Object oriented design project

Jakub Nowosielski

B00153676

Application overview

Introduction

The Task Management Application is a comprehensive Java-based desktop application designed to help users organize, track, and manage their tasks efficiently. Built with a focus on object-oriented design principles, the application implements four key design patterns to create a flexible, maintainable, and extensible architecture.

Design Pattern Implementation

The application leverages four design patterns to achieve its functionality:

Factory Method Pattern: Creates different task types through a unified interface.

Decorator Pattern: Adds features to tasks dynamically at runtime.

Observer Pattern: Notifies relevant components about task changes.

Singleton Pattern: Provides a global access point to application settings.

These patterns work together to create a robust, flexible application that can be easily extended with new features while maintaining a clean, maintainable codebase.

Functional requirements

my task manager application provides the user with a user friendly graphical interface that allows the user to do the following

Create different types of tasks:

- Standard tasks for one-time activities
- Recurring tasks with customizable recurrence intervals
- Project tasks that can contain subtasks

Manage task properties:

- Set titles, descriptions, and due dates
- Mark tasks as completed
- Add priority levels (LOW, MEDIUM, HIGH, URGENT)
- Flag tasks with custom reasons for special attention

Edit existing tasks:

- Modify basic task properties
- Change priority levels
- Update flag reasons
- Adjust recurrence intervals for recurring tasks

View and filter tasks:

- See all tasks in a centralized list
- Filter tasks due today
- View upcoming tasks (next 7 days)
- Visually distinguish tasks by priority, completion status, and due date

Receive notifications:

- Get updates when tasks are created, modified, or completed
- View a notification history in the application

Persist task data:

- Save tasks to a file for retrieval between application sessions

- Load previously saved tasks when the application starts

Customize application settings:

- Set username
- Choose theme preferences (light/dark)

User Requirements and User Interface

User Requirements

The Task Management Application was designed to meet the needs of users who require an efficient way to organize and track their tasks. The primary user requirements include:

Simplicity: Users need an intuitive interface that allows them to quickly create, view, and manage tasks without a steep learning curve.

Task Organization: Users need to categorize tasks based on type (standard, recurring, project) and add attributes like priority and flags to help with organization.

Visual Differentiation: Users need to easily distinguish between different types of tasks and their statuses (completed, due today, overdue).

Customization: Users need to personalize the application with settings like username and theme preferences.

Data Persistence: Users need their task data to be saved and available when they reopen the application.

Notifications: Users need to be informed about task-related events to stay updated on their progress.

Task Editing: Users need to modify existing tasks when requirements or deadlines change.

User Interface

The application's user interface is designed to be clean, intuitive, and responsive to user actions. The main components include:

Main Window

Task List: The central component displaying all tasks with visual indicators for priority, completion status, and due date.

Toolbar: Contains buttons for common actions (Add, Complete, Delete, Edit, Add Priority, Flag).

Notification Area: Displays recent notifications about task changes.

Menu Bar: Provides access to additional functionality and settings.

Task Creation and Editing

Add Task Dialog: Allows users to create new tasks with all necessary properties.

Edit Task Dialog: Enables modification of existing tasks, including:

Basic properties (title, description, due date)

Priority level

Flag reason

Recurrence interval (for recurring tasks)

Settings and Customization

Settings Dialog: Allows users to customize:

Username

Application theme (light/dark)

Visual Indicators

Priority Colors: Different colors indicate task priority (green for LOW, blue for MEDIUM, orange for HIGH, red for URGENT).

Completion Status: Strikethrough text for completed tasks.

Due Date Indicators: Special formatting for tasks due today or overdue.

Flag Symbol: Visual indicator for flagged tasks.

The user interface follows these design principles:

Consistency: Similar actions are performed in similar ways throughout the application.

Feedback: Users receive clear feedback for all actions through notifications and visual changes.

Simplicity: The interface focuses on essential functionality without unnecessary complexity.

Accessibility: Controls are clearly labeled and easy to understand.

Design pattern explanations

Factory Method Pattern

What is the Factory Method Pattern?

The Factory Method pattern is a creational design pattern that provides an interface for creating objects in a superclass, but allows subclasses to alter the type of objects that will be created. It encapsulates object creation logic, making the system more flexible and extensible.

How We Applied It

In our Task Management Application, we implemented the Factory Method pattern through the TaskFactory class, which creates different types of tasks (Standard, Recurring, Project) through a unified interface:

Why We Chose This Pattern

We chose the Factory Method pattern for several reasons:

Encapsulation of Creation Logic: It centralizes the task creation code, making it easier to maintain and modify.

Type Flexibility: It allows the application to create different types of tasks without knowing the specific classes.

Extensibility: New task types can be added by extending the factory without modifying client code.

Consistency: It ensures that tasks are created with all necessary properties and in a consistent manner.

Compared to alternatives like direct instantiation or the Abstract Factory pattern, the Factory Method provides the right balance of simplicity and flexibility for our application's needs. The Abstract Factory would be overly complex for our requirements, while direct instantiation would scatter creation logic throughout the codebase.

Decorator Pattern

What is the Decorator Pattern? The Decorator pattern is a structural design pattern that allows behavior to be added to individual objects, dynamically, without affecting the behavior of other objects from the same class. It's a flexible alternative to subclassing for extending functionality. How We Applied It In our application, we implemented the Decorator pattern to add features like priority and flags to tasks:

Why We Chose This Pattern

We chose the Decorator pattern for several reasons:

Dynamic Feature Addition: It allows features like priority and flags to be added to tasks at runtime.

Composition Over Inheritance: It uses composition instead of inheritance, avoiding class explosion.

Open/Closed Principle: It follows the open/closed principle by allowing new features to be added without modifying existing code.

Combinatorial Flexibility: It allows features to be combined in any order (e.g., a task can have both priority and flag).

Compared to alternatives like inheritance or the Strategy pattern, the Decorator provides more flexibility for combining features. Inheritance would lead to a combinatorial explosion of classes (StandardTask, PriorityTask, FlaggedTask, PriorityFlaggedTask, etc.), while the Strategy pattern wouldn't allow for the same level of feature composition.

Observer Pattern

What is the Observer Pattern?

The Observer pattern is a behavioral design pattern where an object, called the subject, maintains a list of its dependents, called observers, and notifies them automatically of any state changes. It's commonly used to implement distributed event handling systems.

How We Applied It

In our application, we implemented the Observer pattern to notify components about task changes:

Why We Chose This Pattern

We chose the Observer pattern for several reasons:

Loose Coupling: It decouples the notification system from the task management system.

Real-time Updates: It ensures that the UI is always updated when tasks change.

Extensibility: New observers can be added without modifying the subject.

Selective Notification: Task-specific observers allow for targeted notifications.

Compared to alternatives like direct method calls or the Mediator pattern, the Observer provides better decoupling and scalability. Direct method calls would create tight coupling between components, while the Mediator would centralize too much control in a single class.

Singleton Pattern

What is the Singleton Pattern?

The Singleton pattern is a creational design pattern that ensures a class has only one instance and provides a global point of access to it. It's useful when exactly one object is needed to coordinate actions across the system.

How We Applied It

In our application, we implemented the Singleton pattern for the AppSettings class:

Why We Chose This Pattern

We chose the Singleton pattern for several reasons:

Global Access: It provides a single point of access to application settings.

Consistency: It ensures that all components use the same settings.

Resource Management: It prevents multiple instances from accessing the settings file simultaneously.

Lazy Initialization: It creates the instance only when needed.

Compared to alternatives like static methods or a global variable, the Singleton provides better encapsulation and control over instantiation. Static methods would not allow for proper initialization and cleanup, while a global variable would be less secure and harder to manage.

UML Diagrams Description

1. Factory Method Pattern UML Diagram

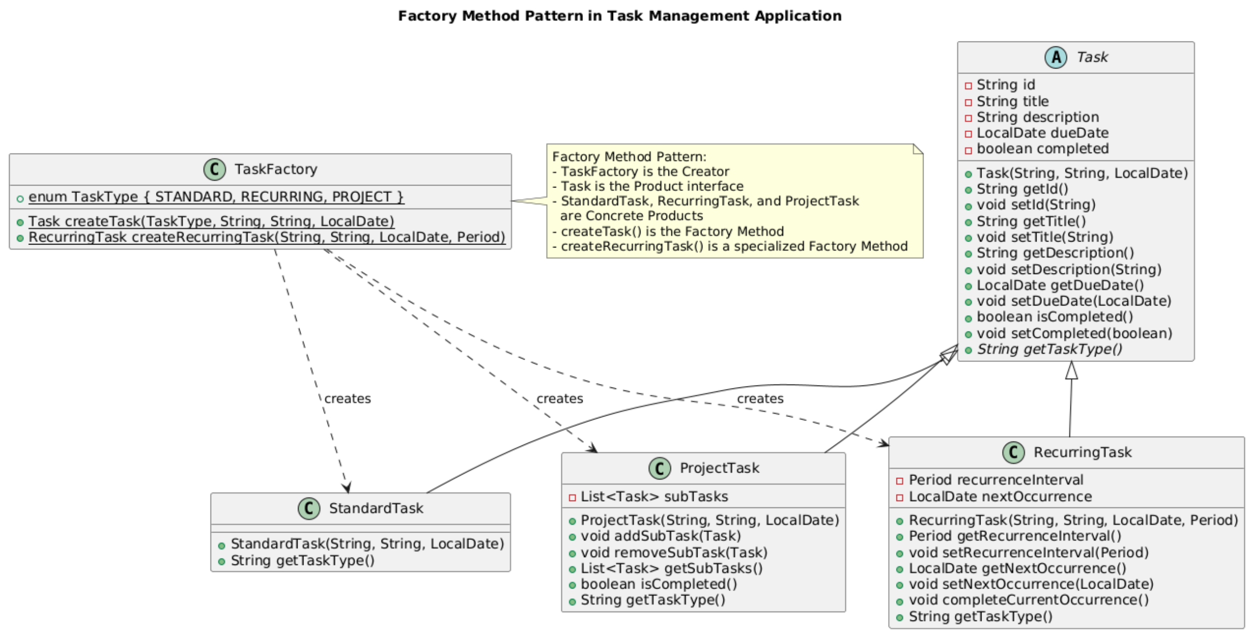
The Factory Method UML diagram illustrates the relationship between the TaskFactory class and the various task types it creates. Key elements include:

TaskFactory: The creator class with static factory methods for creating different task types.

Task: The abstract product that defines the interface for all task types.

StandardTask, RecurringTask, ProjectTask: The concrete products created by the factory.

The diagram shows how the factory creates different task types through a unified interface, encapsulating the creation logic and allowing for easy extension with new task types.


Factory Method Pattern in Task Management Application
Task
String id
String title
String description
LocalDate dueDate
boolean completed
Task(String, String, LocalDate)
String getId()
void setId(String)
String getTitle()
void setTitle(String)
String getDescription()
void setDescription(String)
LocalDate getDueDate()
void setDueDate(LocalDate)
boolean isCompleted()
void setCompleted(boolean)
String getTaskType()
TaskFactory
enum TaskType { STANDARD, RECURRING, PROJECT }
Task createTask(TaskType, String, String, LocalDate)
RecurringTask createRecurringTask(String, String, LocalDate, Period)
Factory Method Pattern:
- TaskFactory is the Creator
- Task is the Product interface
- StandardTask, RecurringTask, and ProjectTask are Concrete Products
- createTask() is the Factory Method
- createRecurringTask() is a specialized Factory Method
creates
creates
creates
StandardTask
StandardTask(String, String, LocalDate)
String getTaskType()
ProjectTask
List<Task> subTasks
ProjectTask(String, String, LocalDate)
void addSubTask(Task)
void removeSubTask(Task)
List<Task> getSubTasks()
boolean isCompleted()
String getTaskType()
RecurringTask
Period recurrenceInterval
LocalDate nextOccurrence
RecurringTask(String, String, LocalDate, Period)
Period getRecurrenceInterval()
void setRecurrenceInterval(Period)
LocalDate getNextOccurrence()
void setNextOccurrence(LocalDate)
void completeCurrentOccurrence()
String getTaskType()

2. Decorator Pattern UML Diagram

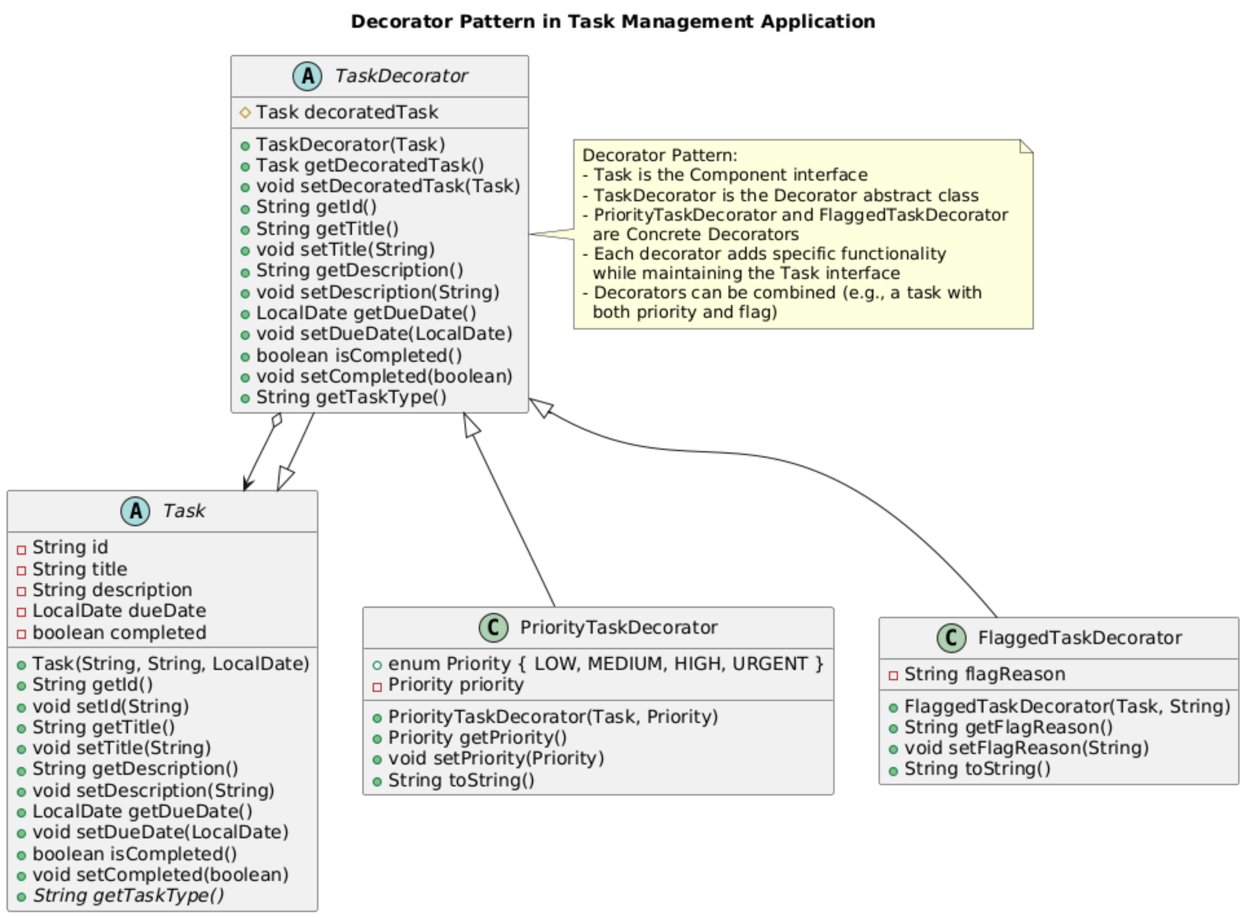
The Decorator Pattern UML diagram shows how task functionality can be extended dynamically. Key elements include:

Task: The component interface that defines the basic task behavior.

TaskDecorator: The abstract decorator class that maintains a reference to a decorated task.

PriorityTaskDecorator, FlaggedTaskDecorator: Concrete decorators that add specific behaviors.

The diagram illustrates the composition relationship between decorators and tasks, showing how decorators can be stacked to add multiple features to a task.



Decorator Pattern in Task Management Application
A TaskDecorator
◇ Task decoratedTask
● TaskDecorator(Task)
● Task getDecoratedTask()
● void setDecoratedTask(Task)
● String getId()
● String getTitle()
● void setTitle(String)
● String getDescription()
● void setDescription(String)
● LocalDate getDueDate()
● void setDueDate(LocalDate)
● boolean isCompleted()
● void setCompleted(boolean)
● String getTaskType()
Decorator Pattern:
- Task is the Component interface
- TaskDecorator is the Decorator abstract class
- PriorityTaskDecorator and FlaggedTaskDecorator are Concrete Decorators
- Each decorator adds specific functionality while maintaining the Task interface
- Decorators can be combined (e.g., a task with both priority and flag)
A Task
□ String id
□ String title
□ String description
□ LocalDate dueDate
□ boolean completed
● Task(String, String, LocalDate)
● String getId()
● void setId(String)
● String getTitle()
● void setTitle(String)
● String getDescription()
● void setDescription(String)
● LocalDate getDueDate()
● void setDueDate(LocalDate)
● boolean isCompleted()
● void setCompleted(boolean)
● String getTaskType()
C PriorityTaskDecorator
○ enum Priority { LOW, MEDIUM, HIGH, URGENT }
□ Priority priority
● PriorityTaskDecorator(Task, Priority)
● Priority getPriority()
● void setPriority(Priority)
● String toString()
C FlaggedTaskDecorator
□ String flagReason
● FlaggedTaskDecorator(Task, String)
● String getFlagReason()
● void setFlagReason(String)
● String toString()

3. Observer Pattern UML Diagram

The Observer Pattern UML diagram demonstrates the notification system in the application. Key elements include:

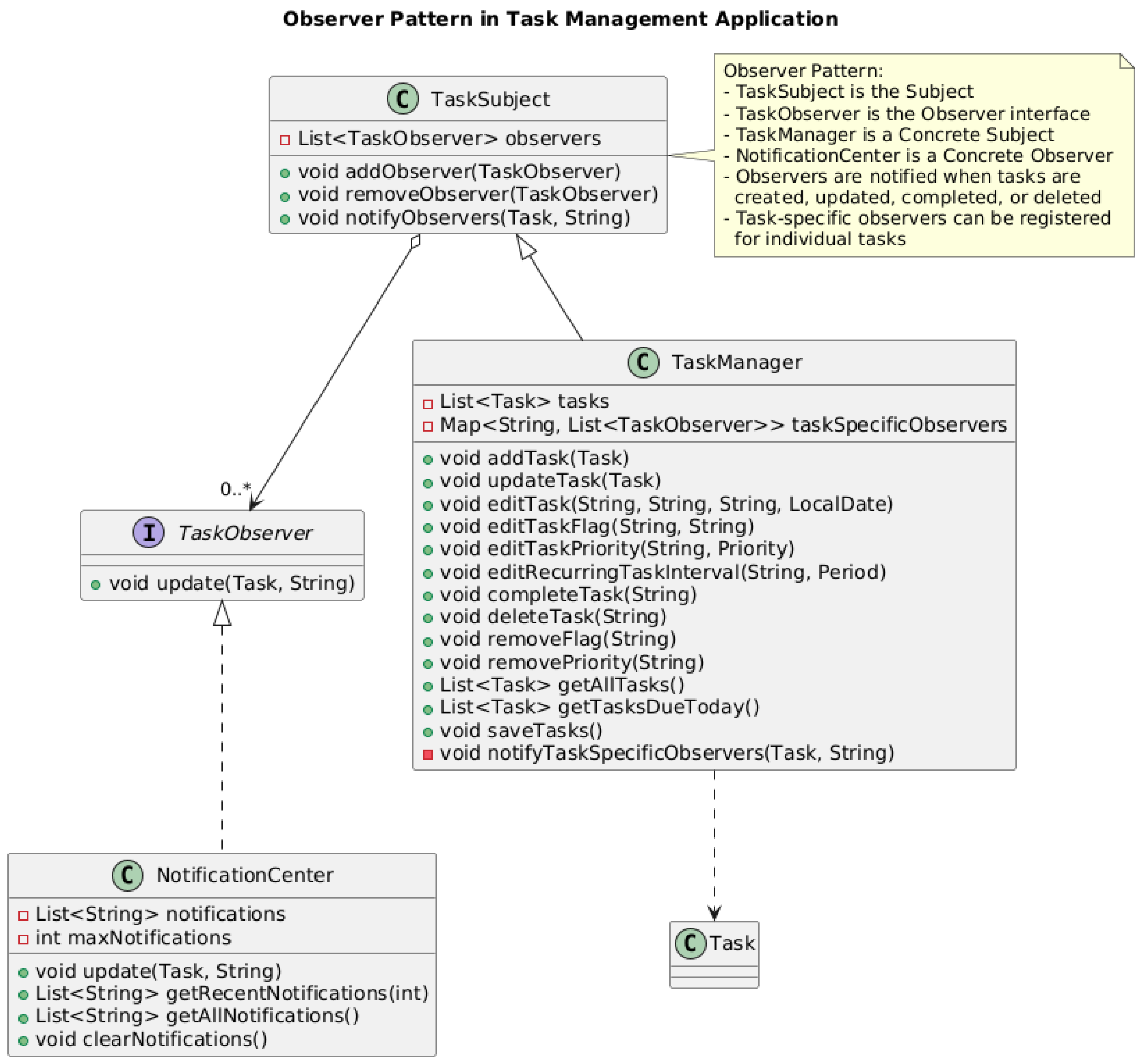
TaskObserver: The observer interface that defines the update method.

TaskSubject: The subject class that maintains a list of observers and notifies them of changes.

TaskManager: The concrete subject that extends TaskSubject and manages tasks.

NotificationCenter: The concrete observer that receives and processes notifications.

The diagram shows how observers are notified when tasks are created, updated, completed, or deleted, enabling loose coupling between components.


Observer Pattern in Task Management Application
TaskSubject
- List<TaskObserver> observers
+ void addObserver(TaskObserver)
+ void removeObserver(TaskObserver)
+ void notifyObservers(Task, String)
Observer Pattern:
- TaskSubject is the Subject
- TaskObserver is the Observer interface
- TaskManager is a Concrete Subject
- NotificationCenter is a Concrete Observer
- Observers are notified when tasks are created, updated, completed, or deleted
- Task-specific observers can be registered for individual tasks
TaskManager
- List<Task> tasks
- Map<String, List<TaskObserver>> taskSpecificObservers
+ void addTask(Task)
+ void updateTask(Task)
+ void editTask(String, String, String, LocalDate)
+ void editTaskFlag(String, String)
+ void editTaskPriority(String, Priority)
+ void editRecurringTaskInterval(String, Period)
+ void completeTask(String)
+ void deleteTask(String)
+ void removeFlag(String)
+ void removePriority(String)
+ List<Task> getAllTasks()
+ List<Task> getTasksDueToday()
+ void saveTasks()
- void notifyTaskSpecificObservers(Task, String)
0..*
TaskObserver
+ void update(Task, String)
NotificationCenter
- List<String> notifications
- int maxNotifications
+ void update(Task, String)
+ List<String> getRecentNotifications(int)
+ List<String> getAllNotifications()
+ void clearNotifications()
Task

4. Singleton Pattern UML Diagram

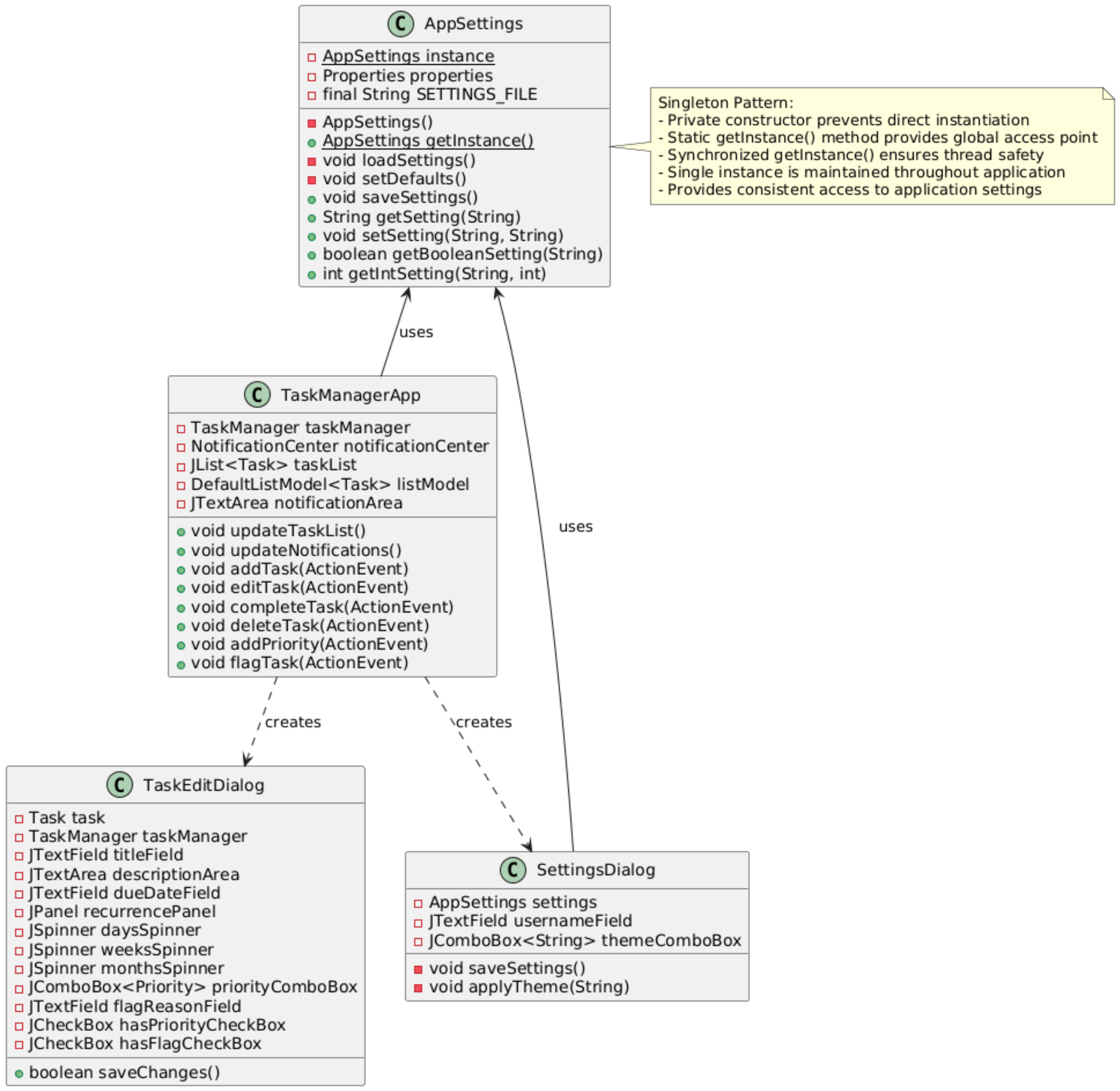
The Singleton Pattern UML diagram illustrates the global access to application settings. Key elements include:

AppSettings: The singleton class with a private constructor and static getInstance method.

TaskManagerApp, SettingsDialog: Classes that use the AppSettings singleton.

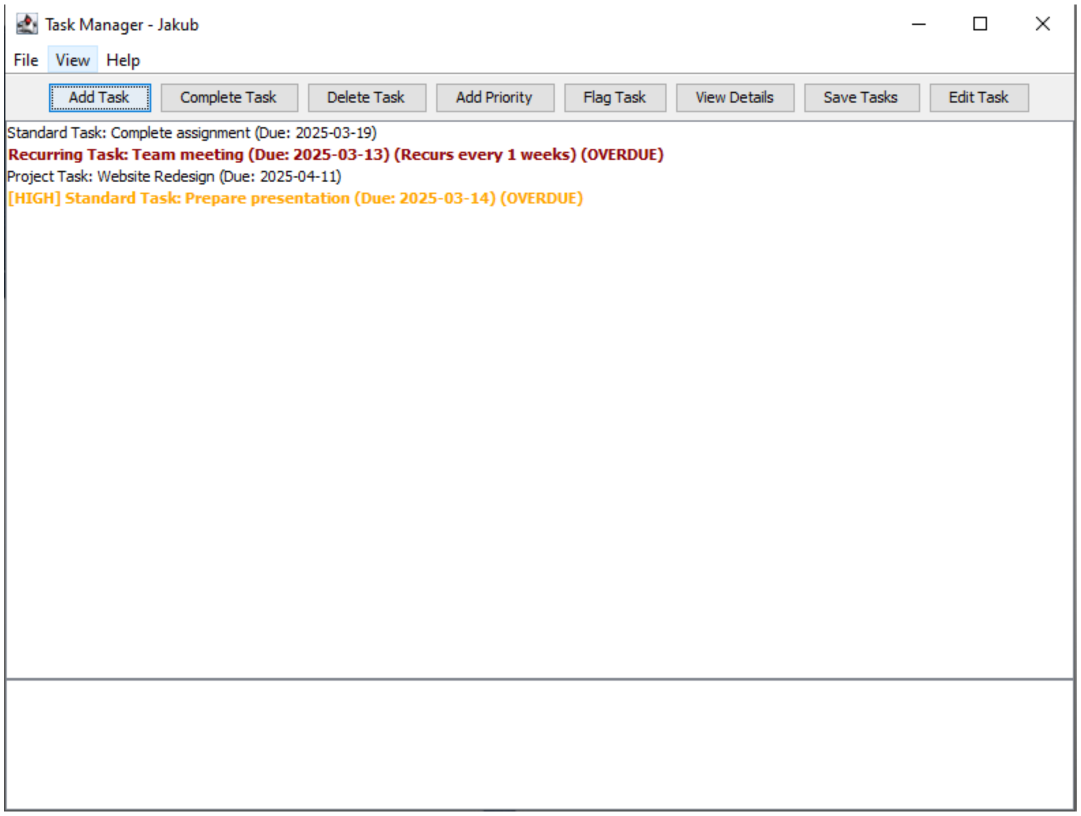
The diagram shows how the singleton ensures that only one instance of AppSettings exists and provides a global access point to it, maintaining consistency across the application.

Singleton Pattern in Task Management Application


AppSettings
- AppSettings instance
- Properties properties
- final String SETTINGS_FILE
- AppSettings()
+ AppSettings getInstance()
- void loadSettings()
- void setDefaults()
+ void saveSettings()
+ String getSetting(String)
+ void setSetting(String, String)
+ boolean getBooleanSetting(String)
+ int getIntSetting(String, int)
Singleton Pattern:
- Private constructor prevents direct instantiation
- Static getInstance() method provides global access point
- Synchronized getInstance() ensures thread safety
- Single instance is maintained throughout application
- Provides consistent access to application settings
uses
uses
TaskManagerApp
- TaskManager taskManager
- NotificationCenter notificationCenter
- JList<Task> taskList
- DefaultListModel<Task> listModel
- JTextArea notificationArea
+ void updateTaskList()
+ void updateNotifications()
+ void addTask(ActionEvent)
+ void editTask(ActionEvent)
+ void completeTask(ActionEvent)
+ void deleteTask(ActionEvent)
+ void addPriority(ActionEvent)
+ void flagTask(ActionEvent)
creates
creates
TaskEditDialog
- Task task
- TaskManager taskManager
- JTextField titleField
- JTextArea descriptionArea
- JTextField dueDateField
- JPanel recurrencePanel
- JSpinner daysSpinner
- JSpinner weeksSpinner
- JSpinner monthsSpinner
- JComboBox<Priority> priorityComboBox
- JTextField flagReasonField
- JCheckBox hasPriorityCheckBox
- JCheckBox hasFlagCheckBox
+ boolean saveChanges()
SettingsDialog
- AppSettings settings
- JTextField usernameField
- JComboBox<String> themeComboBox
- void saveSettings()
- void applyTheme(String)

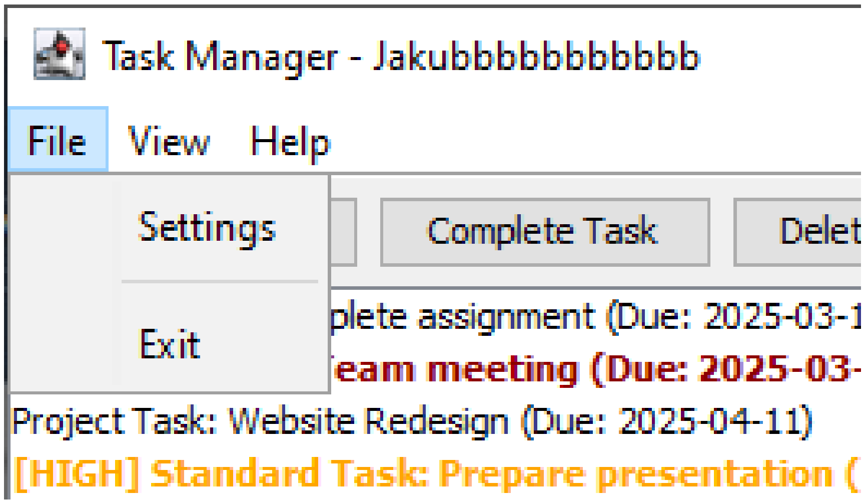
Screenshots of application during execution

Gui view after opening the task manager application



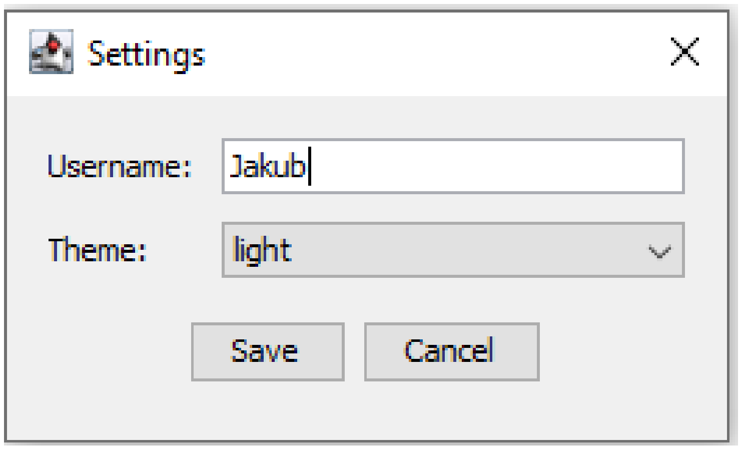
```
Standard Task: Complete assignment (Due: 2025-03-19)
Recurring Task: Team meeting (Due: 2025-03-13) (Recurs every 1 weeks) (OVERDUE)
Project Task: Website Redesign (Due: 2025-04-11)
[HIGH] Standard Task: Prepare presentation (Due: 2025-03-14) (OVERDUE)
```

File tab



```
plete assignment (Due: 2025-03-1
eam meeting (Due: 2025-03-
Project Task: Website Redesign (Due: 2025-04-11)
[HIGH] Standard Task: Prepare presentation (
```

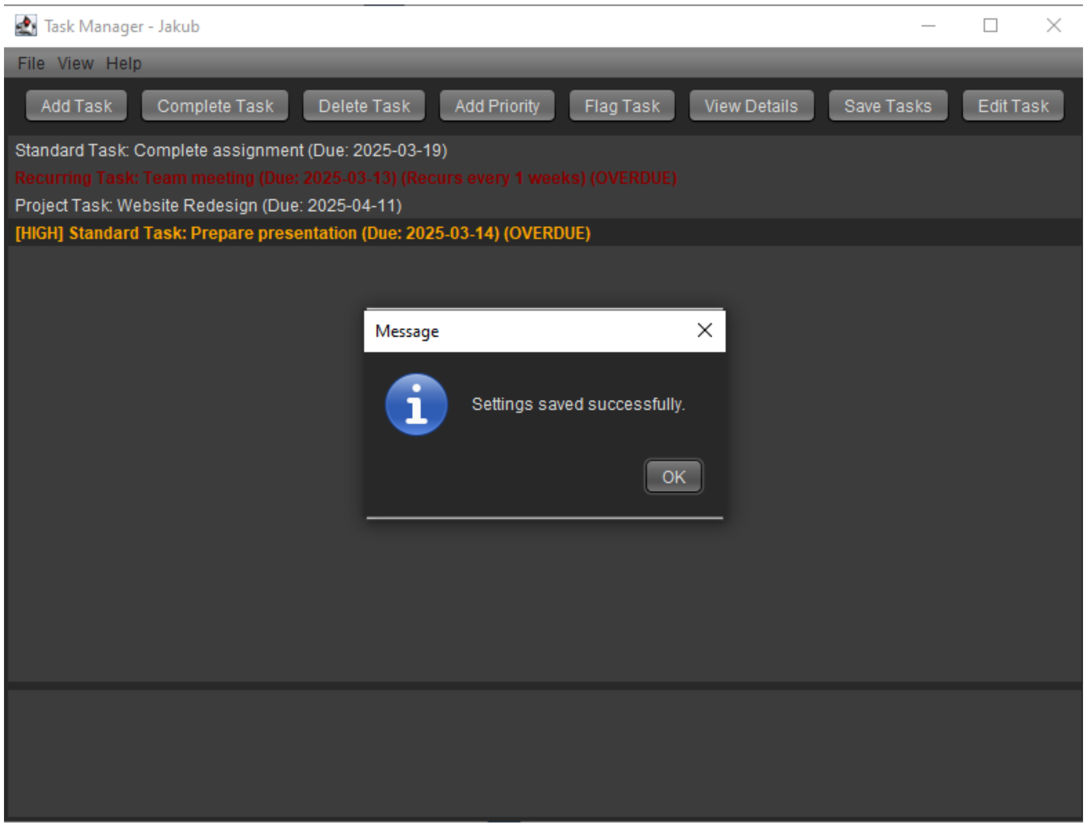
Settings tab



Username: Jakub

Theme: light

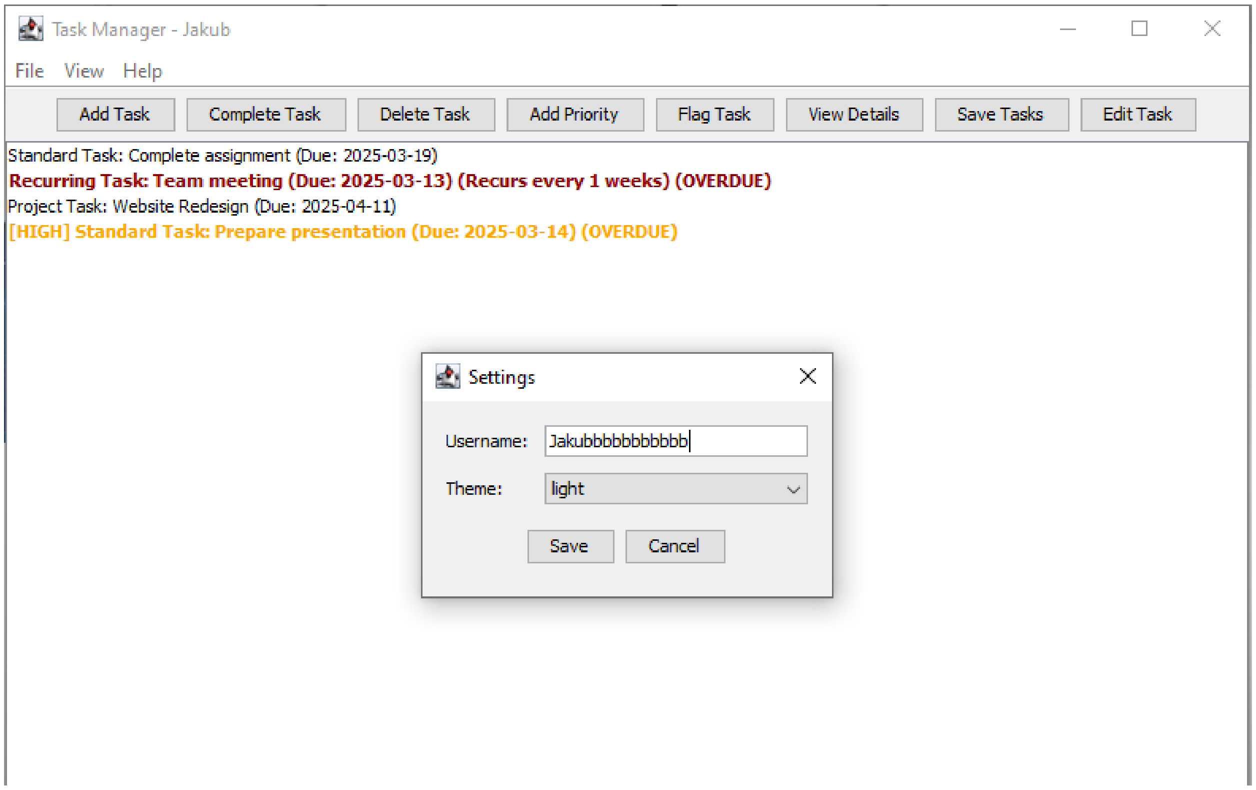
After changing to dark mode



Standard Task: Complete assignment (Due: 2025-03-19)
Recurring Task: Team meeting (Due: 2025-03-13) (Recurs every 1 weeks) (OVERDUE)
Project Task: Website Redesign (Due: 2025-04-11)
[HIGH] Standard Task: Prepare presentation (Due: 2025-03-14) (OVERDUE)

Settings saved successfully.

Username change

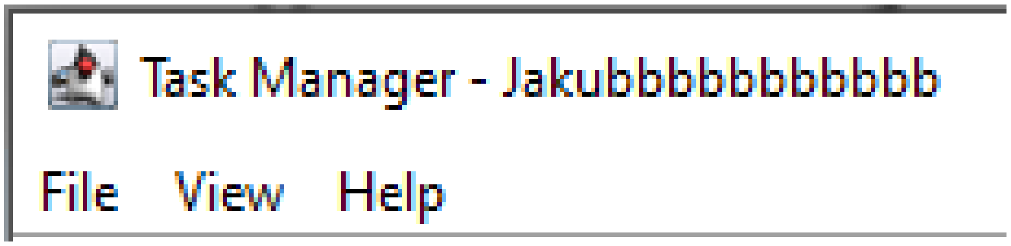


```
Standard Task: Complete assignment (Due: 2025-03-19)
Recurring Task: Team meeting (Due: 2025-03-13) (Recurs every 1 weeks) (OVERDUE)
Project Task: Website Redesign (Due: 2025-04-11)
[HIGH] Standard Task: Prepare presentation (Due: 2025-03-14) (OVERDUE)
```

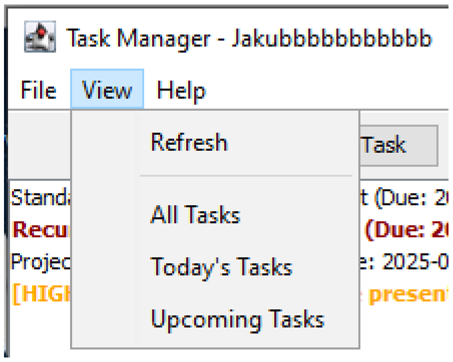
Username: Jakubbbbbbbbbbbb

Theme: light

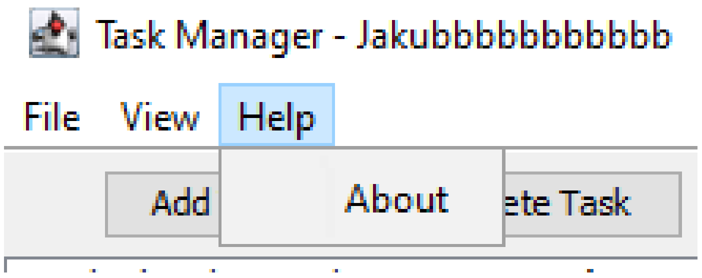
After changing username



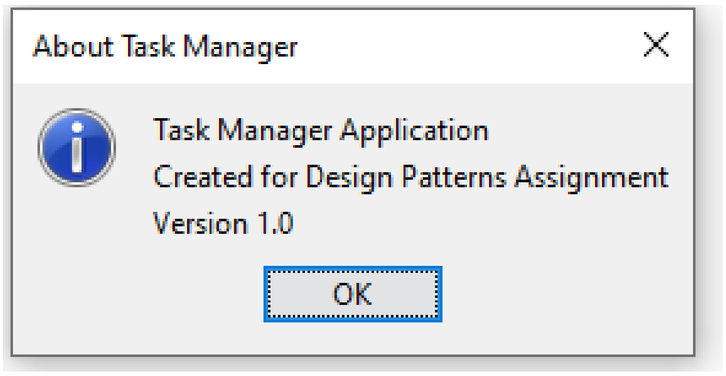
View tab



Help tab



About tab

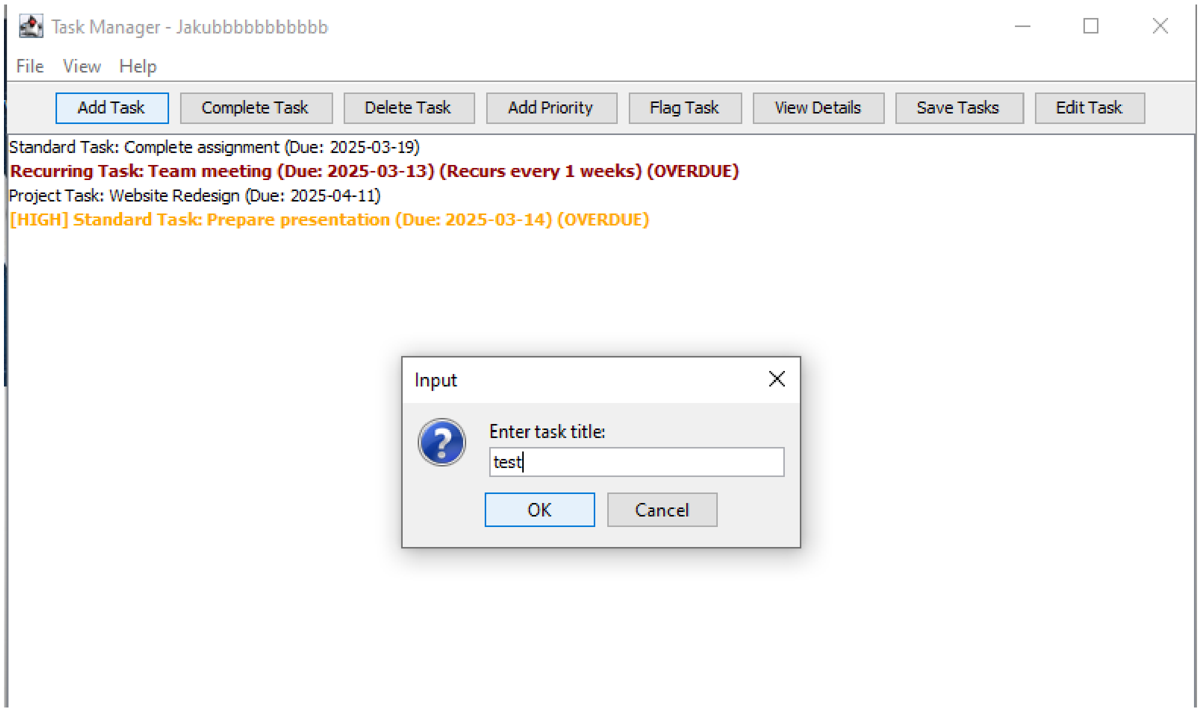

About Task Manager

Task Manager Application

Created for Design Patterns Assignment

Version 1.0

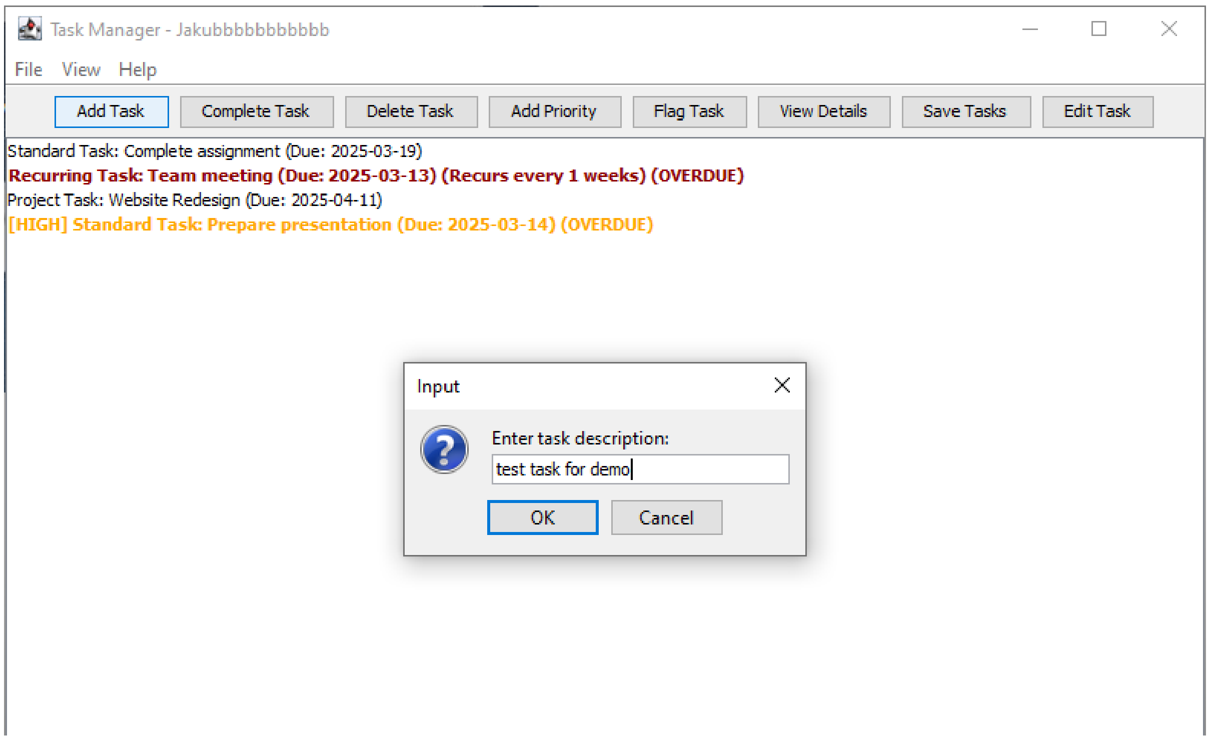
Add task



```
Standard Task: Complete assignment (Due: 2025-03-19)
Recurring Task: Team meeting (Due: 2025-03-13) (Recurs every 1 weeks) (OVERDUE)
Project Task: Website Redesign (Due: 2025-04-11)
[HIGH] Standard Task: Prepare presentation (Due: 2025-03-14) (OVERDUE)
```

Enter task title: test

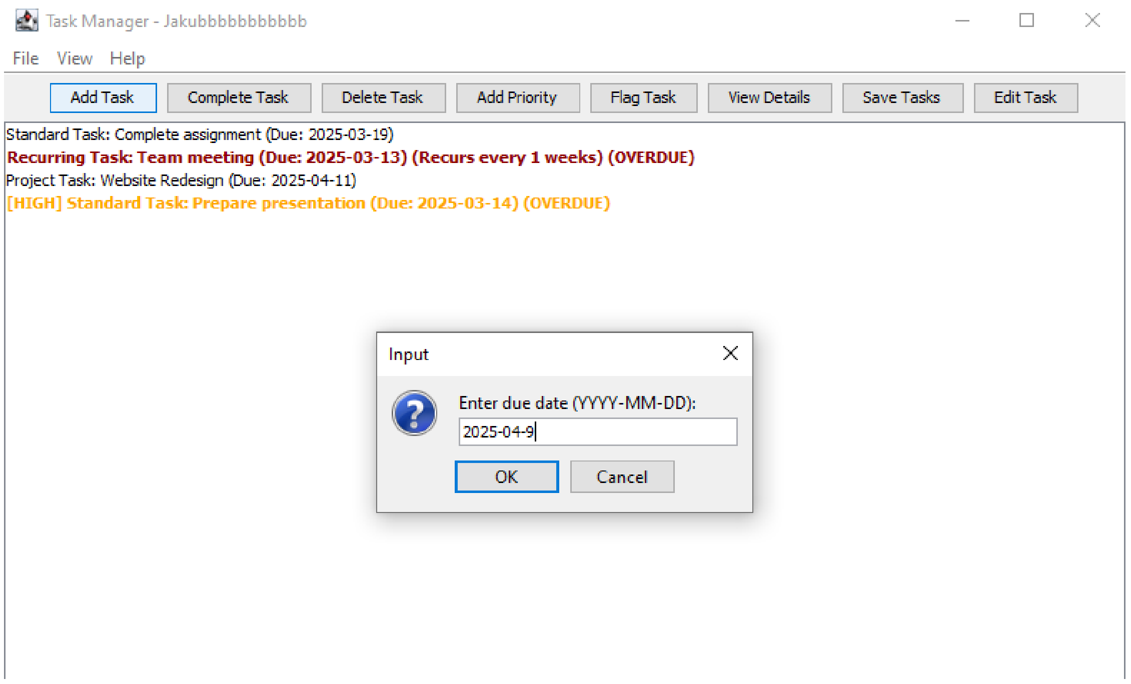
Add description to task



```
Standard Task: Complete assignment (Due: 2025-03-19)
Recurring Task: Team meeting (Due: 2025-03-13) (Recurs every 1 weeks) (OVERDUE)
Project Task: Website Redesign (Due: 2025-04-11)
[HIGH] Standard Task: Prepare presentation (Due: 2025-03-14) (OVERDUE)
```

Enter task description: test task for demo

Add date to a task

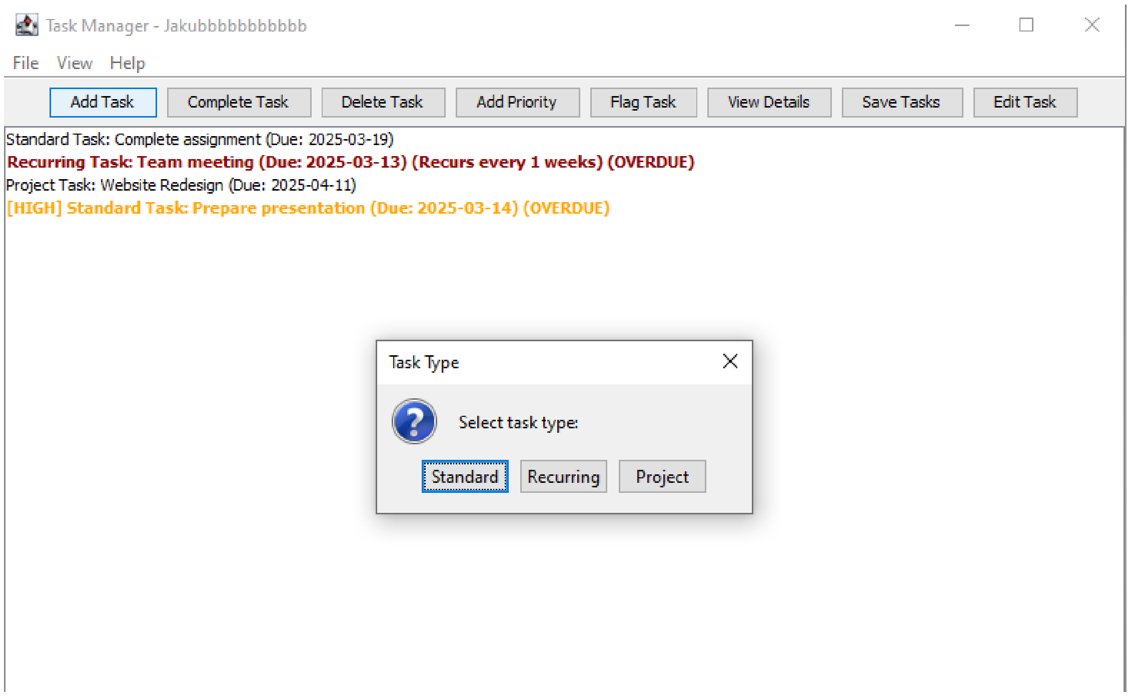

Task Manager - Jakubbbbbbbbbbb
File View Help
Add Task Complete Task Delete Task Add Priority Flag Task View Details Save Tasks Edit Task

Standard Task: Complete assignment (Due: 2025-03-19)
Recurring Task: Team meeting (Due: 2025-03-13) (Recurs every 1 weeks) (OVERDUE)
Project Task: Website Redesign (Due: 2025-04-11)
[HIGH] Standard Task: Prepare presentation (Due: 2025-03-14) (OVERDUE)

Input

Enter due date (YYYY-MM-DD):
2025-04-9

Add type to task


Task Manager - Jakubbbbbbbbbbb
File View Help
Add Task Complete Task Delete Task Add Priority Flag Task View Details Save Tasks Edit Task

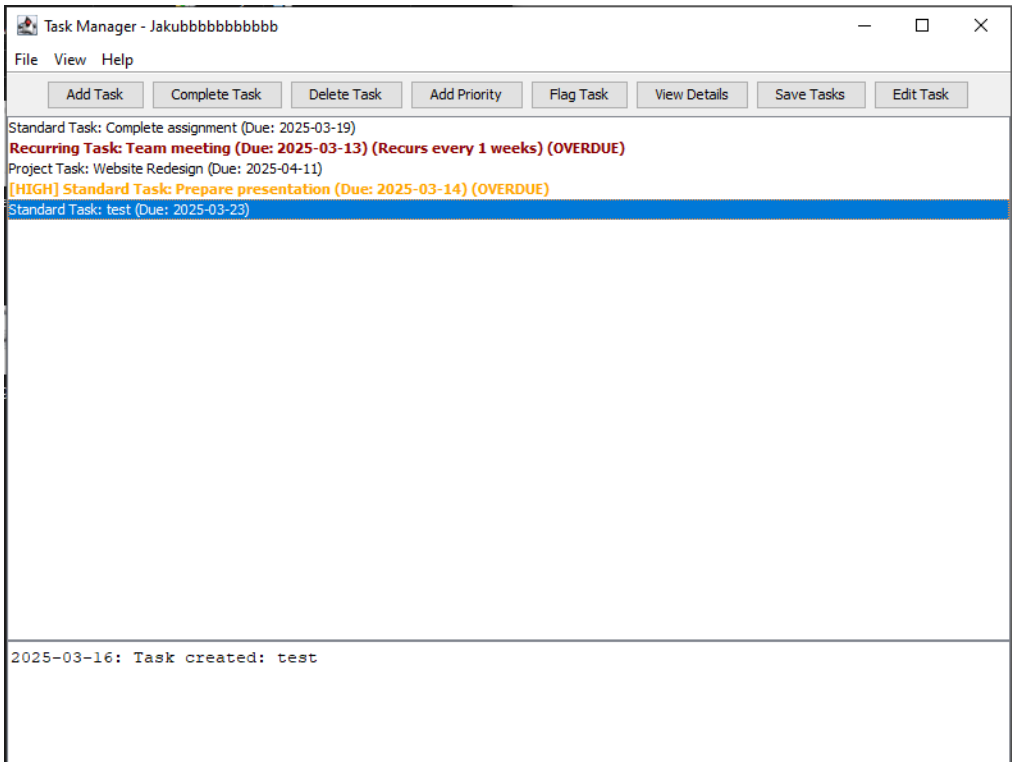
Standard Task: Complete assignment (Due: 2025-03-19)
Recurring Task: Team meeting (Due: 2025-03-13) (Recurs every 1 weeks) (OVERDUE)
Project Task: Website Redesign (Due: 2025-04-11)
[HIGH] Standard Task: Prepare presentation (Due: 2025-03-14) (OVERDUE)

Task Type

Select task type:

Standard Recurring Project

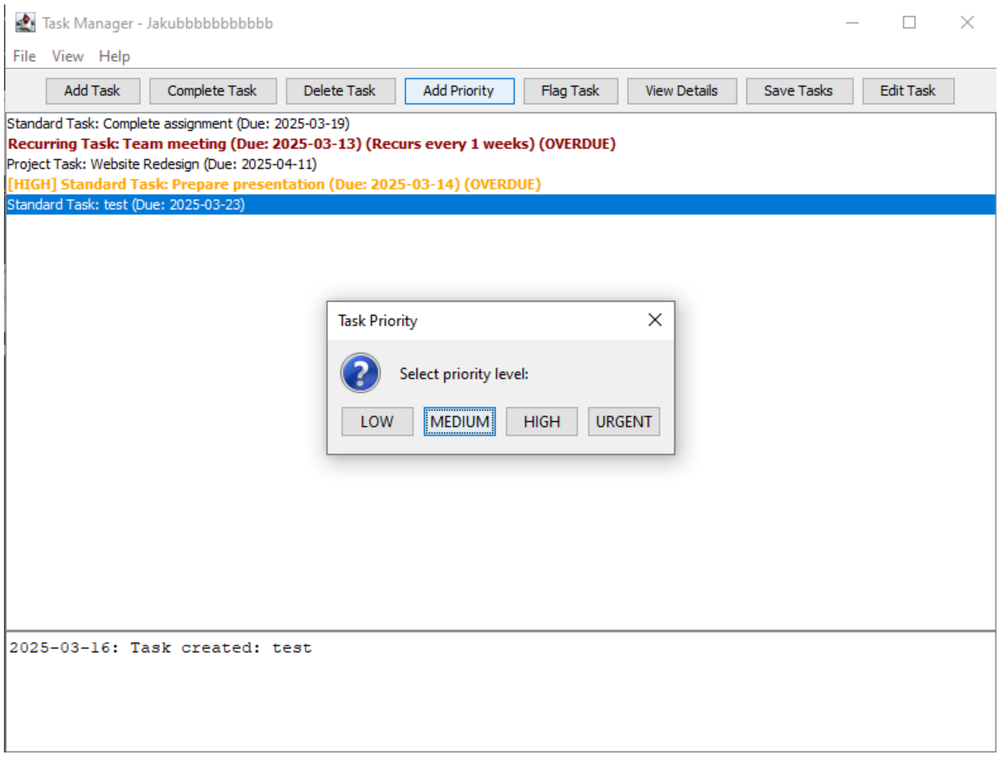
Adding a task completed



```
Standard Task: Complete assignment (Due: 2025-03-19)
Recurring Task: Team meeting (Due: 2025-03-13) (Recurs every 1 weeks) (OVERDUE)
Project Task: Website Redesign (Due: 2025-04-11)
[HIGH] Standard Task: Prepare presentation (Due: 2025-03-14) (OVERDUE)
Standard Task: test (Due: 2025-03-23)
```

```
2025-03-16: Task created: test
```

Adding priority to task

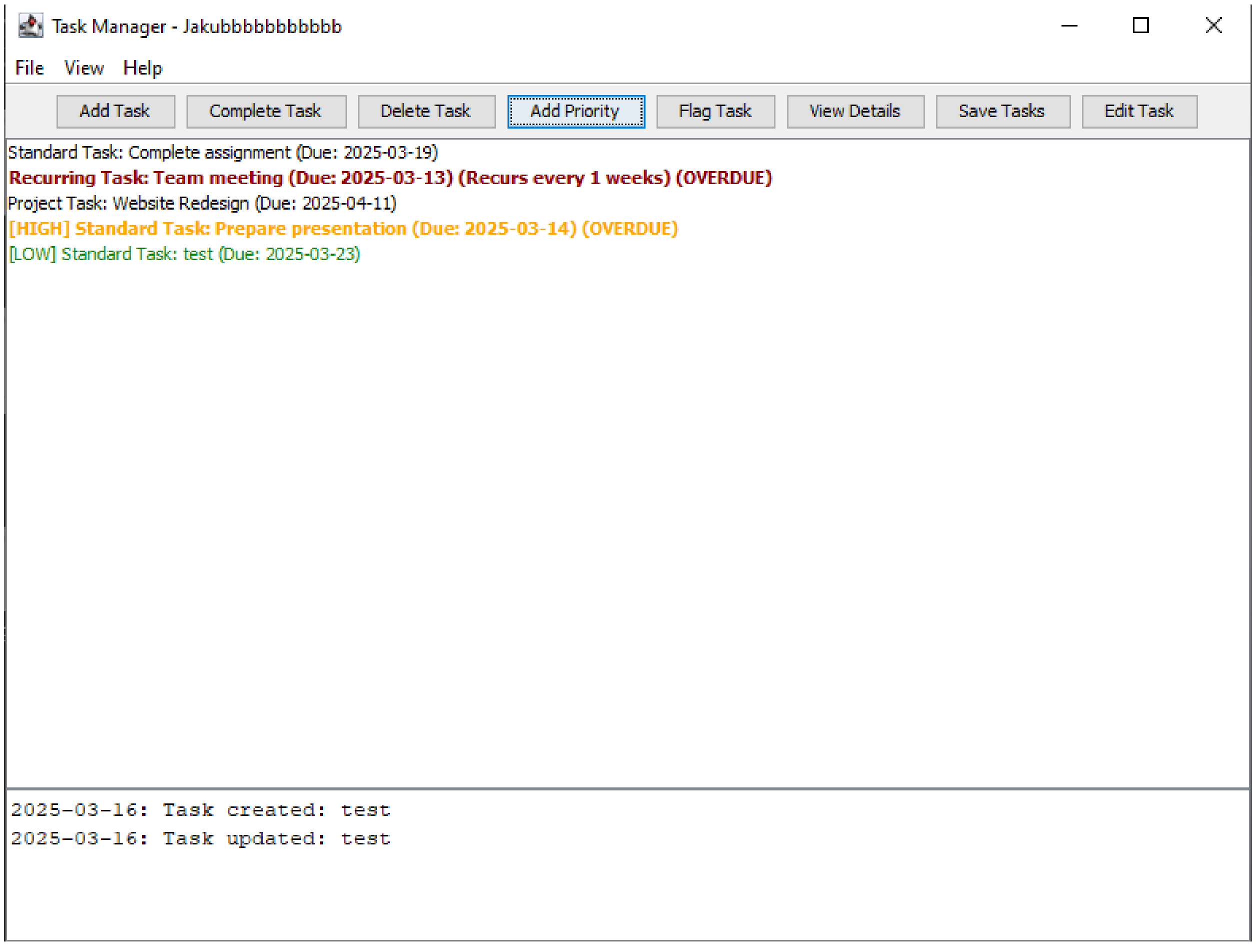


```
Standard Task: Complete assignment (Due: 2025-03-19)
Recurring Task: Team meeting (Due: 2025-03-13) (Recurs every 1 weeks) (OVERDUE)
Project Task: Website Redesign (Due: 2025-04-11)
[HIGH] Standard Task: Prepare presentation (Due: 2025-03-14) (OVERDUE)
Standard Task: test (Due: 2025-03-23)
```

Select priority level:

```
2025-03-16: Task created: test
```

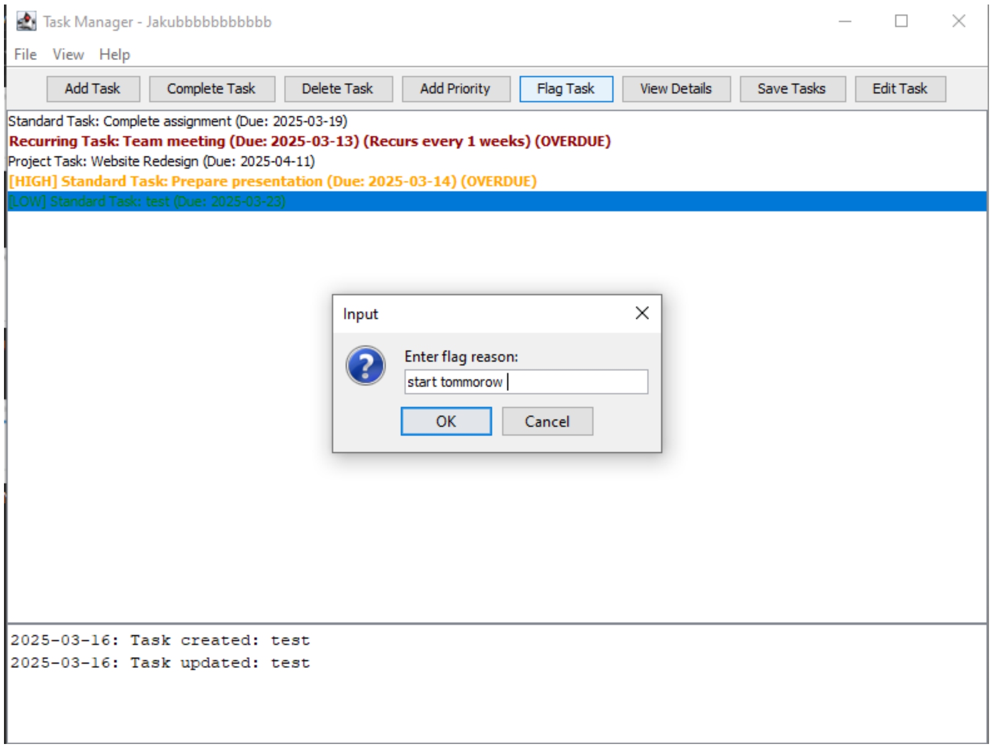
Priority decorator added



```
Standard Task: Complete assignment (Due: 2025-03-19)
Recurring Task: Team meeting (Due: 2025-03-13) (Recurs every 1 weeks) (OVERDUE)
Project Task: Website Redesign (Due: 2025-04-11)
[HIGH] Standard Task: Prepare presentation (Due: 2025-03-14) (OVERDUE)
[LOW] Standard Task: test (Due: 2025-03-23)
```

```
2025-03-16: Task created: test
2025-03-16: Task updated: test
```

Adding flag to task

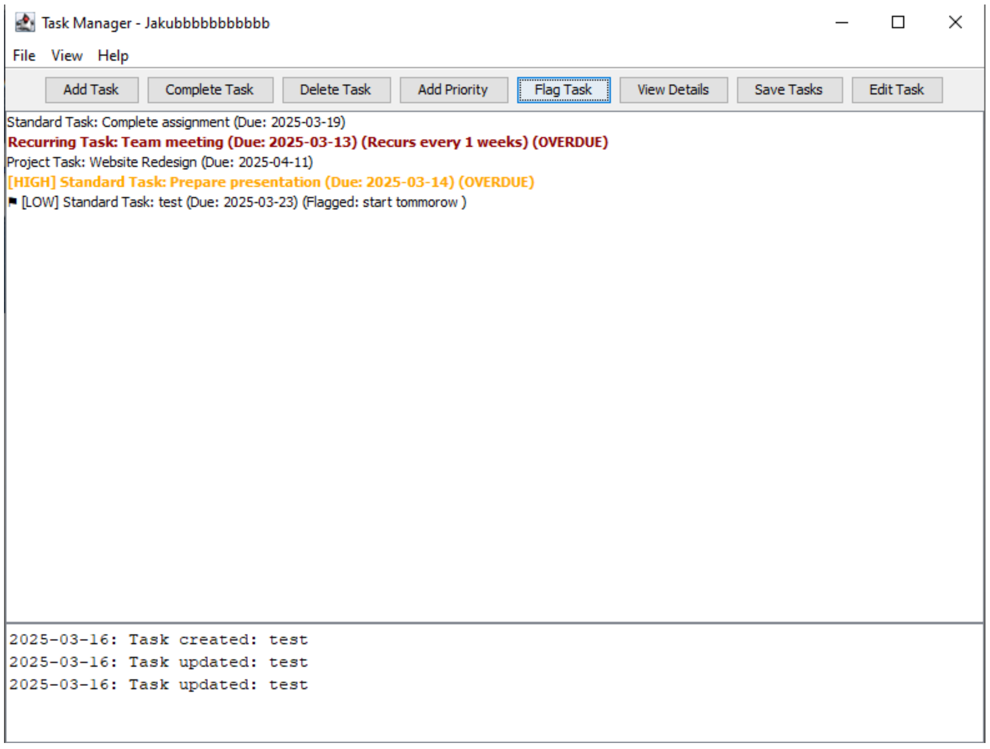


Standard Task: Complete assignment (Due: 2025-03-19)
Recurring Task: Team meeting (Due: 2025-03-13) (Recurs every 1 weeks) (OVERDUE)
Project Task: Website Redesign (Due: 2025-04-11)
[HIGH] Standard Task: Prepare presentation (Due: 2025-03-14) (OVERDUE)
[LOW] Standard Task: test (Due: 2025-03-23)

Enter flag reason:
start tommorow

```
2025-03-16: Task created: test
2025-03-16: Task updated: test
```

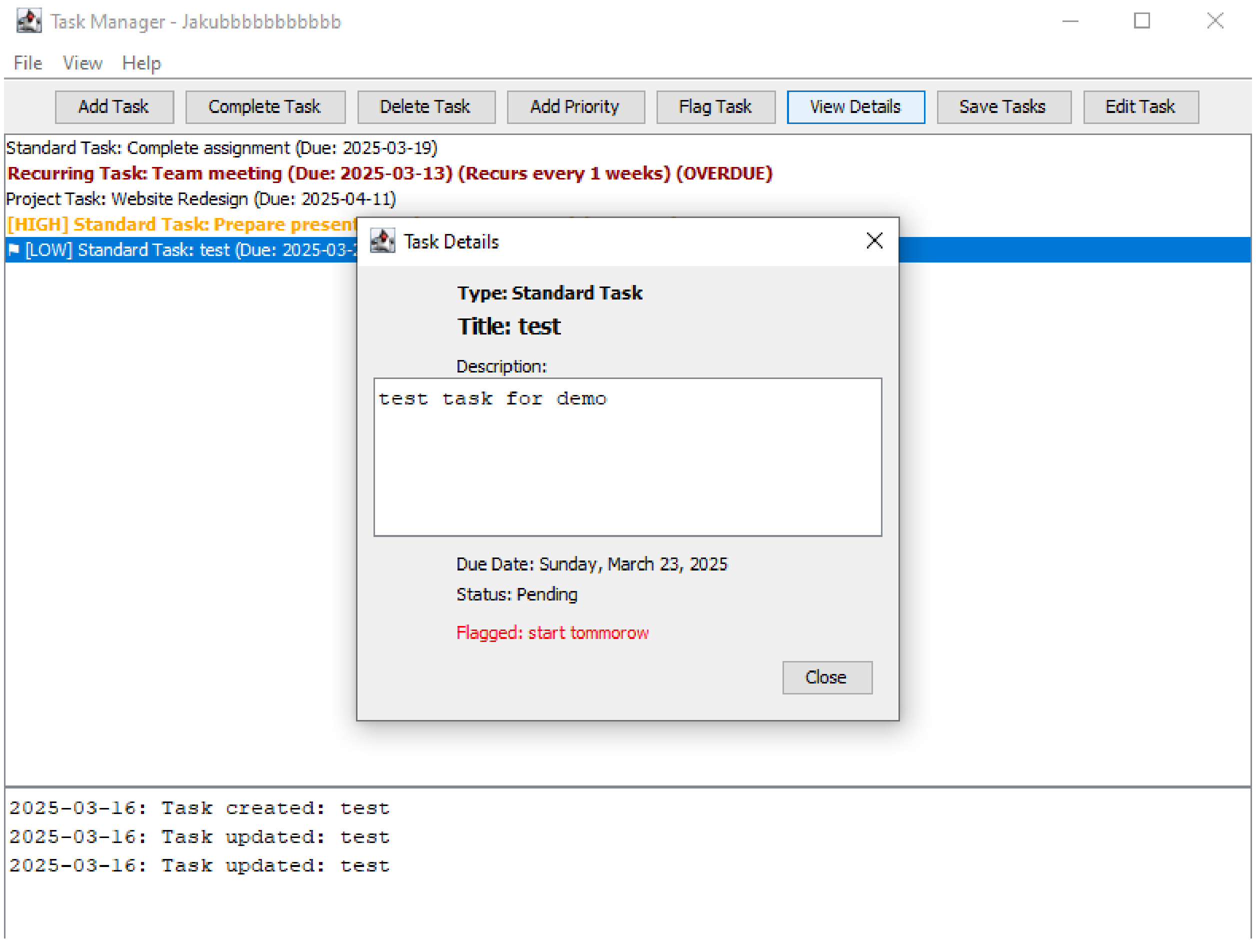
Custom flag added



Standard Task: Complete assignment (Due: 2025-03-19)
Recurring Task: Team meeting (Due: 2025-03-13) (Recurs every 1 weeks) (OVERDUE)
Project Task: Website Redesign (Due: 2025-04-11)
[HIGH] Standard Task: Prepare presentation (Due: 2025-03-14) (OVERDUE)
⚑ [LOW] Standard Task: test (Due: 2025-03-23) (Flagged: start tommorow)

```
2025-03-16: Task created: test
2025-03-16: Task updated: test
2025-03-16: Task updated: test
```

View details of a task

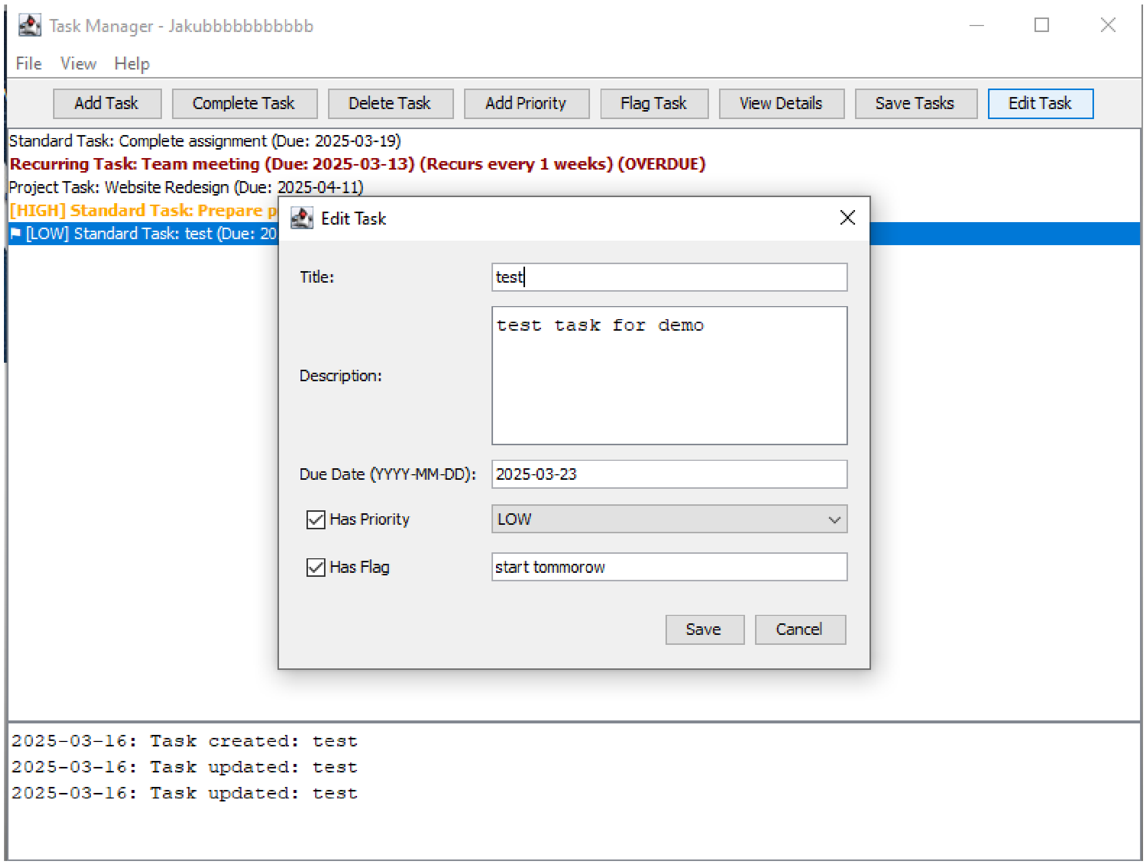


Standard Task: Complete assignment (Due: 2025-03-19)
Recurring Task: Team meeting (Due: 2025-03-13) (Recurs every 1 weeks) (OVERDUE)
Project Task: Website Redesign (Due: 2025-04-11)
[HIGH] Standard Task: Prepare present
⚑ [LOW] Standard Task: test (Due: 2025-03-2

Type: Standard Task
Title: test
Description:
test task for demo
Due Date: Sunday, March 23, 2025
Status: Pending
Flagged: start tommorow

```
2025-03-16: Task created: test
2025-03-16: Task updated: test
2025-03-16: Task updated: test
```

Editing a task

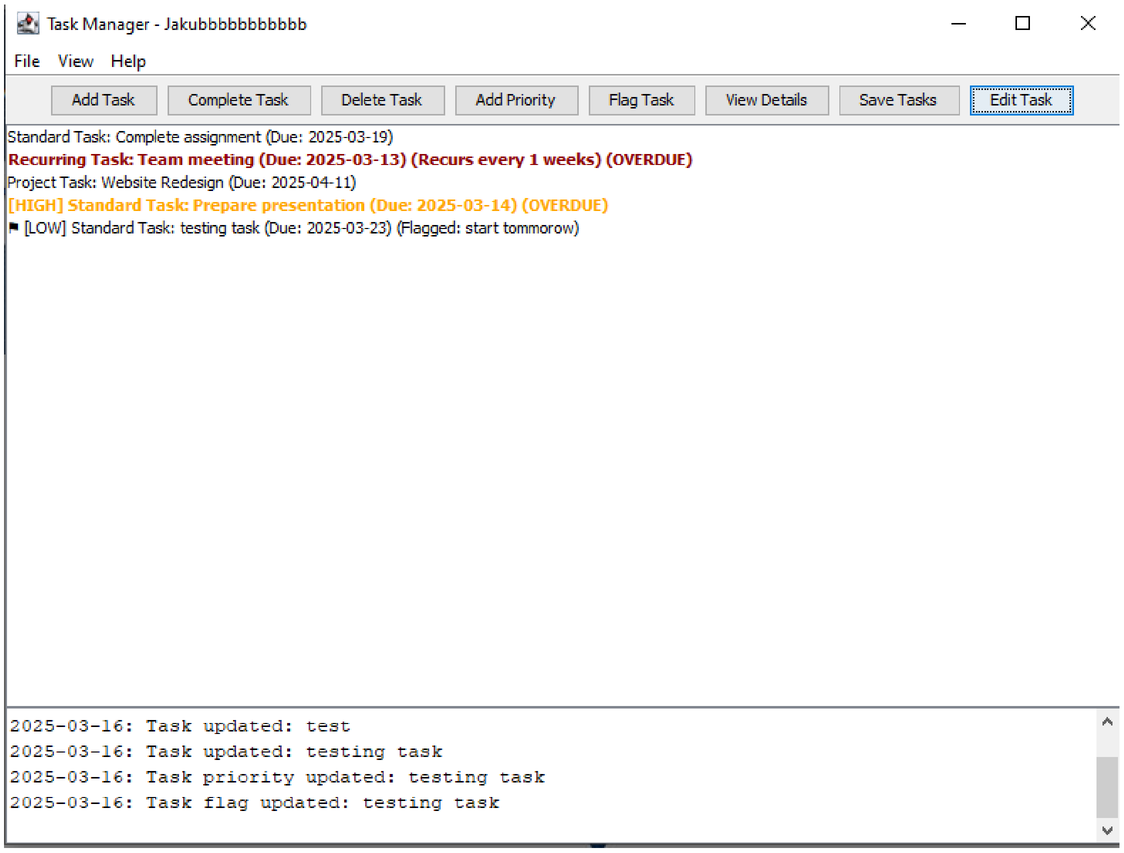


Standard Task: Complete assignment (Due: 2025-03-19)
Recurring Task: Team meeting (Due: 2025-03-13) (Recurs every 1 weeks) (OVERDUE)
Project Task: Website Redesign (Due: 2025-04-11)
[HIGH] Standard Task: Prepare p
⚑ [LOW] Standard Task: test (Due: 20

Title:	test
Description:	test task for demo
Due Date (YYYY-MM-DD):	2025-03-23
☑ Has Priority	LOW
☑ Has Flag	start tommorow

```
2025-03-16: Task created: test
2025-03-16: Task updated: test
2025-03-16: Task updated: test
```

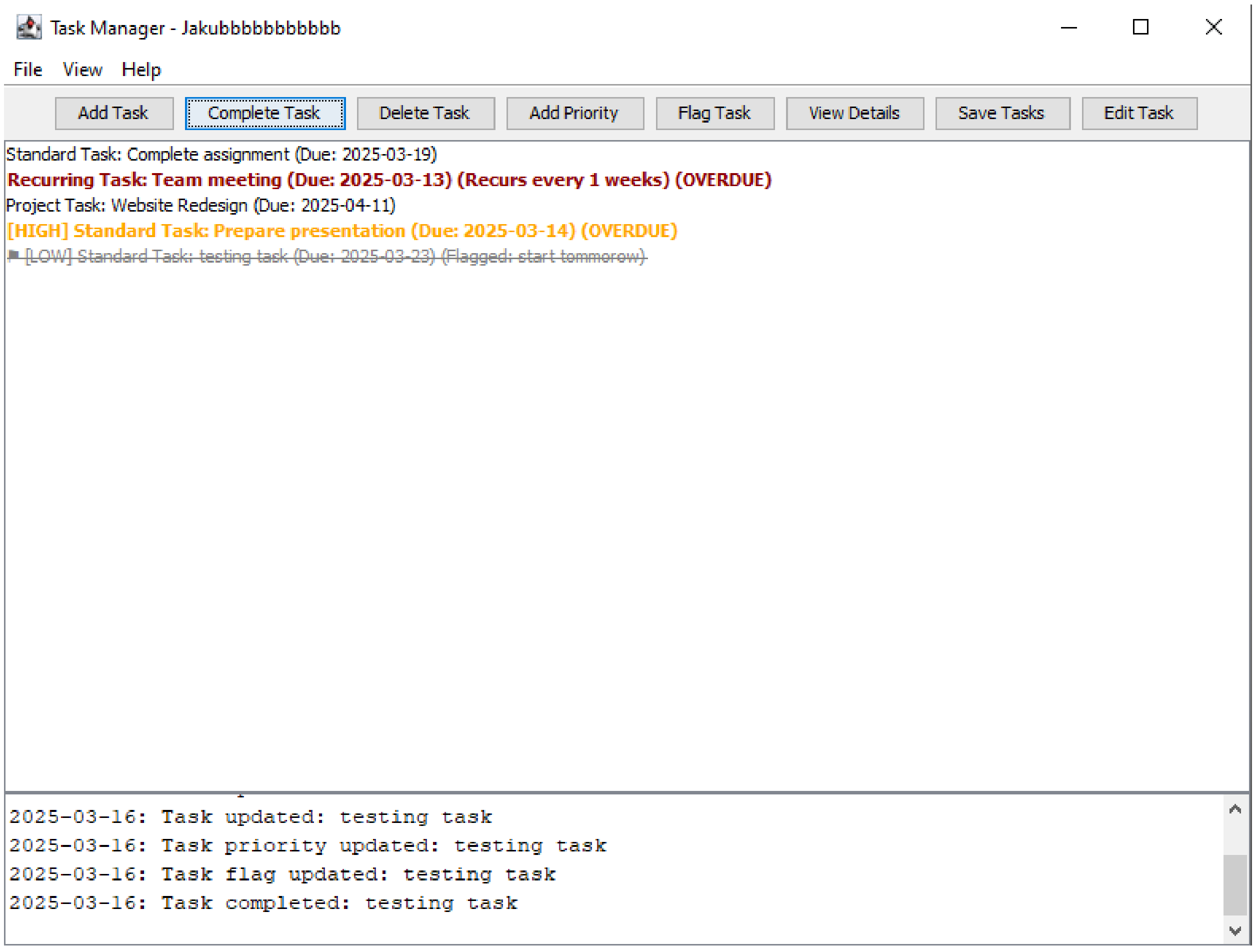
Edited task



```
Standard Task: Complete assignment (Due: 2025-03-19)
Recurring Task: Team meeting (Due: 2025-03-13) (Recurs every 1 weeks) (OVERDUE)
Project Task: Website Redesign (Due: 2025-04-11)
[HIGH] Standard Task: Prepare presentation (Due: 2025-03-14) (OVERDUE)
⚑ [LOW] Standard Task: testing task (Due: 2025-03-23) (Flagged: start tommorow)
```

```
2025-03-16: Task updated: test
2025-03-16: Task updated: testing task
2025-03-16: Task priority updated: testing task
2025-03-16: Task flag updated: testing task
```

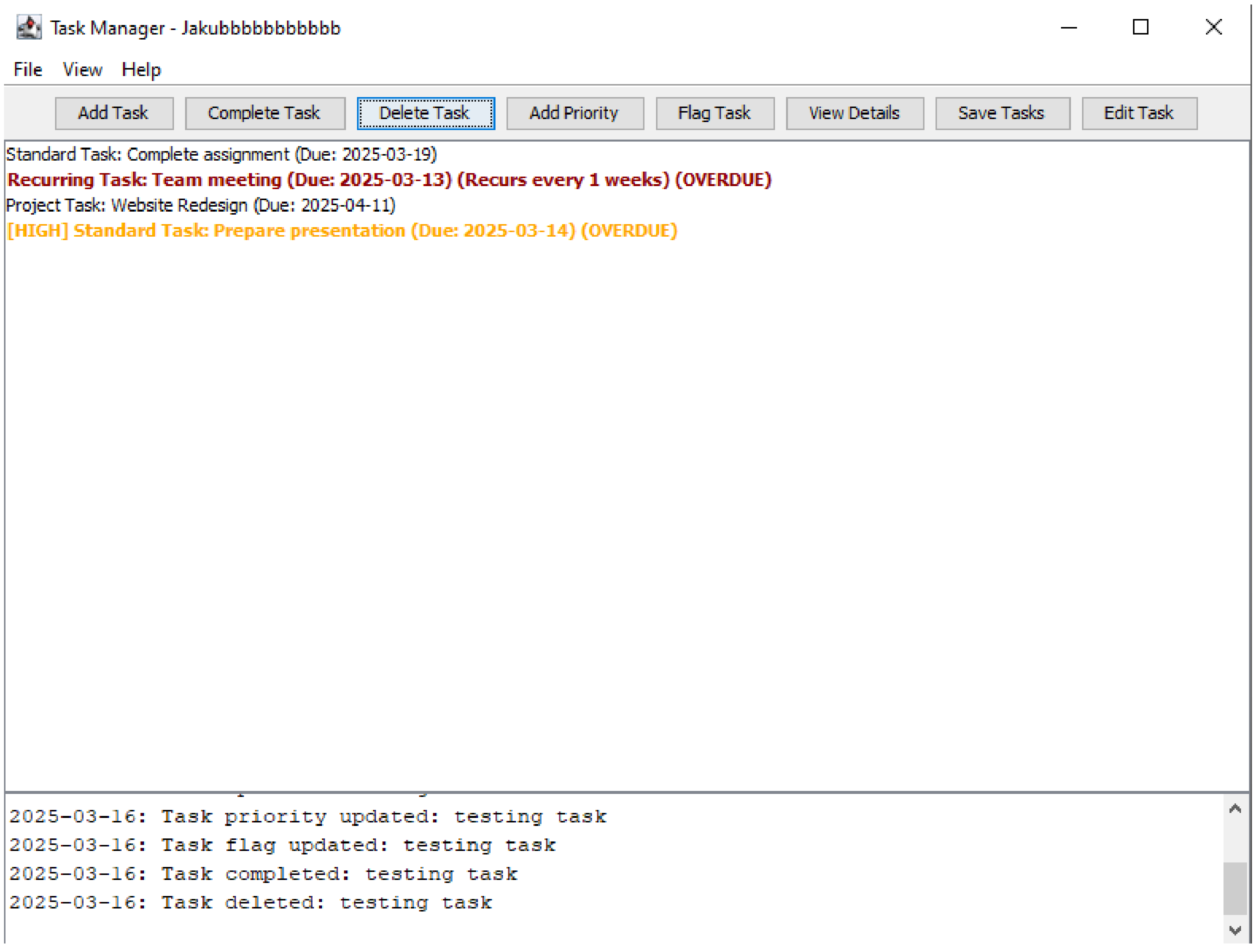
Marking task as complete after completion



Standard Task: Complete assignment (Due: 2025-03-19)
Recurring Task: Team meeting (Due: 2025-03-13) (Recurs every 1 weeks) (OVERDUE)
Project Task: Website Redesign (Due: 2025-04-11)
[HIGH] Standard Task: Prepare presentation (Due: 2025-03-14) (OVERDUE)
~~⚑ [LOW] Standard Task: testing task (Due: 2025-03-23) (Flagged: start tommorow)~~

```
2025-03-16: Task updated: testing task
2025-03-16: Task priority updated: testing task
2025-03-16: Task flag updated: testing task
2025-03-16: Task completed: testing task
```

Deleting completed task



Standard Task: Complete assignment (Due: 2025-03-19)
Recurring Task: Team meeting (Due: 2025-03-13) (Recurs every 1 weeks) (OVERDUE)
Project Task: Website Redesign (Due: 2025-04-11)
[HIGH] Standard Task: Prepare presentation (Due: 2025-03-14) (OVERDUE)

```
2025-03-16: Task priority updated: testing task
2025-03-16: Task flag updated: testing task
2025-03-16: Task completed: testing task
2025-03-16: Task deleted: testing task
```