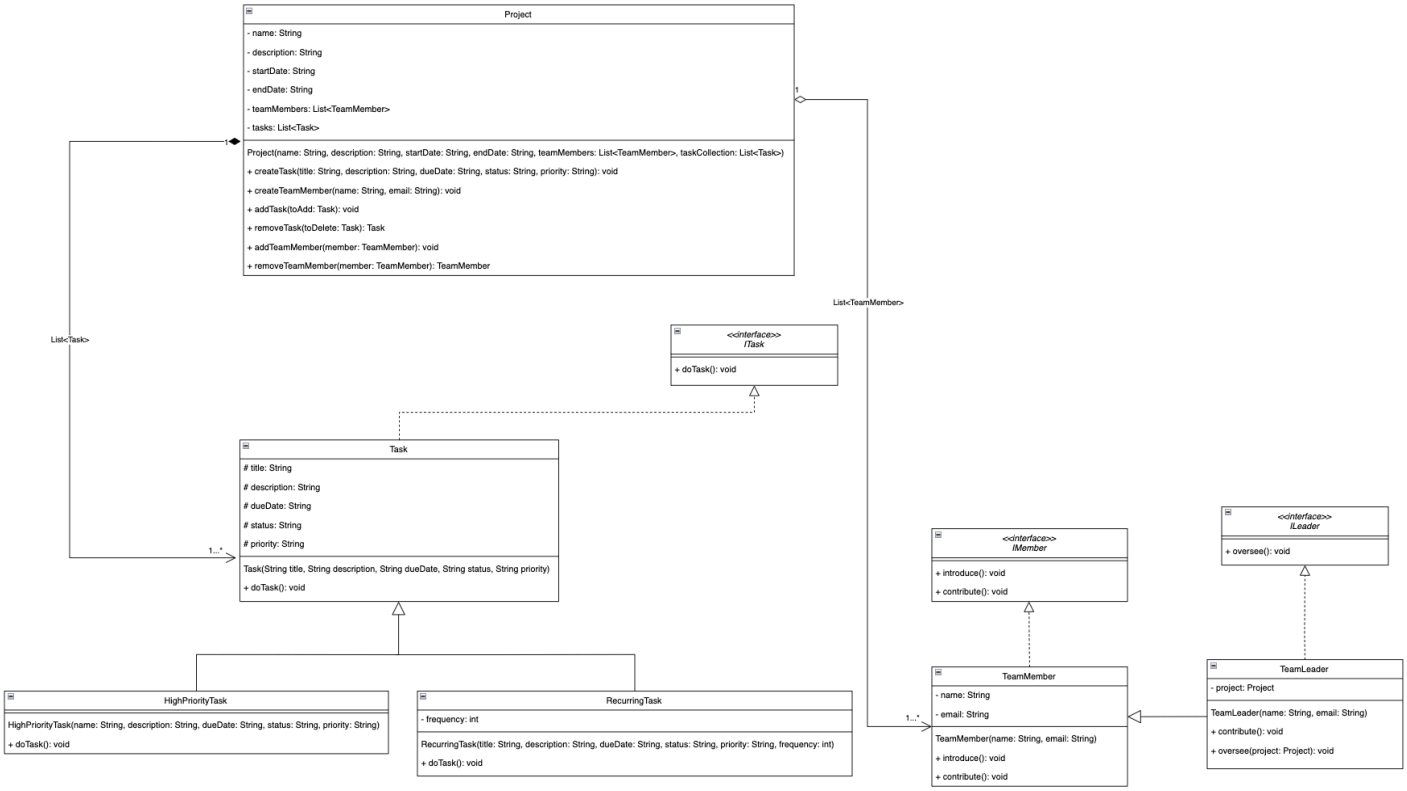
Contributors: Tanmay Gupta, Sarvesh Gade, Aryika Kumar, Evan Romero, Tri Nguyen, Usman Rashid

Design Class Diagram



SOLID and GRASP Principles Used

Single Responsibility Principle:
Our implementation of Task and its subclasses reflect the single responsibility principle. We have a Task class which maintains the behaviors and attributes of tasks in general, and smaller subclasses, HighPriorityTask and RecurringTask, which are each responsible for a single more specific function. For instance, RecurringTask focuses on tasks which need to be repeated as shown by the frequency attribute along with the getter and setter. Similarly TeamLeader is a specific class only for the leader of the project, compared to the generic TeamMember class catering to all members.

Open Closed Principle:
The open closed principle was implemented by using inheritance to extend functionality without having to modify existing classes. For example, the two task types, HighPriorityTask and RecurringTask were made to extend the Task class, and they also implement the ITask interface which is used to ensure consistent behavior. The ITask interface defines the doTask() function, which allows for a standard way for a certain task to be implemented, but also allows the implementation of new tasks without altering the system, for example if we wanted to add a LowPriorityTask.

Liskov Substitution Principle:
The Liskov substitution principle states that any subclasses should be able to functionally replace their superclass without editing the code or program. In our case, all of the child subclasses implement their parent's methods so their instances would work in place of their parent's. For example, both HighPriorityTask and RecurringTask inherit and implement the doTask() method, so their instances could replace Task instances. TeamLeader also inherits the introduce() and contribute() methods from TeamMember, which makes sense as well because team leaders should also be able to do what team members do, and could therefore replace a TeamMember without breaking the code.

Interface Segregation:
Instead of one big monolithic interface structure, we separated the interfaces to require smaller and specific methods. This makes it so that there is no single interface that forces a class under it to implement unnecessary methods that otherwise would not make sense for that class. In our case, we had two different interfaces for team members and team leaders, IMember and ILeader, instead of one interface that encompasses all the methods for project members as a whole. This way, team members and leaders can implement their own specific interfaces and methods instead of overlapping requirements.

Creator Principle:
The Creator Principle helps decide which class should create a new instance of another class. In our system, the Creator Principle was applied to the Project class since it is responsible for aggregating the Task and Member list objects. We did so by implementing a createTask and

createTeamMember method in the Project class, initializing those objects with their respective instance variables.

Information Expert Principle:
The information expert principle says that we should assign responsibility to a class that has the information necessary to fulfill that responsibility. In our case, the project class maintains a list of team members and tasks, making it the information expert as it has the relevant information for adding and removing team members and tasks. The task class is also an information expert as it knows the information necessary to provide task details such as dueDate, status, and priority.