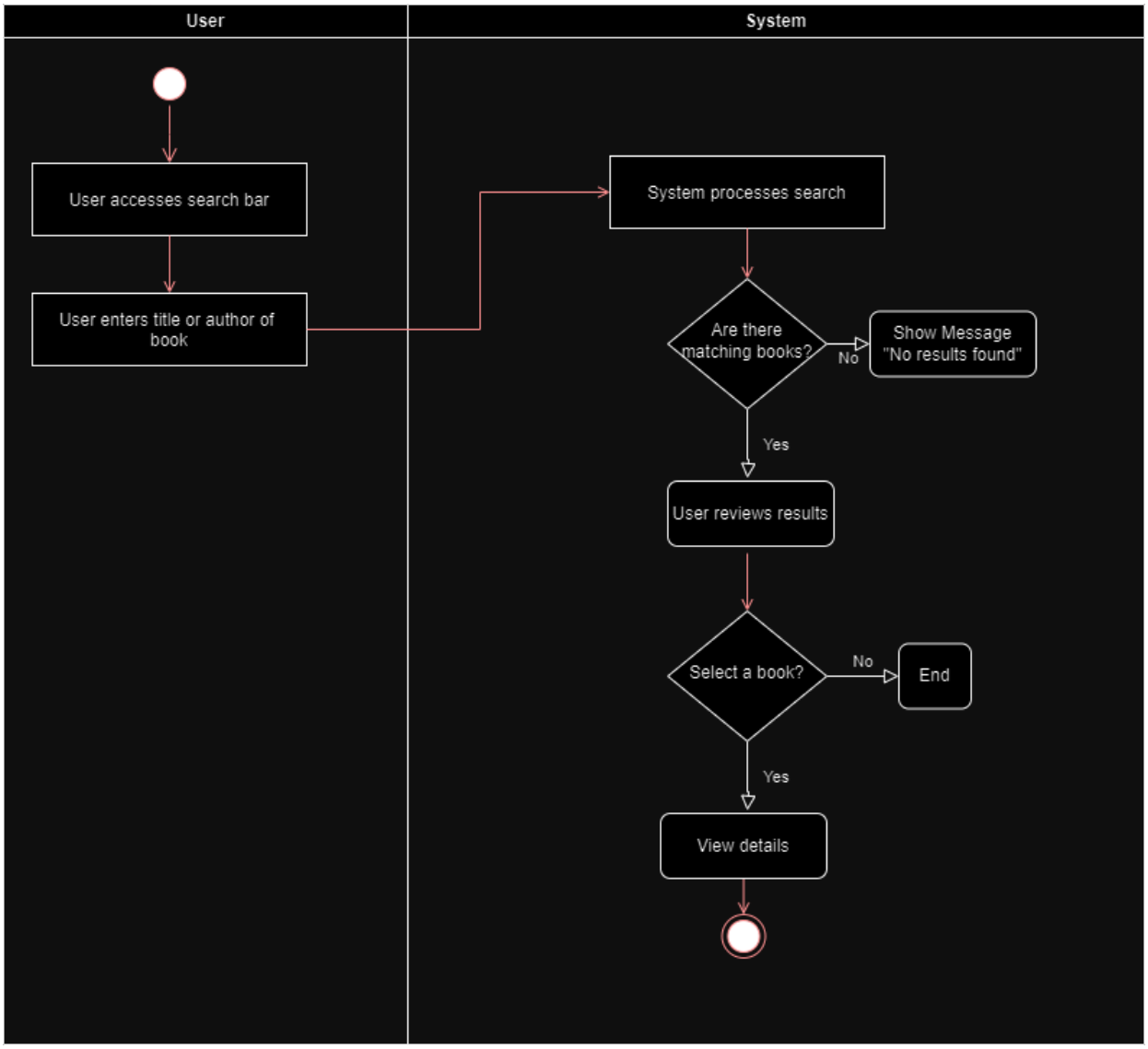
Modify details books:


User
System
Administrator logs into the system
System verifies administrator credentials
Are credentials valid?
No
The system displays an error message and asks to try again
Yes
The system grants access to the admin panel
Administrator selects "Modify book" option
System displays a list of books available in the catalog
Administrator selects the book to modify
System displays current details of the selected book (price, stock, description, etc.)
Administrator modifies book details
Is data missing or incorrect?
Yes
System displays an error message
No
System saves new book to catalog

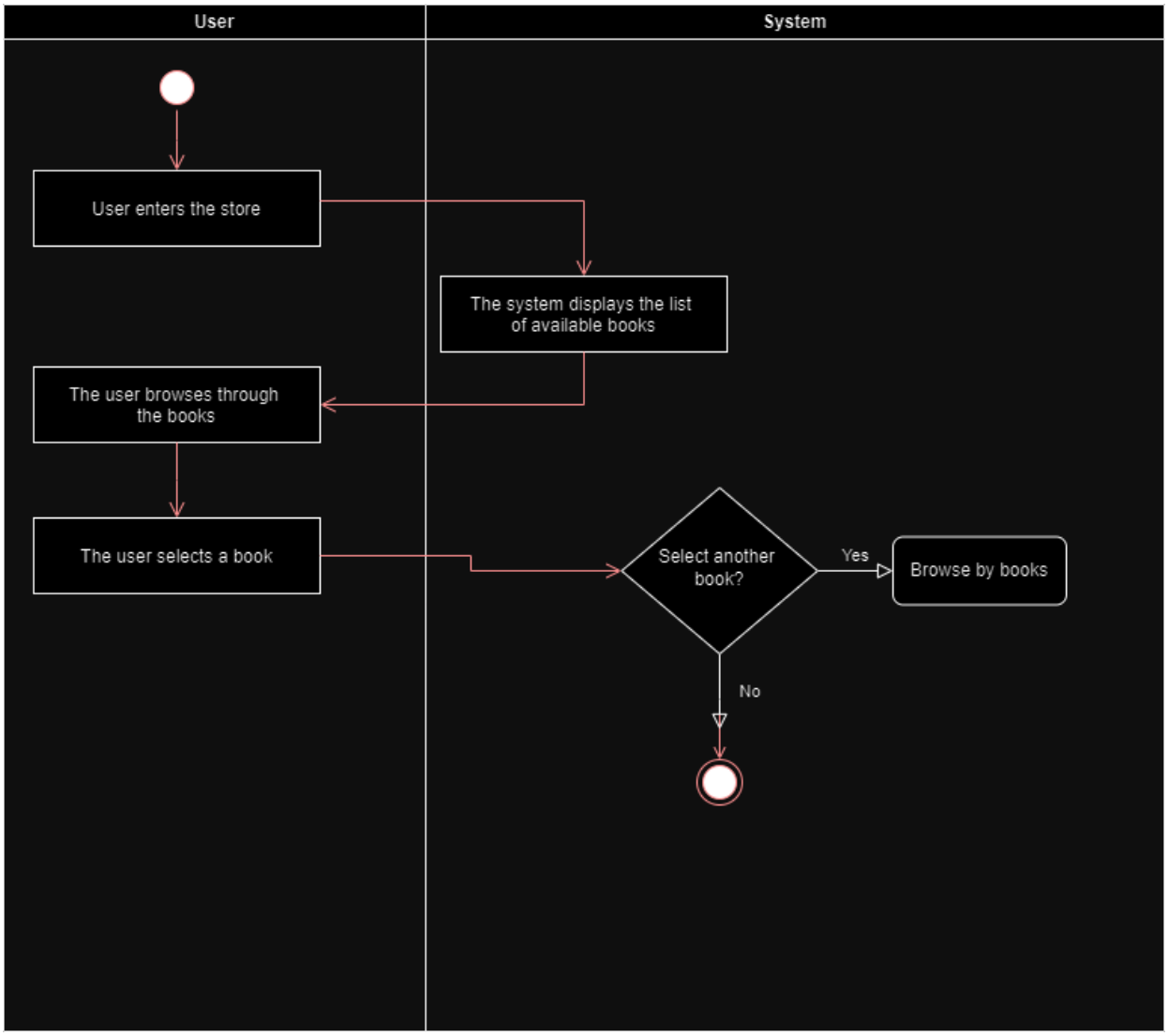
Review books cart



Search books:

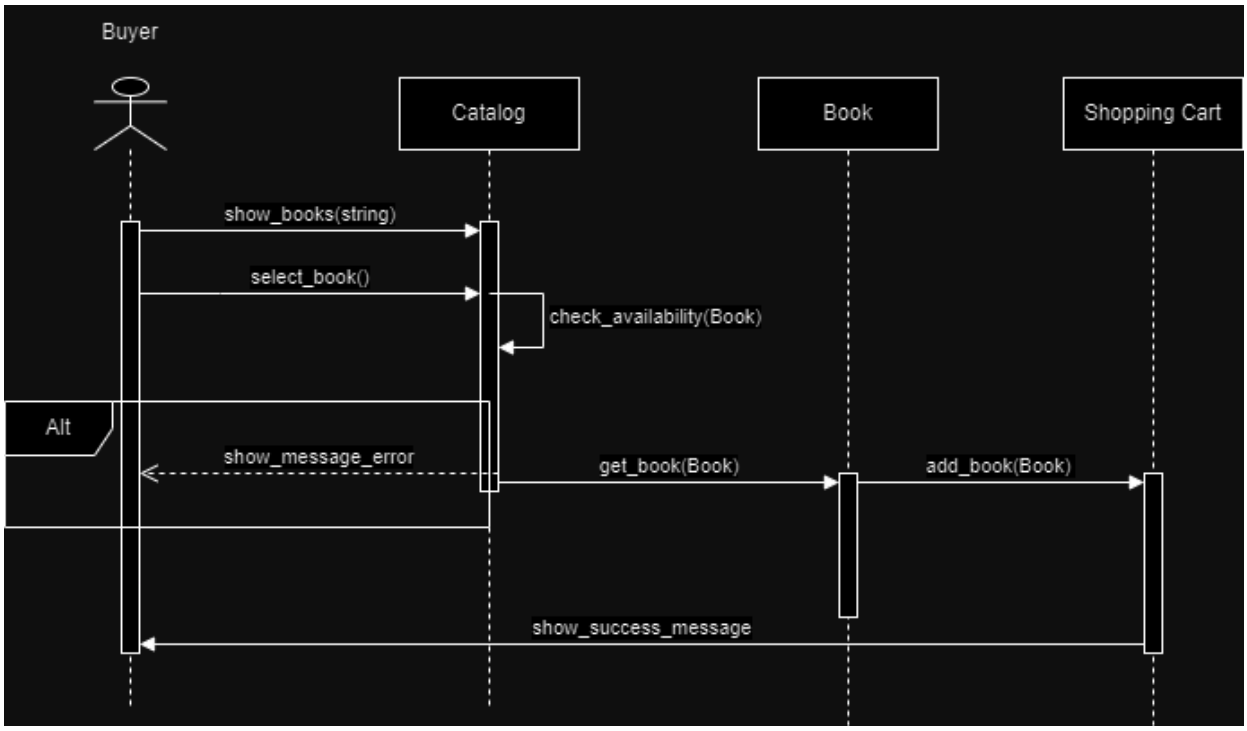

User
System
User accesses search bar
User enters title or author of book
System processes search
Are there matching books?
No
Show Message "No results found"
Yes
User reviews results
Select a book?
No
End
Yes
View details

Show books

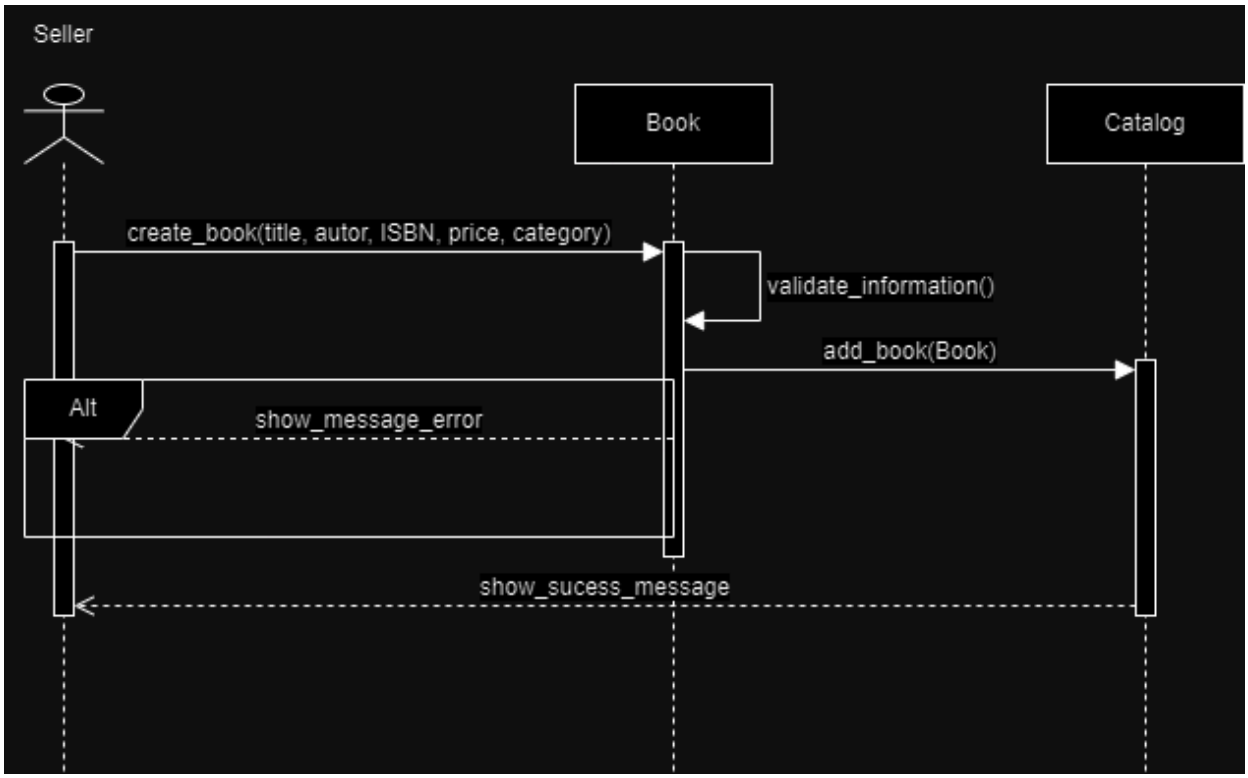


Sequence Diagrams:

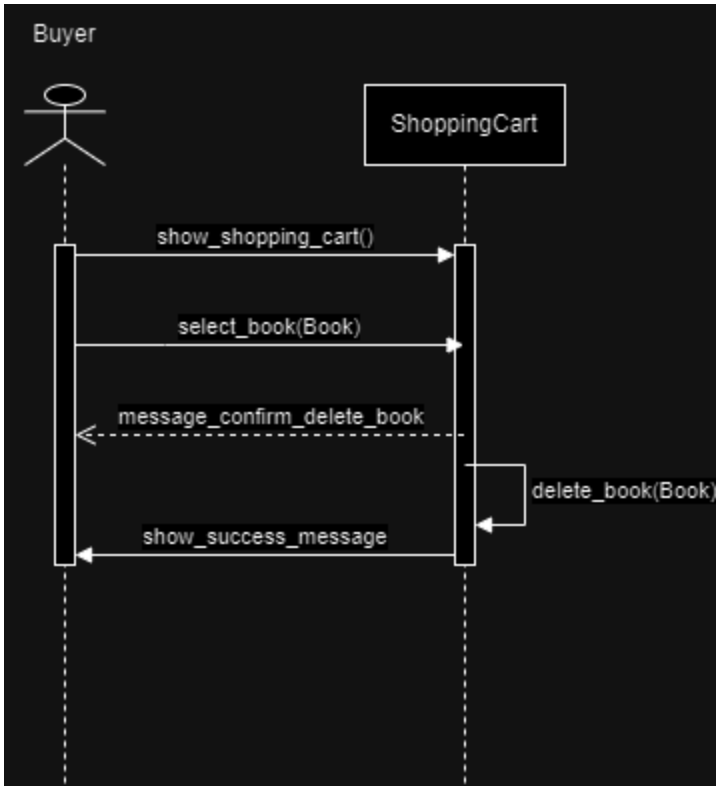
Add books cart:



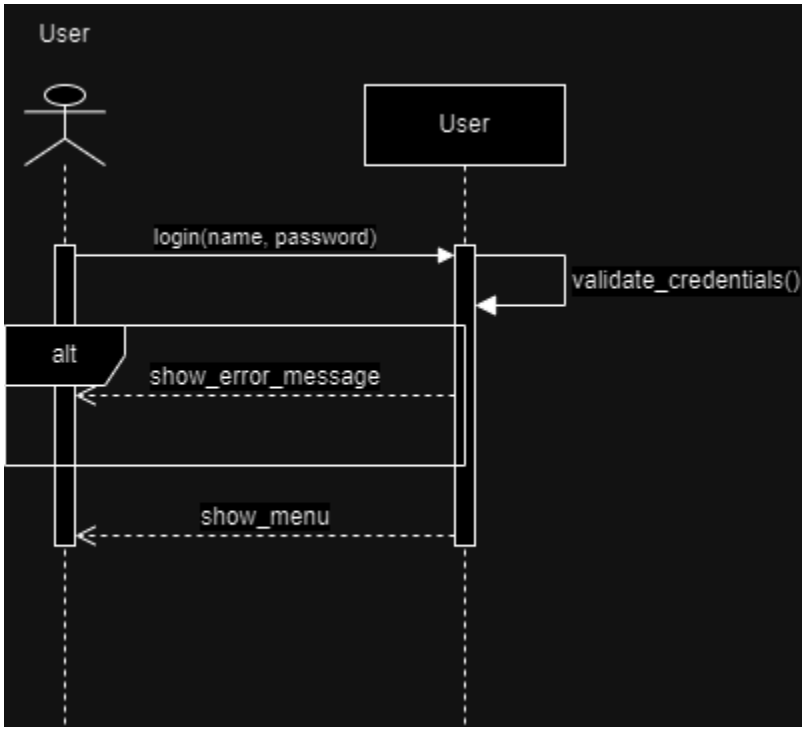
Add new books:



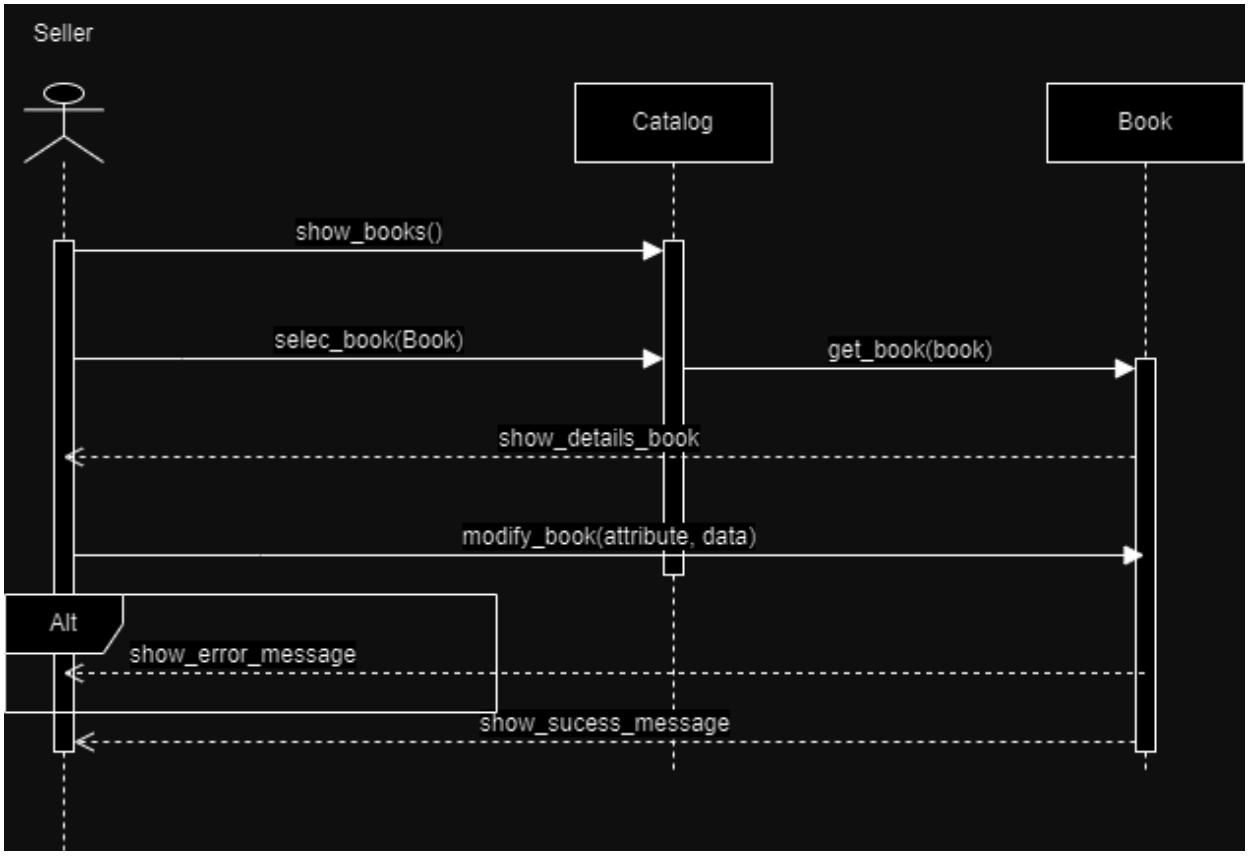
Delete book cart:



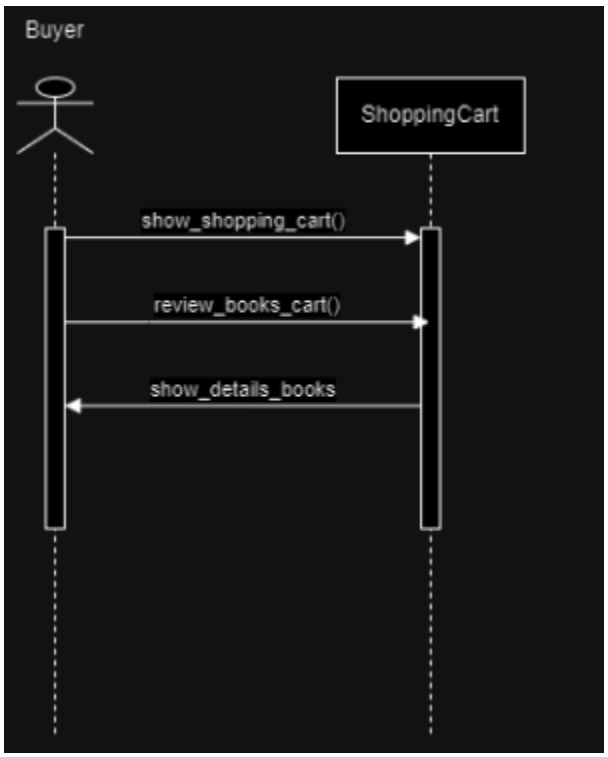
Login



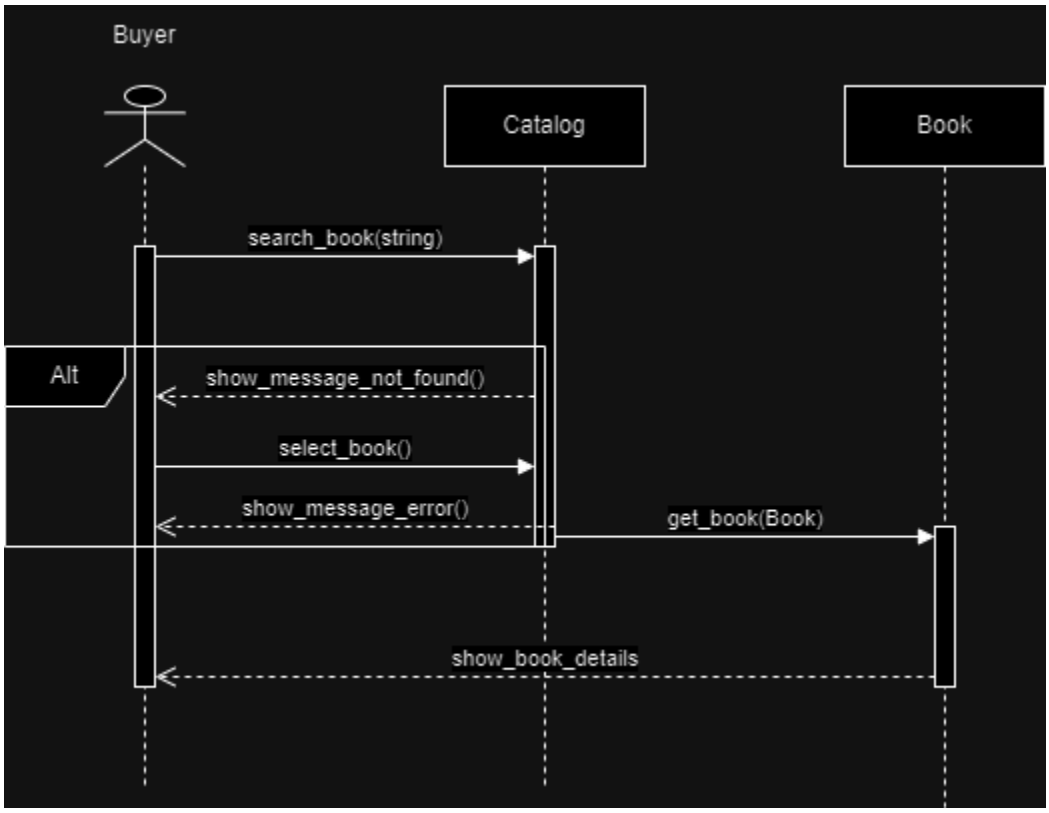
Modify details book:



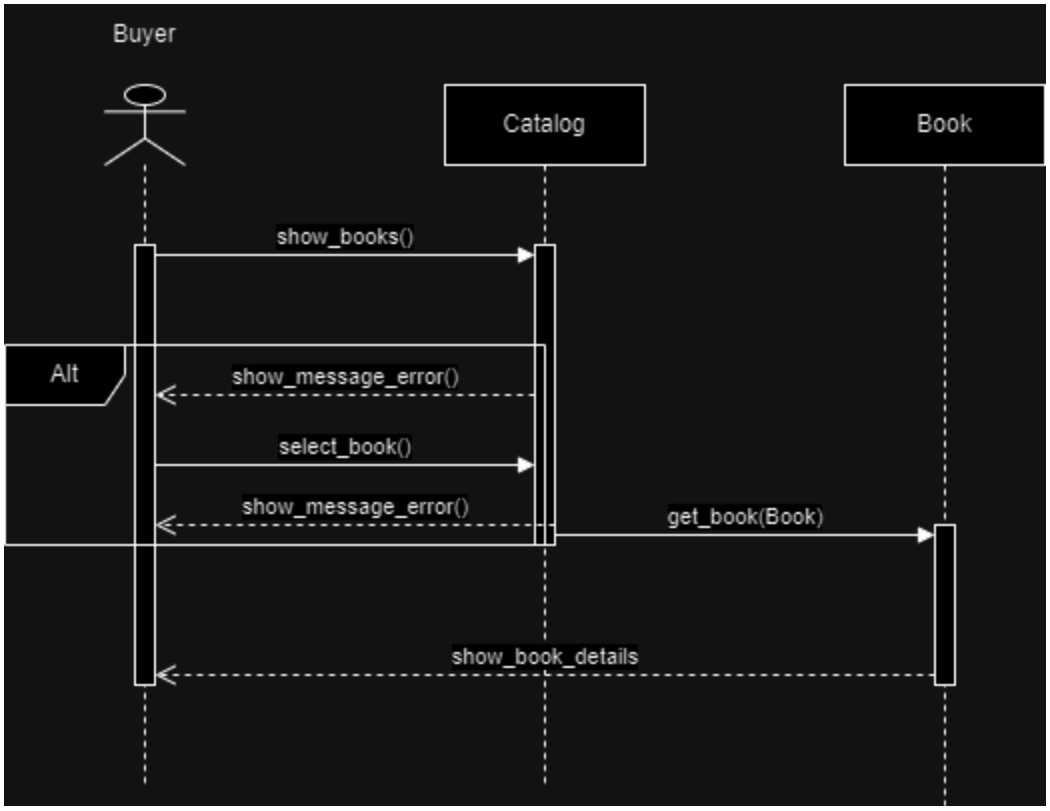
Review books cart:



Search books

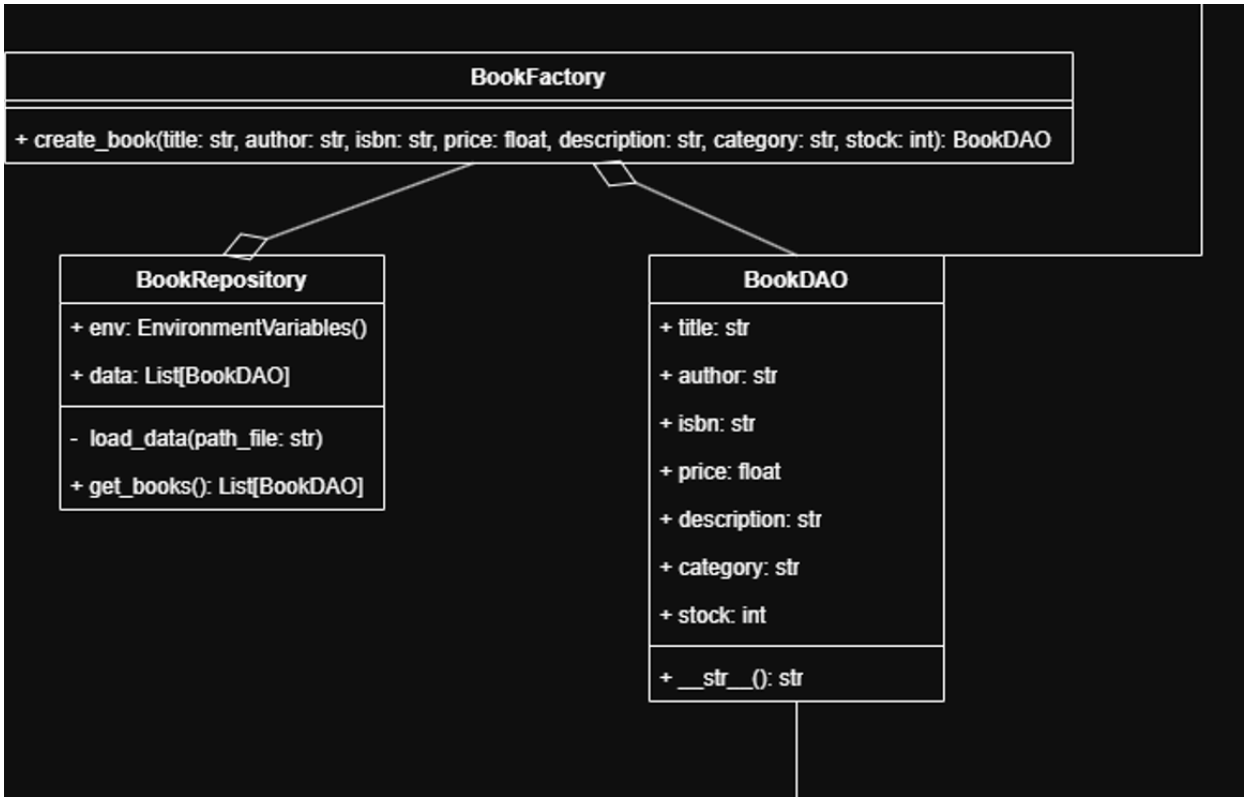


Show Books:

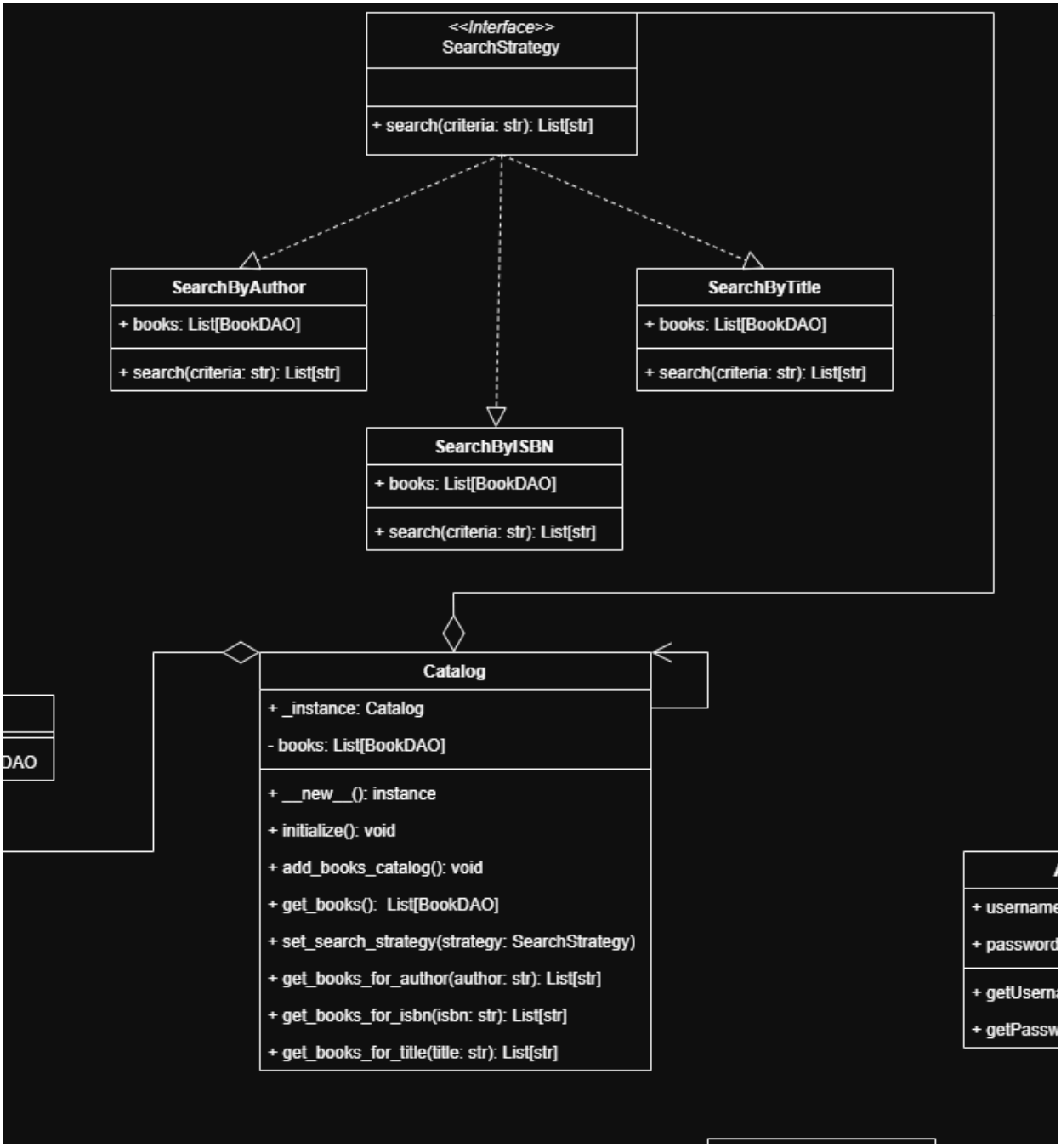


Components Diagram:

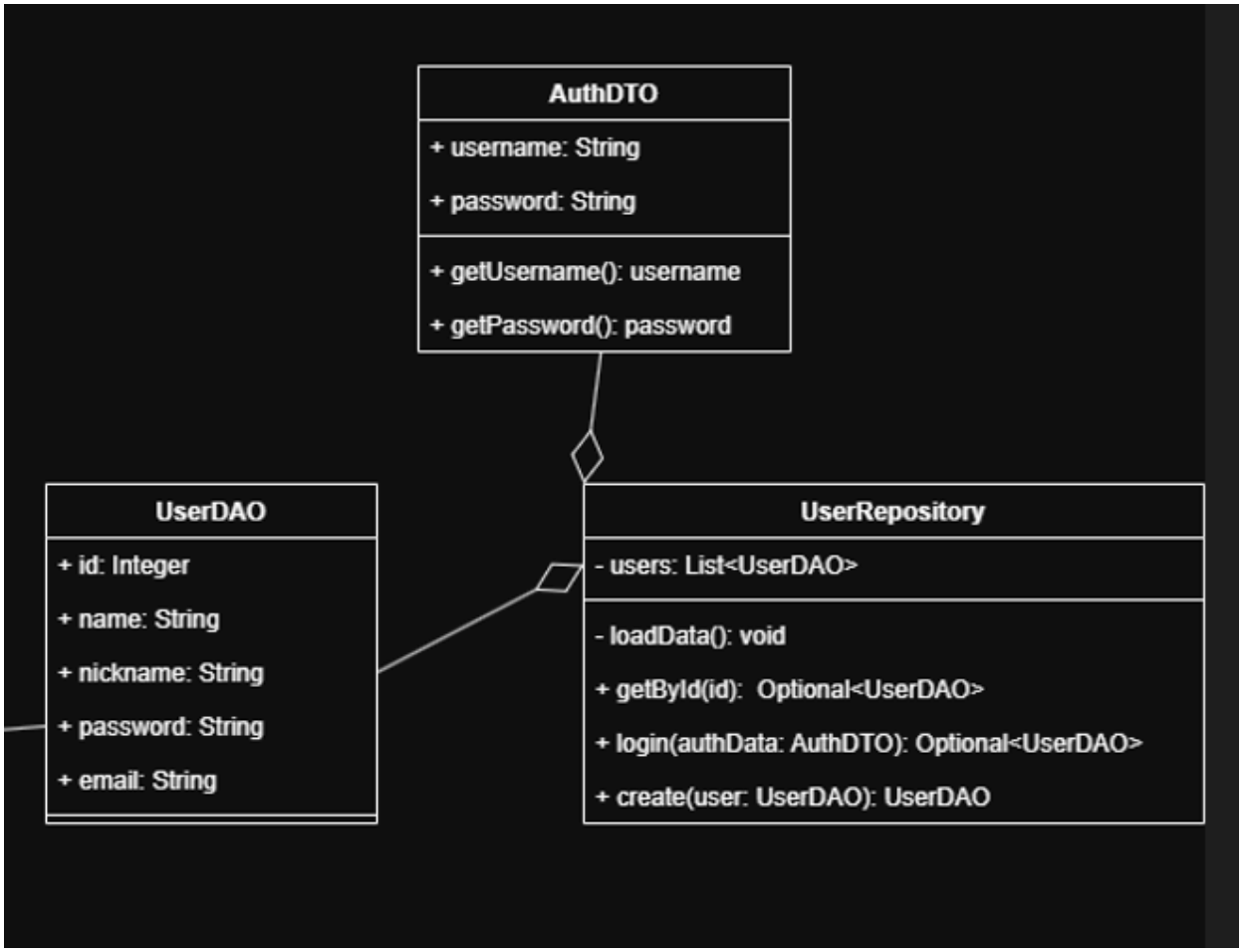
Book Component :



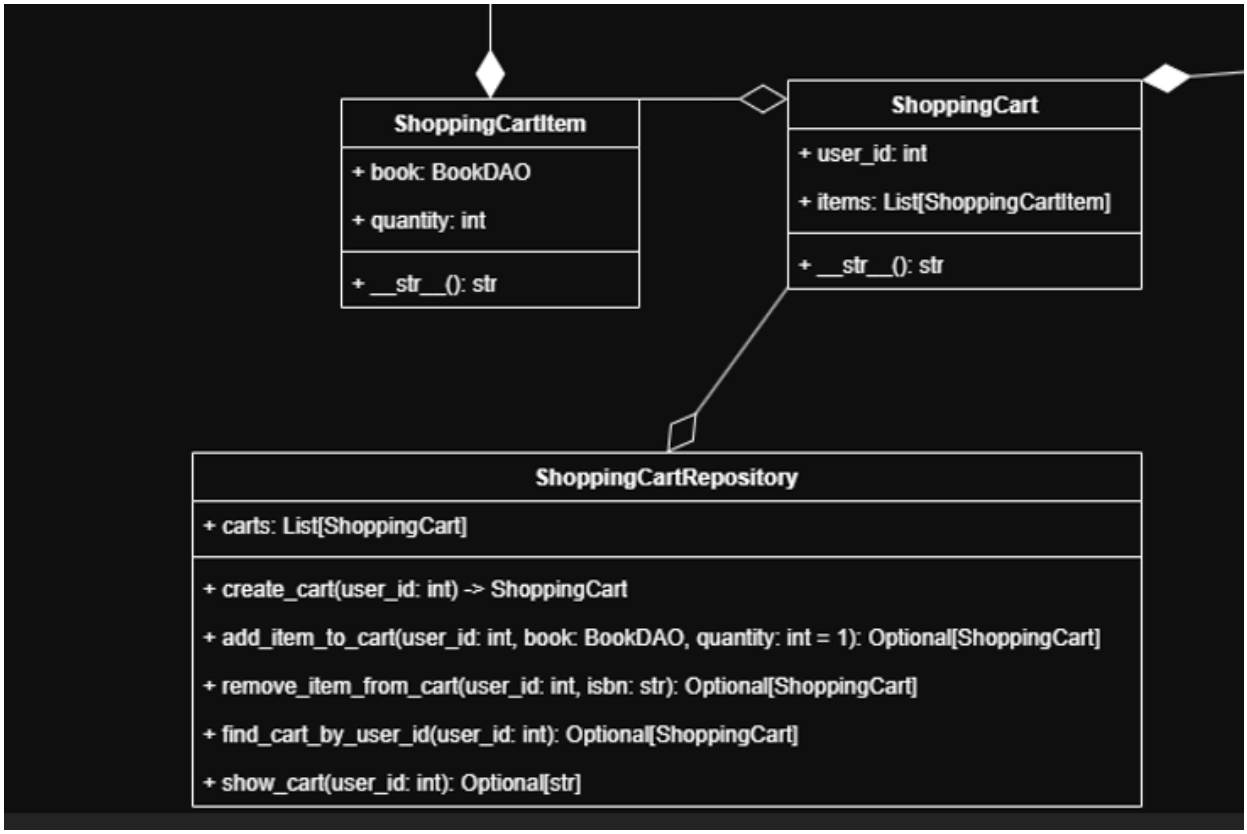
Catalog Component:



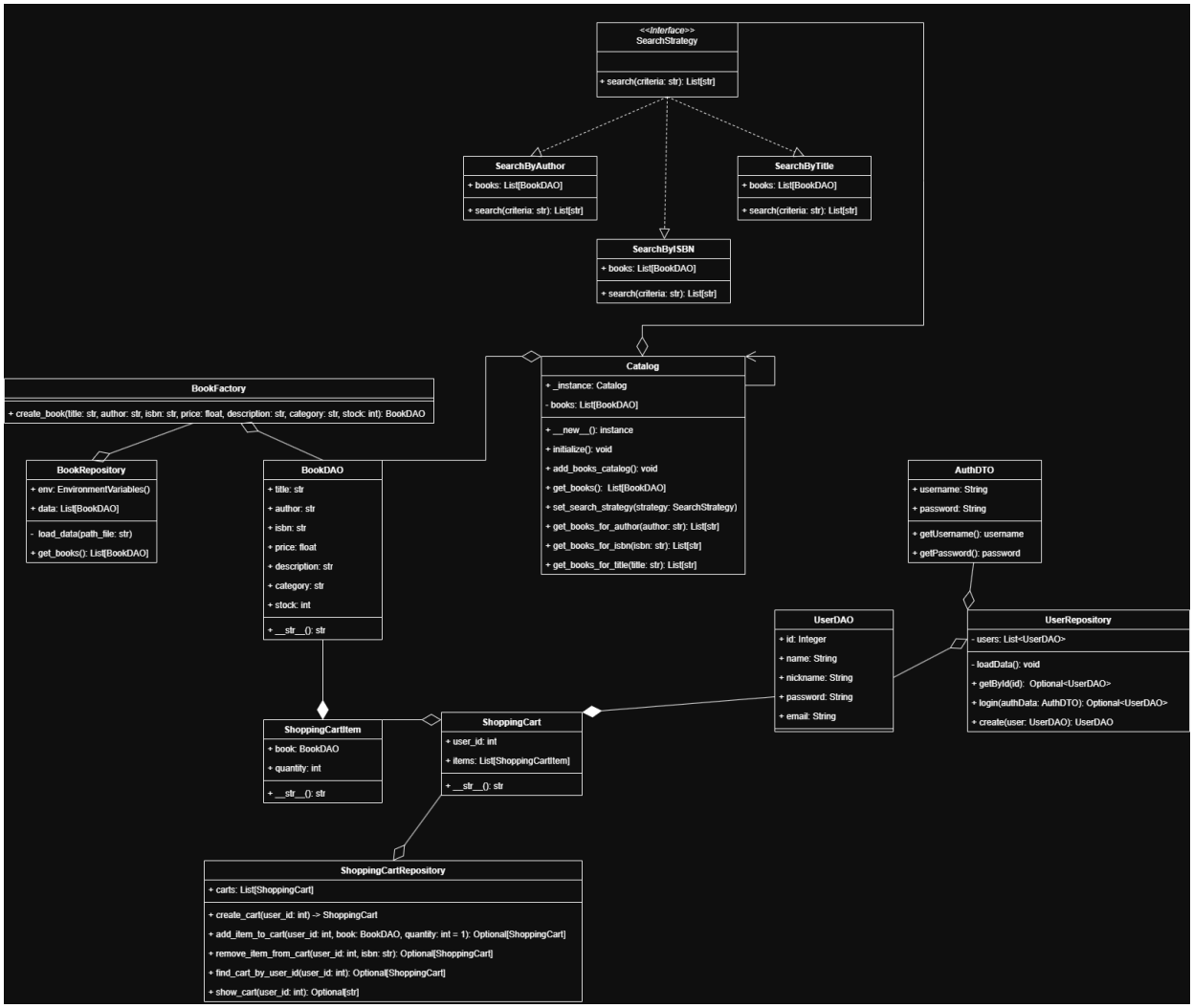
User Component:



ShoppingCart Component:


ShoppingCartItem
+ book: BookDAO
+ quantity: int
+ __str__(): str
ShoppingCart
+ user_id: int
+ items: List[ShoppingCartItem]
+ __str__(): str
ShoppingCartRepository
+ carts: List[ShoppingCart]
+ create_cart(user_id: int) -> ShoppingCart
+ add_item_to_cart(user_id: int, book: BookDAO, quantity: int = 1): Optional[ShoppingCart]
+ remove_item_from_cart(user_id: int, isbn: str): Optional[ShoppingCart]
+ find_cart_by_user_id(user_id: int): Optional[ShoppingCart]
+ show_cart(user_id: int): Optional[str]

Class Diagram:



Conclusions:

The Web Book Store project features a well-structured, modular and scalable architecture, aligned with SOLID principles and supported by design patterns such as Factory Method, Strategy and Singleton, which optimize the creation, search and management of the book catalog. The application facilitates user interaction through an efficient shopping cart and an adaptable search system, while administrators can manage the product offering in an intuitive way. Its modular design based on layers (repositories, services, controllers) allows easy expansion and maintenance, ensuring that new functionalities can be incorporated without affecting the existing structure. In addition, common anti-patterns have been avoided, maintaining clean, understandable and well-documented code, which guarantees its long-term maintainability.

In terms of technology, the adoption of FastAPI as the main framework provides an agile and efficient backend, while Docker reinforces the portability of the system, ensuring its execution in various environments without conflicts. Proper dependency management through Poetry (Python) and Maven (Java) ensures a stable and reproducible ecosystem for developers. However, the system can still benefit from the implementation of more thorough unit and integration tests, as well as improved security through a robust authentication system to control access to critical functionalities. Also, large-scale query optimization, through techniques such as indexing and caching, would help improve performance on databases with a high volume of data. In conclusion, this project not only follows best practices in software design, but is also prepared to scale and evolve over time, consolidating itself as a solid foundation for the development of efficient and sustainable e-commerce applications.