Darsh Gangakhedkar
12/5/2024
Kathleen Kanemoto
CPSC-39

Library Management Final Report

What is it?

The Library Management System is an interactive application designed to manage books efficiently by providing features such as adding, removing, searching, listing, borrowing, and returning books. It also handles scenarios where books are unavailable by using a queue system for borrowing. The application is useful for organizing library operations and educational for understanding object-oriented programming, data structures, and algorithms.

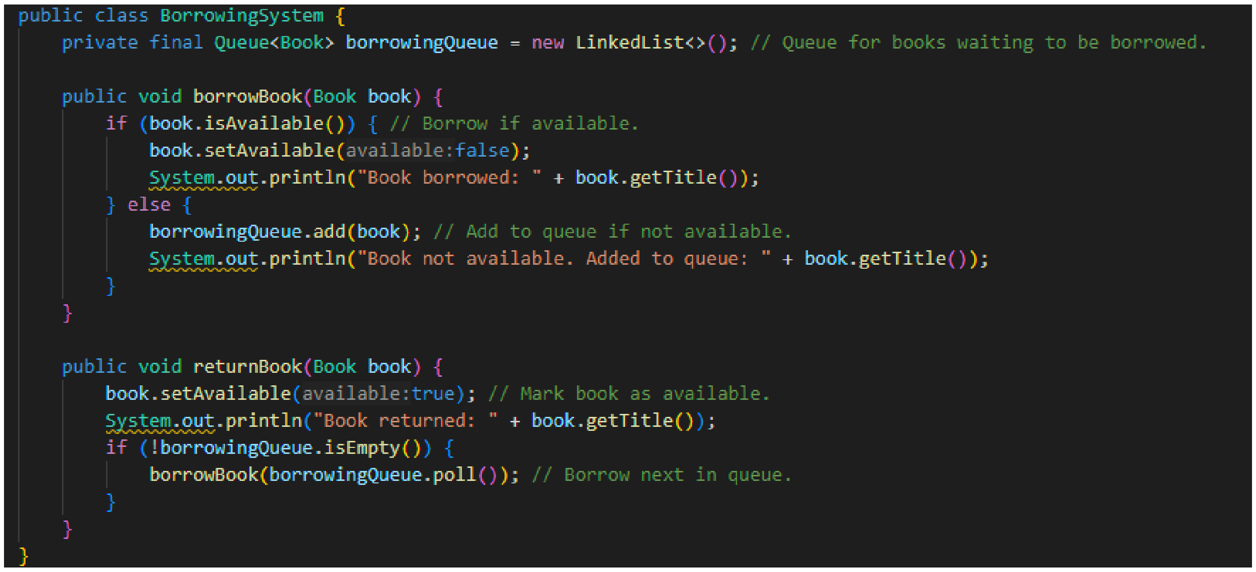
Algorithm Design

Algorithm 1: Borrowing System with Queue Management

This algorithm manages the borrowing and returning of books, ensuring availability and handling a queue for unavailable books.

Steps:

1. Check if the book is available using `isAvailable()`.
2. If the book is available:
    - Mark it as unavailable using `setAvailable(false)`.
    - Print a message confirming the book has been borrowed.
3. If the book is unavailable:
    - Add the book to the queue (`borrowingQueue.add()`).
    - Print a message that the book has been added to the queue.
4. When returning a book:
    - Mark it as available using `setAvailable(true)`.
    - If the queue is not empty, poll the next book in the queue and automatically borrow it.



```java
public class BorrowingSystem {
    private final Queue<Book> borrowingQueue = new LinkedList<>(); // Queue for books waiting to be borrowed.

    public void borrowBook(Book book) {
        if (book.isAvailable()) { // Borrow if available.
            book.setAvailable(available:false);
            System.out.println("Book borrowed: " + book.getTitle());
        } else {
            borrowingQueue.add(book); // Add to queue if not available.
            System.out.println("Book not available. Added to queue: " + book.getTitle());
        }
    }

    public void returnBook(Book book) {
        book.setAvailable(available:true); // Mark book as available.
        System.out.println("Book returned: " + book.getTitle());
        if (!borrowingQueue.isEmpty()) {
            borrowBook(borrowingQueue.poll()); // Borrow next in queue.
        }
    }
}
```

Algorithm 2: Library Book Management System

This algorithm integrates multiple library management functionalities, including adding, removing, searching, and listing books.

Steps:

1. Add a Book:
    - Add the book to the `ArrayList` of books.
    - Insert it into the `HashMap` for efficient lookup by title.
2. Remove a Book:
    - Remove the book from the `HashMap` using its title as the key.
    - Remove the book from the `ArrayList` if it exists.
3. Search for a Book:
    - Use the `HashMap` to perform a case-insensitive search by title.
4. List All Books:
    - Iterate through the `ArrayList` and print each book.

Code Snapshot:

```java
public void addBook(Book book) {
    books.add(book);
    bookMap.put(book.getTitle().toLowerCase(), book);
}
```

```java
public void removeBook(String title) {
    Book book = bookMap.remove(title.toLowerCase());
    if (book != null) {
        books.remove(book);
        System.out.println("Book removed: " + title);
    } else {
        System.out.println("Book not found.");
    }
}

public Book searchBook(String title) {
    return bookMap.get(title.toLowerCase());
}

public void listBooks() {
    books.forEach(System.out::println);
}
```

Algorithm 3: Library Management System Control Flow

This algorithm controls the flow of the application, handling user inputs and delegating tasks to various subsystems.

Steps:

1. Display a menu of options (e.g., Add Book, Remove Book, Search Book, etc.).
2. Read user input using a scanner.
3. Match the user input to an action using a `switch` statement.
   - Add, remove, search, list, borrow, or return a book by calling the appropriate methods from `Library` and `BorrowingSystem`.
4. Exit the program if the user selects the exit option.

Code Snapshot:

```java
public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in); // Scanner for reading user input
        int choice; // Stores the user's menu selection

        while (true) { // Infinite loop to keep the program running until the user chooses to exit
            logger.info(() -> String.join(System.lineSeparator(),
                "Library Management System",
                "1. Add Book",
                "2. Remove Book",
                "3. Search Book",
                "4. List Books",
                "5. Borrow Book",
                "6. Return Book",
                "7. Sort Books",
                "8. Exit",
                "Enter your choice:"
            ));
            choice = scanner.nextInt(); // Get the user's choice
            scanner.nextLine(); // Del the newline character so it doesn't mess with input

            // Handle the user's choice
            switch (choice) {
                case 1 -> addBook(scanner); // Add a book to the library
                case 2 -> removeBook(scanner); // Remove a book from the library
                case 3 -> searchBook(scanner); // Search for a book by its title
                case 4 -> LIBRARY.listBooks(); // List all books in the library
                case 5 -> borrowBook(scanner); // Borrow a book (or queue for it if unavailable)
                case 6 -> returnBook(scanner); // Return a borrowed book
                case 7 -> LIBRARY.sortBooks(); // Sort the books by title
                case 8 -> {
                    logger.info("Exiting Library Management System. Goodbye!"); // Friendly exit message
                    scanner.close(); // Close the scanner to free up resources
                    return; // End the program
                }
                default -> logger.warning("Invalid choice. Please try
again."); // Warn the user about invalid input
            }
        }
    }
```

The Library Management System has three main algorithms: the Borrowing System with Queue Management, the Library Book Management System, and the Control Flow Mechanism, all of which make the app functional and efficient. The **Borrowing System** handles book availability and borrowing requests. If a book is available, it's marked as borrowed. If it's not, the book is placed in a queue, ensuring fairness by automatically lending it to the next user when it's returned. To create this, I used Java's `Queue` interface along with logical checks to manage availability and transitions effectively. The operations in this system, like checking availability and updating the queue, have a time complexity of O(1).

The **Library Book Management System** is the heart of the app, managing all the book records. It uses an `ArrayList` to maintain an ordered collection for sorting and listing and a `HashMap` for fast case-insensitive searching by title. I made sure the methods for adding and removing books updated both structures to keep everything consistent. The search functionality is efficient with O(1) complexity due to the `HashMap`, while listing and sorting books involves iterating over the `ArrayList`, making listing O(n) and sorting O(n log n) using Java's TimSort.

The **Control Flow Mechanism** ties everything together, creating an interactive user experience. It displays a menu, reads the user's choice, and uses a `switch` statement to handle different actions like adding, removing, or borrowing books. I got the idea to use a `switch` statement for organizing menu options from an example on GeeksforGeeks. It was really helpful to see a structured approach for managing menu interactions. While I used ChatGPT for guidance and optimization, I wrote and refined the algorithms myself, making sure they were tailored to the specific needs of this project. The control flow's time complexity for handling user input and delegating tasks is O(1) for action selection since the `switch` statement evaluates the user's choice in constant time. However, the overall complexity depends on the invoked operations, such as O(n log n) for sorting or O(n) for listing books, making the control flow efficient for all interactions.

Data Structures Used

The Library Management System uses several data structures, each chosen to fit the specific needs of the application efficiently. An `ArrayList` is used to store the collection of books, as it provides dynamic resizing and allows for straightforward iteration when listing or sorting books. The `HashMap` is employed for case-insensitive book lookups by title, offering O(1) time complexity for search operations. By storing titles as lowercase keys, it ensures quick and accurate retrieval, even with user input variations in case. A `Queue` (implemented using a `LinkedList`) is used in the Borrowing System to manage unavailable books. This ensures a First In, First Out order, maintaining fairness when lending books to users waiting in line. These data structures were chosen for their ability to handle frequent operations—searching, sorting, and updating the library catalog—while keeping the implementation intuitive. Together, they create a system that balances simplicity with efficiency, ensuring the library operations run smoothly.

Opportunity?

While working on the Library Management System, I noticed a issue in how the borrowing process handled unavailable books. Initially, the system only allowed users to borrow or return books based on whether they were available, but it didn't account for situations where multiple users might request the same unavailable book. This felt like an opportunity to make the system more realistic and user-friendly by introducing a queue for borrowing.

To address this, I decided to implement a Queue to manage borrowing requests for unavailable books. This way, when a book is returned, it automatically gets lent to the next user waiting for it. Adding this feature required revisiting the return logic to check for queued users and update the borrowing state seamlessly. It was a small but meaningful improvement that made the system fairer and more aligned with how real libraries operate. This addition not only improved the overall functionality but also added depth to the system, making it feel more complete.

Issues?

During the development of the Library Management System, I encountered a significant issue with ensuring the correct directory structure matched the `package` declaration in all files. Despite placing all source files in a `library` folder under the `src` directory, I repeatedly faced errors like "Package declaration should match source file directory" and "Cannot find symbol", which prevented the program from compiling and running correctly. The problem was caused by a mismatch between the Java package declaration (`package library;`) and the physical directory structure or incorrect IDE settings. Additionally, cached or incorrectly compiled files from earlier attempts were interfering with the following runs, compounding the issue.

To resolve this, I first verified the directory structure, ensuring that all files were located in `src/library`, matching the `package library;` declaration. I then updated my VS Code settings by modifying `.vscode/settings.json` to mark `src` as the source root, ensuring the IDE correctly recognized the package structure. Next, I deleted all previously compiled `.class` files to avoid conflicts from earlier builds and recompiled the project using the command `javac -d bin src/library/*.java`, which placed the compiled files in the correct `bin/library` directory. Finally, I tested the program to confirm the issue was resolved and that no package-related errors persisted.

This process taught me the importance of maintaining a proper directory structure, performing clean compilations, and correctly configuring the development environment. Resolving this issue not only allowed the program to run successfully but also highlighted the critical role of environment setup in software development, reinforcing that attention to these details is as essential as writing clean code.

New Features?

In the next version of the Library Management System, I would add persistent storage using a database or file-based system to save book and user data across sessions. I would also implement a graphical user interface to make the application more user-friendly. Additionally, I would automate penalty tracking by adding due dates for borrowed books and calculating overdue charges(a feature I was trying to implement but scrapped). These changes would make the system more practical and user-friendly for real-world use.