Design Principles in the E-Learning Platform Codebase

1. Single Responsibility Principle (SRP)

Each class has a single responsibility or reason to change:

Example 1: Model Classes

Each model class represents exactly one entity:

```
@Data
@Entity
@Table(name = "mentoring_sessions")
@NoArgsConstructor
@AllArgsConstructor
public class MentoringSession {
    // Only responsible for mentoring session data
}

// filepath: /backend/src/main/java/com/elearning/model/Certificate.java
@Data
@Entity
@Table(name = "certificates")
@NoArgsConstructor
@AllArgsConstructor
public class Certificate {
    // Only responsible for certificate data
}
```

Example 2: Controller Separation

Controllers are divided by domain area:

```
@RestController
@RequestMapping("/api/instructor")
```

```
@PreAuthorize("hasAuthority('ROLE_INSTRUCTOR')")
public class InstructorController {
    // Only handles instructor-related operations
}

// filepath: /backend/src/main/java/com/elearning/controller/AdminControl
@RestController
@RequestMapping("/api/admin")
@PreAuthorize("hasRole('ADMIN')")
public class AdminController {
    // Only handles admin-related operations
}
```

2. Open/Closed Principle (OCP)

The code is structured to be extended without modifying existing code:

Example: Status Enums

Using enums for status types allows extending behavior without modifying existing code:

```
public enum MentoringSessionStatus {
    PENDING,
    APPROVED,
    REJECTED,
    COMPLETED,
    CANCELLED
    // New status can be added without changing how status is used
}
```

The controller code is structured to handle all enum values through consistent patterns:

```
// Update the session status
if (updateRequest.getStatus() != null) {
    session.setStatus(updateRequest.getStatus());
}
```

3. Liskov Substitution Principle (LSP)

This principle is demonstrated through the consistent use of interfaces:

Example: Repository Interfaces

The controllers depend on repository interfaces, not implementations:

```
@Autowired
private CourseRepository courseRepository;
@Autowired
private EnrollmentRepository enrollmentRepository;
```

Any implementation of these repositories can be substituted without breaking the controller's behavior.

4. Interface Segregation Principle (ISP)

The codebase uses specific interfaces rather than general ones:

Example: Repository Interfaces

Each repository interface has specific methods relevant to only that entity:

```
// Implied in the code where different specialized repositories are used
@Autowired
private UserRepository userRepository;
@Autowired
private CourseRepository courseRepository;
@Autowired
private ChapterDetailRepository chapterDetailRepository;
@Autowired
private MentoringSessionRepository mentoringSessionRepository;
```

Instead of having one large repository interface, each entity has its own tailored repository.

5. Dependency Inversion Principle (DIP)

High-level modules depend on abstractions, not concrete implementations:

Example: Controller Dependencies

Controllers depend on repository interfaces, not concrete implementations:

```
@Autowired
private UserRepository userRepository;
@Autowired
private CourseRepository courseRepository;
@Autowired
private EnrollmentRepository enrollmentRepository;
```

The controller uses these interfaces without needing to know their concrete implementations.

Example: Service Layer Abstraction

```
@Autowired
private StudentService studentService;
@Autowired
private InstructorService instructorService;
@Autowired
private CourseService courseService;
```

Controllers depend on service abstractions rather than directly implementing business logic.

Additional SOLID-Aligned Patterns

DTO Pattern for SRP

DTOs separate data transfer concerns from domain entities:

```
CourseResponse response = new CourseResponse(
    course.getId(),
    course.getTitle(),
    course.getDescription(),
    // other properties...
    course.getUpdatedAt()
);
```

Factory Methods for OCP/SRP

Creating response objects through factory-like methods:

```
List<MentoringSessionResponse> responses = sessions.stream()
    .map(session -> new MentoringSessionResponse(
        session.getId(),
        session.getStudent().getId(),
        // other mappings...
        session.getNotes()
    ))
    .collect(Collectors.toList());
```

Usage of SOLID principles has been fulfilled, particularly through the clear separation of concerns, dependency injection, and interface-based design. These practices contribute to a more maintainable and extensible application.