Documentation for the Adapter Pattern: External Grading System Integration

---

Here's the detailed documentation for the Adapter Pattern implementation, which will be very helpful for your project viva:

4. Adapter Pattern: External Grading System Integration

- **Files:**
  - src/main/java/com/ums/GradingSystem.java (Target Interface)
  - src/main/java/com/ums/ExternalGradingSystemAPI.java (Adaptee)
  - src/main/java/com/ums/GradingSystemAdapter.java (Adapter)

Purpose of the Pattern

The Adapter Pattern (also known as Wrapper) allows two incompatible interfaces to work together. It acts as a bridge between two classes by converting the interface of one class into another interface that the client expects.

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

- **Integrating Incompatible Systems:** The UMS needs to interact with a grading system, but an existing ExternalGradingSystemAPI has different method signatures and data formats (e.g., uses numerical scores and ext_studentID_ext_courseID keys, while UMS expects String grades like "A" and direct studentId, courseCode).
- **Facilitating Communication:** The Adapter bridges this gap, enabling the UMS to "talk" to the ExternalGradingSystemAPI using its familiar GradingSystem interface, without needing to modify the existing external system's code or the core UMS client code.
- **Promoting Code Reusability:** It allows you to reuse existing third-party or legacy components (like the simulated ExternalGradingSystemAPI) even if their interfaces don't perfectly match your new system's design.

How it Works (Implementation Details)

The Adapter Pattern is implemented through the following components:

1. **Target Interface (GradingSystem.java):**
   ```
   Java
   public interface GradingSystem {
       String getStudentGrade(String studentId, String courseCode);
       boolean updateStudentGrade(String studentId, String courseCode, String newGrade);
   }
   ```

   - This is the interface that your UMS client code (e.g., your App.java's testing logic, or later a GradeService in your UMS) expects to interact with. It defines the desired

Documentation for the Adapter Pattern: External Grading System Integration

contract for any grading system.

2. **Adaptee (ExternalGradingSystemAPI.java):**

```
Java
public class ExternalGradingSystemAPI {
    public int fetchScore(String studentExternalID, String courseExternalID) { /* ... */ }
    public boolean setScore(String studentExternalID, String courseExternalID, int score) { /* ... */ }
    // ... helper methods for ID mapping ...
}
```

   - This represents the existing "external" or "legacy" system that you cannot (or should not) modify. Its methods (fetchScore, setScore) have different names, parameter types (e.g., int score instead of String grade), and potentially different ID formats compared to what the GradingSystem interface expects.

3. **Adapter (GradingSystemAdapter.java):**

```
Java
public class GradingSystemAdapter implements GradingSystem {
    private ExternalGradingSystemAPI externalAPI; // Holds a reference to the Adaptee

    public GradingSystemAdapter(ExternalGradingSystemAPI externalAPI) {
        this.externalAPI = externalAPI;
    }

    @Override
    public String getStudentGrade(String studentId, String courseCode) {
        // 1. Map UMS IDs to external IDs
        String externalStudentId = externalAPI.getExternalStudentId(studentId);
        String externalCourseId = externalAPI.getExternalCourseId(courseCode);
        // 2. Call the Adaptee's method (e.g., fetchScore)
        int score = externalAPI.fetchScore(externalStudentId, externalCourseId);
        // 3. Convert the Adaptee's output (int score) to the Target's expected format (String grade)
        return convertScoreToLetterGrade(score);
    }

    @Override
    public boolean updateStudentGrade(String studentId, String courseCode, String newGrade) {
        // 1. Map UMS IDs to external IDs
        // 2. Convert the Target's input (String newGrade) to the Adaptee's expected format (int
score)
        int score = convertLetterGradeToScore(newGrade);
        // 3. Call the Adaptee's method (e.g., setScore)
        return externalAPI.setScore(externalStudentId, externalCourseId, score);
    }
```

Documentation for the Adapter Pattern: External Grading System Integration

```
    // ... Helper methods: convertScoreToLetterGrade, convertLetterGradeToScore ...
}
```

- This is the core of the pattern. It **implements the GradingSystem (Target) interface**, meaning it provides the methods that the UMS expects.
- It **contains an instance of the ExternalGradingSystemAPI (Adaptee)**.
- When a method from the GradingSystem interface is called on the adapter (e.g., getStudentGrade), the adapter:
  1. Performs any necessary **data transformations** (e.g., converting UMS IDs to external IDs, String grades to int scores, or vice-versa).
  2. Calls the corresponding method on the **Adaptee's instance** (externalAPI).
  3. Transforms the Adaptee's return value (if any) back into the format expected by the Target interface.

Justification for Adapter Pattern Implementation

- **Interface Compatibility:** Solves the problem of incompatible interfaces without modifying existing code. This is critical for integrating third-party libraries or legacy systems.
- **Client Decoupling:** The UMS client code interacts only with the GradingSystem interface, completely unaware that an ExternalGradingSystemAPI with a different interface is being used behind the scenes. This promotes loose coupling.
- **Flexibility and Maintainability:** If the external grading system's API changes in the future, only the GradingSystemAdapter needs to be modified. The core UMS logic and other components remain untouched. If a new external grading system needs to be integrated, a new adapter can be written for it, again without impacting existing code.
- **Improved Code Organization:** Keeps the translation logic encapsulated within the adapter, preventing this logic from being scattered throughout your application.
- **Reusability of Existing Code:** Allows you to leverage the functionality of existing classes even if their API doesn't fit your current design, avoiding rewriting functional code.

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

This warning is a common occurrence when a simple Java application using JDBC drivers (like the MySQL Connector/J) finishes execution quickly. The mysql-cj-abandoned-connection-cleanup thread is a background daemon thread that the JDBC driver starts to manage connection resources. When your main method completes, this daemon thread might not have had enough time to gracefully shut down, leading to the "interrupted but still alive" message.

This is generally not an error in your application's logic or the design patterns you've implemented. It's more of an informational message from the JVM about a lingering

background thread. In a long-running server application or one using a robust connection pool, these threads are managed over the application's lifetime and usually shut down cleanly. For your current demo project, you can safely disregard this warning; it does not indicate a flaw in your Singleton, Factory, Builder, or Adapter pattern implementations.

With the Adapter Pattern successfully implemented and documented, you've completed all the Creational Patterns and the first Structural Pattern! We're ready to move to the **Composite Pattern** next. Let me know when you're prepared!