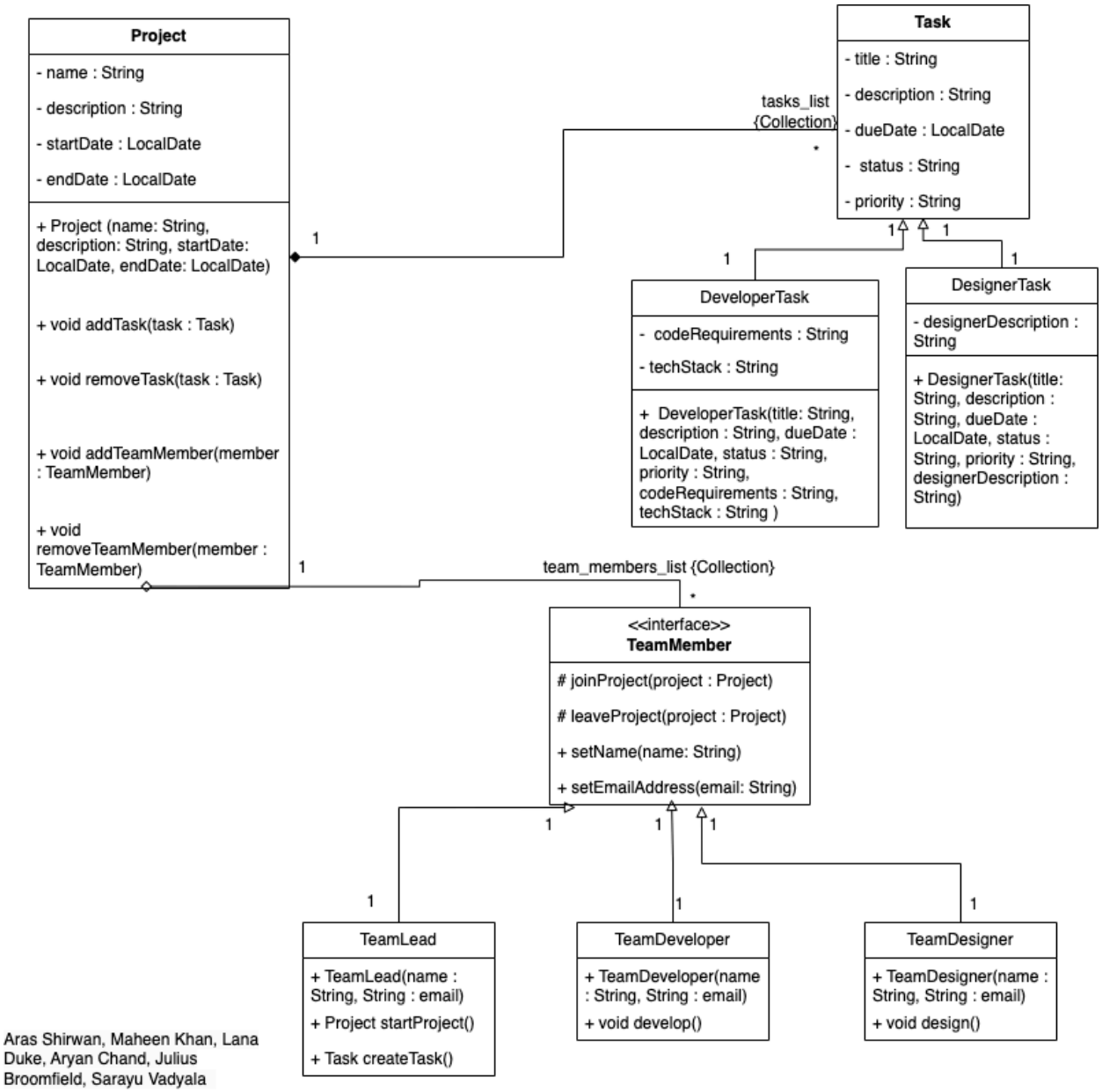
**Team Members: Maheen Khan, Julius Broomfield, Lana Duke, Sarayu Vadyala, & Aryan Chand**

**A write-up explaining how your design and implementation incorporate the SOLID and GRASP principles.The minimum number of design principles in the write-up should equal the number of team members participating in the assignment.**

1. Information Expert
    - Each of the following classes have methods describing their responsibilities: TeamMember, TeamLead, TeamDeveloper, TeamDesigner, Project.
    - **TeamMember**
        - This class has the methods (i.e. jobs) “joinProject(project)” and “leaveProject(project)”.
        - Based on the principle of IE from GRASP, it makes sense for the abstract (template) class TeamMember to definitively have the jobs (methods) for joining or leaving some project, since it is up to that team member to join/leave a project.
    - **TeamLead**
        - This class has the methods of “Project startProject()” and “Task createTask()”.
        - Similarly, it realistically makes sense that the team leader has the responsibilities of starting a project and of creating a task for the project.
        - It wouldn’t make sense for any other TeamMember to have these responsibilities.
    - **TeamDeveloper**
        - This class has the method “develop()”.
        - It makes sense that TeamDeveloper would have this responsibility of actually developing the project/program, since that is the job description! That is their *expertise*, hence following the IE principle exactly.
    - **TeamDesigner**
        - This class has the method “design()”.
        - It makes sense that TeamDesigner would have such responsibility of working on the design for the project.
    - **Project**
        - This class has the methods “addTask()”, “removeTask()”, “addTeamMember()”, and “removeTeamMember()”.
        - Based on the connection lines, a Project object has a list of tasks (tasks_list) and a list of members (team_members_list), so it makes sense that Project is the one that has methods that can add/remove members and tasks.
        - The other classes do not have direct access to these lists. So they would not have these responsibilities.
2. Stamp Coupling (low coupling)
    - Two parts of attaining Low Coupling:

        - Note that we have no bidirectional relationships between modules; we only have one-directional relations between Project → TeamMember and Project → Task
            - Project has a list team_members_list of TeamMember's (Aggregation)
            - Similarly, Project has a list tasks_list of Task's (Composition)
        - We use the interface TeamMember instead of directly relating Project to TeamLead/TeamDeveloper/TeamDesigner to **decouple**
            - Specifically, we do not require Project to know the specifics / work with the specifics of a type of TeamMember's responsibilities. So we can abstract TeamLead/TeamDeveloper/TeamDesigner into the interface TeamMember, and then use that interface to connect with Project.
    - Our type of coupling is Stamp Coupling since we pass data/structures between Project/TeamMember (as arguments), and pass tasks to Project objects and their methods (as arguments). But we are not exceeding our coupling to any higher level like that of Control or Common.
3. High Cohesion
    - High cohesion is when the operations of an element are functionally related. One part of our design and implementation that demonstrates this is the Task class. The task class only contains attributes related to itself, and its child classes, DeveloperTask and DesignerTask, only have operations that are related to how a specific task functions. None of the classes within the "Task module" have operations that are related coincidentally, logically, temporally, procedurally, or communicationally. Since the components of the "Task module" are related functionally, the module follows the high cohesion principle.
4. Liskov Substitution Principle
    - The Liskov Substitution Principle states that any class that implements an interface should be able to substitute any reference in code that implements that same interface. This is demonstrated in our code through the usage of the TeamMember.java interface. This interface outlines the methods joinProject() and leaveProject() which must be concretely defined in any classes implementing the interface (in addition to setters and getters for the name and email string attributes). The classes TeamLead.java, TeamDesigner.java, and TeamDeveloper.java all implement the TeamMember.java interface and concretely define the methods outlined in the interface. As a result, all of the concrete implementations of the TeamMember interface are interchangeable with any reference to a TeamMember, thus satisfying the Liskov Substitution Principle.
5. SRP
    - SRP, or Single Responsibility Principle, says that a class should be responsible for only one thing and should have only one reason to change, guaranteeing that there's a place for everything and that everything is in its place. The **TeamLead** class has the sole responsibility of **starting a project**, as well as **creating tasks**. Likewise, the **TeamDeveloper** class and **TeamDesigner** class have the sole

roles of **developing** and **designing**, respectively. Lastly, the **Project** class has the sole responsibility of **keeping track of team members as well as the different tasks** that have been created (while the team lead has the responsibility of creating tasks, the project class acts as a container to store them).




Aras Shirwan, Maheen Khan, Lana Duke, Aryan Chand, Julius Broomfield, Sarayu Vadyala