To-Do List Application

A To-Do list application allows users to track their tasks. Such application can be used to track personal errands or professional tasks for software developers, newspaper employees, social media managers, and more.

In the application, each task is represented as an object with multiple properties like description, due date, and others. It is possible to link different tasks to indicate that they are interdependent.

Usage scenarios

A user launches the application and runs specific commands to:

- List the existing tasks
- Create a new task
- Request details for an existing task
- Update an existing task with new data
- Remove an existing task
- Query the tasks database, applying filters and sorting criteria

Design of data structure and algorithms

The following data structure for a task object is proposed to support the required functionality. Each task contains a title, description, created date, due date, priority, and state. It is possible to link different tasks to each other using different connection types: related to, subtask of, parent for. A connection is stored as an object that contains the connection type and the identifier of the target task. The values for connection type, task state, and task priority are represented as enumeration entries.

The classes required to describe a task properly are depicted on the diagram in Figure 1.

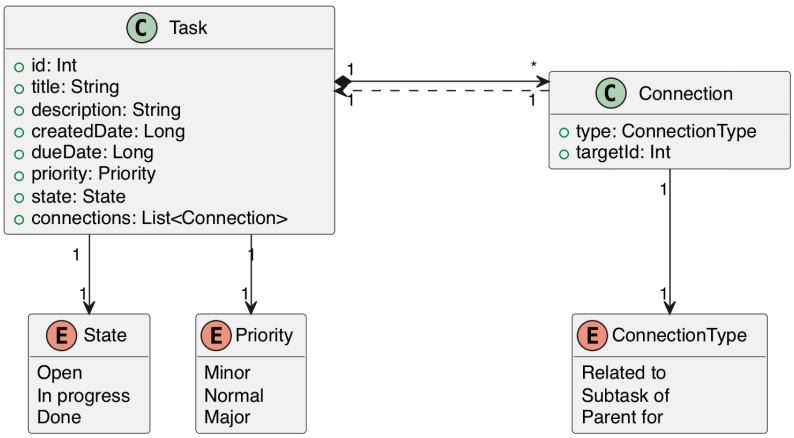

Figure 1: Task class diagram

Tasks will be stored in a database that assigns identifiers to new tasks and enables searching for task objects based on their properties.

Ideally, this would be a database like MySQL where it is possible to set up the primary key with autoincrement and additional indexes for certain fields. In this implementation, the database functionality will be implemented using the built-in JSON support of Python.

A dedicated object will implement the necessary functionality as displayed on the Figure 2. This object will be serialized and deserialized as necessary. Tasks are stored in a dictionary to allow for fast lookup by identifier.

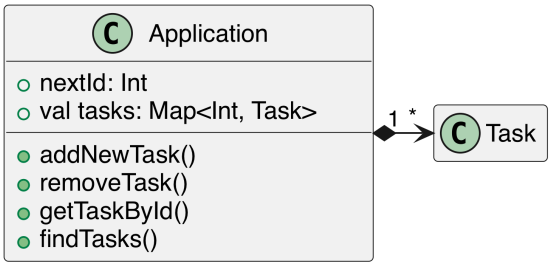

Figure 2: The Application object

Algorithm design

The application will support the functionality for data insertion, deletion, sorting and searching. These tasks will be implemented via the methods of the Application object.

Data insertion

New Task object can be added to the database by calling the addNewTask() method with the metadata for the new task or a draft Task object. The algorithm is depicted on the Figure 3 below.

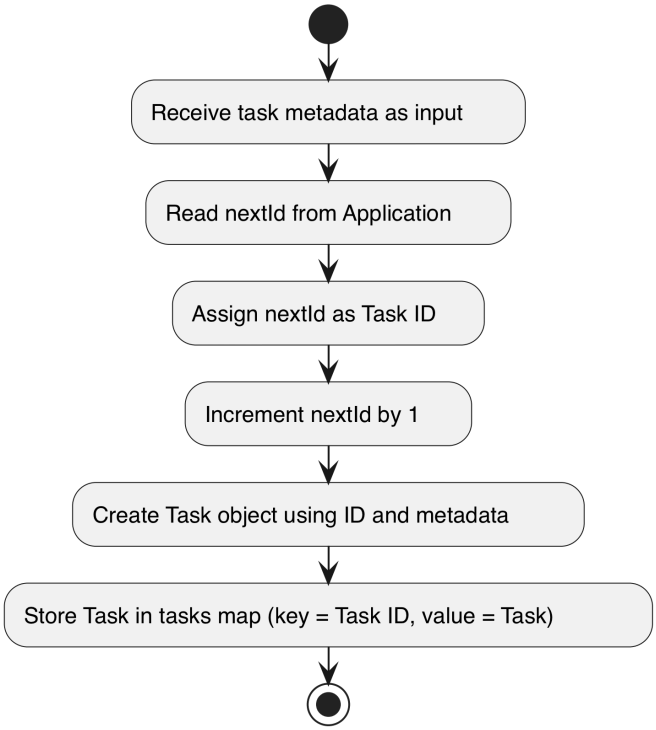

Figure 3: Data insertion algorithm

Data deletion

To remove a Task object from the database, the removeTask() method of the Application object is called specifying the Task identifier. The method will also ensure there are no orphan connections after the task is removed. Figure 4 describes the algorithm for the method. If no task is found with the specified identifier, the application will display an error message.

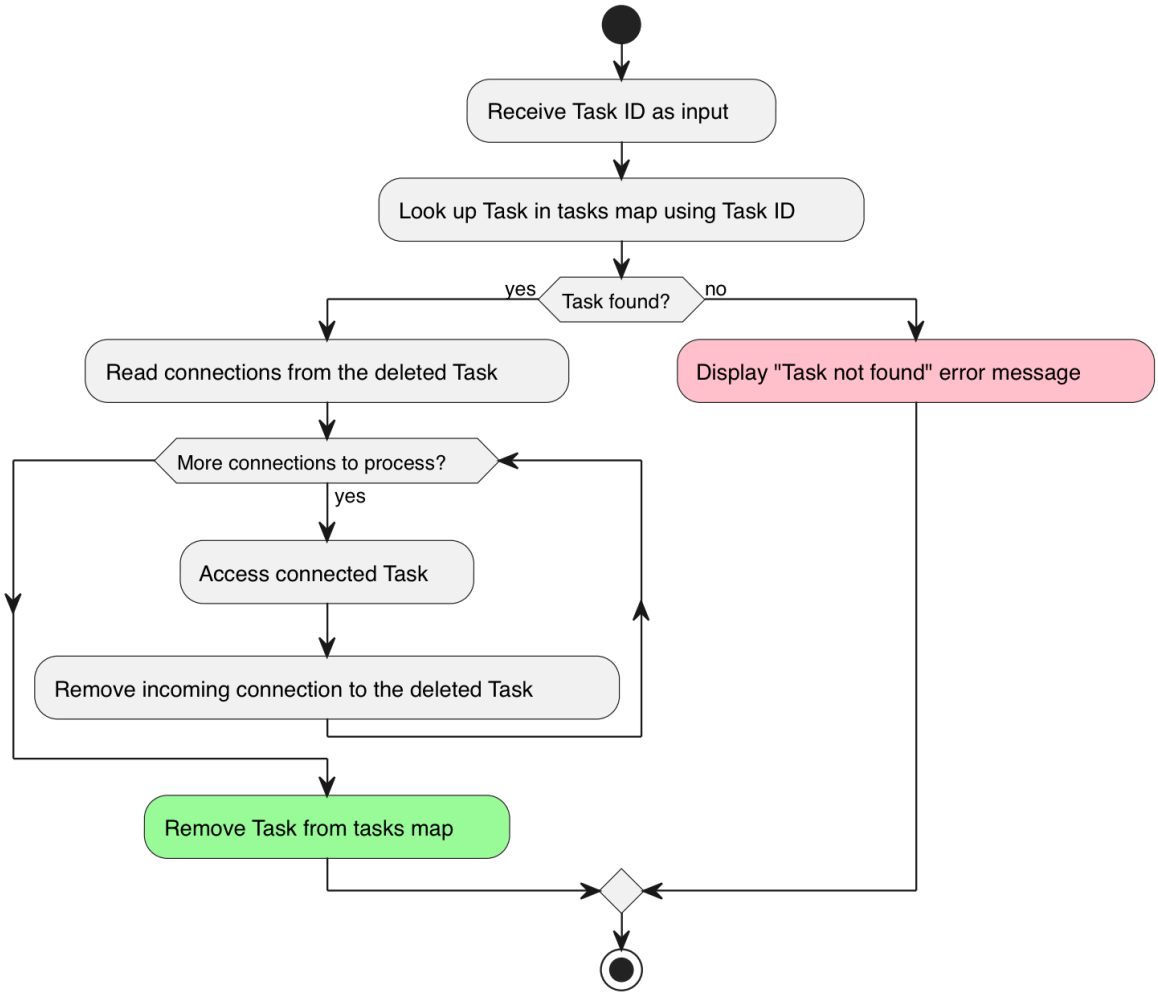

Figure 4: Data deletion algorithm

Data sorting and searching

The findTasks() method will implement an algorithm to look up tasks by criteria such as title or description text, creation date, due date, or priority. The entries can be sorted using the criteria specified by the user based on date or priority fields. The algorithm is depicted on the Figure 5. If there are no tasks matching the search criteria, the result will be an empty list.

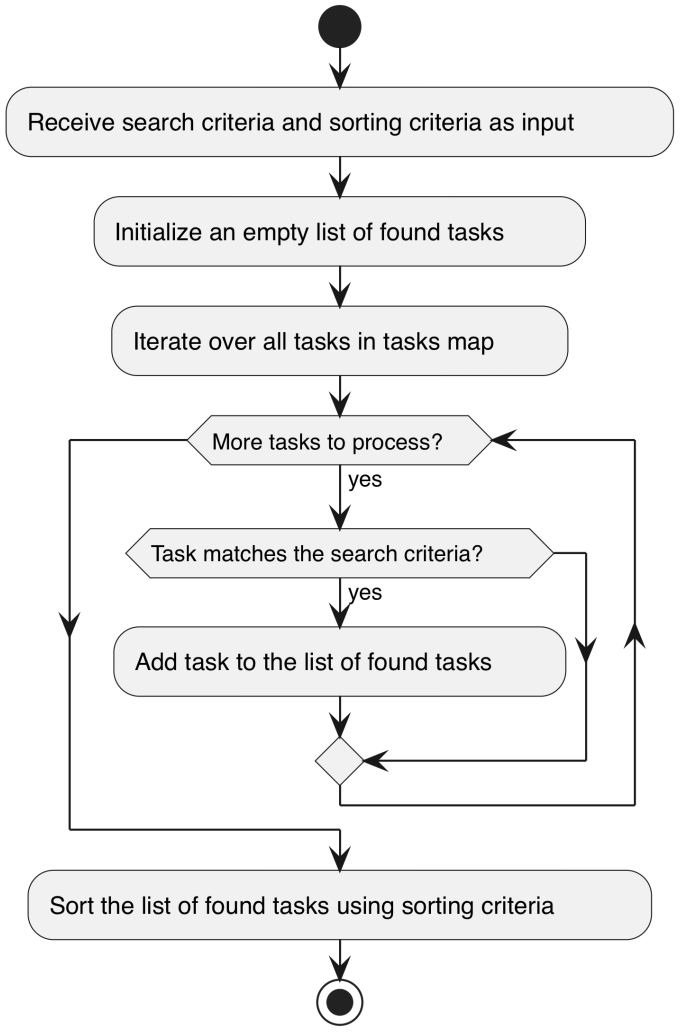

Figure 5: Data sorting and searching algorithm

Test plan

Based on the operations described above it is now possible to outline the test plan for the application and define the results of calling each of the operations.

Operation	Original state	Expected result

Add new task	Database:
	
	`{ "nextId": 1, "tasks": { } }`
	
	Algorithm input:
	
	`{ "id": 0, "title": "Test task", //... }`
		`{ "nextId": 2, "tasks": { "1": { "id": 1, "title": "Test task", //... } } }`

Remove an existing task	Database: `{ "nextId": 2, "tasks": { "1": { "id": 0, "title": "Test task", //... } } }` Algorithm input: “Remove task with identifier ‘1’”	`{ "nextId": 2, "tasks": { } }`
Remove a non-existing task	Database: `{ "nextId": 2, "tasks": { } }` Algorithm input:	Error message and no changes to the database

	“Remove task with identifier ‘10’”	
Data searching and sorting	Database: `{ "nextId": 2, "tasks": { "1": { "id": 1, "title": "Test task 1", "priority": "Minor", //... }, "2": { "id": 2, "title": "Test task 2", "priority": "Major", //... }, "3": { "id": 3, "title": "Task 3", "priority": "Major", //... } } }` Algorithm input: “Find tasks containing text ‘Test’ sorting by their priority descending”	`[ { "id": 2, "title": "Test task 2", "priority": "Major" //... }, { "id": 1, "title": "Test task 1", "priority": "Minor" //... } ]`

Additional test cases may be required, depending on implementation details, to ensure full coverage for the Application object and Task class methods.

Conclusion

This proposal outlines the functionality of a basic task-tracker application, along with the implementation of key algorithms and test coverage. The application’s architecture supports extending its functionality based on user needs.

To-Do List Application implementation

Application functionality

The developed application implements the following functionality:

- Creating tasks, displaying, editing, and removing them based on user inputs,
- Establishing connections between tasks, including maintaining the database integrity when a task or a connection is removed,
- Filtering and sorting the list of tasks based on the user input,
- Reading the database from a JSON file and saving it there when the application exits.

To facilitate access to the functionality, command-based user interface is implemented.

Testing strategy

Various testing approaches were used for testing the application.

To verify the proper performance of the data structures and algorithms used, automated testing strategies were used, such as unit tests to ensure individual functions work as expected. The following scenarios are tested:

- Adding or removing a task is reflected in the database,
- Attempting to remove a non-existing task raises an error,
- Filtering tasks returns a set where all tasks fulfill the search criteria,
- Sorting tasks returns a list that is sorted according to the specified criteria.

Integration and functional tests are used to verify the behavior of more complex functionality:

- Task identifier increments for consecutive tasks,

- Setting up a connection results in establishing a bidirectional relation between tasks,
- Removing a connection or a task triggers the removal of an incoming connection or connections,
- Setting up hierarchical connections is possible unless results in cyclic dependencies.

To test the user interface of the application, manual testing was used to verify proper behavior of the available commands:

- Each command performs the action it is designed for,
- Commands repeatedly prompt user for correctly formatted input if any is needed,
- If the action cannot be performed, a command displays an error message.

README

The application was tested using Python 3.9.6.

Steps to run the application

1. Execute the notebook.
2. When prompted, specify the file name for the JSON file where the database contents are or should be stored.
3. In the application user interface, run the commands to interact with the application.
4. Use the exit command in the interface to terminate the application and write the database to the file system.

Implementation details

Built-in Python functionality is used for working with enumeration classes. Alternatively, string values for fields like connection type or state can be used, or numerical values for priority. However, enumeration classes allow to declare a predefined set of values for a class, thus providing consistency for the values and alleviating the need for implementing additional checks.

Task class

The class describes the tasks that are managed by the application and implements methods such as add or remove connection. However, they only perform the basic operations to modify the connections without providing any safeguards, which are a part of the Application object logic: editing connections requires access to all existing tasks, which can be seamlessly done from the Application object, and implementing this logic within the Task class would add unnecessary complexity to its definition.

Application object

The Application object stores the tasks in a dictionary where key is task ID, and the value is task object. This allows for quickly accessing the tasks by their identifiers. Alternatively, other collections such as list or set could be used, but looking up an element in such collection generally requires performing a linear search which has O(n) complexity on average. On the other hand, Python dictionaries offer O(1) complexity for retrieving object on average, and O(n) in the worst-case scenario (Python Software Foundation, 2023).

Connections between tasks in the application form a graph, i.e. a set of edges (connections) each associated with two vertices (tasks). Moreover, the different types of

connections ("Parent for", "Subtask of", and "Related to") allow to consider it a directed graph (Bondy and Murty, 1976). While the "Related to" connections are bidirectional and do not form any hierarchical structure, it is important to ensure when setting up other connections that a task does not become its own parent directly or transitively, i.e. that a loop is not formed. For this reason, before such a connection is established, a depth-first search is performed to traverse the descendant nodes of the new child node and ensure the new parent node isn't present in the subgraph. The application of this algorithm allows to effectively traverse the nodes in a graph visiting each node only once (Erickson, 2019, p. 225).

ApplicationInterface class

This class implements the functionality for user interaction with an Application object. The application user interface contains an Application object and provides a command-based user interface for the user to perform different actions as per the available commands.

The searching, sorting, and viewing functionality is interconnected: search results are saved in the current view, which can be displayed on screen and sorted.

References and bibliography

Atlassian (no date) *The different types of testing in software*, *Atlassian*. Available at: https://www.atlassian.com/continuous-delivery/software-testing/types-of-software-testing (Accessed: 10 January 2025).

Bondy, J.A. and Murty, U.S.R. (1976) *Graph theory with applications*. Macmillan London.

Erickson, J. (2019) *Algorithms*. Available at: http://jeffe.cs.illinois.edu/teaching/algorithms/ (Accessed: 20 December 2024).

IBM (2021) *What Is Software Testing?* Available at: https://www.ibm.com/think/topics/software-testing (Accessed: 10 January 2025).

Python Software Foundation (2023) *TimeComplexity - Python Wiki*. Available at: https://wiki.python.org/moin/TimeComplexity (Accessed: 10 January 2025).