

Documentation

for Task Management Application

Overview

The **Task Management Application** is a Java-based desktop application designed to facilitate task creation, assignment, and management. It allows administrators to assign tasks to employees, sort and organize tasks, and switch user roles. The application leverages various **design patterns** to achieve modularity, scalability, and maintainability.

Design Patterns Used

This section provides an overview of the design patterns utilized in the application, along with relevant code snippets for each.

1. Singleton Pattern

The **Singleton Pattern** ensures that only one instance of the Task Management Application class is created during the application's lifecycle.

Code Snippet:

```
private static TaskManagementApplication instance;

public static TaskManagementApplication getInstance() {
    if (instance == null) {
        instance = new TaskManagementApplication();
    }
    return instance;
}

private TaskManagementApplication() {
    // Constructor logic
}
```

- **Purpose:** Ensures a single instance of the application is shared across the entire program, simplifying state management.
-

2. Factory Pattern

The **Factory Pattern** is used to create different types of tasks (Bug or Feature) dynamically based on the input provided by the admin.

Code Snippet:

```
String taskType = JOptionPane.showInputDialog(this, "Enter Task Type (Bug/Feature):");
Task newTask = TaskFactory.createTask(taskType, task, assignee, deadline, completed);
```

Factory Implementation:

```
public class TaskFactory {  
    public static Task createTask(String type, String task,  
                                  String assignee, String deadline,  
                                  boolean completed) {  
        if ("Bug".equalsIgnoreCase(type)) {  
            return new Bug(task, assignee, deadline, completed);  
        } else if ("Feature".equalsIgnoreCase(type)) {  
            return new Feature(task, assignee, deadline, completed);  
        }  
        return null;  
    }  
}
```

- **Purpose:** Decouples task creation logic, allowing the addition of new task types without modifying existing code.

3. MVC (Model-View-Controller) Pattern

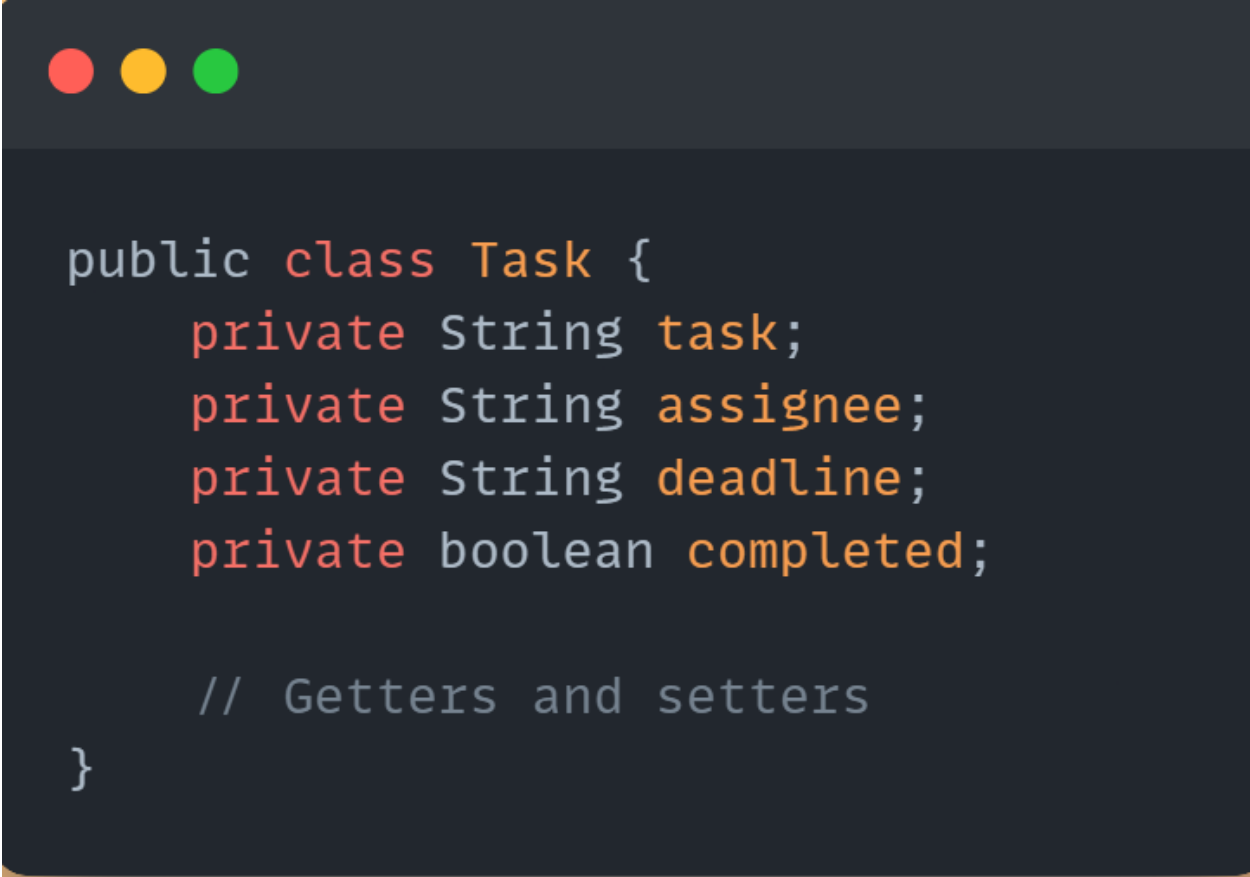
The **MVC Pattern** separates the application into three interconnected components:

- **Model:** Represents the application's data (e.g., Task, Bug, Feature).
- **View:** The graphical interface for user interaction (JTable, JPanel).

- **Controller:** Handles user input and updates the view and model (actionPerformed methods).

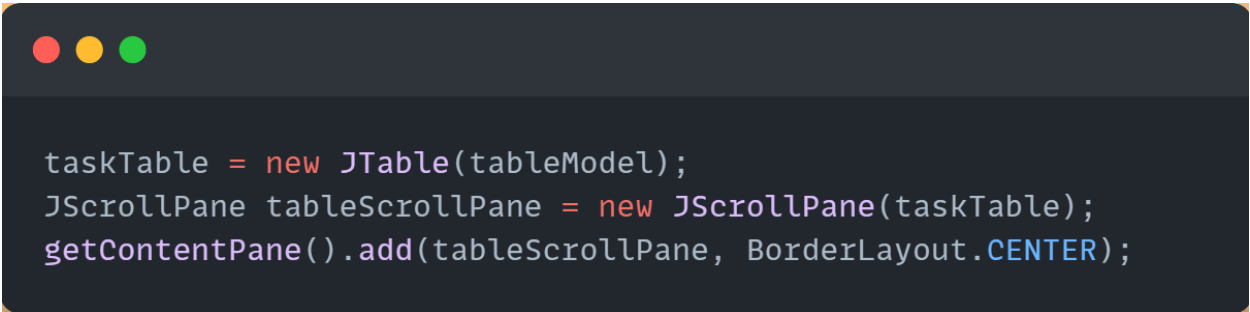
Code Snippet:

- **Model:**



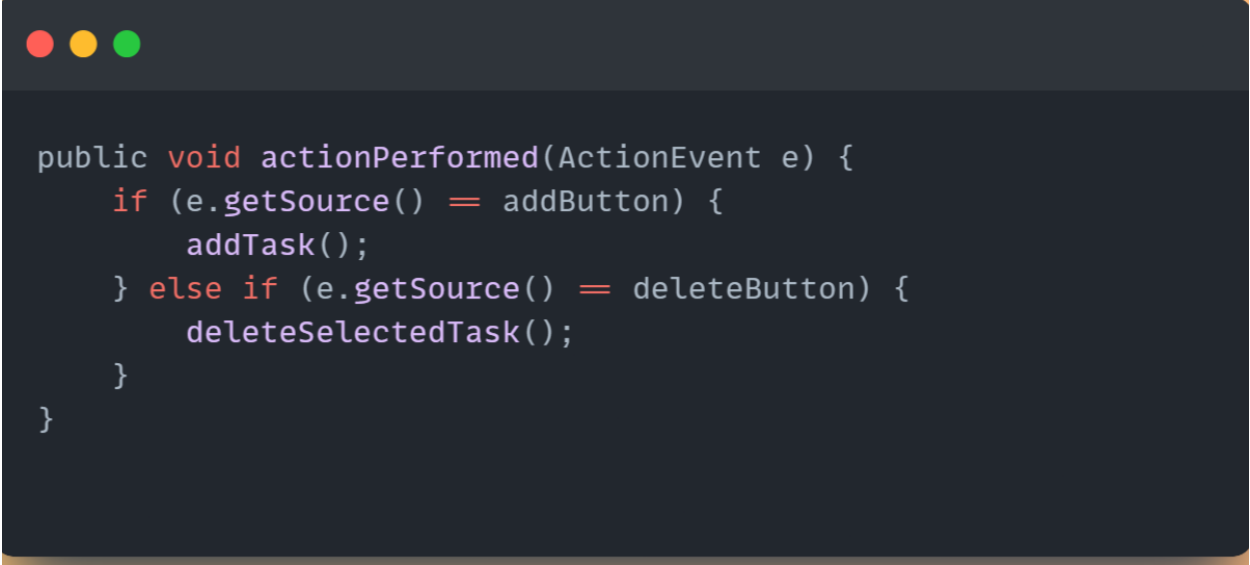
```
public class Task {  
    private String task;  
    private String assignee;  
    private String deadline;  
    private boolean completed;  
  
    // Getters and setters  
}
```

View:



```
taskTable = new JTable(tableModel);  
JScrollPane tableScrollPane = new JScrollPane(taskTable);  
getContentPane().add(tableScrollPane, BorderLayout.CENTER);
```

- **Controller:**



```
public void actionPerformed(ActionEvent e) {  
    if (e.getSource() == addButton) {  
        addTask();  
    } else if (e.getSource() == deleteButton) {  
        deleteSelectedTask();  
    }  
}
```

- **Purpose:** Organizes the code into clear and distinct responsibilities, enhancing testability and maintainability.

4. Strategy Pattern

The **Strategy Pattern** is applied for sorting tasks based on different criteria (task name, assignee, deadline).

Code Snippet:

```

switch (selectedIndex) {
    case 0:
        tasks.sort((t1, t2) → t1.getTask().compareToIgnoreCase(t2.getTask()));
        break;
    case 1:
        tasks.sort((t1, t2) → t1.getAssignee().compareToIgnoreCase(t2.getAssignee()));
        break;
    case 2:
        tasks.sort((t1, t2) → t1.getDeadline().compareTo(t2.getDeadline()));
        break;
}

```

- **Purpose:** Allows different sorting algorithms to be dynamically applied without altering the task list structure.

5. DAO (Data Access Object) Pattern

The **DAO Pattern** is used for database interactions, such as validating user credentials and retrieving role information.

Code Snippet:

```

private String validateLogin(String username, String password) {
    String query = "SELECT role FROM Users WHERE username = ? AND password = ?";
    try (Connection connection = getConnection();
        PreparedStatement statement = connection.prepareStatement(query)) {
        statement.setString(1, username);
        statement.setString(2, password);
        ResultSet resultSet = statement.executeQuery();
        if (resultSet.next()) {
            return resultSet.getString("role");
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
    return null;
}

```

- **Purpose:** Abstracts database logic into a separate method, promoting reusability and reducing code duplication.
-

Features

1. Task Management:

- Add, delete, and sort tasks.
- Mark tasks as complete/incomplete.

2. User Management:

- Switch between admin and employee roles.
- Admin-only sign-up feature for new users.

3. Database Integration:

- Secure user authentication.
 - Persistent data storage.
-

Conclusion

This project demonstrates how **design patterns** enhance the modularity and scalability of applications. By utilizing patterns like Singleton, Factory, MVC, Strategy, and DAO, the application achieves a clear separation of concerns, ensuring it is robust and adaptable to future requirements.