Design Patterns in the E-Learning Platform Codebase

1. Adapter Pattern

The Adapter pattern converts the interface of a class into another interface that clients expect, allowing classes with incompatible interfaces to work together.

Example from codebase: The conversion of domain entities to DTOs in the InstructorController

```
List<CourseResponse> response = courses.stream()
        .map(course -> new CourseResponse(
                course.getId(),
                course.getTitle(),
                course.getDescription(),
                // other properties...
                course.getUpdatedAt()))
        .collect(Collectors.toList());
```

Here, `CourseResponse` acts as an adapter that transforms the complex `Course` entity (with potential circular references and JPA-specific fields) into a client-friendly format. This adapter pattern shields API consumers from internal domain model changes and provides only the necessary data.

2. Decorator Pattern

The Decorator pattern attaches additional responsibilities to objects dynamically without modifying their structure, offering a flexible alternative to subclassing.

Example from codebase: The Spring annotations used on InstructorController

```
@CrossOrigin(origins = "http://localhost:3000", maxAge = 3600)
@RestController
@RequestMapping("/api/instructor")
@PreAuthorize("hasAuthority('ROLE_INSTRUCTOR')")
public class InstructorController {
```

```
    // Methods...
}
```

Each annotation decorates the InstructorController class with additional behavior:

- `@CrossOrigin` adds CORS support for cross-domain requests
- `@RestController` marks it as a component that handles web requests and automatically serializes responses
- `@RequestMapping` routes requests to this controller
- `@PreAuthorize` adds security checks before method execution

These decorators enhance the class's functionality without changing its core implementation.

3. Factory Method Pattern

The Factory Method pattern defines an interface for creating objects but lets subclasses decide which classes to instantiate.

Example from codebase: Creating DTO objects in controller methods

```
@PostMapping("/courses")
public ResponseEntity<?> createCourse(
        @AuthenticationPrincipal UserDetails userDetails,
        @RequestBody CourseRequest courseRequest) {
    // ...
    CourseResponse response = new CourseResponse(
        savedCourse.getId(),
        savedCourse.getTitle(),
        // other properties...
        savedCourse.getUpdatedAt());

    return ResponseEntity.ok(response);
}
```

While not a classic factory method pattern, this follows the factory concept where controller methods act as "factories" for creating response objects. The controller encapsulates the knowledge of how to create appropriately formatted responses from domain objects, allowing the creation logic to vary independently from client code that uses these objects.

4. Singleton Pattern

The Singleton pattern ensures a class has only one instance and provides a global point of access to it.

Example from codebase: Spring-managed components like InstructorController

```
@RestController
@RequestMapping("/api/instructor")
@PreAuthorize("hasAuthority('ROLE_INSTRUCTOR')")
public class InstructorController {
    @Autowired
    private UserRepository userRepository;

    @Autowired
    private CourseRepository courseRepository;
    // ...
}
```

Spring's dependency injection system maintains singleton instances of components by default. When you use `@Autowired` to inject repositories and other dependencies, you're accessing singleton instances. The InstructorController itself is a singleton - Spring creates only one instance of it for the entire application and reuses it for all requests, managing its lifecycle. This ensures consistency and reduces resource consumption.

These patterns work together to create a maintainable, flexible architecture in your e-learning platform. The combination of adapters for data transformation, decorators for cross-cutting concerns, factory methods for object creation, and singletons for resource management demonstrates good application of object-oriented design principles.