<u>SOFTWARE DOCUMENTATION</u>

Task Manager – Simple Command-Line App

Prepared by Group 7

July 25th, 2025

Table of Contents..

1. INTRODUCTION

1.1 PURPOSE

The purpose of this document is to provide a comprehensive overview of the Task Manager CLI Application, an offline tool designed to help students and workers efficiently manage their daily tasks. This documentation aims to guide developers, testers, and end-users by detailing the system's architecture, requirements, design, and usage instructions. By clearly outlining the system's features and structure, this document ensures that all stakeholders have a shared understanding of the application and its intended use.

1.2 Scope

The Task Manager CLI Application is a lightweight, command-line-based tool that allows users to create, view, edit, and delete tasks without requiring an internet connection. The application is specifically tailored for users in regions with limited or unreliable internet access, such as many areas in Africa. By focusing on simplicity and offline functionality, the tool ensures accessibility and ease of use for a wide range of users, including those who may not be technologically advanced.

1.3 System Architecture Document

Overview

The Task Manager application follows a modular, object-oriented architecture with clear separation of concerns. The system is designed using SOLID principles and implements comprehensive file management for data persistence.

Component Architecture

Core Components

1. Task Class (src/Task.js)

    Purpose: Encapsulates individual task data and behavior
    Key Features:
        Private fields using JavaScript's # syntax for encapsulation
        Validation methods for data integrity
        Serialization/deserialization for file storage
        Business logic for overdue detection and date calculations

2. Specialized Task Classes

    WorkTask: Extends Task with project-specific properties

PersonalTask: Extends Task with location-specific properties
Demonstrates: Inheritance and polymorphism principles

3. TaskManager Class (src/TaskManager.js)

Purpose: Central business logic coordinator
Responsibilities:

- CRUD operations for tasks
- Search and filtering algorithms
- Data validation coordination
- File persistence coordination

4. FileHandler Class (src/FileHandler.js)

Purpose: Abstracts all file I/O operations
Features:

- JSON file management
- Automatic backup system
- Import/export functionality
- Error recovery mechanisms

5. Validator Class (src/Validator.js)

Purpose: Centralized data validation
Functions:

- Input sanitization
- Data format validation
- Business rule enforcement

6. CLI Class (src/CLI.js)

Purpose: User interface management
Technologies: Commander.js, Inquirer.js, Chalk
Features:

- Interactive menu system
- Command-line argument parsing
- Colored output for better UX

Data Flow

```
User Input → CLI → TaskManager → Validator → Task Classes
          ↓
      FileHandler → JSON Files
```

File Structure

```
task-manager/
├── index.js           # Application entry point
├── src/               # Source code
│   ├── Task.js        # Task classes
│   ├── TaskManager.js  # Business logic
│   ├── FileHandler.js  # File operations
│   ├── Validator.js    # Data validation
│   └── CLI.js         # User interface
├── tests/             # Test suite
├── docs/              # Documentation
├── exports/           # Export directory
├── tasks.json         # Main data file
└── tasks_backup.json   # Backup data file
```

Design Patterns Used

1. Singleton Pattern

    FileHandler ensures a single instance for file operations

2. Factory Pattern

    TaskManager creates different task types based on input

3. Observer Pattern

    CLI observes TaskManager state changes

4. Strategy Pattern

    Different sorting and filtering strategies

Error Handling Strategy

1. Graceful Degradation

    Application continues running even if non-critical operations fail

2. Backup Recovery

    Automatic fallback to backup files if the main data is corrupted

3. User-Friendly Messages

   Technical errors are translated into user-understandable messages

4. Validation Layers

   Multiple validation points prevent invalid data propagation

Security Considerations

1. Input Sanitization

   All user inputs are sanitized before processing

2. File Path Validation

   Safe file operations with path validation

3. Error Information Disclosure

   Sensitive information is not exposed in error messages

Performance Optimizations

1. Efficient Algorithms

   Optimized search and filter operations

2. Memory Management

   Proper object cleanup and memory usage

3. File I/O Optimization

   Minimized file operations with intelligent caching

Scalability Considerations

1. Modular Design

   Easy to add new task types and features

2. Plugin Architecture

   Extensible through additional modules

3. Configuration Management

   Centralized configuration for easy customization

1.4 Intended Audience and Reading Suggestions

This document is intended for several audiences. Developers and contributors can use it to understand the system's architecture and design decisions, making it easier to maintain or extend the application. Testers will find the requirements and use cases helpful for designing test cases. End-users and instructors can refer to the setup and usage instructions to get started with the application. It is recommended that readers begin with the introduction and overall description before moving on to the more technical sections.

2. Overall Description

2.1 Product Perspective

The Task Manager CLI is a standalone application that does not depend on any external services or databases. It is designed to be self-contained, with all data stored locally on the user's device. This approach ensures that users retain full control over their data and can use the application regardless of their internet situation.

2.2 User Classes and Characteristics

The primary users of this application are students and workers who need a straightforward way to manage their daily tasks. These users may have limited access to technology or the internet, so the application is designed to be as simple and intuitive as possible. Users are expected to have basic familiarity with using a computer terminal or command prompt.

2.3 Operating Environment

The application is cross-platform and can run on any operating system that supports Node.js, including Windows, Linux, and macOS. It requires minimal system resources and does not rely on any external network connections, making it suitable for use on older or less powerful computers.

2.4 Design and Implementation Constraints

To ensure maximum accessibility, the application is limited to a command-line interface and uses only local file storage (JSON format). No graphical user interface or online features are included. The application must be implemented in JavaScript using Node.js, and all dependencies should be open-source and freely available.

2.5 User Documentation

Comprehensive user documentation is provided in the project's README file, which includes setup instructions, usage examples, and troubleshooting tips. Additionally, the application includes a built-in help command that users can access at any time for guidance on available commands.

2.6 Assumptions and Dependencies

It is assumed that users have Node.js installed on their computers and possess basic knowledge of using the command line. The application depends on Node.js and standard Node.js modules for file handling and input/output operations.

3. External Interface Requirements

3.1 User Interfaces

The user interface is entirely command-line based. Users interact with the application by typing commands such as add, view, edit, delete, and complete, followed by relevant parameters. The application provides clear prompts and feedback messages to guide users through each operation.

3.2 Hardware Interfaces

No special hardware is required to run the application. It is designed to work on any standard personal computer capable of running Node.js.

3.3 Software Interfaces

The application interfaces with the local file system to read and write task data in JSON format. It does not require or support integration with other software systems or APIs.

3.4 Communications Interfaces

There are no network or communication interface requirements, as the application is designed to function entirely offline.

4. Requirements Specification

4.1. Functional Requirements

There are no network or communication interface requirements, as the application is designed to function entirely offline.

Functional Requirements	**Description**
Add Task	Users can create a new task by specifying a title and an optional description. Each task is assigned a unique identifier.
View Tasks	Users can list all tasks, with options to filter by completed or pending status.
Edit Task	Users can modify the title or description of an existing task.
Delete Task	Users can remove a task from the list permanently.

Mark Task as Complete	Users can mark a task as completed, helping them track their progress.
Backup Tasks	Users can create a backup of their tasks to a separate file for safekeeping.

4.2 Non-functional Requirement

In addition to the functional requirements, the application must meet the following non-functional requirements:

Non-Functional Requirements	**Description**
Usability	The application should be easy to use, with clear instructions and helpful error messages.
Reliability	The application must reliably save and retrieve tasks, ensuring that data is not lost or corrupted.
Performance	The application should respond quickly to user commands, even with a large number of tasks.
portability	The application must run on any operating system that supports Node.js.
Security	The app is for personal use, so no passwords are needed; all data stays on the user's computer.
Safety	The app should warn users before deleting a task to prevent mistakes.

5. Appendix

5.1 Glossary

I. CLI (Command Line Interface): A way for users to interact with the application by typing text commands.
II. Task: An item representing something the user needs to do, such as an assignment or work-related activity.
III. JSON (JavaScript Object Notation): A lightweight data format used to store tasks on the user’s computer.

5.2 Analysis Models

5.2.1 Class Diagram

Description: The class diagram for the Task Manager CLI application shows the main building blocks of the system and how they are connected.

There are four main classes:

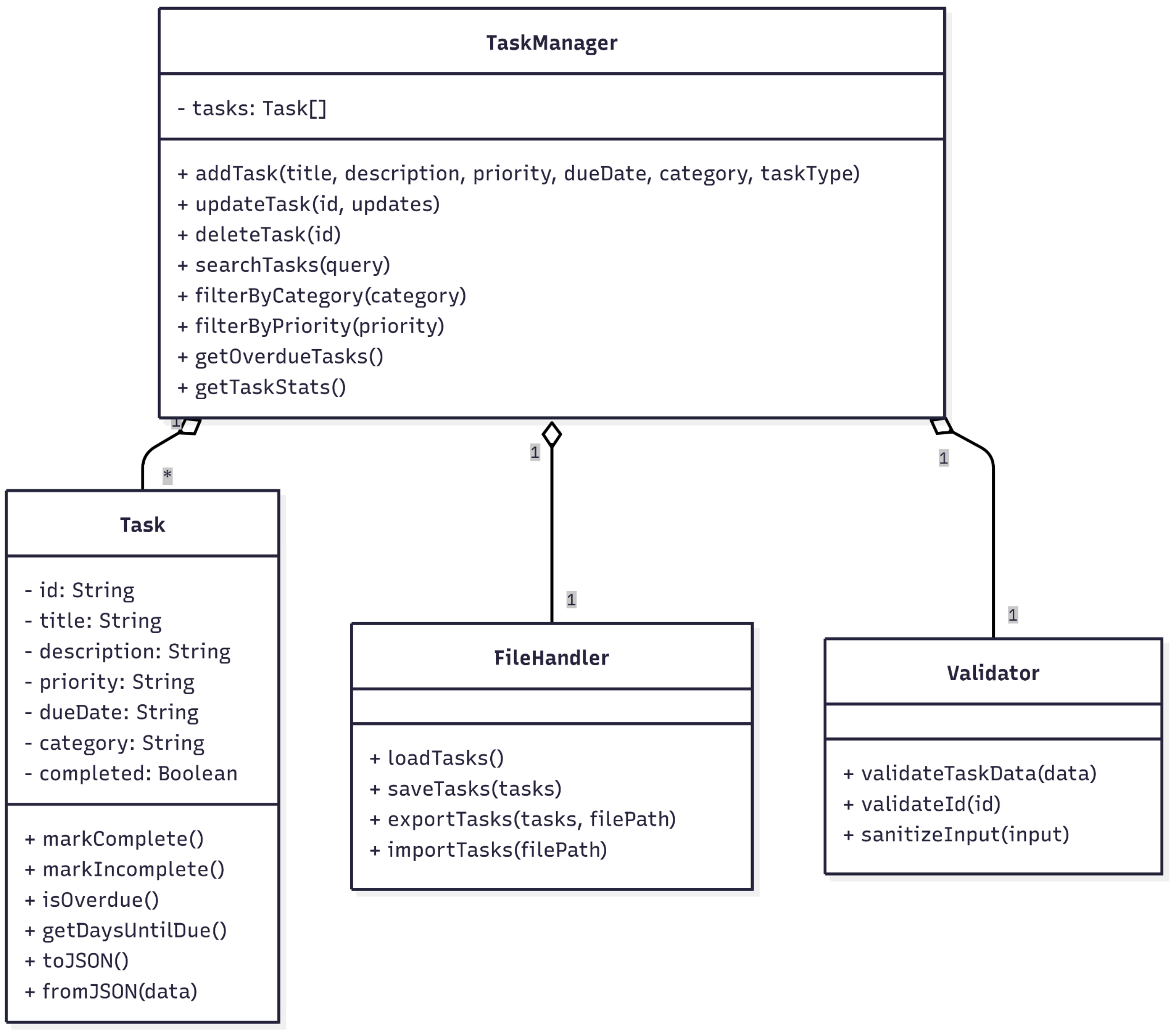
I. Task: This class represents a single task. It stores information like the task’s title, description, priority, due date, category, and whether it is completed. It also has methods to mark the task as complete or incomplete, check if it’s overdue, and convert the task to and from a format that can be saved to a file.
II. TaskManager: This class manages all the tasks. It keeps a list of tasks and provides methods to add, update, delete, search, and filter tasks. It also generates statistics about the tasks, like how many are completed or overdue.

III. FileHandler: This class is responsible for saving tasks to a file and loading them back when the application starts. It also handles importing and exporting tasks to and from other files.

IV. Validator: This class checks that the data for each task is correct and safe before it is added or updated. It makes sure things like the task title and ID are valid and cleans up user input.

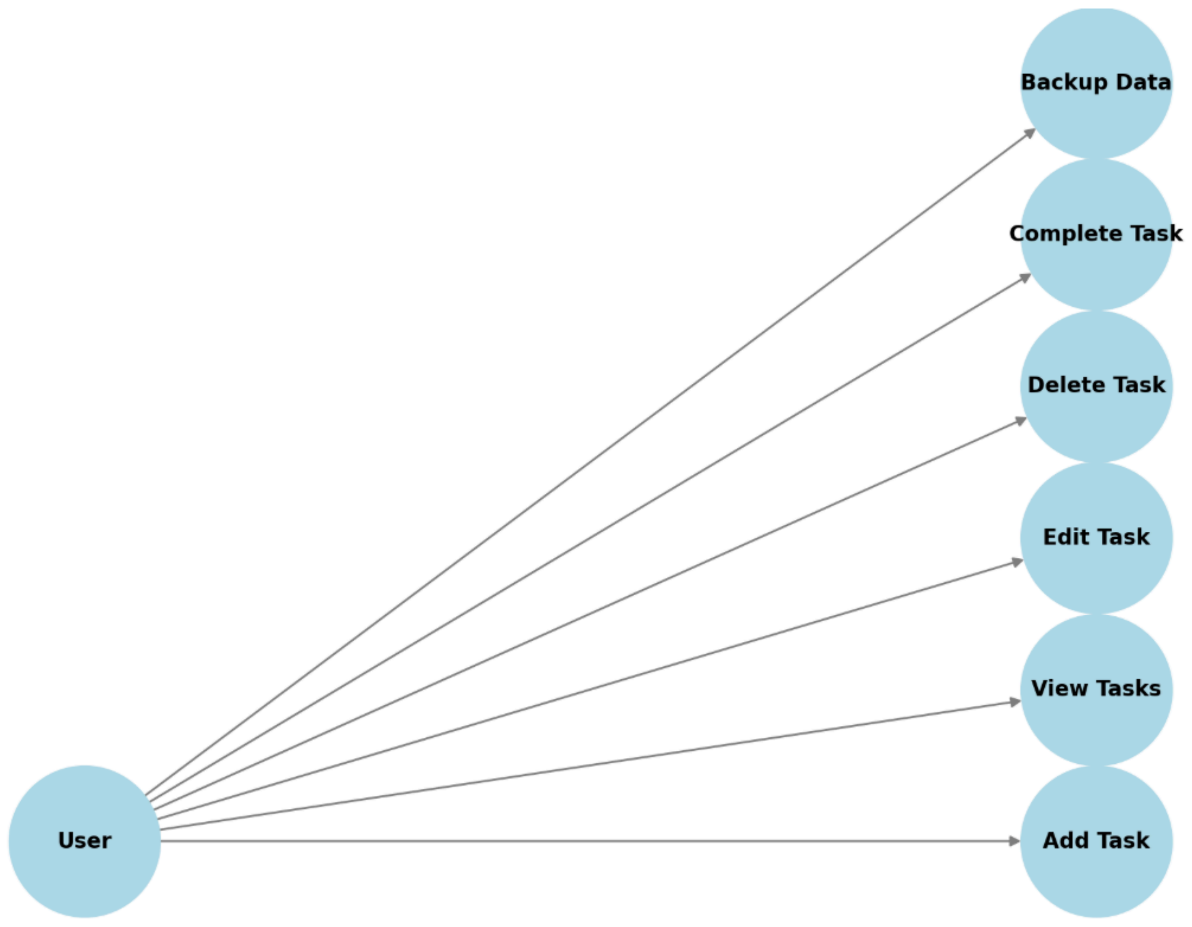
The TaskManager works closely with the FileHandler to store tasks and with the Validator to make sure all data is correct. It also manages multiple Task objects. This structure keeps the code organized and makes it easier to maintain and update the application.

Diagram:



5.2.2 Use Case Diagram

Description: The use case diagram below shows the interactions between the user and the system. The user can perform various actions such as adding, viewing, editing, deleting, completing tasks, as well as backing up their data.



5.2.3 Sequence Diagram

Description: The sequence diagram for adding a new task shows how different parts of the Task Manager CLI application interact when a user wants to create a new task.

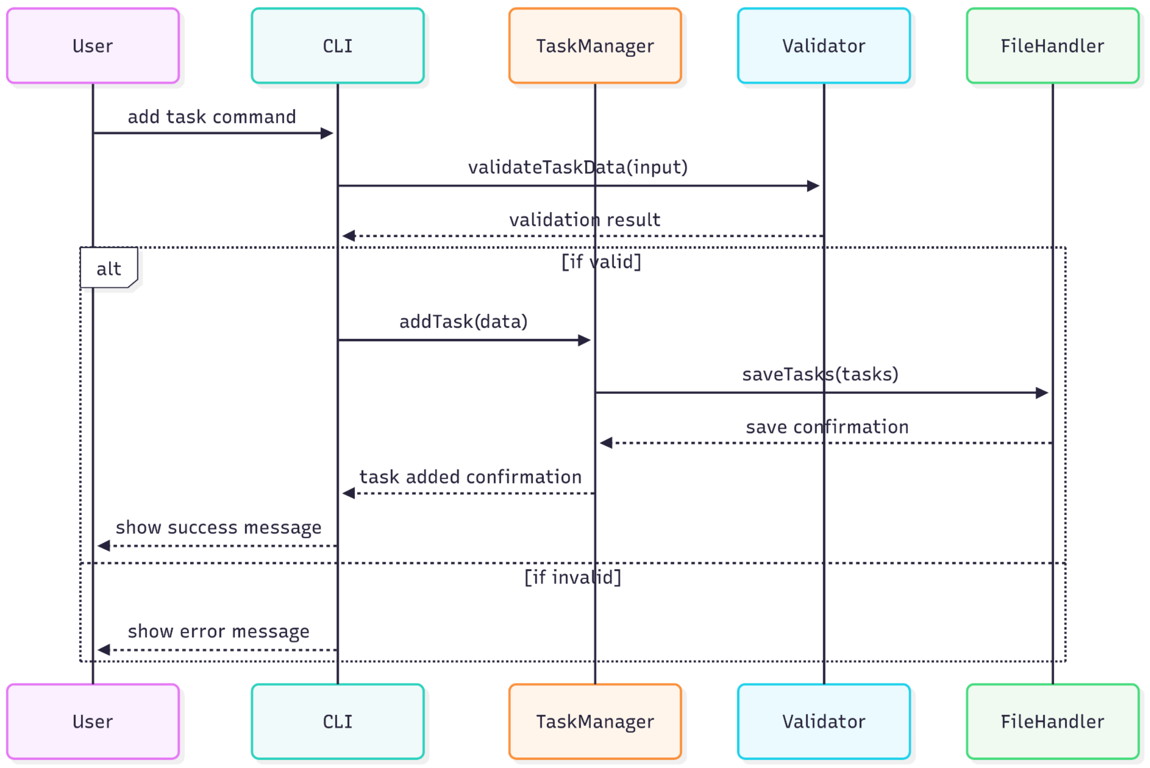
The process starts with the user entering the command to add a task in the CLI. The CLI receives this command and sends the task details to the Validator to check if the input is correct and complete. The Validator returns the result of the validation.

If the input is valid, the CLI passes the task data to the TaskManager, which adds the new task to its list. The TaskManager then calls the FileHandler to save the updated list of tasks to the local file. The FileHandler confirms whether the save was successful.

Finally, the TaskManager informs the CLI that the task was added, and the CLI displays a confirmation message to the user. If at any point the input is invalid, the CLI will instead show an error message and prompt the user to try again.

This sequence ensures that every new task is properly checked, added, and saved, and that the user is kept informed throughout the process.

Diagram:



5.2.4 Activity Diagram

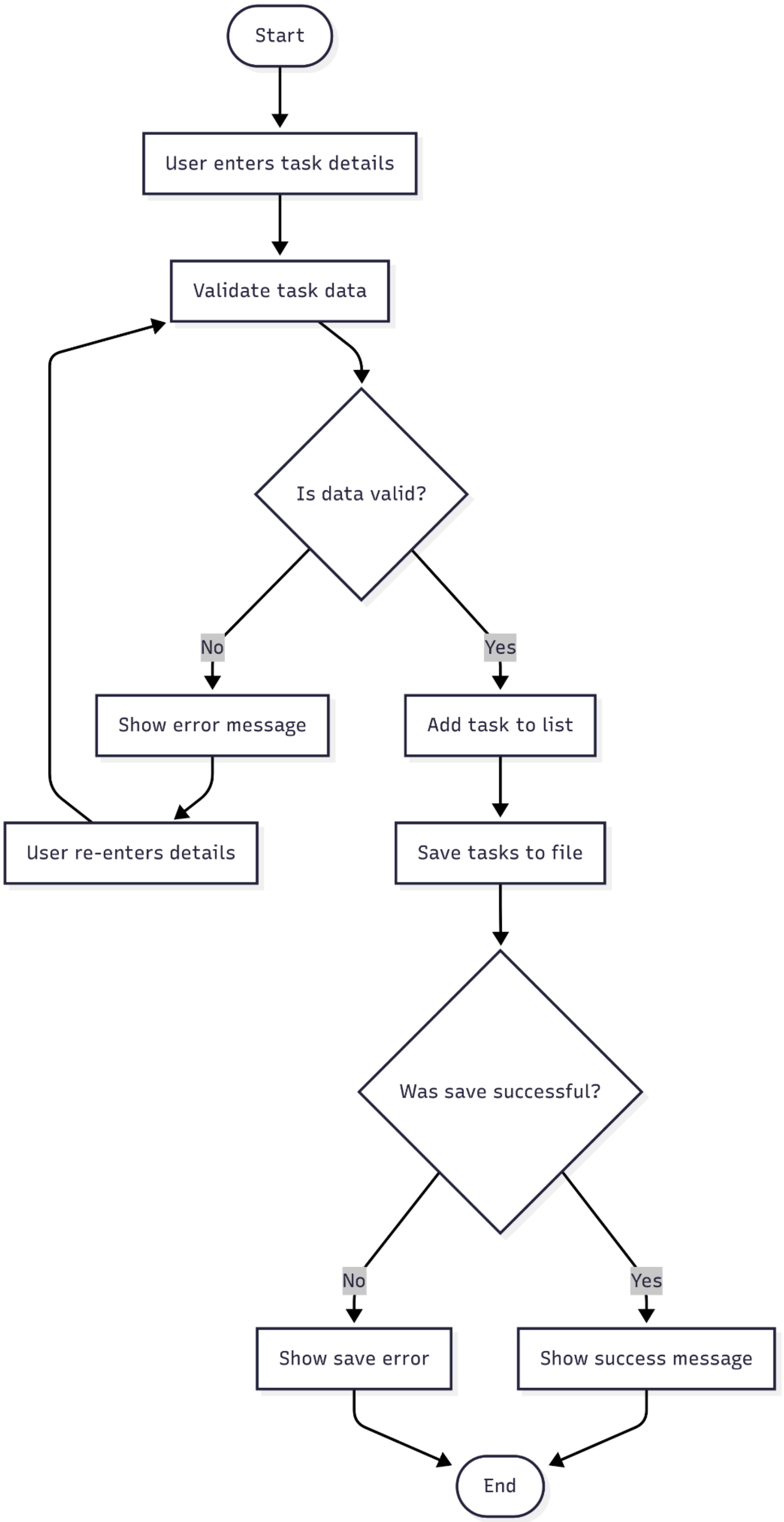
Description: This activity diagram illustrates the step-by-step process a user follows to add a new task in the Task Manager CLI application, including how the system handles errors and user input validation.

The process begins when the user starts the application and enters the details for a new task, such as the title, description, priority, and due date. The system then validates the entered data to ensure all required information is present and correctly formatted.

If the data is not valid (for example, if the title is missing or the date format is incorrect), the system displays an error message and prompts the user to re-enter the details. This loop continues until the user provides valid input.

Once the task data is valid, the system adds the new task to the list and attempts to save all tasks to the local file. If the save operation is successful, the user receives a confirmation message indicating that the task was added successfully. If there is a problem saving the data (such as a file system error), the system displays an error message to inform the user.

The diagram includes decision points for checking if the input is valid and if the save operation was successful, making the process robust and user-friendly. This ensures that users are always informed about what went wrong and how to fix it, leading to a smoother experience when managing their tasks


Start
User enters task details
Validate task data
Is data valid?
No
Yes
Show error message
Add task to list
User re-enters details
Save tasks to file
Was save successful?
No
Yes
Show save error
Show success message
End

6. Instructions for Setting Up and Running the Application

1. **Install Node.js**:

   Download and install Node.js from [https://nodejs.org/]

2. Download the Project:

   Clone or download the project files to your computer.

3. Install Dependencies:

   Open a terminal, navigate to the project directory, and run: npm install

4. Run the Application:

   Start the CLI by running: node src/CLI.js

5. Using the Application:

   - To add a task: follow the prompts or use the `add` command.

   - To view tasks: use the `view` command.

   - To edit, delete, or mark tasks as complete: use the corresponding commands as shown in the help menu.

   - For help: type `help` or `--help` to see all available commands.

6. Data Storage:

All tasks are saved in `tasks.json` in the project directory. Backups are stored in `tasks_backup.json`.

7. API Documentation

Task Class

Constructor

new Task(title, description, priority, dueDate, category)

Methods

markComplete()

Marks the task as completed.

markIncomplete()

Marks the task as incomplete.

isOverdue()

Returns true if the task is overdue, false otherwise.

getDaysUntilDue()

Returns the number of days until the task is due, or null if no due date.

toJSON()

Serializes the task to JSON format for storage.

static fromJSON(data)

Creates a Task instance from JSON data.

TaskManager Class

Methods

async addTask(title, description, priority, dueDate, category, taskType)

Adds a new task to the system.
Parameters:
- title (string): Task title (required)
- description (string): Task description (optional)
- priority (string): Priority level (High/Medium/Low)
- dueDate (string): Due date in YYYY-MM-DD format (optional)
- category (string): Task category
- taskType (string): Type of task (regular/work/personal)

Returns: Task instance

async updateTask(id, updates)

Updates an existing task.
Parameters:
- id (string): Task ID
- updates (object): Object containing fields to update

async deleteTask(id)

Deletes a task by ID.
Parameters:
- id (string): Task ID

Returns: Deleted task instance

searchTasks(query)

Searches tasks by keyword.
Parameters:
- query (string): Search query

Returns: Array of matching tasks

filterByCategory(category)

Filters tasks by category.
Parameters:
- category (string): Category name

Returns: Array of filtered tasks

filterByPriority(priority)

Filters tasks by priority level.
Parameters:
 priority (string): Priority level (High/Medium/Low)
Returns: Array of filtered tasks

getOverdueTasks()

Returns all overdue tasks.
Returns: Array of overdue tasks

getTaskStats()

Returns task statistics.
Returns: Object with statistics:

```
{
  total: number,
  completed: number,
  pending: number,
  overdue: number,
  dueSoon: number,
  completionRate: number
}
```

FileHandler Class

Methods

async loadTasks()

Loads tasks from the JSON file.
Returns: Array of Task instances

async saveTasks(tasks)

Saves tasks to the JSON file.
Parameters:
 tasks (Array): Array of Task instances
Returns: Boolean indicating success

async exportTasks(tasks, filePath)

Exports tasks to a specified file.
Parameters:

tasks (Array): Array of Task instances
filePath (string): Export file path
Returns: Boolean indicating success

async importTasks(filePath)

Imports tasks from a specified file.
Parameters:
filePath (string): Import file path
Returns: Array of Task instances or null if failed

Validator Class

Static Methods

validateTaskData(data)

Validates task data.
Parameters:
data (object): Task data to validate
Returns: Object with validation result:

```
{
 isValid: boolean,
 errors: Array<string>
}
```

validateId(id)

Validates task ID format.
Parameters:
id (string): Task ID
Returns: Boolean indicating validity

sanitizeInput(input)

Sanitizes user input.
Parameters:
input (string): Input to sanitize
Returns: Sanitized string

8. References

1. GitHub Repo
2. Video Presentation Link