Project Documentation: Task Management System

Introduction

The Task Management System is a Java-based desktop application designed to manage tasks, assign them to team members, and prioritize them effectively. The project uses a combination of design patterns to ensure scalability, maintainability, and efficient user interactions.

This documentation provides an overview of the implemented classes and the design patterns used, along with justifications for their inclusion.

Class Descriptions

1. Task (model)

- **Description**: Represents a task with attributes like title, type, priority, and assigned user.
- **Attributes**:
  - `title`: The name of the task.
  - `type`: The category of the task (e.g., Bug, Feature, Improvement).
  - `priority`: The importance level (Low, Medium, High).
  - `assignedTo`: The user responsible for the task.

2. Bug, Feature, Improvement (model)

- **Description**: These classes extend the Task class, representing specific task types.

3. TaskFactory (factory)

- **Description**: Factory class responsible for creating different task objects (Bug, Feature, Improvement) based on user input.

4. TaskManager (singleton)

- **Description**: Manages all tasks in the system. Implemented as a Singleton to ensure a single instance across the application.

- **Responsibilities**:
  - Adding, deleting, and fetching tasks.
  - Handling task persistence with SQLite.

5. DatabaseManager (database)

- **Description**: Handles the connection to the SQLite database and initializes the required tables (tasks table).

6. TaskManagementUI (ui)

- **Description**: Implements the graphical user interface (GUI) for the system.
- **Features**:
  - Adding tasks via input fields.
  - Displaying tasks in a table.
  - Searching for tasks using multiple strategies.
  - Deleting tasks.
  - Displaying task statistics.

7. DynamicNotification (observer)

- **Description**: Implements dynamic notifications to update the user when tasks are added or deleted.

Design Patterns

1. Singleton Pattern

- **Class**: TaskManager
- **Purpose**:
  - Ensures that only one instance of TaskManager exists throughout the application.
  - Provides a global point of access for task management operations.
- **Justification**:
  - Centralizing task management operations ensures consistency and avoids data conflicts caused by duplicate instances.

2. Factory Pattern

- **Class**: TaskFactory
- **Purpose**:
    - Encapsulates the creation logic of different task types (Bug, Feature, Improvement).
- **Justification**:
    - Simplifies the process of creating new task objects.
    - Makes the system extensible; new task types can be added without modifying the existing logic.

3. Observer Pattern

- **Classes**: Notifier, Observer, DynamicNotification
- **Purpose**:
    - Notifies the UI dynamically whenever a task is added or deleted.
- **Justification**:
    - Improves user experience by providing real-time feedback.
    - Decouples the notification logic from the core task management operations.

4. Strategy Pattern

- **Classes**: SearchStrategy, SearchByTitle, SearchByPriority
- **Purpose**:
    - Allows dynamic switching between different search methods.
- **Justification**:
    - Makes the search functionality modular and easy to extend.
    - Enhances flexibility by enabling multiple search strategies (e.g., by title, by priority).

5. Decorator Pattern

- **Classes**: TaskDecorator, DeadlineTask
- **Purpose**:
    - Adds new functionality (e.g., deadlines) to tasks without modifying the core Task class.
- **Justification**:
    - Adheres to the Open/Closed Principle (OCP) by allowing new features to be added to tasks without altering the original class.
    - Provides flexibility for extending task functionality.

Justification for the Three Additional Patterns

1. Observer Pattern:

- Provides dynamic notifications to enhance user interaction.
- Decouples the UI from the task management logic, making the system more maintainable.

2. Strategy Pattern:

- Offers flexibility in how tasks are searched (e.g., by title or by priority).
- Makes the system more modular by separating search logic from the core UI.

3. Decorator Pattern:

- Adds new features (e.g., deadlines) to tasks without modifying their base class.
- Supports future enhancements without risking changes to the existing system.

Conclusion

This project implements a scalable and maintainable task management system by leveraging well-known design patterns. Each pattern was chosen based on its ability to address specific requirements, such as task creation, real-time notifications, flexible searching, and extending functionality.