Documentation for the Builder Pattern: Course Object Construction

---

Here's the detailed documentation for the Builder Pattern implementation, crucial for your project viva:

3. Builder Pattern: Course Object Construction

- **Files:**
  - src/main/java/com/ums/Course.java (Contains both the Course class and its nested CourseBuilder class)

Purpose of the Pattern

The Builder Pattern is a creational design pattern designed to construct complex objects step by step. It separates the construction of a complex object from its representation, allowing the same construction process to create different representations (i.e., different configurations of the object).

In the context of the University Management System (UMS) and Course objects:

- **Handling Complex Objects with Many Optional Parameters:** A Course object can have numerous attributes: courseCode, title, credits, department (required), but also optional ones like prerequisites, isElective, description. Using a traditional constructor for all these would lead to a "telescoping constructor" problem (many constructors with increasing numbers of parameters), which is hard to read and maintain.
- **Improving Readability and Maintainability:** The Builder pattern makes object creation code much cleaner and more readable, especially when many parameters are involved and some are optional. Instead of long, confusing constructor calls, you get descriptive setter-like methods.
- **Ensuring Object Immutability:** The Course object itself can be made immutable (its state cannot be changed after creation). The builder handles all the modifications during the construction phase, and once build() is called, the Course object is fully formed and unchangeable. This is a best practice for thread safety and predictable behavior.
- **Separation of Concerns:** The responsibility of constructing a Course object is delegated to the CourseBuilder class, keeping the Course class focused purely on representing its data and behavior.

How it Works (Implementation Details)

The Builder Pattern for Course objects is implemented with two main components:

1. **The Course Class (The Product - Course.java):**
   Java
   ```
   public class Course {
       private final String courseCode;
   ```

Documentation for the Builder Pattern: Course Object Construction

```
    private final String title;
    // ... other final fields (required and optional) ...
    private final List<String> prerequisites; // Note: made unmodifiable

    private Course(CourseBuilder builder) { // Private constructor
        this.courseCode = builder.courseCode;
        this.title = builder.title;
        // ... assign all fields from the builder ...
        this.prerequisites = builder.prerequisites != null ?
                             Collections.unmodifiableList(builder.prerequisites) :
Collections.emptyList();
    }
    // ... Getters and toString() ...

    public static class CourseBuilder { // Nested static Builder class
        // ... builder's fields (copies of Course's fields) ...

        public CourseBuilder(String courseCode, String title, int credits, String department) { /* ... */ }
// Builder's constructor for required fields

        public CourseBuilder withPrerequisites(List<String> prerequisites) { /* ... */ return this; }
        public CourseBuilder asElective(boolean isElective) { /* ... */ return this; }
        public CourseBuilder withDescription(String description) { /* ... */ return this; }

        public Course build() { // Final method to create Course
            return new Course(this);
        }
    }
}
```

- **Private Constructor:** The Course class has a private constructor that accepts a CourseBuilder object. This ensures that a Course object *cannot* be instantiated directly from outside the class, enforcing that it must be built via the builder.
- **Final Fields:** All fields in the Course class are final. This guarantees that once a Course object is created by the builder, its state cannot be modified, making it **immutable**. For mutable collections like prerequisites, Collections.unmodifiableList() is used to ensure the list itself cannot be changed after the Course is built.
- **Getters and toString():** Standard methods to access the Course's data and provide a string representation for debugging/logging.

2. **The CourseBuilder Class (The Builder - Nested in Course.java):**
   - **Nested Static Class:** The CourseBuilder is implemented as a public static nested class within Course. This groups the builder logically with the object it builds and allows it to be instantiated without an instance of Course (new

Course.CourseBuilder(...)).
    - **Builder's Fields:** It mirrors the fields of the Course class. Optional fields are often initialized to default values (e.g., isElective = false) or null.
    - **Builder's Constructor:** It takes all the **required** parameters for a Course. This ensures that a builder cannot even be started without providing the essential information.
    - **Fluent API (Chaining Methods):** It provides public methods (e.g., withPrerequisites(), asElective(), withDescription()) for setting each **optional** attribute. These methods return this (the CourseBuilder instance itself), allowing multiple calls to be chained together in a fluent, readable manner.
    - **build() Method:** This is the final method called to construct the Course object. It takes all the parameters accumulated in the CourseBuilder and passes them to the Course's private constructor, returning the fully assembled Course object.

Justification for Builder Implementation

- **Constructor Overload Avoidance:** Eliminates the need for multiple overloaded constructors (the "telescoping constructor" anti-pattern) when dealing with a large number of optional parameters, making Course construction much cleaner.
- **Enhanced Readability:** Instead of a long list of anonymous parameters in a constructor call, the builder provides named methods (withPrerequisites, asElective), making the code highly self-documenting.
    - Compare: new Course("CS101", "PF", 3, "CS", null, false, null); (unclear what nulls mean)
    - With Builder: new Course.CourseBuilder("CS101", "PF", 3, "CS").build(); (clear, concise)
- **Flexibility in Object Construction:** You can set parameters in any order, and only set the ones you need. This is especially useful for objects with many optional attributes.
- **Promotes Immutability:** By making the Course object immutable (all final fields, private constructor), it becomes inherently thread-safe and more predictable. The builder handles all the mutable state during the construction phase.
- **Guaranteed Valid State (if implemented carefully):** The builder can include validation logic in its build() method to ensure that the Course object is only constructed if all necessary conditions are met, preventing the creation of invalid objects.

This completes the documentation for the Builder Pattern. You've now covered all three creational patterns as per your roadmap!

We are now ready to move into **Phase 3: Structural Patterns Integration**, starting with the **Adapter Pattern**. Let me know when you're ready!