**An Academic Report on**

**Design by Contract in a Layered Library Management System**

1. **ABSTRACT**

This report details the design and implementation of a modern Library Management System (LMS) built using a layered architectural pattern and guided by Design by Contract (DbC) principles. The system addresses key challenges often found in traditional LMS solutions, such as inconsistencies, poor maintainability, and lack of formal verification. Users can register, search for books, borrow and return them, leave feedback, and manage fines, while librarians can handle book catalog management, monitor loans and borrow rules. The architecture includes a React.js frontend, a Python Flask microservice backend, and a cloud-hosted MongoDB Atlas database. To ensure correctness and support future scalability, the system applies formal specification techniques for defining critical operations and uses UML diagrams to validate the system’s structure and behavior. Altogether, the approach enhances software reliability, simplifies debugging, and lays a solid foundation for future development.

**Keywords:** Library Management System (LMS), Design by Contract (DbC), MongoDB Atlas, Flask API, React.js Frontend, Book Loan Management, User Feedback System, Borrowing Constraints, Fine Calculation Logic, User Authentication, UML Design.

1. **INTRODUCTION**

Library Management Systems (LMS) have become essential tools in academic and public libraries, helping manage critical operations such as lending and returning books, user registration, fine calculation, and maintaining borrowing histories. As library usage grows and user expectations shift toward seamless digital interactions, modern LMS solutions must also handle complex policies like borrowing limits, varied loan durations, and role-based access. While these needs may seem straightforward, implementing them consistently and accurately presents significant technical challenges.

Many traditional LMS implementations rely on informal development approaches, where business rules are scattered and enforced inconsistently across different modules. This often leads to policy violations, logic errors, and maintenance difficulties particularly in large systems handled by multiple developers. Consequences such as incorrect fine calculations, unauthorized borrowing, or data integrity issues can seriously impact the reliability of the system, which is unacceptable in library environments that depend on accuracy and trust.

To address these limitations, this project proposes a modern LMS built around **Design by Contract (DbC)** principles and a layered software architecture. DbC ensures correctness by defining precise preconditions, postconditions, and invariants for all critical operations, making the system easier to test, maintain, and extend. The layered architecture separates the presentation, service, and data layers, which enhances modularity and simplifies future development.

The system is implemented using a modern technology stack: React.js for an intuitive frontend, Flask for a lightweight backend API, and MongoDB Atlas as a cloud-hosted NoSQL database for scalable and persistent data storage. To reinforce software correctness and clarity, formal specification techniques using Z notation are employed to rigorously define key operations. UML diagrams are also used to visualize system architecture and workflows. Altogether, this approach provides a reliable, scalable, and future-ready LMS that addresses the shortcomings of traditional systems and supports long-term adaptability.

1. **PROBLEM STATEMENT**

Traditional Library Management Systems (LMS) often struggle with enforcing complex borrowing rules, maintaining data consistency, and ensuring long-term maintainability. Users may have varying loan limits, due dates, and fine structures based on membership types, all of which must be accurately enforced. However, in many systems, business rules are implemented informally or inconsistently, leading to issues such as users exceeding borrowing limits, incorrect fine calculations, or double loaning of books. These problems are compounded by weak validation mechanisms that fail to enforce rules uniformly across components, allowing for bypasses, policy violations, or data integrity issues particularly in areas like book returns or user feedback handling.

The root cause of these issues lies in the lack of structured design and formal behavior contracts in traditional LMS architectures. Without clearly defined preconditions, postconditions, and invariants, software components operate with uncertain guarantees, making the system difficult to test, debug, or evolve. As a result, maintaining and scaling such systems becomes increasingly error-prone over time. To address these challenges, there is a clear need for a more systematic approach one that promotes correctness, clarity, and maintainability through formal methods and well-defined contracts, while still supporting flexibility for future enhancements.

1. **OBJECTIVES & GOALS**

The primary objective of this project is to design and implement a modern Library Management System (LMS) that is robust, maintainable, and scalable. To achieve this, the system adopts a layered architectural approach, separating the presentation, service, and data layers to enhance modularity and maintain clarity in design. Design by Contract (DbC) principles are applied throughout the system to enforce correctness by clearly defining preconditions, postconditions, and invariants for critical operations. This ensures predictable behavior and helps prevent logic errors during development and future maintenance. A cloud-based NoSQL database, MongoDB Atlas, is integrated to provide reliable and scalable data storage with global accessibility and consistent performance.

In addition to architectural and design principles, the system is built using a modern technology stack featuring React.js for the frontend, offering users an intuitive and interactive interface. The LMS supports a comprehensive set of features, including user registration, login, book searching, borrowing and returning, fine calculation, user feedback, and librarian administration. To ensure formal correctness, key operations are specified using Z notation, while UML diagrams are used to visualize the system’s architecture and workflows. The final system is validated through test cases, interface demonstrations, and discussion of results, showcasing its effectiveness in meeting the needs of modern library environments.

1. **LITERATURE REVIEW**

Design by Contract (DbC) has emerged as a powerful methodology in software engineering, offering formal guarantees about program behavior through well-defined contracts. This literature review examines key research contributions that explore contract-based development, practical implementation techniques, and alternative architectural models for Library Management Systems (LMS). The review highlights the relevance of DbC in enhancing modularity, correctness, and maintainability, while also contrasting it with other formal and semantic approaches to system design. The following three studies provide a strong foundation for understanding how this project’s architectural and methodological choices align with current research trends.

**5.1 A Systematic Mapping Study on Contract-Based Software Design (Okumuş et al., 2025)**

Okumuş et al. conducted an extensive systematic mapping study, analyzing 288 research papers on contract-based software design. Their work focused on the use of preconditions, postconditions, and invariants to improve software correctness and modular boundaries. The study revealed that formal contracts play a significant role in reducing defect rates and improving code clarity, particularly in enterprise and embedded systems. However, the authors identified a notable research gap in applying DbC principles to modern, cloud-native systems that use NoSQL databases precisely the architectural context of this LMS project, which integrates Flask with MongoDB Atlas.

Key insights from their review emphasize that contract-based development enhances component reusability and promotes safe interchangeability between modules. Additionally, the presence of tool support such as icontract or Eiffel’s native support was strongly associated with fewer runtime errors and better enforcement of logic constraints. The combination of DbC with automated verification tools was found to bridge the gap between informal design and formally correct implementation. This study validates the decision to apply icontract in the LMS backend, confirming that such tools not only help enforce business logic but also position the system for future scalability, including microservice migration, without compromising correctness.

**5.2 Applying Design by Contract: Insights from Engineering Practice (Meyer, 1992)**

Bertrand Meyer’s seminal work on Design by Contract laid the foundation for understanding contracts as both formal specifications and executable components in software development. In his exploration of Eiffel a programming language built around DbC Meyer demonstrates how preconditions, postconditions, and invariants act as self-verifying contracts between software components. These contracts serve as executable documentation, making assumptions and guarantees explicit, thereby acting as both unit tests and detailed design specifications.

Meyer further discusses how Eiffel’s environment enforces contracts at runtime, enabling early detection of violations and improving software reliability. While contracts may be disabled in production environments for performance optimization, their presence during development and testing phases proves critical in catching defects early. A key takeaway from Meyer’s work is the ability of contracts to clearly define the source of errors in distributed or layered systems, improving traceability and debugging. These principles are directly reflected in this LMS project, where backend service methods are decorated using icontract decorators such as @require and @ensure to enforce constraints like “a user cannot borrow more than three books.” Meyer’s methodology strongly supports the structured, contract-enforced backend design adopted here.

**5.3 Ontology-Based Library Management: Building a Semantic LMS (Mohd et al., 2020)**

While many systems rely on procedural or contract-based logic, Mohd et al. propose an entirely different approach to library management using semantic web technologies and ontologies. Their work introduces LMSO, an ontology-driven architecture modeled using OWL (Web Ontology Language), designed to represent entities such as books, users, and loans in a formal, machine-readable format. This model organizes domain entities into modular, class-based structures with logical relationships and constraints. For example, a loan must reference both a book and a user, and inheritance is used to generalize shared properties among entities.

One of the key strengths of this ontology-based approach is reasoning. Through formal axioms and inference engines like HermiT, the system can automatically detect inconsistencies, such as overdue items or invalid user actions. The use of semantic standards like RDF and OWL also makes the system highly interoperable with external catalogs and linked data sources. Mohd et al. validated their model by implementing over 200 instances and running consistency checks using Protégé. While this semantic approach does not follow the Design by Contract paradigm, it offers a powerful alternative for ensuring correctness through logic-based reasoning. It also presents opportunities for integration with broader knowledge systems, making it a valuable contrast to the DbC-based architecture implemented in this LMS project.

1. **SYSTEM ARCHITECTURE**

This Library Management System (LMS) is designed using a three-tier layered architecture, separating the system into the presentation, service, and data layers. This separation improves clarity, maintainability, and flexibility for future updates.

* 1. **Layered Architectural Model:**

**Presentation Layer (React.js):** Provides a responsive, component-based user interface. It handles user interactions like searching for books or initiating loans.

**Service Layer (Flask):** Acts as the middleware, exposing a REST API. It applies business logic, validates inputs using Design by Contract (icontract), and handles communication between the frontend and database.

**Data Layer (MongoDB Atlas):** A cloud-based NoSQL database that stores structured collections for users, books, loans, loan history, and feedback. Its schema-less design allows easy data model evolution.

Each layer communicates through well-defined APIs, allowing components to operate independently. For instance, the React frontend could be replaced without altering the backend, if API contracts are preserved.

A diagram of a data flow

AI-generated content may be incorrect.

**Fig-6.1:** Layered Architecture Diagram of LMS

* 1. **Data Persistence Strategy:** MongoDB Atlas offers flexibility through its document-oriented storage and global accessibility. Key collections includeusers, books, loans, loan\_history, and book\_feedback.Indexes on fields like username and book\_id ensure fast data retrieval, while replica sets enhance fault tolerance and availability.

A screenshot of a computer

AI-generated content may be incorrect.**Fig-6.2:** The above figure showcases all the collections in our database.

* 1. **Communication Flows:** The React frontend interacts with Flask through RESTful APIs and retrieve the necessary information from the database based on user requests and functions.

1. **METHODOLOGY & SYSTEM DESIGN** 
   1. **Requirements Analysis:** The requirements for this Library Management System were derived by analyzing traditional library workflows and common user interactions. Core functionalities identified include user registration, book searching, borrowing and returning books, fine calculation, and administrative tasks for librarians. In addition to these core functions, input from stakeholders highlighted the need for a modern, responsive user interface, consistent enforcement of borrowing policies, and clearly defined user constraints to avoid misuse or confusion.

To ensure clarity and structured development, the requirements were divided into two categories. Functional requirements cover specific user actions the system must support, such as borrowing and returning books, viewing loan history, paying fines, submitting feedback, and allowing librarians to manage the book catalog. On the other hand, non-functional requirements address broader system qualities like maintainability, scalability, security, cloud accessibility, and the ability to verify correctness through formal validation methods. These non-functional goals are especially important to ensure the system remains reliable and adaptable as it evolves over time.

* 1. **Design by Contract Implementation:** In this project, Design by Contract (DbC) is implemented using the icontract library in Python. This approach ensures that each critical function within the system such as user registration, book borrowing, and returns strictly follows predefined rules. These rules are enforced using decorators that define both preconditions (what must be true before the function runs) and postconditions (what must be true after the function completes).

For instance, in the register function, preconditions ensure that the provided username, password are non-empty strings, and postconditions confirm that the function returns a dictionary representing a valid user record. If any of these conditions are not met such as if the input is invalid or the function fails to return the expected result icontract raises an exception immediately. This stops the error from spreading further into the system and makes debugging much easier. Overall, this approach adds a layer of reliability by making the system’s logic self-validating and easier to maintain. Here is the example:

A screen shot of a computer code

AI-generated content may be incorrect.**Fig-7.2: Registration Function implemented by using DBC through icontracts in python**

* 1. **System Modeling with UML:** To effectively visualize and plan the system’s structure and behavior, Unified Modeling Language (UML) was used throughout the design phase. Three key diagrams were created to support different aspects of the system. A class diagram illustrates the relationships between core entities such as User, Book, Loan, and Feedback, providing a clear view of the system’s data model. A sequence diagram captures the flow of operations over time, particularly focusing on dynamic processes like borrowing and returning books. Additionally, a use case diagram maps out how various users such as members and librarians interact with different system functionalities. These diagrams played a crucial role in building a shared understanding among team members before the development phase began. They helped identify missing requirements, clarified system behavior, and minimized potential rework by highlighting inconsistencies early in the design process.

Class Diagram: