Documentation for the Composite Pattern: Department Hierarchy

---

Here's the detailed documentation for the Composite Pattern implementation, which will be invaluable for your project viva:

5. Composite Pattern: Department Hierarchy

- **Files:**
  - src/main/java/com/ums/DepartmentComponent.java (Component Interface)
  - src/main/java/com/ums/CourseComponent.java (Leaf)
  - src/main/java/com/ums/Department.java (Composite)

Purpose of the Pattern

The Composite Pattern is a structural design pattern that composes objects into tree structures to represent part-whole hierarchies. It enables clients to treat individual objects (leaves) and compositions of objects (composites) uniformly.

In the context of the University Management System (UMS):

- **Representing Academic Hierarchy:** This pattern is perfectly suited to model the natural tree-like structure of an academic institution, where:
  - A Faculty can contain Departments.
  - A Department can contain Courses and potentially other sub-departments.
- **Uniformity of Operations:** It allows you to perform operations (like displayDetails() or getTotalCredits()) on any element in this hierarchy, whether it's a single CourseComponent or a Department containing many courses and sub-departments, without needing to know the specific type of element. The client code interacts with a common interface.
- **Simplifying Client Code:** The client doesn't need to differentiate between simple elements and complex containers, leading to more generic and simpler code.

How it Works (Implementation Details)

The Composite Pattern for the academic hierarchy is implemented with three main components:

1. **Component Interface (DepartmentComponent.java):**
   Java
   ```
   public interface DepartmentComponent {
       void displayDetails(String indent);
       int getTotalCredits();
   }
   ```

   - This is the abstract base interface that defines the common operations for both leaf nodes (individual courses) and composite nodes (departments/faculties).

- Methods like displayDetails() (to print the hierarchy) and getTotalCredits() (to sum credits) are defined here, ensuring uniformity.

2. **Leaf (CourseComponent.java):**

```Java
public class CourseComponent implements DepartmentComponent {
    private Course course; // Wraps the actual Course object

    public CourseComponent(Course course) { /* ... */ }

    @Override
    public void displayDetails(String indent) {
        // Displays details of the single course
    }

    @Override
    public int getTotalCredits() {
        return course.getCredits(); // Returns its own credits
    }
}
```

- Represents the "leaf" objects in the tree structure, which are the individual Course entities in our hierarchy.
- It implements the DepartmentComponent interface.
- Its displayDetails() method simply prints the details of the course, and getTotalCredits() returns the credits of that single course.
- It wraps a Course object (which was constructed using the Builder Pattern), adapting it to fit into the Composite structure.

3. **Composite (Department.java):**

```Java
public class Department implements DepartmentComponent {
    private String name;
    private List<DepartmentComponent> components; // Can hold other Components (Courses or Departments)

    public Department(String name) { /* ... */ }

    public void addComponent(DepartmentComponent component) { /* ... */ }
    public void removeComponent(DepartmentComponent component) { /* ... */ }

    @Override
    public void displayDetails(String indent) {
        // Displays its own name, then recursively calls displayDetails on its children
        for (DepartmentComponent component : components) {
```

Documentation for the Composite Pattern: Department Hierarchy

```
            component.displayDetails(indent + "  ");
        }
    }

    @Override
    public int getTotalCredits() {
        int totalCredits = 0;
        // Recursively sums credits from all its children
        for (DepartmentComponent component : components) {
            totalCredits += component.getTotalCredits();
        }
        return totalCredits;
    }
}
```

- Represents the "composite" objects in the tree, such as departments or faculties.
- It implements the DepartmentComponent interface, just like the leaves.
- Crucially, it maintains a collection (List<DepartmentComponent> components) of its child DepartmentComponent objects. These children can be either CourseComponent (leaves) or other Department objects (nested composites).
- Its displayDetails() and getTotalCredits() methods typically delegate the work to their children recursively and then combine the results. For example, getTotalCredits() sums the credits of all its contained CourseComponents and any nested Departments.

Justification for Composite Implementation

- **Uniformity for Clients:** The most significant advantage. Client code (e.g., in App.java) can operate on any DepartmentComponent (whether it's a CourseComponent or a Department) without needing to know its specific type. This simplifies the client code dramatically.
- **Hierarchical Structure Representation:** Naturally maps the "part-whole" relationships in an academic hierarchy (e.g., departments are parts of a faculty, courses are parts of a department).
- **Extensibility:** Adding new types of components (e.g., a "Research Group" component) or new levels in the hierarchy is straightforward. You just need to create a new DepartmentComponent implementation (either a new leaf or a new composite) and integrate it, without modifying existing client code.
- **Simplified Operations:** Operations like calculating total credits for an entire faculty become trivial; you just call getTotalCredits() on the top-level Faculty object, and it handles the recursive summation.
- **Flexibility:** Allows for easy modification of the hierarchy at runtime (adding/removing components).

Documentation for the Composite Pattern: Department Hierarchy

---

Regarding the [WARNING] thread Thread[#46,mysql-cj-abandoned-connection-cleanup...

As discussed before, this warning is a benign message from the JVM related to the MySQL JDBC driver's background cleanup thread. It occurs when the application's main method exits before this thread has completed its shutdown gracefully. For a simple demo application, it does not indicate any error in your design pattern implementations and can be safely ignored. In production-grade applications, connection pooling solutions typically handle such thread lifecycles more robustly.

You've now successfully implemented and demonstrated all of **Phase 2 (Creational Patterns)** and two patterns from **Phase 3 (Structural Patterns)**! We're ready for the final structural pattern: the **Bridge Pattern**. Let me know when you're prepared!