VIET NAM NATIONAL UNIVERSITY - HO CHI MINH CITY
UNIVERSITY OF SCIENCE
FACULTY OF INFORMATION TECHNOLOGY

OBJECT-ORIENTED PROGRAMMING

REPORT
LIBRARY MANAGEMENT SYSTEM

GROUP 4

Theory Lecturer	Dr.	Bui Tien Len
Lab Lecturer	M.Sc.	Truong Tan Khoa
	B.Sc.	Vo Nhat Tan
Students	24127104	Du Hoai Phuc - Leader
	24127272	Ly Huynh Minh Dang
	24127163	Nguyen Son Hai
	24127347	Ta Duc Dung

Contents

Group Introduction

Group details:

ID	Full name	Email	Roles
24127104	Du Hoai Phuc	dhphuc2424@clc.fitus.edu.vn	Backend Developer, Leader
24127272	Ly Huynh Minh Dang	lhmdang2404@clc.fitus.edu.vn	Backend Developer
24127347	Ta Duc Dung	tddung2426@clc.fitus.edu.vn	UI/UX Developer
24127163	Nguyen Son Hai	nshai2431@clc.fitus.edu.vn	Designer, Report Writing

Table 1: Group details

Presentation Video

This is the link to our team's demo gameplay video:
https://drive.google.com/file/d/1u-jRM0ugusIxY0lzi2Ym1pQjp-EEtaHl/view?usp=sharing

Source Code (In Github)

This is the link to our team's source code: https://github.com/nightshade94/Knight-Prequel

Project Requirements Overview

Group	Requirement	Progress
Problem Statement	Library Management System	100%
Programming & Design Requirements	Programming Language	C++
	Object-Oriented Principles	4/4 principles
	Design Patterns	5/2 patterns
User Interface Requirements	Friendly, clearly formatted, and fully interactive user interface	100%
	Console-based UI requirements	100%
	GUI frameworks	ImGUI

Table 2: Project Requirements table

Individual Contributions

Name	Contribution	Detail
Du Hoai Phuc (Team Leader & Core Architecture)	Core Classes Design	User, Member, Librarian, Book, Author, Category, Loan
	Polymorphism Implementation	Virtual functions, runtime behavior selection
	Project Coordination	Timeline management, code integration
	Design Patterns Architect	Observer, Strategy, Decorator patterns
	User Experience Design	System workflow and interaction design
Ly Huynh Minh Dang (Backend Systems Developer)	Data Persistence	CSV-based storage with CSVHandler utility
	Library Management Services	LibraryManager singleton implementation
	Loan Management	Borrowing/returning system with fine calculation
	Business Logic	Core operations and member status management
	Input Validation	Form validation and error handling
Ta Duc Dung (UI/UX Developer)	GUI Implementation	Complete ImGui-based interface
	Authentication System	Login/logout with password validation
	Visual Design	Consistent styling and layout
	Responsive Interface	Smooth navigation and interaction
	User Experience Design	Intuitive menu systems
Nguyen Son Hai (Designer & Tester)	Slide Maker	Presentation materials and diagrams
	Report Writing	Technical documentation
	Tester	System testing and quality assurance

Table 3: Individual Contributions

Self-evaluation

Group	Requirement	Progress
Problem Statement	Library Management System	100%
OOP Principles Implementation	Encapsulation - All classes have private members with controlled public interfaces	10/10
	Inheritance - Clear inheritance hierarchy with User base class	10/10
	Polymorphism - Virtual functions, interface implementations	10/10
	Abstraction - Abstract interfaces for strategies and observers	10/10
	Overall OOP Implementation	**10/10**
Design Patterns	Singleton Pattern - Thread-safe LibraryManager instance management	10/10
	Facade Pattern - LibraryManager as unified subsystem interface	10/10
	Observer Pattern - Book and loan status notifications	10/10
	Strategy Pattern - Search algorithms and penalty systems	10/10
	Decorator Pattern - Enhanced book information display	10/10
	Total Patterns Implemented	**5/2**
User Interface Requirements	GUI Framework Implementation	ImGui 10/10
	Menu Navigation - Multi-level menu with clear structure	10/10
	User Input Handling - Structured data input with validation	10/10
	Output Formatting - Clear display with proper formatting	10/10
	Error Handling - Smooth interaction with error checking	10/10
	Overall User Interface	10/10

Group	Requirement	Progress
Technical Implementation	Programming Language - Modern C++ features and best practices	10/10
	Code Quality - Clean architecture, proper memory management	10/10
	Modularity - Clear separation of concerns with layered architecture	10/10
	Data Persistence - CSV-based storage with automatic loading/saving	10/10
	Overall Technical Implementation	10/10
Documentation & Testing	Code Documentation - Comprehensive comments and class documentation	10/10
	System Testing - Manual testing performed	10/10
	User Manual - Clear usage instructions and examples	10/10
	Overall Documentation & Testing	10/10
Overall Project Completion	Core requirements met with excellent implementation	**100/100**

Table 4: Self-Evaluation Overview

1 Introduction

1.1 Project Overview

This report details the design, implementation, and technical analysis of an **Electronic Library Management System**. The primary motivation for this project is to address the common inefficiencies and limitations found in traditional, manual library operations. These challenges often include error-prone record-keeping, difficulty in tracking resource availability, and a lack of proactive communication with library members.

This project leverages Object-Oriented Programming (OOP) principles and established software design patterns to deliver a robust, maintainable, and efficient software solution [1]. The resulting application serves as a practical demonstration of OOP concepts applied to a real-world management problem [3].

1.2 Goals and Objectives

The project was guided by a set of clear technical and academic objectives. Figure 1 visually summarizes the core goal and its supporting objectives. The primary goals were:

- **Functional:** To develop a fully functional console-based application capable of managing core library operations.
- **Academic:** To apply and demonstrate a strong, practical understanding of the four pillars of OOP.
- **Technical:** To explore and implement appropriate software design patterns to improve the system's design.
- **Practical:** To gain hands-on experience in the software development lifecycle.

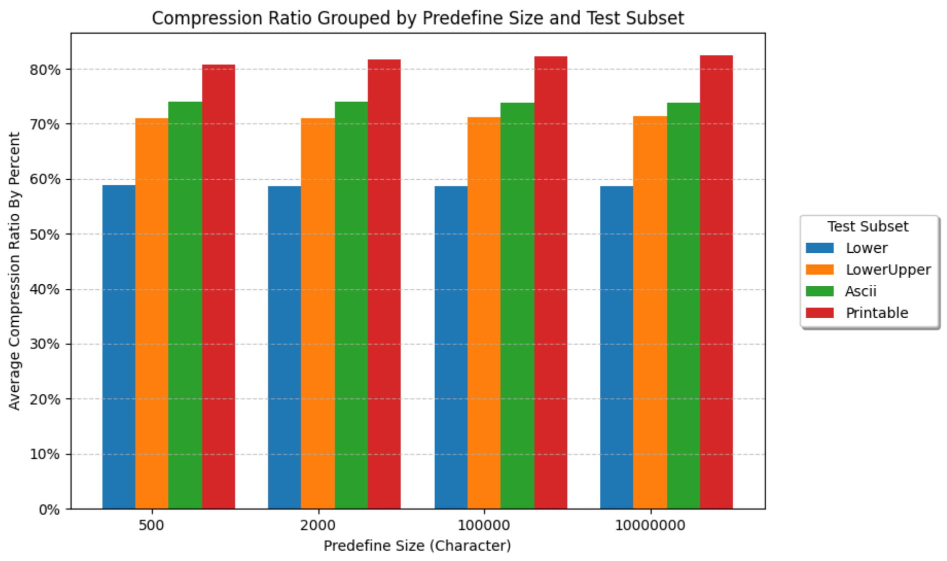

Figure 1: Visual Representation of Project Objectives.

1.3 Scope of the Project

1.3.1 In Scope

The features and functionalities implemented in the final system include:

- A secure authentication system for both members and librarians.
- Full CRUD (Create, Read, Update, Delete) functionality for the book catalogue.
- A complete loan management system.
- An interactive console-based user interface (CLI).
- Data persistence using local CSV files with a hashing mechanism for integrity.

1.3.2 Out of Scope

The following features were considered but explicitly excluded:

- A Graphical User Interface (GUI).
- Integration with a real-time relational database.
- Web-based or mobile client interfaces.

1.4 Report Organization

This report is structured into the following sections to guide the reader from the project's conception to its technical details and conclusion:

Section 1 - Introduction: Outlines the project's purpose, goals, scope, and provides a roadmap for the document.

Section 2 - System Analysis and Design: Delves into the high-level design of the application, including the software architecture and the primary UML class diagram.

Section 3 - Implementation and OOP Principles: Focuses on the implementation details, providing concrete code examples to demonstrate how the core principles of OOP were applied.

Section 4 - Software Design Patterns: Provides a detailed analysis of the design patterns used in the project, explaining their purpose and implementation.

Section 5 - Application Walkthrough: Demonstrates the system from a user's perspective, using sequence diagrams and screenshots to illustrate key functionalities.

Section 6 - Conclusion: Summarizes the project's outcomes, discusses the challenges faced, and proposes directions for future work.

2 System Analysis and Design

This section outlines the high-level design of the Electronic Library Management System, covering the functional requirements, the software architecture, and the static structure of the core classes.

2.1 Functional Requirements

The system is designed to fulfill a comprehensive set of functional requirements, enabling efficient management of library operations. The core functionalities are:

- **User Authentication:** Provides secure registration and login mechanisms for both Members and Librarians.
- **Book Management:** Supports full CRUD (Create, Read, Update, Delete) operations for the library's book collection.
- **Search Functionality:** Allows users to search the book catalog by title, author, and category, facilitating easy resource discovery.
- **Loan Management:** Manages the entire lifecycle of a book loan, including borrowing, returning, tracking loan status (active, returned, overdue), and calculating fines for overdue items.
- **User Account Management:** Enables librarians to manage member accounts and allows members to view their profiles and loan history.
- **Data Integrity:** Implements a CSV file hashing and verification mechanism to protect against unauthorized data tampering.
- **Notification System:** Capable of notifying users about important events, such as impending loan due dates.

2.2 System Architecture

The application employs a modular architecture to separate concerns and enhance maintainability[4]. The codebase is organized into logical components, each with a distinct responsibility:

`core/` This directory contains the domain models, which represent the fundamental entities of the system. Key classes include `Book`, `User`, `Member`, `Librarian`, and `Loan`.

`services/` This component encapsulates the primary business logic. It includes service classes like `LibraryManager` (which acts as a central facade for library operations), `AuthManager` for handling authentication, and `NotificationService`.

`patterns/` Contains the implementations of the software design patterns used throughout the project, such as `Observer`, `Strategy`, and `Decorator`. This isolates pattern-specific code.

`utils/` Provides various utility classes that support the core application logic. This includes the `CSVHandler` for data persistence, date utilities, and input validators.

`data/` This directory serves as the data storage layer, containing the CSV files (`books.csv`, `members.csv`, etc.) that the application uses for data persistence.

2.3 UML Class Diagram

The static structure of the system is visualized in the UML Class Diagram below [1] (Figure 2). It illustrates the key classes, their attributes, methods, and the relationships between them, such as inheritance and association.

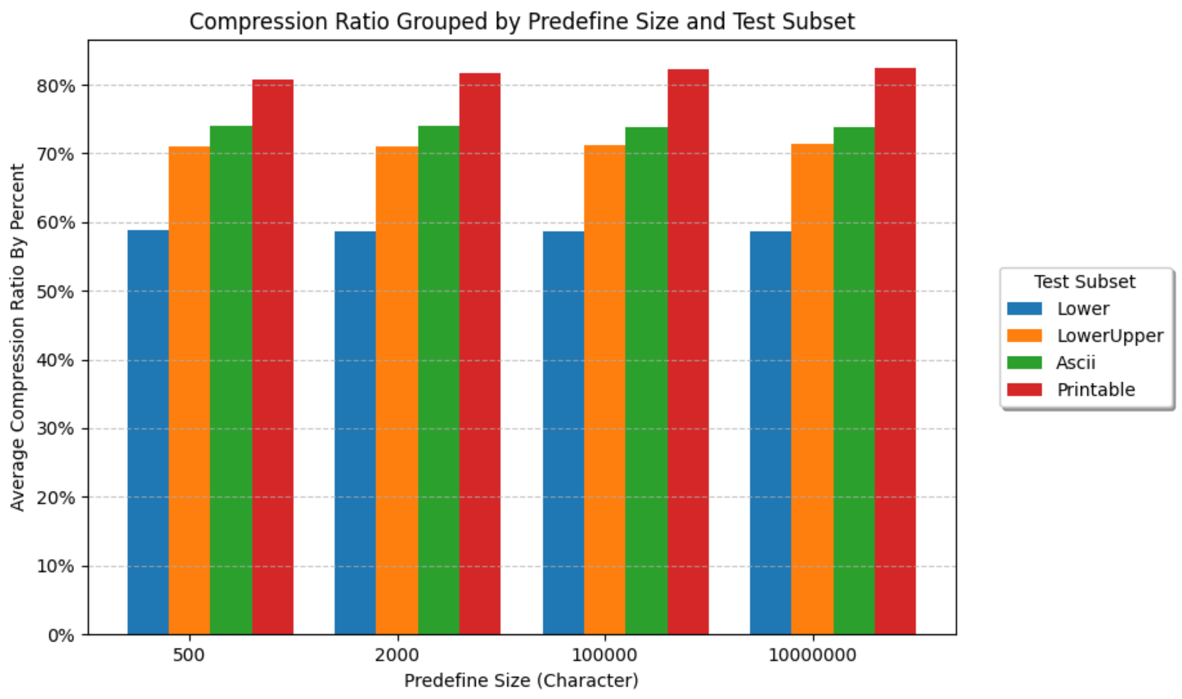

Figure 2: UML Class Diagram of the Library Management System.

3 Implementation and Object-Oriented Principles

This section delves into the implementation details of the system, with a specific focus on how the four fundamental principles of Object-Oriented Programming (OOP) [1] were realized using the features of the C++ programming language [6]. The use of these principles was crucial for developing a system that is not only functional but also robust, modular, and maintainable.

3.1 Encapsulation

Encapsulation is the practice of bundling data and the methods that operate on that data into a single unit, while restricting direct access. This is achieved in C++ through access specifiers (‘public‘, ‘private‘, ‘protected‘).

In this project, all domain models enforce encapsulation. For example, the `User` class declares its attributes `userID` and `password` as `private`. Access from outside the class is only permitted through its public interface, protecting the object’s internal state.

```
// -- Demonstrating Encapsulation in C++ --
class User {
private:
    // Data members are hidden from outside access
    std::string userID;
    std::string password;

public:
    // Public methods provide a controlled interface
    std::string getUserID() const;
    void setPassword(const std::string& newPassword);
};
```

3.2 Inheritance

Inheritance allows a new class (subclass) to be based on an existing class (base class), promoting code reusability. C++ supports this directly through class derivation.

The system utilizes inheritance to model user types. The `Member` and `Librarian` classes both inherit publicly from the base `User` class. They acquire common attributes and functionalities while also implementing their own specialized methods.

```
// -- Demonstrating Inheritance in C++ --

// Base class
class User { /* ... */ };

// Derived class Member "is-a" User
class Member : public User {
private:
    // Member-specific attributes
    std::vector<Loan*> borrowedBooks;
public:
    // Member-specific behavior
    void borrowBook(const std::string& isbn);
};
```

3.3 Polymorphism

Polymorphism allows objects of different classes to be treated as objects of a common superclass. In C++, this is primarily achieved at runtime through `virtual` functions.

A clear example is the Decorator pattern for books. The base `Book` class declares a `virtual` method `getFullDescription()`. Decorator classes override this method. When `getFullDescription()` is called on a `Book*` pointer, C++'s dynamic dispatch mechanism ensures the correct overridden version is invoked at runtime.

```
// -- Demonstrating Polymorphism in C++ --

// Base class defines the virtual interface
class Book {
public:
    virtual std::string getFullDescription() const {
        return "Title: " + this->title;
    }
};

// Decorator class overrides the virtual function
class SpecialTagDecorator : public Book {
private:
    Book* bookComponent;
    std::string tag;
public:
    std::string getFullDescription() const override {
        // Dynamically calls the wrapped object's method
        return bookComponent->getFullDescription() + " [Tag: " + tag + "]";
    }
};
```

3.4 Abstraction

Abstraction involves hiding complex implementation details and exposing only essential functionalities. In C++, this is often achieved using abstract classes (classes with one or more pure virtual functions).

This principle is used in the Strategy pattern. An abstract class, `ISearchStrategy`, is defined with a pure virtual function `search() = 0`. This creates an interface, or a contract, that all concrete search strategy classes must implement, decoupling the client from the specific search algorithms.

```
// -- Demonstrating Abstraction in C++ --

// Abstract base class (Interface)
class ISearchStrategy {
public:
    virtual ~ISearchStrategy() = default;

    // Pure virtual function defines a contract
    virtual std::vector<Book*> search(
        const std::vector<Book*>& books,
        const std::string& query) = 0;
};

// Concrete implementation must provide the search method
class TitleSearchStrategy : public ISearchStrategy {
public:
    std::vector<Book*> search(
        const std::vector<Book*>& books,
        const std::string& query) override {
        // ... specific implementation for searching by title
    }
};
```

4 Software Design Patterns

To address recurring design challenges and enhance the system's modularity, flexibility, and maintainability, several established software design patterns were implemented [3]. This section provides a detailed analysis of each pattern, explaining its purpose and its specific implementation within the project.

4.1 Singleton Pattern

4.1.1 Purpose

The Singleton pattern ensures that a class has only one instance and provides a single, global point of access to it. This is useful for objects that need to coordinate actions across the system, such as a central manager or a configuration handler.

4.1.2 Implementation

In this project, the Singleton pattern is applied to the `LibraryManager` class. Since the library's state (its collection of books, members, and loans) must be consistent throughout the application, it is critical to have only one object managing it. The `LibraryManager` class implements a private constructor to prevent external instantiation and a public static method, `getInstance()`, which returns the sole instance of the class.

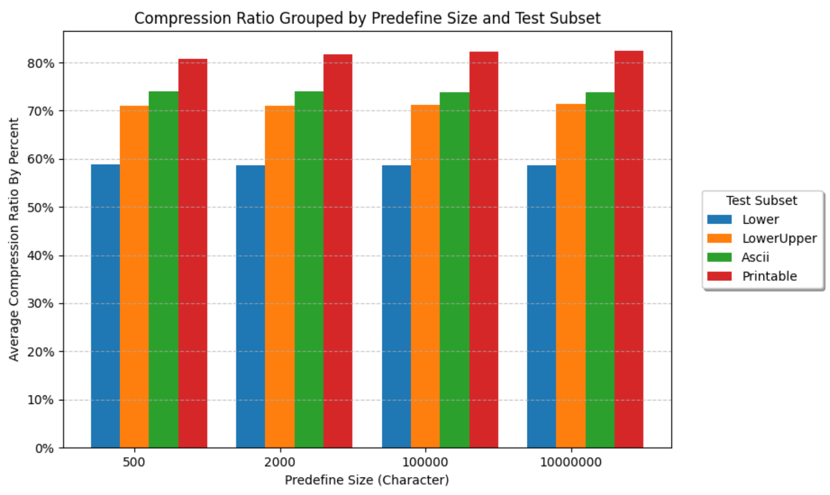

Figure 3: UML Diagram for the Singleton Pattern.

4.2 Observer Pattern

4.2.1 Purpose

The Observer pattern defines a one-to-many dependency between objects. When one object (the "subject") changes its state, all its dependents (the "observers") are notified and updated automatically. This pattern is ideal for creating distributed event-handling systems.

4.2.2 Implementation

The Observer pattern is the backbone of the `NotificationService`. In our system, entities like `Loan` act as subjects. Observers, such as a `MemberObserver` or `LibrarianObserver`, can register themselves with a subject. When a loan's status changes (e.g., becomes overdue), the loan subject notifies all its registered observers, allowing the system to send alerts or take other actions without tightly coupling the `Loan` class to the notification logic.

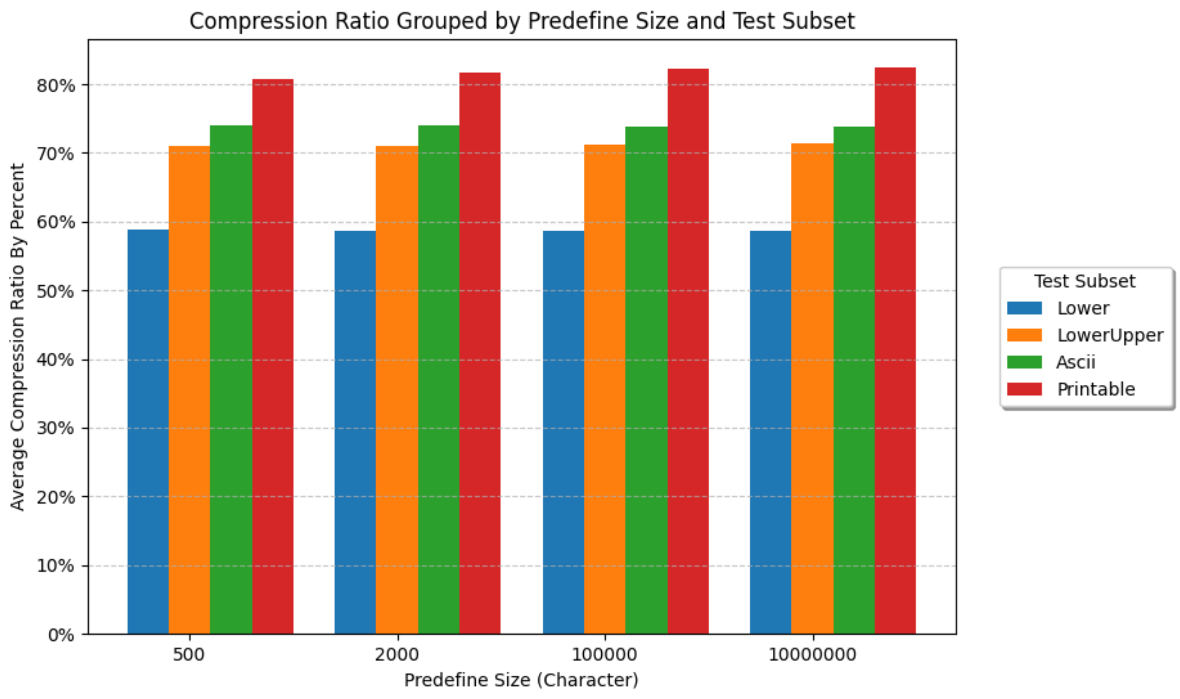

Figure 4: UML Diagram for the Observer Pattern.

4.3 Strategy Pattern

4.3.1 Purpose

The Strategy pattern defines a family of algorithms, encapsulates each one, and makes them interchangeable. It lets the algorithm vary independently from the clients that use it. This is useful when you have multiple ways to perform a task and want to choose one at runtime.

4.3.2 Implementation

The Strategy pattern is used for implementing the book search functionality. An abstract interface, `ISearchStrategy`, defines a common `search()` method. Concrete strategy classes like `TitleSearchStrategy` and `AuthorSearchStrategy` implement this interface, each providing a different search algorithm. The client code can then select and use the desired search strategy at runtime without being coupled to its specific implementation.

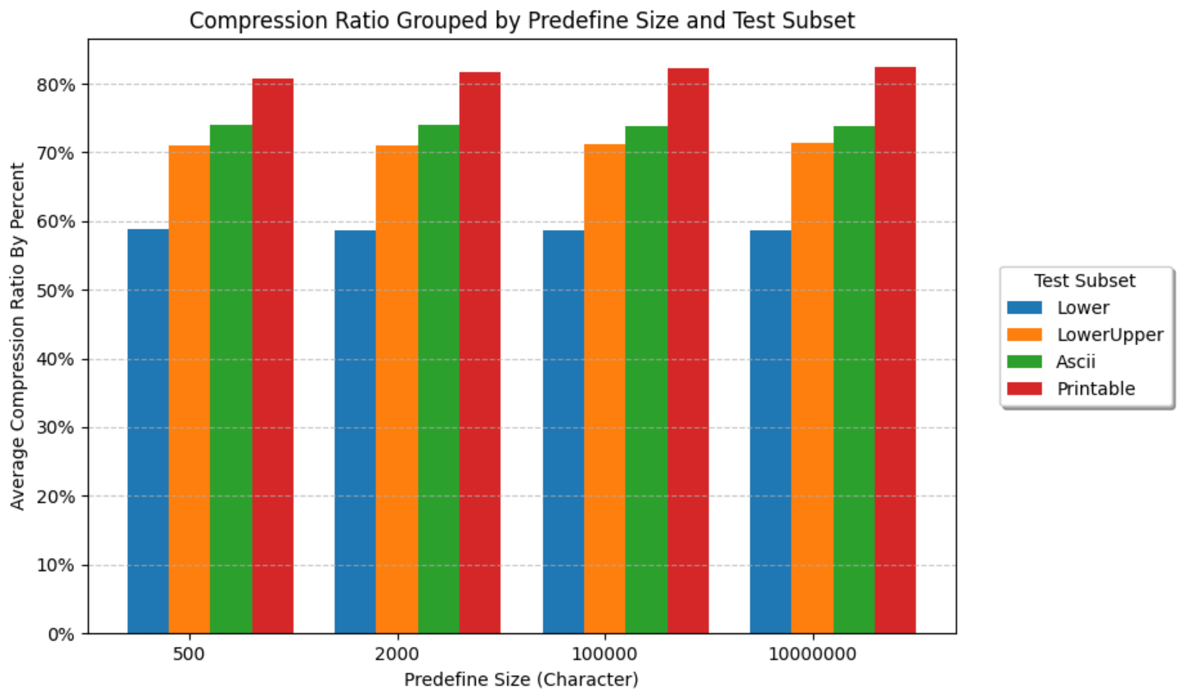

Figure 5: UML Diagram for the Strategy Pattern.

4.4 Decorator Pattern

4.4.1 Purpose

The Decorator pattern allows behavior to be added to an individual object, either statically or dynamically, without affecting the behavior of other objects from the same class. It is used to extend an object's functionality.

4.4.2 Implementation

This pattern is used to add extra information to `Book` objects in a flexible way. A base `BookDecorator` class inherits from `Book` and also contains a pointer to a `Book` object. Concrete decorator classes like `DifficultyLabelDecorator` and `SpecialTagDecorator` inherit from `BookDecorator`. They override methods like `getFullDescription()` to first call the wrapped object's method and then add their own information (e.g., a difficulty label or a "Bestseller" tag) to the result.

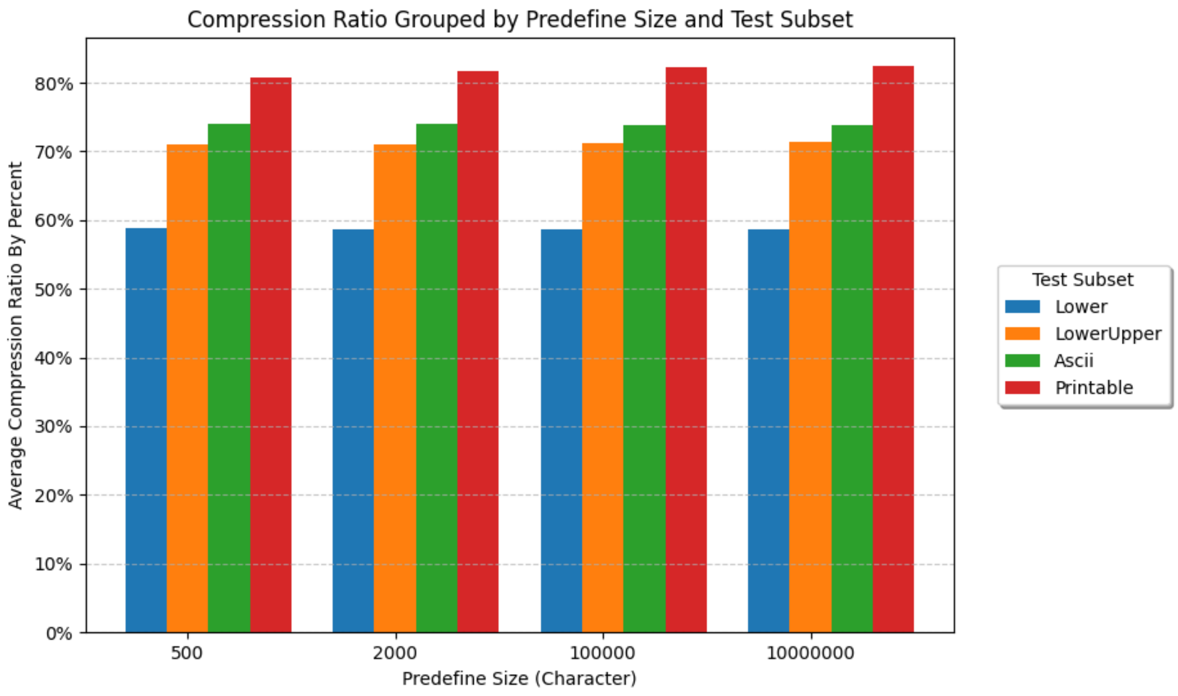

Figure 6: UML Diagram for the Decorator Pattern.

5 Application Walkthrough and Usage

This section demonstrates the system in action by walking through its primary use cases. For each key feature, we provide a description of the user interaction, a UML Sequence Diagram to illustrate the technical flow of method calls between objects, and screenshots to showcase the console-based user interface.

5.1 Registering a New Member

A new user interacts with the system to create a member account. The user selects the registration option from the main menu and provides the required credentials (username and password). The system then validates the input and creates a new member record, assigning a unique Member ID. The sequence of object interactions for this process is detailed in Figure 7.

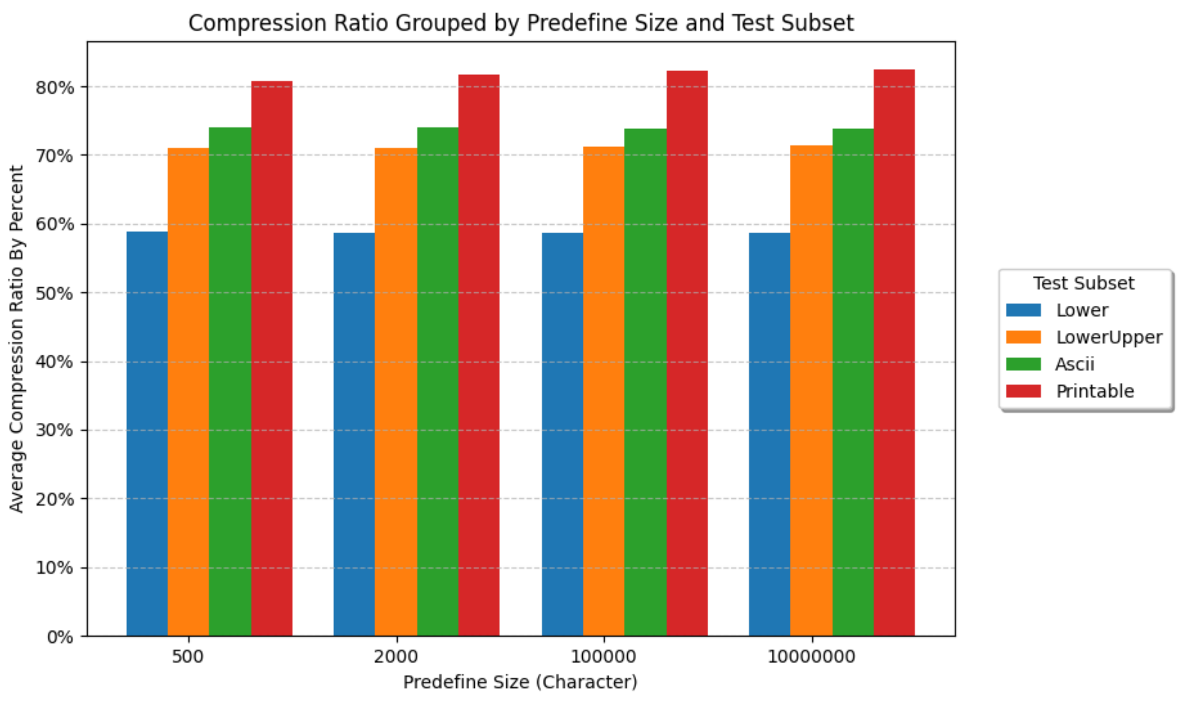

Figure 7: Sequence Diagram for New Member Registration.

5.2 Borrowing a Book

A logged-in member can borrow a book from the library. The typical flow involves the member searching for a book and then selecting the "borrow" option, providing the book's ISBN. The system validates that the book is available, checks the member's borrowing eligibility, creates a new loan record, and finally updates the book's status. This workflow is visualized in Figure 8.

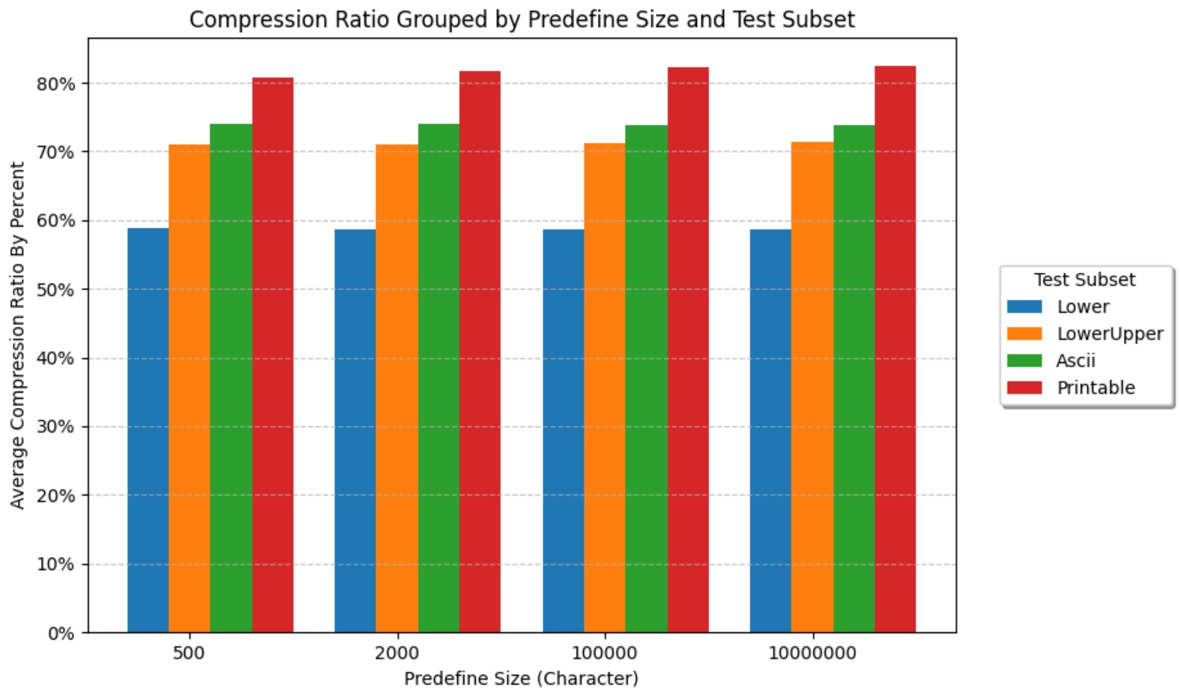

Figure 8: Sequence Diagram for Borrowing a Book.

5.3 Returning a Book

When a member returns a book, they select the return option and provide the book's ISBN. The system finds the corresponding active loan record, updates its status to "returned," records the return date, and increments the number of available copies for that book. Figure 9 illustrates the object interactions for this process.

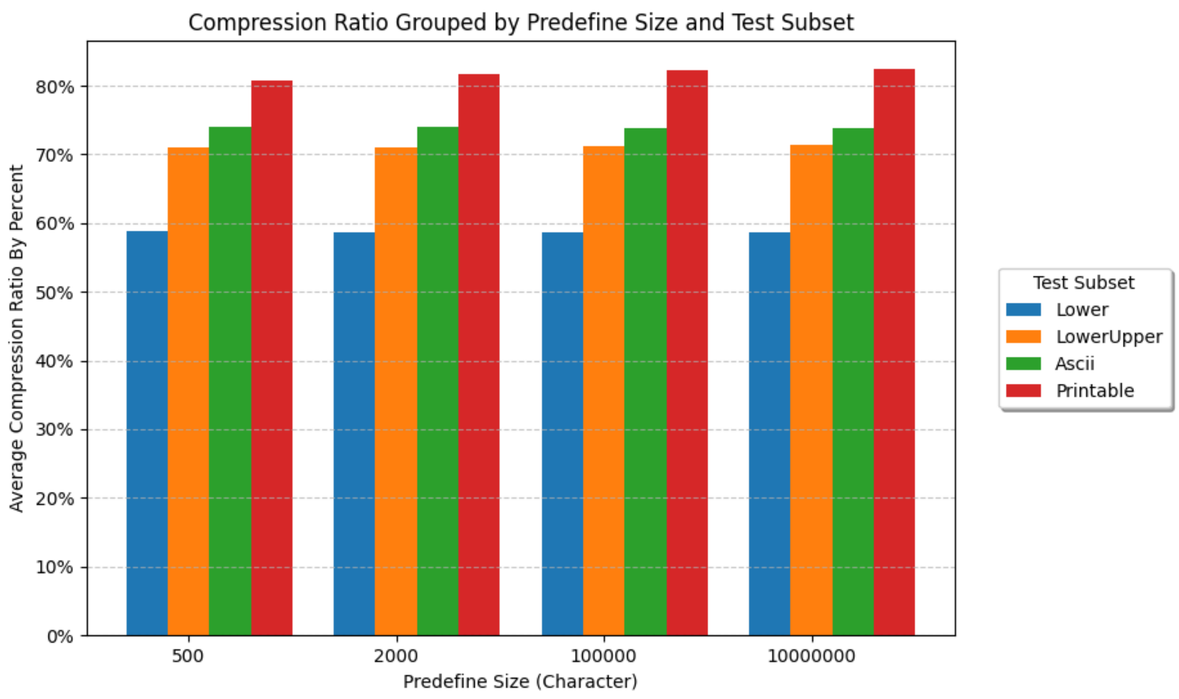

Figure 9: Sequence Diagram for Returning a Book.

5.4 User Interface Showcase

The following screenshots provide a glimpse into the application's interactive console-based user interface, demonstrating its clarity and ease of use.

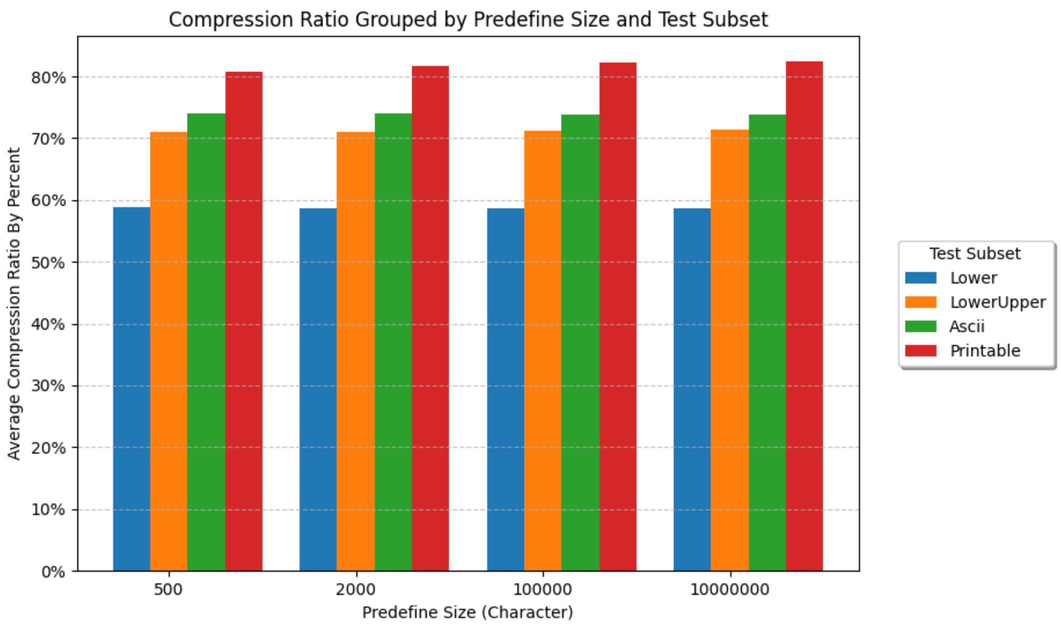

Figure 10: The main menu provides clear options for the user.

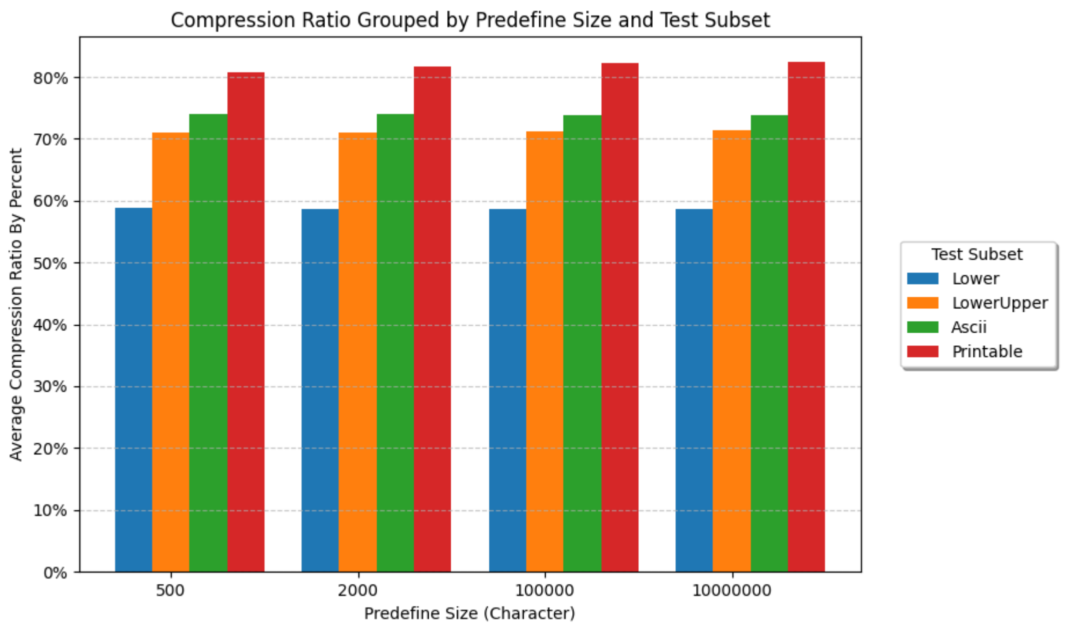

Figure 11: An example of a book search result display.

6 Conclusion

This project has successfully culminated in the development of a functional and robust Electronic Library Management System. By adhering to the principles of Object-Oriented Programming and leveraging established software design patterns, the project has achieved its primary goal of creating a maintainable and extensible software solution for a real-world problem. The final application effectively demonstrates the practical application of key academic concepts in a complete software development lifecycle.

6.1 Challenges Faced During Development

The development process, while successful, presented several challenges that provided valuable learning experiences:

- **Design Pattern Selection:** One of the initial challenges was selecting the most appropriate design patterns. It required careful analysis to determine which pattern best suited each specific problem, such as choosing the Strategy pattern for search functionality versus a simpler conditional approach.
- **Data Persistence Logic:** Implementing a reliable data handler for CSV files was more complex than anticipated. Ensuring data integrity through file hashing, handling potential file I/O exceptions, and managing data consistency across multiple files required careful planning and implementation.
- **Collaborative Workflow:** As a team project, managing the source code via version control (Git) occasionally led to merge conflicts. Establishing and adhering to a consistent coding style across all modules also required continuous communication and discipline.

6.2 Future Enhancements

The current system serves as a solid foundation that can be extended with numerous features to increase its value and utility. Potential directions for future work include:

- **Graphical User Interface (GUI):** The most significant improvement would be to replace the current console-based interface with a user-friendly GUI. This could be developed using a framework like Qt for C++ or JavaFX/Swing for Java, which would dramatically improve the user experience.
- **Database Integration:** To enhance performance, scalability, and data integrity, migrating from CSV files to a relational database system is a logical next step. Using a lightweight database like SQLite or a more powerful one like MySQL would be a major architectural improvement.
- **Book Reservation System:** A valuable new feature would be to allow members to reserve a book that is currently on loan. The system could automatically notify the member once the book becomes available.
- **Advanced Reporting and Analytics:** The reporting module could be expanded to generate more detailed analytics, such as identifying the most popular books, tracking peak borrowing times, or creating activity reports for members.

- **REST API for Remote Access:** Exposing the system's core functionalities through a RESTful API would enable the development of web-based or mobile clients, allowing users to interact with the library from anywhere.

In summary, this project has not only met its specified requirements but has also provided a rich learning experience in software design and object-oriented methodologies, paving the way for future enhancements and development.

7 Project Summary and Team Evaluation

This final section provides a comprehensive overview of the project's management, the team's collaborative process, and an evaluation of the overall workflow from inception to completion.

7.1 Member Contributions and Task Allocation

The project's success was built on a clear allocation of tasks and the dedicated contribution of each team member. The work was divided based on individual strengths and interests to maximize efficiency and quality.

Member 1 - [Tên vai trò, ví dụ: Team Lead / Core Architect] [Mô tả chi tiết đóng góp, ví dụ: Responsible for the overall system architecture, designing the core domain models (Book, User, Loan), and implementing the Singleton and Strategy patterns. Led team meetings and managed the Git repository.]

Member 2 - [Tên vai trò, ví dụ: Backend Developer / Data Layer] [Mô tả chi tiết đóng góp, ví dụ: Developed the entire data persistence layer, including the CSVHandler for file I/O and the data integrity module using file hashing. Implemented the book and member management logic.]

Member 3 - [Tên vai trò, ví dụ: Backend Developer / Business Logic] [Mô tả chi tiết đóng góp, ví dụ: Focused on the business logic for loan management, including borrowing, returning, and overdue calculations. Implemented the Observer and Decorator patterns for the notification and book description features.]

Member 4 - [Tên vai trò, ví dụ: UI & Documentation] [Mô tả chi tiết đóng góp, ví dụ: Developed the entire console-based user interface, ensuring a user-friendly and interactive experience. Was also responsible for writing and formatting the final technical report and creating the UML diagrams.]

7.2 Development Workflow and Collaboration

To ensure a smooth and organized development process, our team adopted a structured workflow utilizing industry-standard tools and practices.

- **Version Control:** All source code was managed using **Git**, a distributed version control system that is the de facto standard in modern software development [2]. A central repository was hosted on **GitHub**, and the team followed a feature-branch workflow to facilitate parallel development and code review before integration.
- **Communication:** Primary communication was conducted through a dedicated **Discord** channel for daily updates and quick questions. We also held weekly online meetings to discuss progress, resolve blocking issues, and plan the next steps.
- **Task Management:** We utilized a simple Kanban-style board on **Trello** to track the status of all tasks (To Do, In Progress, Done). This provided clear visibility into the project's overall progress and helped identify bottlenecks early.

7.3 Key Challenges and Resolutions

Throughout the project, we encountered several challenges that tested our problem-solving and teamwork skills:

Technical Design Decisions Challenge: Choosing the most suitable design patterns from the catalog presented by Gamma et al. [3] was a point of extensive debate. **Resolution:** The team held dedicated design sessions where we would whiteboard different approaches and discuss the pros and cons of each. The final decision was always made collectively to ensure everyone understood the chosen architecture.

Code Integration Challenge: As members completed their features, merging different Git branches occasionally resulted in code conflicts. **Resolution:** We resolved this by adopting a policy of frequent communication. Before starting a major change, members would announce their plans to the team. Before merging, the developer would pull the latest changes from the main branch to resolve conflicts locally first.

Data Integrity Verification Challenge: Ensuring the integrity of data stored in plain text CSV files was a significant security concern. **Resolution:** We implemented a mechanism to compute and store a cryptographic hash (using SHA-256) for each data file. Before reading any file, the system re-computes its hash and compares it to the stored value to detect any tampering, a fundamental technique in data security [5].

References

[1] Grady Booch et al. *Object-Oriented Analysis and Design with Applications*. 3rd. Addison-Wesley Professional, 2007.

[2] Scott Chacon and Ben Straub. *Pro Git*. 2nd. Apress, 2014.

[3] Erich Gamma et al. *Design Patterns: Elements of Reusable Object-Oriented Software*. Boston, MA, USA: Addison-Wesley Professional, 1994.

[4] Robert C. Martin. *Clean Architecture: A Craftsman's Guide to Software Structure and Design*. Prentice Hall, 2017.

[5] William Stallings and Lawrie Brown. *Computer Security: Principles and Practice*. 4th. Pearson, 2017.

[6] Bjarne Stroustrup. *The C++ Programming Language*. 4th. Addison-Wesley Professional, 2013.