Documentation for the Bridge Pattern: Flexible Notification System

---

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

6. Bridge Pattern: Flexible Notification System

- **Files:**
  - src/main/java/com/ums/NotificationSender.java (Implementor Interface)
  - src/main/java/com/ums/EmailSender.java (Concrete Implementor)
  - src/main/java/com/ums/SmsSender.java (Concrete Implementor)
  - src/main/java/com/ums/PortalNotificationSender.java (Concrete Implementor)
  - src/main/java/com/ums/Notification.java (Abstraction - abstract class)
  - src/main/java/com/ums/GradeNotification.java (Refined Abstraction)
  - src/main/java/com/ums/EventNotification.java (Refined Abstraction)

Purpose of the Pattern

The Bridge Pattern is a structural design pattern that aims to decouple an abstraction from its implementation, allowing them to vary independently. This means that changes to one side (the abstraction or its variations) do not affect the other side (the implementation or its variations), and vice-versa.

In the context of the University Management System (UMS), the Bridge Pattern is applied to create a robust and flexible **notification system**:

- **Abstraction (What):** Represents the *type* of notification (e.g., a grade notification, an event announcement).
- **Implementation (How):** Represents the *mechanism* by which the notification is delivered (e.g., email, SMS, internal portal message).

Without the Bridge pattern, you might end up with a class hierarchy that combines both aspects (e.g., EmailGradeNotification, SmsGradeNotification, PortalGradeNotification, EmailEventNotification, etc.), leading to a combinatorial explosion of classes and tight coupling. The Bridge pattern elegantly solves this by separating them.

How it Works (Implementation Details)

The Bridge Pattern consists of two independent hierarchies (Abstraction and Implementation) linked by composition:

1. **Implementor Interface (NotificationSender.java):**
   Java
   ```
   public interface NotificationSender {
       void sendMessage(String recipient, String message);
   }
   ```

Documentation for the Bridge Pattern: Flexible Notification System

- This interface defines the common contract for all concrete implementation strategies. It represents the "how" part of the bridge.
- Any concrete class implementing this interface will provide a specific way to send a message.

2. **Concrete Implementors (EmailSender.java, SmsSender.java, PortalNotificationSender.java):**

```
Java
public class EmailSender implements NotificationSender {
    @Override
    public void sendMessage(String recipient, String message) { /* ... send email ... */ }
}
// Similar implementations for SmsSender and PortalNotificationSender
```

- These classes provide the concrete details for sending messages via different channels (email, SMS, UMS portal).
- They are entirely focused on their specific sending mechanism and do not know about the types of notifications (grade, event, etc.).

3. **Abstraction (Notification.java):**

```
Java
public abstract class Notification {
    protected NotificationSender sender; // The bridge: a reference to the Implementor

    public Notification(NotificationSender sender) { /* ... */ }
    public void setSender(NotificationSender sender) { /* ... */ } // Allows changing implementation at runtime
    public abstract void send(String recipient, String message); // Delegates to sender
}
```

- This is an abstract class that defines the high-level interface for all notification types. It represents the "what" part of the bridge.
- Crucially, it holds a reference (protected NotificationSender sender;) to an object from the NotificationSender (Implementor) hierarchy. This reference is the "bridge."
- The send() method is abstract, but its implementation in concrete subclasses will *delegate* the actual message sending to the sender object.
- The setSender() method allows for dynamic changing of the implementation (e.g., switching from email to SMS for a particular notification object).

4. **Refined Abstractions (GradeNotification.java, EventNotification.java):**

```
Java
public class GradeNotification extends Notification {
    public GradeNotification(NotificationSender sender) { super(sender); }
    @Override
    public void send(String recipient, String message) {
```

Documentation for the Bridge Pattern: Flexible Notification System

```
        String fullMessage = "Grade Update: " + message;
        sender.sendMessage(recipient, fullMessage); // Delegates to the sender
    }
    public void notifyGrade(String studentId, String courseCode, String grade) { /* ... prepares message and calls send ... */ }
}
// Similar for EventNotification
```

- These are concrete classes that extend the Notification abstraction. They represent specific types of notifications.
- They define the content and context specific to their type (e.g., formatting a grade update message).
- When their send() method (or a higher-level specific method like notifyGrade()) is called, they use the sender object (the bridge) to perform the actual message delivery, without needing to know *how* the message is sent.

Justification for Bridge Pattern Implementation

- **Decoupling Abstraction and Implementation:** This is the core benefit. The notification *type* (e.g., GradeNotification) is completely separate from the notification *sending mechanism* (EmailSender). You can add new notification types without changing senders, and new senders without changing notification types.
- **Reduced Class Proliferation:** Avoids a complex hierarchy where you would have classes like EmailGradeNotification, SmsGradeNotification, etc. Instead, you have GradeNotification (abstraction) and EmailSender (implementation), which can be combined at runtime. This significantly reduces the number of classes and makes the system easier to manage.
- **Increased Flexibility:** The ability to change the NotificationSender at runtime (e.g., gradeNotification.setSender(new SmsSender())) means that a GradeNotification object can send its message via email one moment and via SMS the next, without its internal logic being affected.
- **Improved Extensibility:**
  - To add a new notification channel (e.g., WhatsApp Sender), you only need to create a new NotificationSender implementation.
  - To add a new type of notification (e.g., FeeReminderNotification), you only need to create a new Notification subclass.
  - Existing code remains unchanged.
- **Clean Code and Maintainability:** The separation of concerns makes the code cleaner, easier to understand, test, and maintain. Each part has a clear responsibility.

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

Documentation for the Bridge Pattern: Flexible Notification System

As explained previously, this warning is a benign message from the JVM related to the MySQL JDBC driver's background cleanup thread. It's a common occurrence in short-lived applications using JDBC. It indicates that a background thread initiated by the driver didn't shut down gracefully within the JVM's timeout when the main method finished. This does not signify any flaw in your application's logic or the design patterns you've implemented and can be safely ignored for this demo project.

You have now successfully implemented and demonstrated all patterns in **Phase 2 (Creational Patterns)** and **Phase 3 (Structural Patterns)**! This is excellent progress.

Next, we will move into **Phase 4: Behavioral Patterns Implementation**, starting with the **Chain of Responsibility Pattern**. Let me know when you're ready!