## **Low-Level Design Document for DO IT Todo Application**

### **Design Pattern Selection**

#### **Subtask:**

Implementing the **Task Notification System** to ensure customizable and non-intrusive reminders with actionable options.

#### **Design Pattern Family: Behavioral**

We recommend the **Observer Pattern** for the Task Notification System. This pattern ensures that the system can notify multiple observers (e.g., email, push notifications, lock screen banners) whenever a task's state changes (e.g., approaching due date or completed).

### **Justification**

1. **Observer Pattern Characteristics**:
   * Establishes a one-to-many relationship between objects, where one object (Subject) notifies multiple Observers.
   * Easily extensible to add new notification methods (e.g., SMS, webhooks) without modifying the existing code.
   * Decouples the notification mechanism from the task management logic, improving maintainability.
2. **Relevance to Requirements**:
   * Allows users to receive task notifications across various channels (app, email, lock screen).
   * Ensures flexibility in adding/removing observers without disrupting the core functionality.

### **Pseudocode Representation**

#### **1. Interfaces and Classes**

// Observer Interface

interface Observer {

void update(Task task);

}

// Concrete Observers

class EmailNotifier implements Observer {

public void update(Task task) {

System.out.println("Email sent: Task '" + task.getName() + "' is due on " + task.getDueDate());

}

}

class AppNotifier implements Observer {

public void update(Task task) {

System.out.println("App Notification: Task '" + task.getName() + "' is due on " + task.getDueDate());

}

}

// Subject Class

class Task {

private List<Observer> observers = new ArrayList<>();

private String name;

private String dueDate;

public Task(String name, String dueDate) {

this.name = name;

this.dueDate = dueDate;

}

public String getName() {

return name;

}

public String getDueDate() {

return dueDate;

}

// Observer Management

public void attach(Observer observer) {

observers.add(observer);

}

public void detach(Observer observer) {

observers.remove(observer);

}

public void notifyObservers() {

for (Observer observer : observers) {

observer.update(this);

}

}

// Task logic

public void setDueDate(String dueDate) {

this.dueDate = dueDate;

notifyObservers(); // Notify on change

}

}

#### **2. Example Usage**

public class Main {

public static void main(String[] args) {

// Create Task

Task task = new Task("Complete Low-Level Design", "2024-11-20");

// Attach Observers

Observer emailNotifier = new EmailNotifier();

Observer appNotifier = new AppNotifier();

task.attach(emailNotifier);

task.attach(appNotifier);

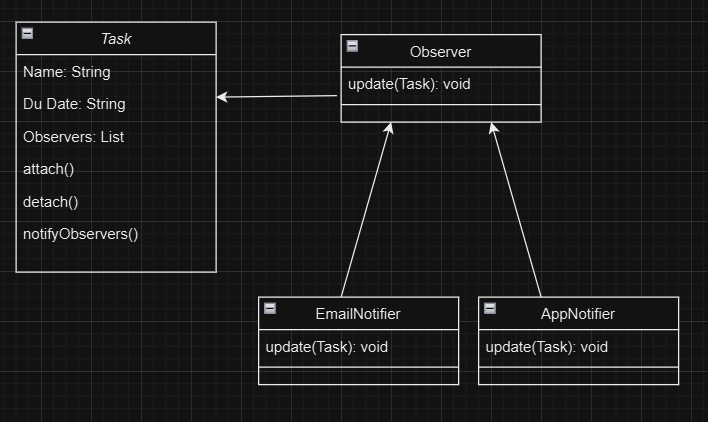
// Trigger Notifications

task.setDueDate("2024-11-19"); // Notifies observers of the change

}

}

**Informal Class Diagram**



### **Benefits of This Design**

1. **Extensibility**: Adding new notification types (e.g., SMS) is as simple as creating a new Observer implementation.
2. **Reusability**: The Task class remains reusable, independent of notification logic.
3. **Scalability**: Easily supports multiple observers without impacting performance.