

**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**

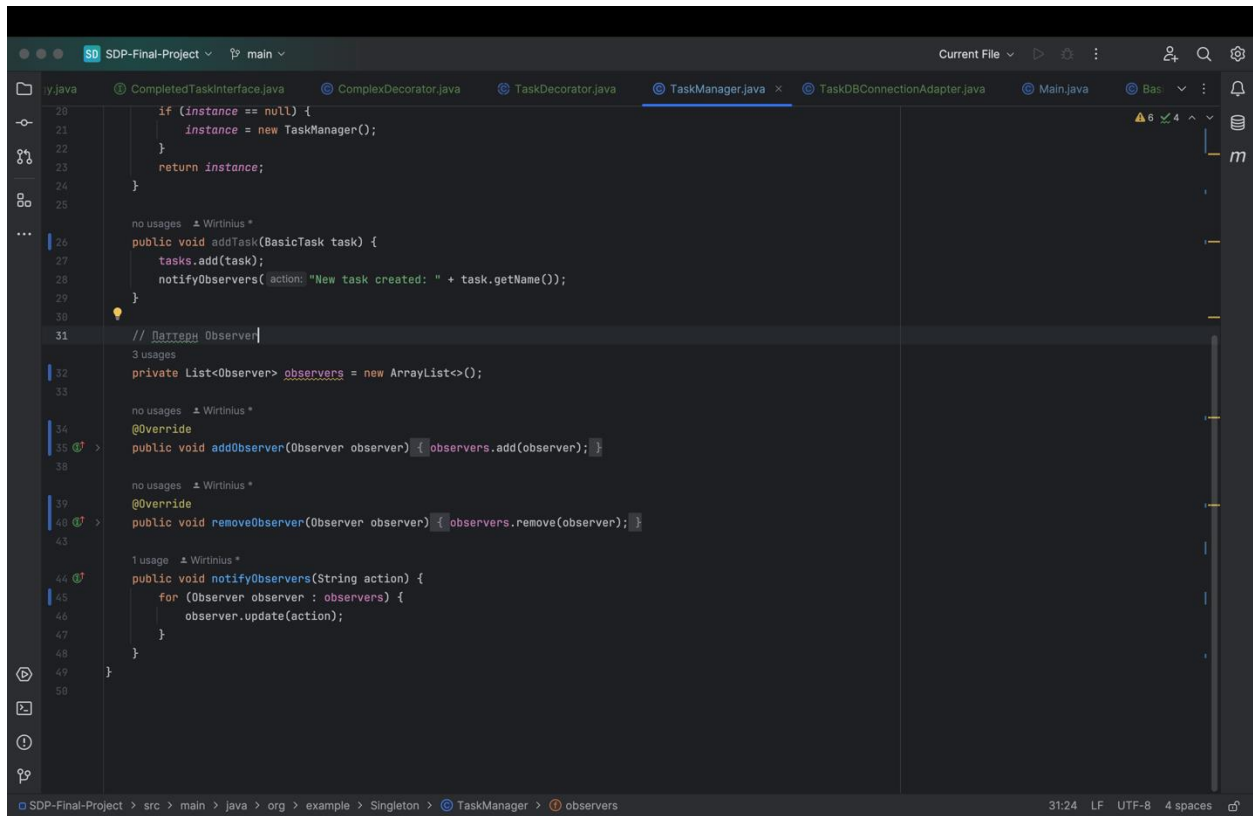
Singleton: The Singleton pattern is used for the TaskManager class to ensure that there is only one instance of this class. This is useful when you need exactly one object to coordinate actions in the system.

SDP-Final-ProjectmainCurrent FileTaskManager.javaTaskDBConnectionAdapter.javaMain.javaBas...

1package org.example.Singleton;  
2  
3> import ...  
4  
5// Singleton для управления регистрацией студентов  
6// 4 usages Wirtinius \*  
7  
8public class TaskManager implements Observable {  
9 3 usages  
10 private static TaskManager instance;  
11 2 usages  
12 private List<BasicTask> tasks;  
13  
14 1 usage Wirtinius  
15 private TaskManager() { tasks = new ArrayList<>(); }  
16  
17 no usages Wirtinius  
18 public static TaskManager getInstance() {  
19 if (instance == null) {  
20 instance = new TaskManager();  
21 }  
22 return instance;  
23 }  
24  
25 no usages Wirtinius \*  
26 public void addTask(BasicTask task) {  
27 tasks.add(task);  
28 notifyObservers( action: "New task created: " + task.getName());  
29 }  
30  
31 // Паттерн Observer  
32 3 usages  
33 private List<Observer> observers = new ArrayList<>();  
34  
35 no usages Wirtinius \*  
36 @Override  
37 public void addObserver(Observer observer) { observers.add(observer); }  
38  
39 no usages Wirtinius \*

SDP-Final-Project > src > main > java > org > example > Singleton > TaskManager > addTask26:34 (9 chars) LF

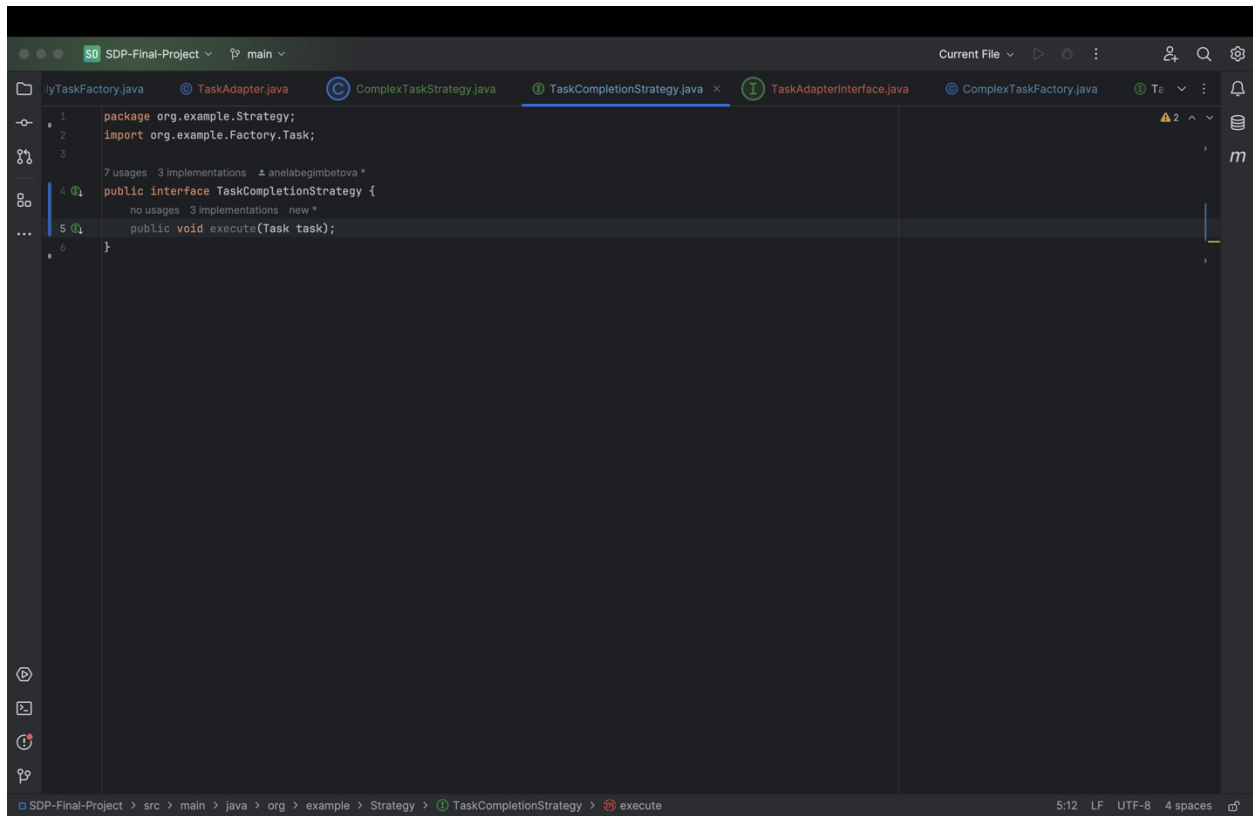
**The Observer pattern:** defines a one-to-many dependency between tasks so that when one task is created, all its dependents are notified and updated automatically. In the context of my project, I applied the Observer pattern to notify interested parties when a task is saved to the database.



```
20  if (instance == null) {
21      instance = new TaskManager();
22  }
23  return instance;
24  }
25
26  no usages  Wirtinius *
27  public void addTask(BasicTask task) {
28      tasks.add(task);
29      notifyObservers(action: "New task created: " + task.getName());
30  }
31
32  // @SuppressWarnings
33  private List<Observer> observers = new ArrayList<>();
34
35  no usages  Wirtinius *
36  @Override
37  public void addObserver(Observer observer) { observers.add(observer); }
38
39  no usages  Wirtinius *
40  @Override
41  public void removeObserver(Observer observer) { observers.remove(observer); }
42
43  1 usage  Wirtinius *
44  public void notifyObservers(String action) {
45      for (Observer observer : observers) {
46          observer.update(action);
47      }
48  }
49  }
50  }
```

SDP-Final-Project > src > main > java > org > example > Singleton > TaskManager > observers 31:24 LF UTF-8 4 spaces

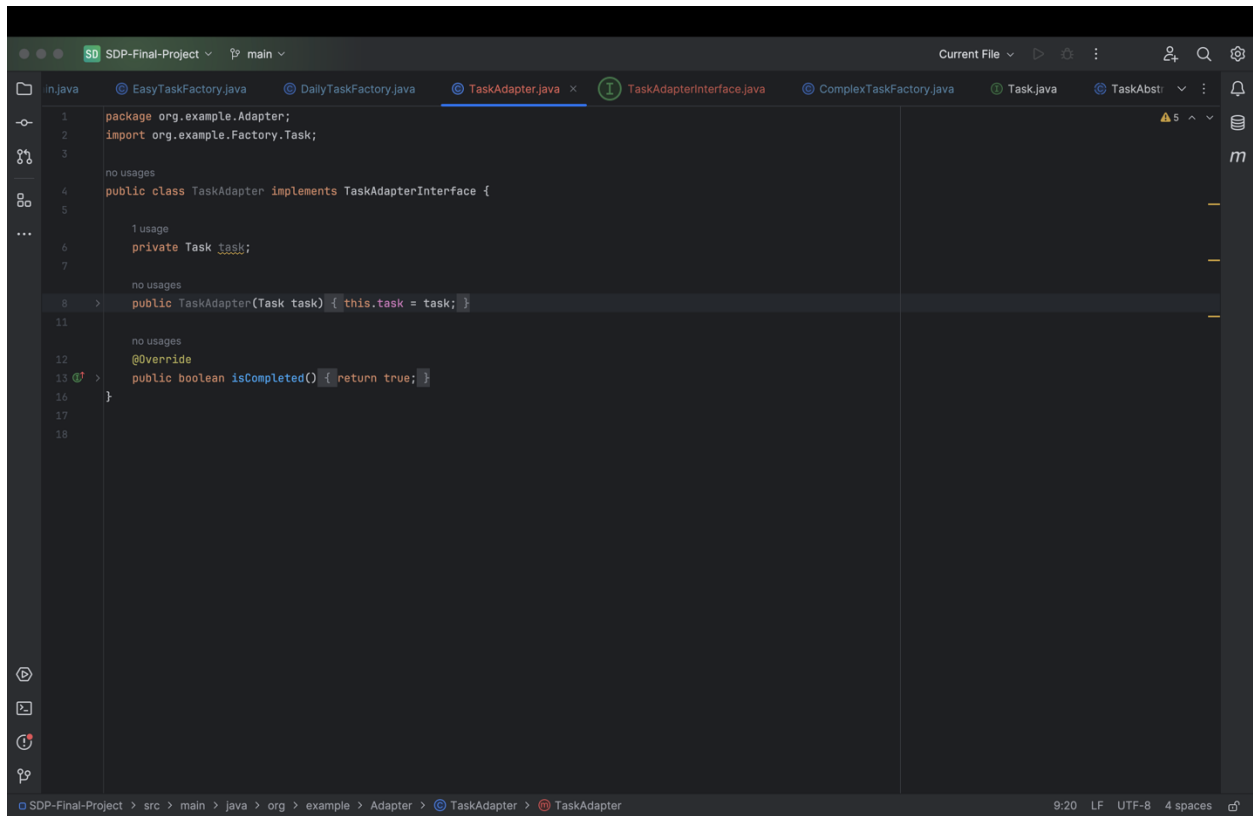
**The Strategy pattern:** this showcases the flexibility of the Strategy Pattern by allowing different strategies for handling task completion and enabling dynamic strategy changes. The pattern promotes encapsulation, flexibility, and maintainability in scenarios where different algorithms need to be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



The screenshot shows an IDE window for a project named "SDP-Final-Project" on the "main" branch. The editor displays the file "TaskCompletionStrategy.java". The code defines a package "org.example.Strategy" and imports "org.example.Factory.Task". It then defines a public interface "TaskCompletionStrategy" with a single method "execute(Task task)". The IDE interface includes a sidebar with project navigation, a top toolbar with standard IDE actions, and a bottom status bar showing the file path "src > main > java > org > example > Strategy > TaskCompletionStrategy.java" and the current cursor position "5:12 LF UTF-8 4 spaces".

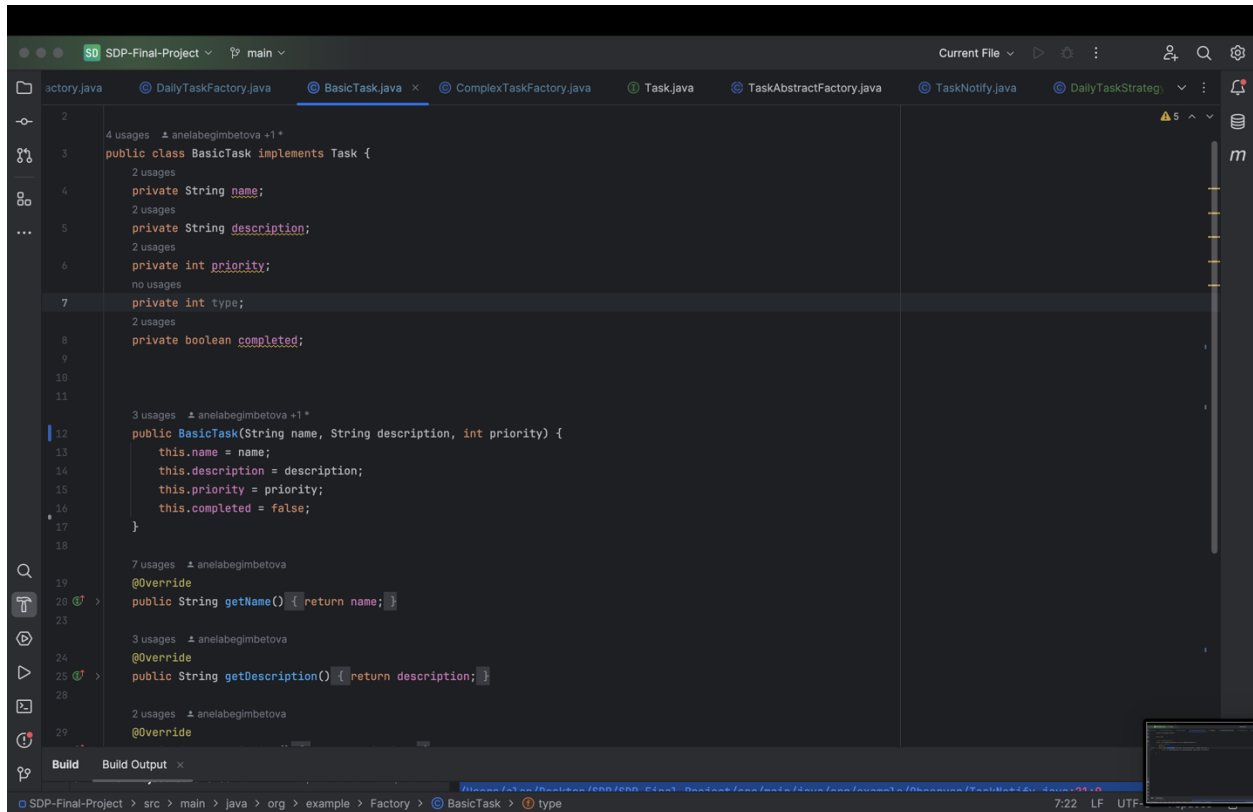
```
1 package org.example.Strategy;
2 import org.example.Factory.Task;
3
4 7 usages 3 implementations 1 anelabegimbetova *
5 public interface TaskCompletionStrategy {
6     no usages 3 implementations new *
7     public void execute(Task task);
8 }
```

**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.



```
1 package org.example.Adapter;
2 import org.example.Factory.Task;
3
4 no usages
5 public class TaskAdapter implements TaskAdapterInterface {
6     1 usage
7     private Task task;
8
9     no usages
10    public TaskAdapter(Task task) { this.task = task; }
11
12    no usages
13    @Override
14    public boolean isCompleted() { return true; }
15 }
16
17
18
```

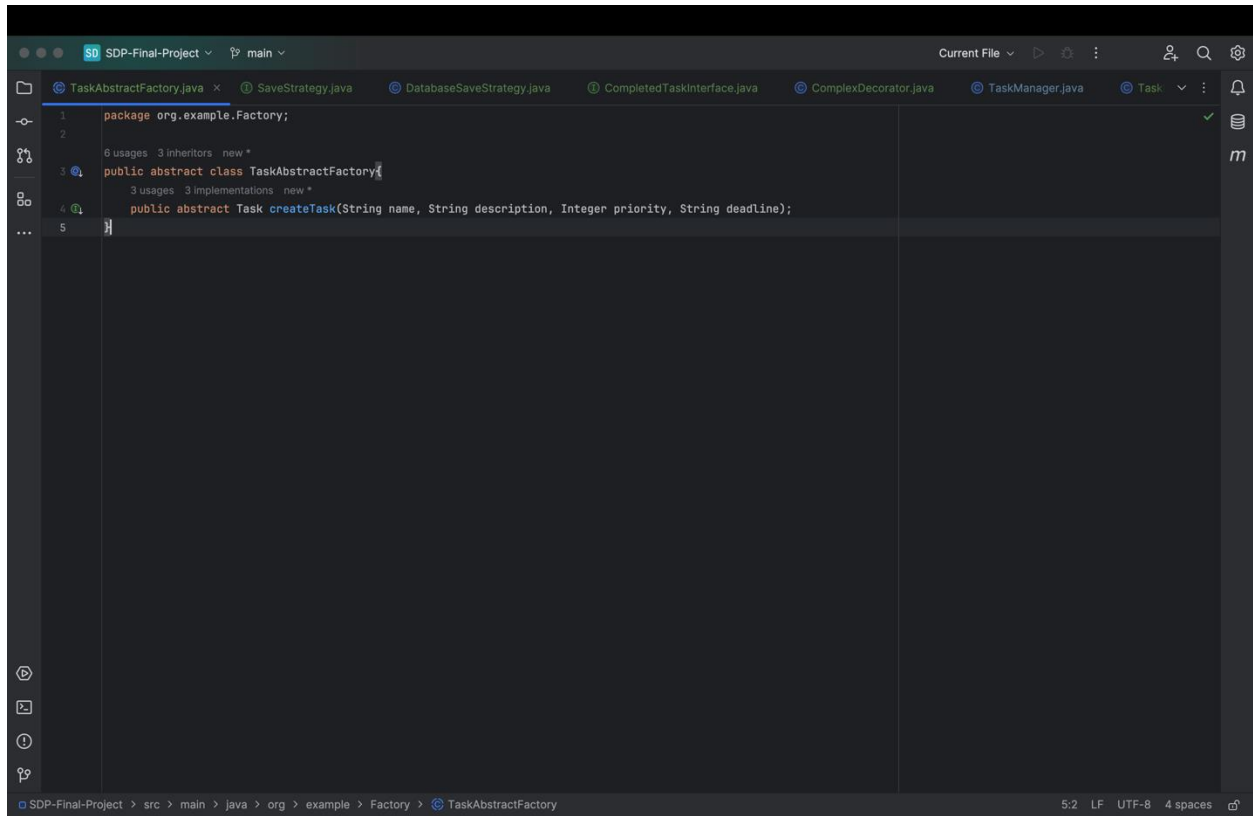
**Factory Method:** The Factory Pattern is used to create objects without specifying the exact class of the object that will be created. In your project, you have implemented the Factory Pattern to create different types of tasks (DailyTask, EasyTask, ComplexTask) without exposing the details of their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



```
1  actory.java  BasicTaskFactory.java  BasicTask.java x  ComplexTaskFactory.java  Task.java  TaskAbstractFactory.java  TaskNotify.java  DailyTaskStrateg...
2
3  4 usages  anelabegimbetova +1 *
4  public class BasicTask implements Task {
5      2 usages
6      private String name;
7      2 usages
8      private String description;
9      2 usages
10     private int priority;
11     no usages
12
13     2 usages
14     private int type;
15
16     2 usages
17     private boolean completed;
18
19
20     3 usages  anelabegimbetova +1 *
21     public BasicTask(String name, String description, int priority) {
22         this.name = name;
23         this.description = description;
24         this.priority = priority;
25         this.completed = false;
26     }
27
28
29     7 usages  anelabegimbetova
30     @Override
31     public String getName() { return name; }
32
33
34     3 usages  anelabegimbetova
35     @Override
36     public String getDescription() { return description; }
37
38
39     2 usages  anelabegimbetova
40     @Override
41
42
43 Build  Build Output x
44
45 SDP-Final-Project > src > main > java > org > example > Factory > BasicTask > type 7:22 LF UTF-
```

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:

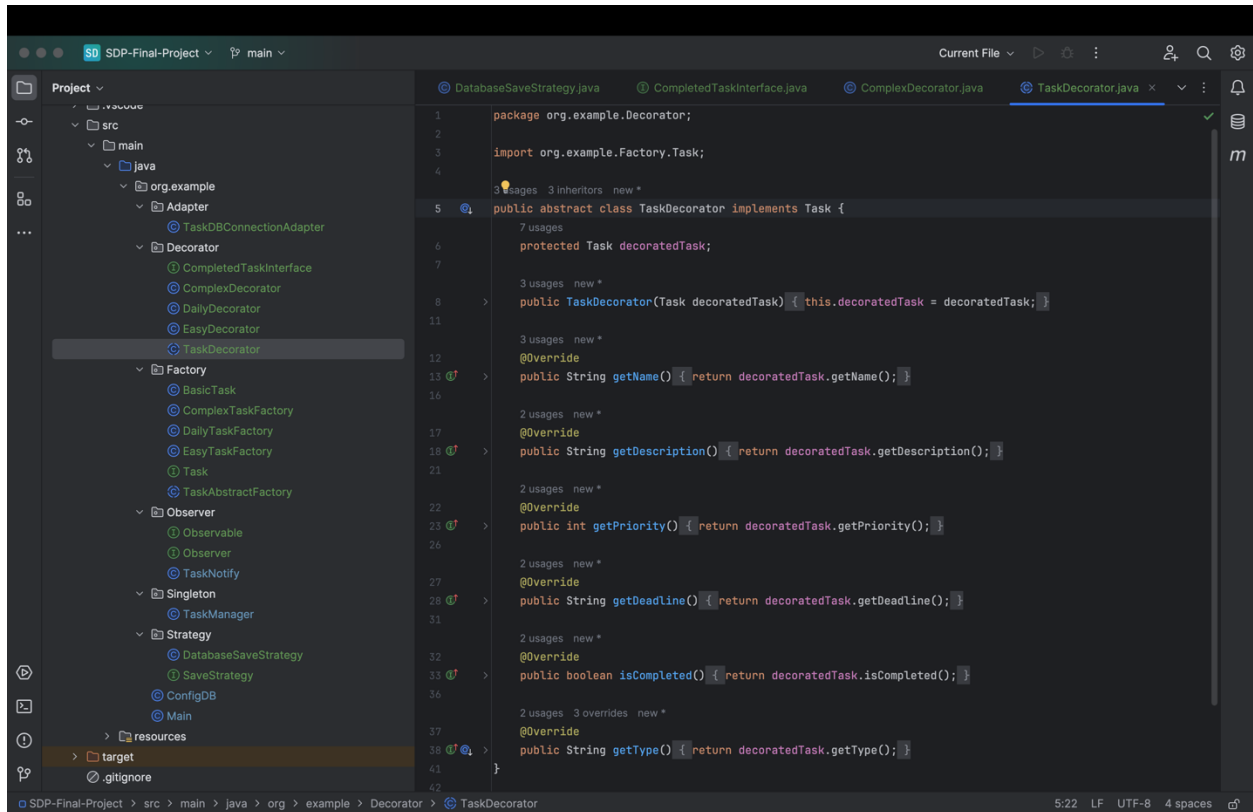


The screenshot shows an IDE window for a project named "SDP-Final-Project". The file explorer on the left shows a package structure: `src > main > java > org > example > Factory`. The editor displays the `TaskAbstractFactory.java` file. The code defines an abstract class `TaskAbstractFactory` in the package `org.example.Factory`. It includes a package-info comment and an abstract method `createTask` that takes parameters for name, description, priority, and deadline. The IDE interface includes a top toolbar with icons for file operations, a search bar, and a bottom status bar showing file encoding (UTF-8) and line length (5:2 LF).

```
1 package org.example.Factory;
2
3 6 usages 3 inheritors new *
4 public abstract class TaskAbstractFactory {
5     3 usages 3 implementations new *
6     public abstract Task createTask(String name, String description, Integer priority, String deadline);
7 }
```

## 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:



The screenshot shows an IDE with the following components:

- Project Explorer (Left):** Displays the project structure. The path is `SDP-Final-Project > src > main > java > org > example > Decorator > TaskDecorator`. The `TaskDecorator` class is highlighted.
- Code Editor (Right):** Shows the source code of `TaskDecorator.java`. The code is as follows:

```
1 package org.example.Decorator;
2
3 import org.example.Factory.Task;
4
5 public abstract class TaskDecorator implements Task {
6     protected Task decoratedTask;
7
8     public TaskDecorator(Task decoratedTask) { this.decoratedTask = decoratedTask; }
9
10    3 usages new *
11
12    @Override
13    public String getName() { return decoratedTask.getName(); }
14
15    2 usages new *
16
17    @Override
18    public String getDescription() { return decoratedTask.getDescription(); }
19
20    2 usages new *
21
22    @Override
23    public int getPriority() { return decoratedTask.getPriority(); }
24
25    2 usages new *
26
27    @Override
28    public String getDeadline() { return decoratedTask.getDeadline(); }
29
30    2 usages new *
31
32    @Override
33    public boolean isCompleted() { return decoratedTask.isCompleted(); }
34
35    2 usages 3 overrides new *
36
37    @Override
38    public String getType() { return decoratedTask.getType(); }
39
40    }
41
42 }
```
- Bottom Bar:** Shows the file path `SDP-Final-Project > src > main > java > org > example > Decorator > TaskDecorator` and the status `5:22 LF UTF-8 4 spaces`.



**UML diagram:**

