Week 9: OOP Architecture and Implementation

1. Architecture Overview

The Gym Database Management System (GDMS) follows a layered architecture pattern with clear separation of concerns:

1.1 Model Layer

- Contains domain entities (`User` , `Member` , `Equipment` , `Payment`)
- Implements business logic within entities
- Ensures data encapsulation and validation

1.2 Service Layer (Planned)

- Will handle business operations
- Implements transaction management
- Provides interface for controllers

1.3 Data Access Layer (Planned)

- Will handle database operations
- Implements repository pattern
- Uses Hibernate for ORM

2. OOP Principles Implementation

2.1 Encapsulation

- Private fields with public getters/setters in all model classes
- Data validation within setter methods
- Protected access for sensitive methods

Example from `User` class:

```
private String password;
protected String getPassword() { return password; }
protected void setPassword(String password) {
    this.password = password;
}
```

2.2 Inheritance

- Abstract `User` class as base for different user types
- Specialized classes (`Member`, `Admin`, `Trainer`) extend `User`
- Common functionality in base class

Example hierarchy:

```
User (Abstract)
├── Member
├── Admin (Planned)
└── Trainer (Planned)
```

2.3 Polymorphism

- Method overriding in derived classes
- Abstract methods in `User` class
- Interface implementations (planned)

Example from `User` class:

```
public abstract boolean authenticate(String password);
public abstract String getRole();
```

2.4 Abstraction

- Abstract base class for users
- Clear method signatures

- Hidden implementation details

3. Design Patterns

3.1 Singleton Pattern (Planned)

- Will be used for database connection
- Ensures single instance
- Thread-safe implementation

3.2 Factory Pattern (Planned)

- Will create user instances
- Centralizes object creation
- Supports different user types

3.3 Observer Pattern (Planned)

- Will handle notifications
- Decouples components
- Supports event-driven architecture

4. Testing Strategy

4.1 Unit Testing

- Comprehensive test coverage
- Tests for all model classes
- Boundary value testing

Example test structure:

```
@Test
void testMembershipActive() {
    assertTrue(member.isMembershipActive());
    Member expiredMember = new Member(...);
    assertFalse(expiredMember.isMembershipActive());
}
```

4.2 Test Categories

- Creation tests
- Business logic tests
- Edge case tests
- Validation tests

5. Security Implementation

5.1 Password Security

- BCrypt encryption
- Salted hashes
- Secure password storage

5.2 Access Control

- Role-based access
- Method-level security
- Protected resources

6. Current Implementation Status

6.1 Completed Components

- Base model classes
- Core business logic
- Unit tests

- Basic security features

6.2 Pending Components

- Service layer implementation
- Database integration
- User interface
- Additional security features

7. Next Steps

7.1 Short-term Tasks

- Implement remaining model classes
- Complete service layer
- Add database connectivity
- Enhance security features

7.2 Long-term Goals

- Add web interface
- Implement real-time notifications
- Add reporting features
- Mobile application integration

8. Code Quality Measures

8.1 Code Standards

- Consistent naming conventions
- Comprehensive documentation
- Clean code principles
- Design pattern adherence

8.2 Best Practices

- SOLID principles
- DRY principle
- Defensive programming
- Exception handling

9. Dependencies

9.1 Core Dependencies

- Spring Security (Password encryption)
- JUnit (Testing)
- Hibernate (ORM)
- MySQL Connector

9.2 Build Tools

- Maven
- Java 17
- Git

10. Conclusion

The Week 9 implementation demonstrates a solid foundation of OOP principles and best practices. The architecture is designed for scalability and maintainability, with clear separation of concerns and robust testing. Future implementations will build upon this foundation to add more features and enhance existing functionality.