1 Overview

This document describes the design and implementation of a **Book Repository Management System**, a C++ project that manages a collection of books and authors. The system's primary purpose is to allow users to add, view, and manage books along with their associated information (title, year of publishing, authors, genre). It supports two user roles: admin with display/edit rights and customer with only display access to the repository. The application is console-based, providing a text menu interface for interactions, and it includes functionality to persist data by saving and loading the repository contents to a file between sessions.

2 Project Structure and Design

The project is organized into a directory structure that separates header files, source files, and the main program. All header files are located in the `include/` directory, implementation files (.cpp) are in the `lib/` directory, and the menu interface (Menu.cpp), utility functions (utils.cpp) and main application code reside in `src/`. After compilation, the output is an executable (e.g., `app.exe`) that uses the compiled classes.

Key files and modules in the system include:

- **Book**: Defines a Book entity with attributes: title, publication year, genre, an auto-incremented bookID and a vector of pointers to its associated authors.
- **Author**: Defines an Author entity with attributes first and last name, auto-incremented authorID and a reference count. Authors and books have a one-to-many relationship (one author can write multiple books). The reference counts stores the number of books the author is associated with and is later used to manage deletion of author objects.
- **Genre**: An enumerated type (defined in `Genre.h`) that classifies books by genre (e.g., fiction, article etc.).
- **BookRepo**: A repository class responsible for managing the collection of Book and Author objects. `BookRepo` provides methods to add new books/authors, display them, and handle persistence (through file load/save functionality).
- **Menu**: Encapsulates the text-based user interface logic. The Menu class displays options to the user and processes user input. It utilizes function pointers (via a `typedef` for a function pointer type) to map menu selections to the appropriate action handlers, making the menu logic easily extensible.

- **User**, **UserState**, **AdminState**/**CustomerState**, **RoleSelector**: These classes work together to manage user roles using the state design pattern. `UserState` is an abstract class defining the interface for user operations. `AdminState` and `CustomerState` are concrete states representing administrator and customer behaviors, respectively. A `User` (or a context managed by `RoleSelector`) holds a current state and delegates actions to it, allowing the program to switch between admin and customer modes without changing the high-level logic.

- **utils**: A utility module (`utils.h`/`utils.cpp`) that contains helper functions used across the system. These utility functions handle string formatting and input validation (toLower, readInt, splitCommaSeparated) to avoid code duplication and keep the main logic in other classes clean.

3 Efficient Data Management and Search

The system is designed to handle data (books and authors) efficiently, especially as the size of the repository grows. An intial hurdle in the design was how to efficiently and safely check if a book/author already exists. To do this, I developed the following workflow:

- **Hash-based author/book lookup**: In BookRepo, I used an `unordered map` to map author keys (first_last) or IDs to pointers to their `Author` objects. Unordered maps are hash tables with an average constant-time complexity for lookups, insertions, and deletions (O(1) on average for search operations. Book lookup is managed in an unordered set of bookKeys which consist of the book's title (lowercased) followed by "_" and the (sorted) IDs of its associated authors separated with dashes. Of course, in reality, this exact issue has been solved with ISBN's which are unique identifier for every book/edition etc.

- **BookRepo::newBook**: This is the heart of the logic for creating new books and authors. I made the decision to allow new authors to only be created when a new book is created which makes creation and deletion of books and authors much safer. BookRepo::newBook takes the following parameters: title, year of publishing, a vector of author's full names as strings and a genre. First, author names are normalized and turned into authorKeys which are used to check if an author already exists. The pointers of either the newly created or existing authors are added to the book's attribute (vector of authors), after which the existence of the book is also checked via a bookKey. Finally, the new book is created, its key is saved in the unordered map of the BookRepo, the reference count for each author is incremented and the function returns a pointer to this book.

- **BookRepo::deleteBookByID**: Since authors can only be created when a book is created, the deletion of authors is also coupled with the deletion

of books. This function takes the bookID as a parameter and first decrements the ref count of the associated authors. If an author's ref count reaches zero, they are automatically deleted as well. Furthermore, this function cleans up the book and author keys, removes their pointers from the member variables of BookRepo and finally deletes the book object itself.

- **Loading and Saving**: The books and authors of the repository can be saved in a .txt file with entries separated by semi-colons. The order of saving the items was a crucial consideration here; since a book may have multiple authors, the authors are last (4th) element to be saved/loaded. When loading, I set this to be the default case in a switch so that everything at the 4th position and beyond gets loaded as an author object. Example: "The Hobbit;Fantasy;1937;J.R.R. Tolkien".

4 Function Pointers

- **Function pointer typedef for menu actions**: In the menu implementation (`Menu.cpp`), a `typedef` (or type alias) is defined to represent the function pointer type for menu command handlers. This allows the menu to store a list or map of menu options to function pointers (of a uniform type). When the user selects an option, the corresponding function is invoked via the pointer. Using a typedef for these pointers improves code readability and maintainability, since it gives a descriptive name to the pointer-to-function type (for example, `"add_book"`) and abstracts away the raw pointer syntax. It also makes adding new menu options straightforward—one can simply add a new function and include it in the menu's dispatch table.

5 Design Principles and Patterns

5.1 State Pattern for User Roles

One of the key design patterns employed in this project is the *State* pattern [**?**] to handle the different behaviors of admin vs. customer users. Instead of using inheritance to model a User base class with Admin and Customer derived classes, a state pattern encapsulates role-specific functionality in separate state classes. The abstract class `UserState` defines the common interface for actions that a user can perform (defined as pure virtual function), and the two concrete classes `AdminState` and `CustomerState` implement these actions in ways appropriate to their roles by overriding the pure virtual functions defined in UserState. At runtime, the current user is associated with one of these states, and the program prompts the user to select a role (via the `RoleSelector` mechanism. The implementation of the functions available to the User is delegated to the respective state of the User object. This design adheres to the open/closed

principle, allowing new user roles to be added with minimal changes to existing code [**?**]. It also makes the code more modular and easier to maintain, since role-specific logic is isolated in the state classes rather than intermingled with conditional logic. While my implementation in this project is minimal, I found the state design pattern to be a very useful principle to learn as it is much more flexible and clean than creating (a potentially infinite) number of derived classes from a base class.

6 Challenges Faced

During the development of the Book Repository Management System, a few challenges were encountered, including:

- **Supporting multiple authors per book:** The initial implementation was for each book has a single author. Extending this to support multiple authors for a single book turned out to be non-trivial. It requires modifying the data model (e.g., having a list of author pointers in the `Book` class), updating the user interface to allow selecting or displaying multiple authors, and changing the file format and parsing logic to handle multiple author entries per book.
- **Memory management and data consistency:** Because the system dynamically allocates objects for books and authors and stores pointers to them in various containers, proper memory management is a critical challenge. Care must be taken to delete allocated Book and Author objects when they are no longer needed (for example, on program termination or when removing entries) to avoid memory leaks.

7 Possible Improvements and Future Expansion

There are several opportunities to enhance the system:

- **Editing records:** Implement the ability to edit existing book or author information (for example, updating a book's title, changing its genre, or correcting an author's name). This would make the system more flexible and user-friendly, allowing corrections or changes without requiring deletion and re-entry of records.
- **Search and filtering:** Introduce more advanced search capabilities, such as searching books by author name, by genre, or by keywords in titles. For example, a user might want to list all books by a particular author or all books in a certain genre.
- **User accounts and authentication:** If the system is expanded for multiple users, implement a login system where administrators and customers

have to authenticate. This could include storing user credentials and differentiating user sessions, ensuring that only authorized users can perform administrative tasks like adding or removing books.

- **BookIDs**: In the current design, BookIDs are auto-incremented, meaning that if a book repository has IDs 1, 2 and 3, and we delete the book with ID 2, the books 1 and 3 are going to remain with their IDs in the repository. This is because the IDs are a unique and persistent identifier for each book and are not reused after deletion. While this simplifies implementation and ensures uniqueness, it could lead to fragmentation or gaps in the ID sequence over time. A possible improvement could be to implement an ID reuse mechanism or switch to a ISBN-based system for better scalability.