FINAL

Serikbayev Alen SE-2221.

Begimbetova Anelya SE-2221

Task Manager

Project Overview

Project Idea: Create a comprehensive task management system that allows users to create, organize, and manage their tasks efficiently. The system will feature different task types, priorities, deadlines, and enhanced notification options to keep users informed about task updates.

Performance Goal: Improve Task Management Efficiency and User Productivity

Performance Objectives:
The objective is to provide an intuitive and efficient platform that empowers users to manage their tasks effectively, leading to increased productivity and a sense of accomplishment.

Main body

Singleton: The Singleton pattern is used for the TaskManager class to ensure that there is only one instance of this class. This is useful when you need exactly one object to coordinate actions in the system.

SDP-Final-Project main Current File

y.java | CompletedTaskInterface.java | ComplexDecorator.java | TaskDecorator.java | TaskManager.java | TaskDBConnectionAdapter.java | Main.java | Bas

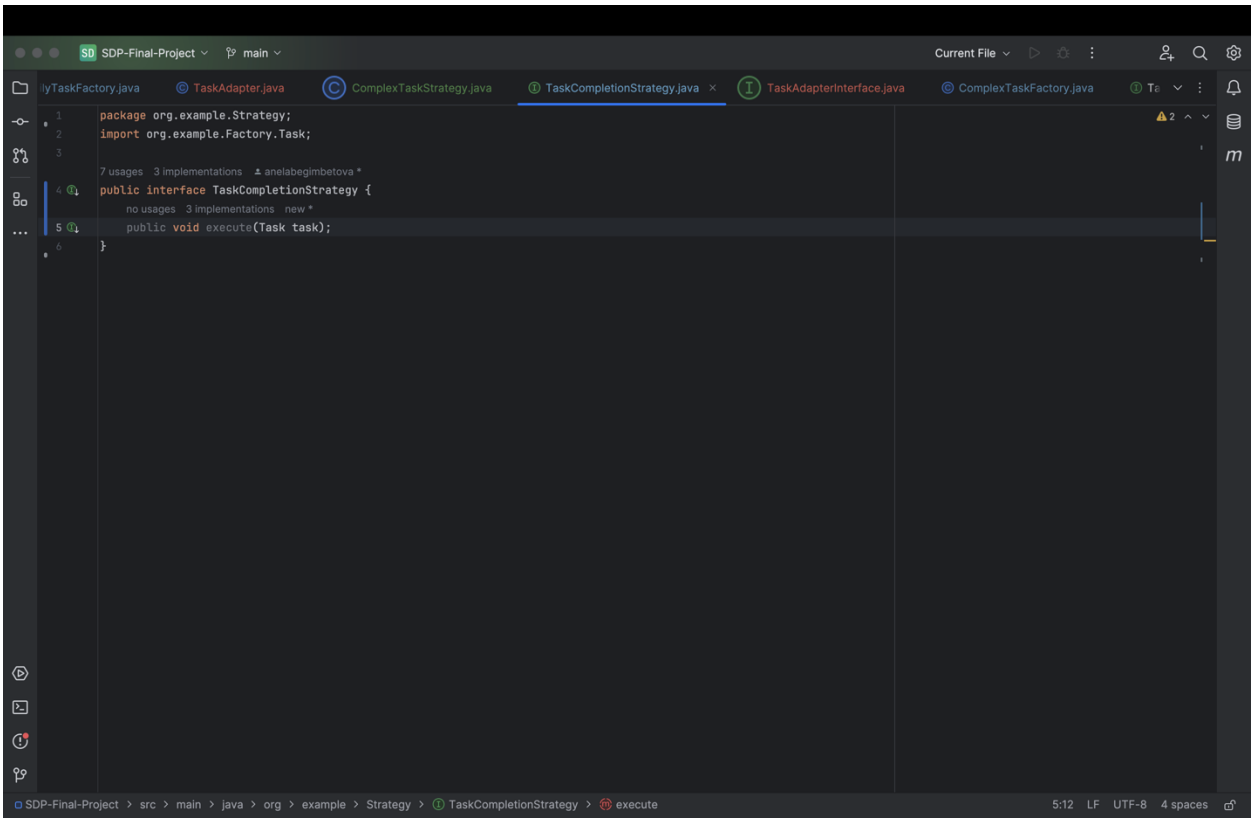
```java
package org.example.Singleton;

import ...
// Singleton для управления регистрацией студентов
4 usages  Wirtinius *
public class TaskManager implements Observable {
    3 usages
    private static TaskManager instance;
    2 usages
    private List<BasicTask> tasks;

    1 usage  Wirtinius
    private TaskManager() { tasks = new ArrayList<>(); }

    no usages  Wirtinius
    public static TaskManager getInstance() {
        if (instance == null) {
            instance = new TaskManager();
        }
        return instance;
    }

    no usages  Wirtinius *
    public void addTask(BasicTask task) {
        tasks.add(task);
        notifyObservers( action: "New task created: " + task.getName());
    }

    // Паттерн Observer
    3 usages
    private List<Observer> observers = new ArrayList<>();

    no usages  Wirtinius *
    @Override
    public void addObserver(Observer observer) { observers.add(observer); }

    no usages  Wirtinius *
```

SDP-Final-Project > src > main > java > org > example > Singleton > TaskManager > addTask 26:34 (9 chars) LF

The Observer pattern: defines a one-to-many dependency between tasks so that when one tasks is created, all its dependents are notified and updated automatically. In the context of my project, I applied the Observer pattern to notify interested parties when a task is saved to the database.

```java
        if (instance == null) {
            instance = new TaskManager();
        }
        return instance;
    }

    no usages  Wirtinius *
    public void addTask(BasicTask task) {
        tasks.add(task);
        notifyObservers( action: "New task created: " + task.getName());
    }

    // Паттерн Observer
    3 usages
    private List<Observer> observers = new ArrayList<>();

    no usages  Wirtinius *
    @Override
    public void addObserver(Observer observer) { observers.add(observer); }

    no usages  Wirtinius *
    @Override
    public void removeObserver(Observer observer) { observers.remove(observer); }

    1 usage  Wirtinius *
    public void notifyObservers(String action) {
        for (Observer observer : observers) {
            observer.update(action);
        }
    }
}
```

The Strategy pattern: this showcases the flexibility of the Strategy Pattern by allowing different strategies for handling task completion and enabling dynamic strategy changes. The pattern promotes encapsulation, flexibility, and maintainability in scenarios where different algorithms need to be applied interchangeably.



The Adapter pattern: In the provided implementation, the TaskAdapter simplifies the interface by providing a default implementation of the isCompleted method. This could be useful in scenarios where you want to provide a consistent behavior for checking completion status, regardless of the actual implementation details of the Task class.

```java
package org.example.Adapter;
import org.example.Factory.Task;

no usages
public class TaskAdapter implements TaskAdapterInterface {

    1 usage
    private Task task;

    no usages
    public TaskAdapter(Task task) { this.task = task; }

    no usages
    @Override
    public boolean isCompleted() { return true; }
}
```

Factory Method: The Factory Pattern is used to create objects without specifying the exact class of the object that will be created. In your project, you have implemented the Factory Pattern to create different types of tasks (DailyTask, EasyTask, ComplexTask) without exposing the details of their creation. They all extends the BasicTaskclass:

```java
4 usages  anelabegimbetova +1 *
public class BasicTask implements Task {
    2 usages
    private String name;
    2 usages
    private String description;
    2 usages
    private int priority;
    no usages
    private int type;
    2 usages
    private boolean completed;



    3 usages  anelabegimbetova +1 *
    public BasicTask(String name, String description, int priority) {
        this.name = name;
        this.description = description;
        this.priority = priority;
        this.completed = false;
    }

    7 usages  anelabegimbetova
    @Override
    public String getName() { return name; }

    3 usages  anelabegimbetova
    @Override
    public String getDescription() { return description; }

    2 usages  anelabegimbetova
    @Override
```

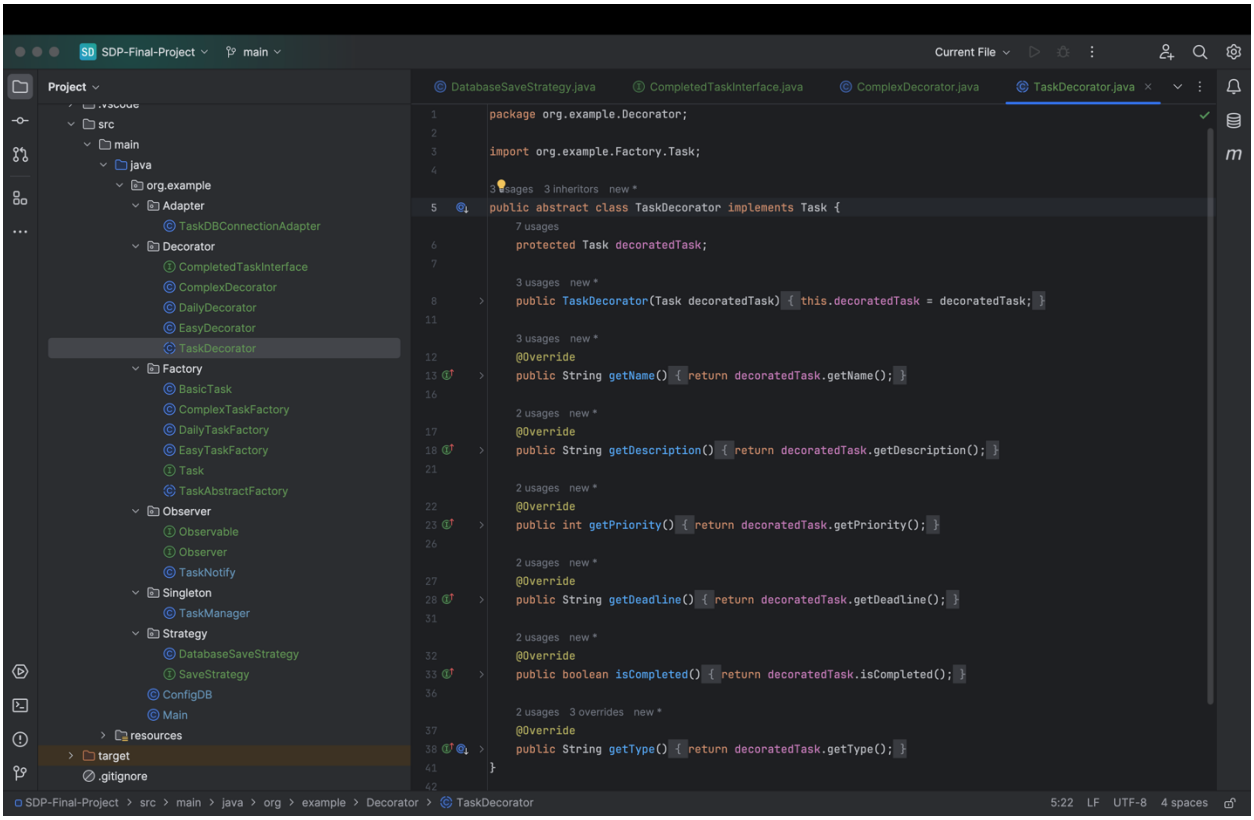
Below is the abstract factory class defining the interface for creating tasks.

It declares an abstract method createTask that concrete factories will implement to create specific types of tasks:

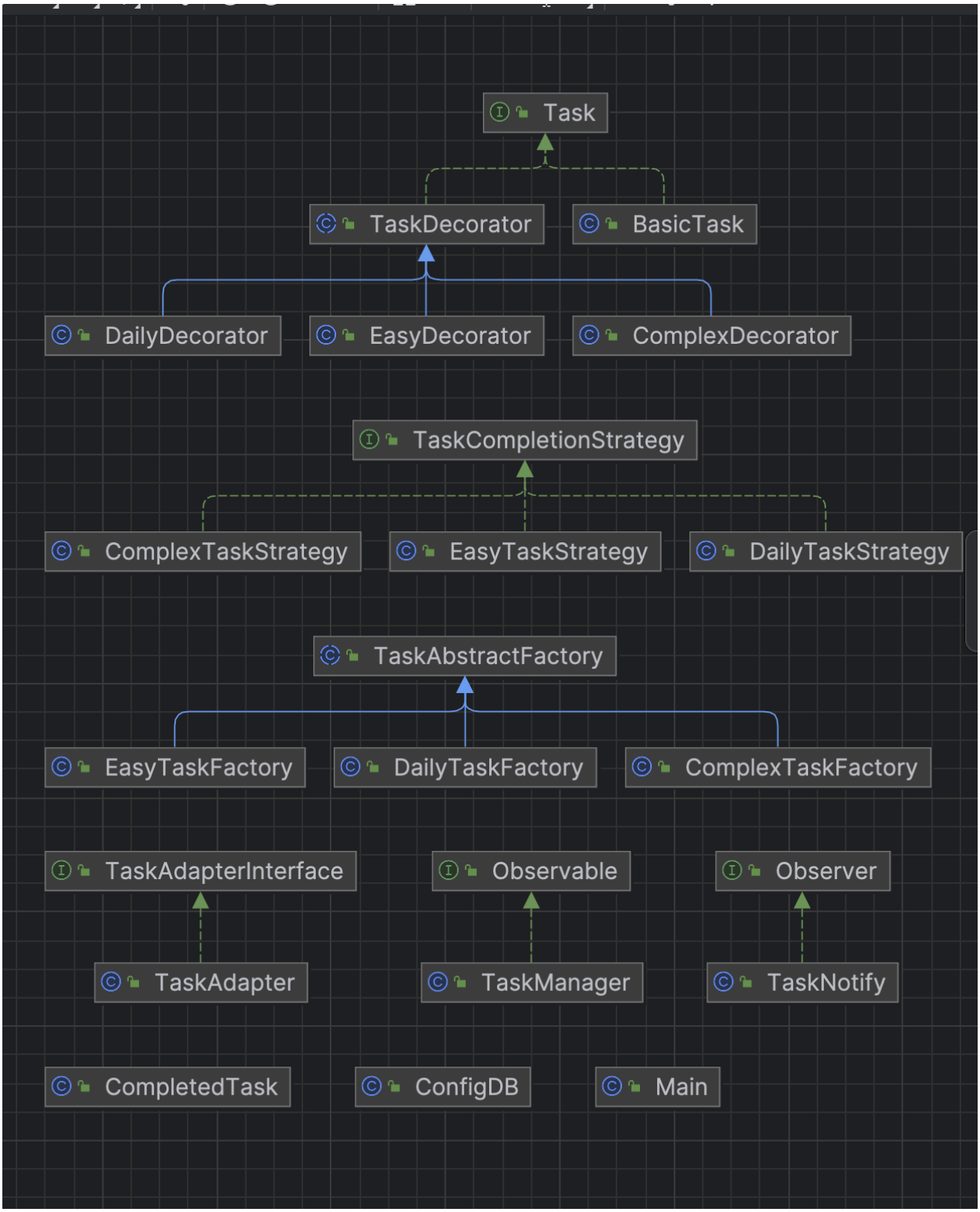
```java
package org.example.Factory;

6 usages  3 inheritors  new *
public abstract class TaskAbstractFactory{
    3 usages  3 implementations  new *
    public abstract Task createTask(String name, String description, Integer priority, String deadline);
}
```

Decorator Pattern:

The Decorator Pattern is used to dynamically attach additional responsibilities to an object. In your project, you have implemented the Decorator Pattern to add different types of behavior (decorators) to a base task (BasicTask). All extends from TaskDecorator class:



UML diagram:


Task
TaskDecorator
BasicTask
DailyDecorator
EasyDecorator
ComplexDecorator
TaskCompletionStrategy
ComplexTaskStrategy
EasyTaskStrategy
DailyTaskStrategy
TaskAbstractFactory
EasyTaskFactory
DailyTaskFactory
ComplexTaskFactory
TaskAdapterInterface
Observable
Observer
TaskAdapter
TaskManager
TaskNotify
CompletedTask
ConfigDB
Main