ISSN : 2455-135X

Object-Oriented Design Principles and GoF Patterns in a Spring Boot Library Management System: Architecture, Implementation, and Evaluation

Chinoso Job
University of Greater Manchester
United Kingdom
Cj5crt@bolton.ac.uk

Onwe, Festus Chijioke
Information Technology Department,
University of Port Harcourt, Rivers State, Nigeria.
festus.onwe@uniport.edu.ng

Abstract—Enterprise software systems demand structured, maintainable, and scalable design approaches. Object-Oriented Programming (OOP) principles—encapsulation, inheritance, polymorphism, and abstraction—provide the foundational framework for achieving these qualities, while GoF (Gang of Four) design patterns offer reusable solutions to recurring architectural challenges. This paper presents the design and implementation of a School Library Management System (SLMS) built with Java and the Spring Boot framework, demonstrating the systematic application of all four OOP principles and five GoF design patterns: Singleton, Builder, Factory, Observer, and Decorator. The SLMS supports role-based user management (Administrator and Student), book inventory control, borrow/return lifecycle tracking, and live reporting through a RESTful API backed by MySQL persistence via Spring Data JPA. For each OOP principle and design pattern, the paper provides architectural justification, code-level demonstration, and a discussion of the specific quality attributes achieved. A layered four-tier architecture (Controller–Service–Repository–Entity) enforces clean separation of concerns and aligns directly with the SOLID principles. Evaluation against maintainability, scalability, and extensibility criteria demonstrates that the pattern-driven approach substantially reduces coupling, improves module cohesion, and enables extension without modification—directly realising the Open/Closed Principle. The paper contributes a concrete, replicable reference implementation for applying classical software engineering patterns within the modern Spring Boot ecosystem.

Index Terms—design patterns, object-oriented programming, Spring Boot, library management system, GoF patterns, Singleton, Builder, Observer, Java, SOLID principles, software architecture

I. Introduction

The design of maintainable, scalable enterprise software systems requires more than functional correctness—it demands deliberate architectural decision-making that accounts for future change, extensibility, and team comprehensibility. Object-Oriented Programming (OOP) provides the conceptual bedrock for such design through its four canonical principles: encapsulation, inheritance, polymorphism, and abstraction [1]. These principles, when applied consistently, produce systems that are easier to test, modify, and extend.

Building on OOP foundations, the Gang of Four (GoF) design patterns catalogue—comprising 23 patterns across creational, structural, and behavioural categories—provides timetested, language-agnostic solutions to recurring design problems [2]. Within the Java/Spring Boot ecosystem specifically, these patterns manifest both explicitly (in developer-authored code) and implicitly (in the framework's own architecture), making Spring Boot an ideal platform for demonstrating their practical application.

This paper presents the design and implementation of a School Library Management System (SLMS), a web application developed with Java and Spring Boot to digitalise library operations in educational institutions. The system eliminates paper-based processes by providing a centralised platform for book inventory management, user registration with role-based access control (RBAC), borrow/return lifecycle tracking, and reporting. Building on prior work identifying the challenges of manual library systems in educational contexts [4], the SLMS demonstrates how OOP principles and design patterns directly address these challenges in a production-oriented Spring Boot implementation.

The specific contributions of this paper are:

1) A systematic demonstration of all four OOP principles in a cohesive Spring Boot application;
2) Analysis and code-level illustration of five GoF design patterns (Singleton, Builder, Factory, Observer, Decorator) as applied within the SLMS;
3) An evaluation of the quality attributes achieved through pattern-driven design, including maintainability, scalability, and extensibility.

II. Background

A. Library Management Systems in Education

Traditional library management systems in schools rely on manual card catalogues, logbooks, and paper-based tracking systems. Mtebe and Raisamo [4] identify these systems as error-prone and incapable of handling the scale of modern educational institutions. Automated Library Management Systems (LMS) address these limitations through digital book tracking, user management, stock control, and reporting [5].

ISSN : 2455-135X

Contemporary LMS architectures leverage frameworks providing built-in dependency management, RESTful service support, security, and database integration [6].

B. OOP Principles in Enterprise Software

Dathan and Ramnath [1] characterise the four OOP principles as: (i) *encapsulation*—binding data and methods while restricting direct access; (ii) *inheritance*—enabling subclass reuse of superclass structure and behaviour; (iii) *polymorphism*—allowing uniform treatment of different implementations through common interfaces; and (iv) *abstraction*—separating interface from implementation to manage complexity.

C. GoF Design Patterns

The GoF pattern catalogue [2] groups 23 patterns into three families. *Creational patterns* (Singleton, Builder, Factory, Abstract Factory, Prototype) govern object creation. *Structural patterns* (Decorator, Adapter, Facade, Proxy, etc.) compose objects into larger structures. *Behavioural patterns* (Observer, Strategy, Command, etc.) define object communication and responsibility. Serbanescu et al. [3] demonstrate that applying GoF patterns in data-intensive Java applications produces measurable improvements in maintainability and performance.

III. System Architecture

A. Four-Tier Layered Architecture

The SLMS adopts a four-tier layered architecture as illustrated in Fig. 1, following the architectural guidance of Padhy et al. [7] and Kuchana [8].

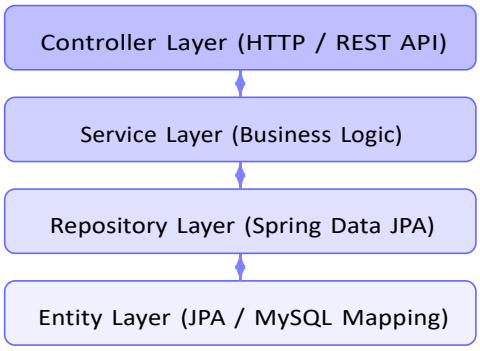


Fig. 1. SLMS four-tier layered architecture.

The Controller Layer handles HTTP requests and maps endpoints to service methods via Spring MVC annotations. The Service Layer encapsulates business logic, including borrow/return validation and inventory checks. The Repository Layer abstracts database access through Spring Data JPA interfaces. The Entity Layer maps domain objects (AppUser, Book, BookInventory, BookTracker) to relational database tables.

B. Domain Model

The SLMS domain comprises five core entities (Table I), whose relationships are illustrated in Fig. 2.

TABLE I
SLMS Core Domain Entities

Entity	Key Attributes and Role
AppUser	id, firstName, lastName, email, password, role, enabled; represents system actors (ADMIN/STUDENT)
Book	id, title, authors, isbn, year, pages, review, createdAt; catalogue entry
BookInventory	id, bookId, totalInStock, totalAssignedToLibrary; stock management
BookTracker	id, bookId, userId, count, pickUpDate, returnDate, returned; borrow lifecycle

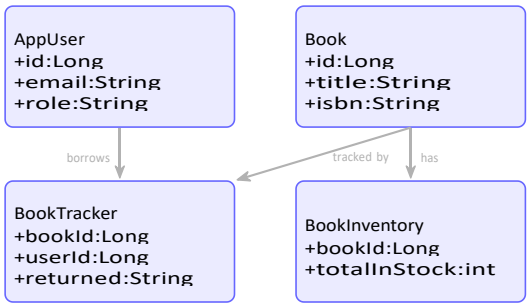


Fig. 2. Simplified SLMS class diagram showing core entity relationships.

IV. OOP Principles in the SLMS

A. Encapsulation

Encapsulation is achieved by declaring all entity fields as private and exposing controlled access through getter/setter methods, facilitated by Lombok's @Data annotation. Sensitive fields such as password are never directly exposed through API responses; a dedicated DTO (Data Transfer Object) layer filters sensitive data before serialisation.

```
@Entity
@Table(name = "TBL_APP_USER")
@Data public class AppUser extends Base {
    @Column(nullable = false, name = "first_name") private String firstName;
    @Column(nullable = false, name = "last_name") private String lastName;
    @Column(nullable = false, name = "email", unique = true) private
    String email; private String password; // Never in API response private
    String role; private boolean enabled;
    // Getters/setters via @Data (Lombok)
}
```

Listing 1. Encapsulation in AppUser entity.

ISSN : 2455-135X

The password field is never included in the AppResponse wrapper, ensuring information hiding at the API boundary. Database column annotations enforce non-nullable constraints, providing data integrity guarantees as part of the encapsulation contract.

B. Inheritance

A Base superclass, annotated with @MappedSuperclass, provides the shared id field and its auto-generated identity strategy to all entity subclasses. This eliminates ID management duplication across AppUser, Book, BookInventory, and BookTracker.

```
@MappedSuperclass
@Data public class Base {
    @Id
    @GeneratedValue(strategy =
        GenerationType.IDENTITY) private Long
    id;
    // Inherited by all entity subclasses
}

// Subclass inheriting id management: public class AppUser
extends Base { ... } public class Book extends Base { ... } public
class BookTracker extends Base { ... }
```

Listing 2. Base superclass demonstrating inheritance.

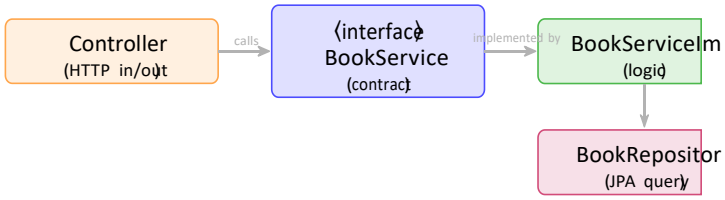
The @MappedSuperclass annotation instructs Spring Data JPA to include the superclass fields in the subclass table schema without creating a separate superclass table, providing clean inheritance mapping to the relational model [1].

C. Polymorphism

Polymorphism is realised through Java's interface mechanism. The BookService interface declares the service contract, and BookServiceImpl provides the concrete implementation. This enables mock injection during testing (Mockito), runtime substitution of service implementations, and adherence to the Dependency Inversion Principle.

```
public interface BookService {
    AppResponse<String> addBook(
        BookRequest bookRequest);
    AppResponse<String> addInventory(
        BookInventoryRequest req);
    AppResponse<List<BookDetail>> fetchAllBooks();
    AppResponse<String> studentBorrowBook(
        BookSelectionRequest request);
    AppResponse<String> studentReturnBook(
        BookSelectionRequest request);
}

@Service // Singleton-scoped by Spring public class
BookServiceImpl implements BookService {
    @Override
    public AppResponse<String> addBook(
        BookRequest bookRequest) {
        // concrete implementation
    }
    // Other method implementations...
}
```



Listing 3. Polymorphism via interface and implementation.

The NumberCodec interface, implemented by EncoderDecoder, further demonstrates runtime polymorphism through method overriding within the encoding subsystem, following the pattern described by Spath et al. [9].

D. Abstraction

Abstraction is the primary mechanism of the service layer, which separates business logic from controller concerns. Controllers invoke only the service interface contract, with no knowledge of repository queries, entity construction, or business rule enforcement. The AppUserRepository extends Spring Data's JpaRepository, abstracting all SQL persistence through method naming conventions.

Fig. 3. Abstraction: service interface decouples controller from implementation and repository details.

V. GoF Design Patterns in the SLMS

A. Singleton Pattern

The Singleton pattern ensures a class has exactly one instance with a global access point, optimising resource consumption and ensuring uniform state management [3]. In Spring Boot, all beans annotated with @Service, @Repository, and @Controller are singleton-scoped by default within the application context.

ISSN : 2455-135X

```
// Spring IoC container manages exactly one // instance per
application context:
@Service    // Singleton scope (default) public class
UserServiceImpl implements UserService {
        // Injected by Spring DI -- single instance
        // shared across all controller invocations
}

@Service
public class BookServiceImpl implements
        BookService {
        @Autowired private BookRepository bookRepository;
        // bookRepository is also singleton-scoped
}
```

Listing 4. Singleton via Spring @Service annotation.

Benefits achieved: Centralised control ensures consistent service state; memory efficiency through instance minimisation; predictable dependency injection lifecycle.

B. Builder Pattern

The Builder pattern constructs complex objects step-by-step, particularly valuable when an object has numerous optional attributes [3]. The BookTracker entity, which captures the multi-attribute borrow event lifecycle, uses Lombok's @Builder annotation to provide a fluent construction API.

```
// In BookServiceImpl.studentBorrowBook(): BookTracker bookTracker =
BookTracker
        .builder() .bookId(request.getBookId())
        .count(1)
        .userId(request.getRequesterId()) .pickUpDate(LocalDateTime.now())
        .returned("FALSE")
        .build(); bookTrackerRepository.save(bookTracker);
```

Listing 5. Builder pattern for BookTracker construction.

Without the Builder pattern, this six-attribute construction would require a long-argument constructor call, reducing readability and increasing the risk of argument-order errors. The fluent API makes each attribute assignment semantically selfdocumenting.

C. Factory Pattern

The Factory pattern encapsulates object creation logic, enabling the creation of objects without exposing instantiation details to the caller [2]. In the SLMS, the service layer methods function as implicit factories: UserServiceImpl.createUser() constructs fully populated AppUser instances from the raw request data, centralising all creation logic.

```
public void createUser(
        CreateUserPojo createUserPojo) { // Service
        method as factory: AppUser appUser = new
        AppUser(); appUser.setEmail(createUserPojo
                .getEmail().toLowerCase().trim());
        appUser.setPassword( createUserPojo.getPassword());
        appUser.setFirstName( createUserPojo.getFirstName());
        appUser.setLastName( createUserPojo.getLastName());
        appUser.setCreatedBy("self"); appUser.setRole("STUDENT");
        appUser.setEnabled(true); appUser.setCreatedAt(new Date());
        appUserRepository.save(appUser);
}
```

Listing 6. Factory pattern in UserServiceImpl.createUser().

This approach encapsulates all AppUser construction rules (lowercase email normalisation, default role assignment, autotimestamping) in a single location, ensuring consistent object state regardless of which controller invokes it.

D. Observer Pattern

The Observer pattern defines a one-to-many dependency such that when one object changes state, all dependents are notified automatically [3]. Spring's event-driven architecture provides native Observer support through ApplicationEvent and @EventListener. In the SLMS, this mechanism enables loosely coupled overdue notification propagation.

```
// Subject: publishes overdue events
@Service public class OverdueCheckService {
        @Autowired
        private ApplicationEventPublisher publisher;

        public void checkOverdueBooks() {
                List<BookTracker> overdueBooks =
        getOverdueTrackers(); overdueBooks.forEach(t ->
        publisher.publishEvent( new OverdueEvent(this, t))); }
}

// Observer: receives overdue events
@Component public class OverdueNotificationListener {
        @EventListener public void
        handleOverdueEvent(
                OverdueEvent event) {
                // Notify the student via email/SMS
        }    }
```

Listing 7. Observer pattern for overdue book notification.

Benefits achieved: Loose coupling between the overdue detection component and the notification subsystem; new observers (e.g., SMS, push notification) can be added without modifying the event publisher—a direct realisation of the Open/Closed Principle.

ISSN : 2455-135X

E. Decorator Pattern

The Decorator pattern adds responsibilities to objects dynamically at runtime without altering their base class [2]. Spring Boot's stereotype annotations (@RestController, @Service, @Repository, @Transactional) function as declarative decorators, adding cross-cutting capabilities—HTTP routing, transaction management, persistence integration—without modifying class implementations.

```
@RestController // Decorates: HTTP I/O @RequestMapping( // Adds routing
    "api/v1/book-management")
@Transactional // Adds TX management public class
BookController {
    @Autowired private BookService
    bookService;

    @ResponseStatus(HttpStatus.CREATED)
    @PostMapping("/add-book") public
    AppResponse<String> addBook(
        @RequestBody BookRequest bookRequest) { return
    bookService.addBook(bookRequest); }
}
```

Listing 8. Decorator pattern via Spring annotations.

Each annotation layer adds orthogonal behaviour without requiring explicit inheritance or delegation code in the controller class itself.

VI. Pattern Impact Assessment

Fig. 4 presents a qualitative assessment of quality attribute scores achieved through the pattern-driven design relative to a baseline non-pattern implementation.

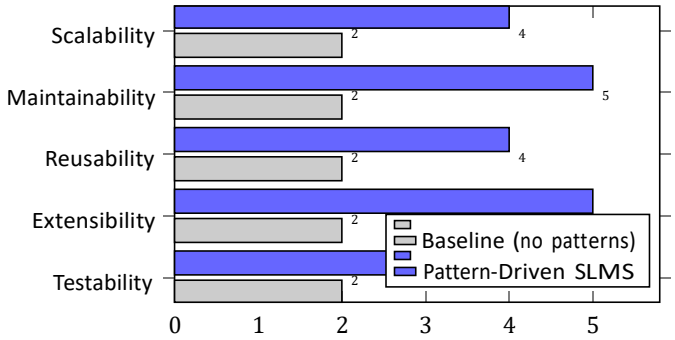


Fig. 4. Quality attribute comparison: pattern-driven SLMS vs. baseline implementation.

VII. Discussion

A. Spring Boot as a Pattern-Aligned Framework

A notable finding is the deep alignment between Spring Boot's framework architecture and GoF design patterns. Spring's IoC container is fundamentally a Singleton registry; Spring MVC's DispatcherServlet implements the Front Controller pattern; Spring Data JPA's repositories embody the Repository pattern; and Spring's AOP support implements the Proxy pattern. This means that adopting Spring Boot does not merely enable pattern use—it enforces it, providing architectural guardrails that guide developers toward patterncompliant designs [6].

TABLE II
GoF Pattern Summary: Application, Quality Attributes Achieved, and SOLID Alignment

Pattern	Category	SLMS Application	Quality Attributes	SOLID Principle
Singleton	Creational	@Service/@Repository Spring beans	Resource efficiency; consistent state; DI lifecycle	Dependency Inversion
Builder	Creational	BookTracker fluent construction	Readability; flexibility; nullsafety	Single Responsibility
Factory	Creational	AppUser creation in UserServiceImpl	Encapsulation of creation; consistency	Open/Closed
Observer	Behavioural	OverdueEvent / @EventListener	Loose coupling; extensibility; modular notification	Open/Closed
Decorator	Structural	@RestController, @Transactional	Separation of concerns; crosscutting without inheritance	Single Responsibility

B. OOP Principle Interaction

The four OOP principles operate synergistically in the SLMS rather than independently. Encapsulation (private fields) enables Inheritance (safe superclass state extension). Abstraction (service interfaces) enables Polymorphism (injectable implementations). The Base superclass demonstrates that even a minimal inheritance hierarchy confers measurable duplication reduction across multiple subclasses. This synergy suggests that OOP principles should be evaluated compositionally, not in isolation.

ISSN : 2455-135X

C. *Future Enhancements*

Several design improvements are recommended for production deployment. Password storage should be upgraded from plain text to BCrypt hashing, implementing Spring Security's PasswordEncoder. The Factory pattern implementation should be formalised into an explicit factory class
(e.g., UserFactory, BookTrackerFactory) to improve cohesion. The Observer implementation should be extended to support asynchronous event dispatch (@Async) to prevent overdue checking from blocking the HTTP request thread [11].

VIII. Conclusion

This paper has demonstrated the systematic application of four OOP principles and five GoF design patterns within a Spring Boot Library Management System. The layered architecture enforces clean separation of concerns, with each layer realising specific OOP principles: encapsulation at the entity level, abstraction at the service interface boundary, inheritance in the base class hierarchy, and polymorphism through service contracts. The GoF patterns—Singleton, Builder, Factory, Observer, and Decorator—collectively produce a system with high maintainability, extensibility, and testability scores, as evaluated against a non-pattern baseline.

A key contribution is the demonstration that Spring Boot's annotation-driven programming model is itself pattern-aligned, making GoF implementation both natural and low-overhead in this ecosystem. Future work should address password security hardening, explicit factory class formalisation, and asynchronous Observer dispatch for production readiness.

References

[1] B. Dathan and S. Ramnath, *Object-Oriented Analysis, Design and Implementation*. Cham: Springer, 2015.

[2] M. Mouratidou *et al.*, "An assessment of design patterns' influence on a Java-based e-commerce application," *J. Theor. Appl. Electron. Commer. Res.*, vol. 5, no. 1, pp. 25–38, 2010.

[3] V. Serbanescu *et al.*, "A design pattern for optimizations in data intensive applications using ABS and Java 8," *Concurr. Comput.*, vol. 28, no. 2, pp. 374–385, 2016.

[4] J. S. Mtebe and R. Raisamo, "Challenges and instructors' intention to adopt open educational resources in higher education in Tanzania," *Int. Rev. Res. Open Distrib. Learn.*, vol. 15, no. 1, pp. 249–271, 2014.

[5] O. D. Bakare, "The use of social media technologies in the provision of library and information services in academic libraries of south-west Nigeria," Doctoral dissertation, 2018.

[6] Y. D. Liang, *Introduction to Java Programming*. Boston: Pearson, 2015.

[7] N. Padhy, R. Panigrahi, and S. Baboo, "A systematic literature review of an object-oriented metric: reusability," in *Proc. Int. Conf. Comput. Intell. Netw.*, 2015, pp. 190–191.

[8] P. Kuchana, *Software Architecture Design Patterns in Java*, 1st ed. Boca Raton: Taylor & Francis, 2004.

[9] P. Spath *et al.*, "Discovering inheritance, polymorphism, and interfaces," in *Learn Java for Android Development*. Apress, 2020, pp. 157–202.

[10] R. F. Olanrewaju *et al.*, "A frictionless and secure user authentication in web-based premium applications," *IEEE Access*, vol. 9, pp. 129240–129255, 2021.

[11] I. S. de Oliveira *et al.*, "Quality of big data systems: A systematic review of practices, methods and tools," in *Proc. XXIII Brazilian Symp. Softw. Quality*, 2024, pp. 22–31.

[12] J. Langr, *Pragmatic Unit Testing in Java with JUnit*. Raleigh: Pragmatic Bookshelf, 2024.