Group 24: Yuang Zhang, Tianrui Qi, Zishuo Wang, Yuwen Ding, Tianqin Yu, Yuxuan Li

Design Principles

Low Coupling

The project class does not have the reference to the classes directly like specializedTasks, recurring tasks and high priority tasks. We used interface baseTask to define the subclasses' behaviors first, and this action allows the project class to work directly with any task type that implements the baseTask interface with no direct reference in the project class.

High Cohesion

In our project, we used high cohesion by ensuring that each class focuses on a specific responsibility. For example, the Project class is dedicated to managing tasks and team members, while the Task class encapsulates all functionalities related to individual tasks. This clear focus enhances the clarity of each class's purpose, making it easier to understand, test, and maintain.

Single Responsibility Principle

Each class focuses on a singular purpose. For instance, the Task class handles task-related functionalities, such as managing the title, description, and due date, as well as executing the task itself. Conversely, the TeamMember class is responsible for managing team member attributes, such as name and email, and their interactions within a project. By separating these responsibilities, we allow for easier modifications and testing of individual components. If a change is needed—such as adding a new feature to the task execution logic—only the Task class is affected, so there's less risk on other parts of the codebase.

Open/ClosedPrinciple

We applied the Open/Closed Principle by designing our classes to be open for extension but closed for modification. This was accomplished through the use of abstract classes and interfaces. For example, we implemented specialized task types, such as RecurringTask and SpecializedTask, which extend the abstract Task class. By creating these subclasses, we can introduce new types of tasks with unique behaviors without altering the existing task management structure within the Project class. This approach allows us to maintain a stable codebase while expanding functionality, thereby reducing the risk of introducing bugs due to changes in existing code.

Liskov Substitution Principle

In our project, we followed the Liskov Substitution Principle by designing derived classes that can be substituted for their base classes without affecting the program's functionality. Both NormalTeamMember and SpecialTeamMember inherit from the abstract TeamMember class. This means that anywhere a TeamMember is expected, either subclass can be used interchangeably without breaking the existing functionality. This flexibility is crucial for managing team members in various contexts, as it allows us to treat different team member types uniformly while still providing specialized behaviors where needed.

Interface Segregation Principle

We created specific interfaces for different functionalities. Instead of having a large, monolithic interface, we defined smaller, focused interfaces like ManageTask and ManageTeamMember. Additionally, we utilized abstract classes like BaseTask, allowing us to define shared behaviors while still enabling specific implementations. This design ensures that classes only implement methods relevant to their roles, reducing complexity and improving maintainability.