SOLID & GRASP Write-Up

Participating Team Members: Sanjay Ramesh, Sami Moussa, Sanat Dhanyamraju, Nikil Kandala, Nikhil Vyas, Benjamin Gunasekara

Creator Pattern - The creator pattern looks at who is responsible for creating a new instance of a class with the main indicator being when a class contains or aggregates another class. Our implementation follows the creator pattern as the classes are directionally connected to classes that they contain. For example, a project contains tasks and team members.

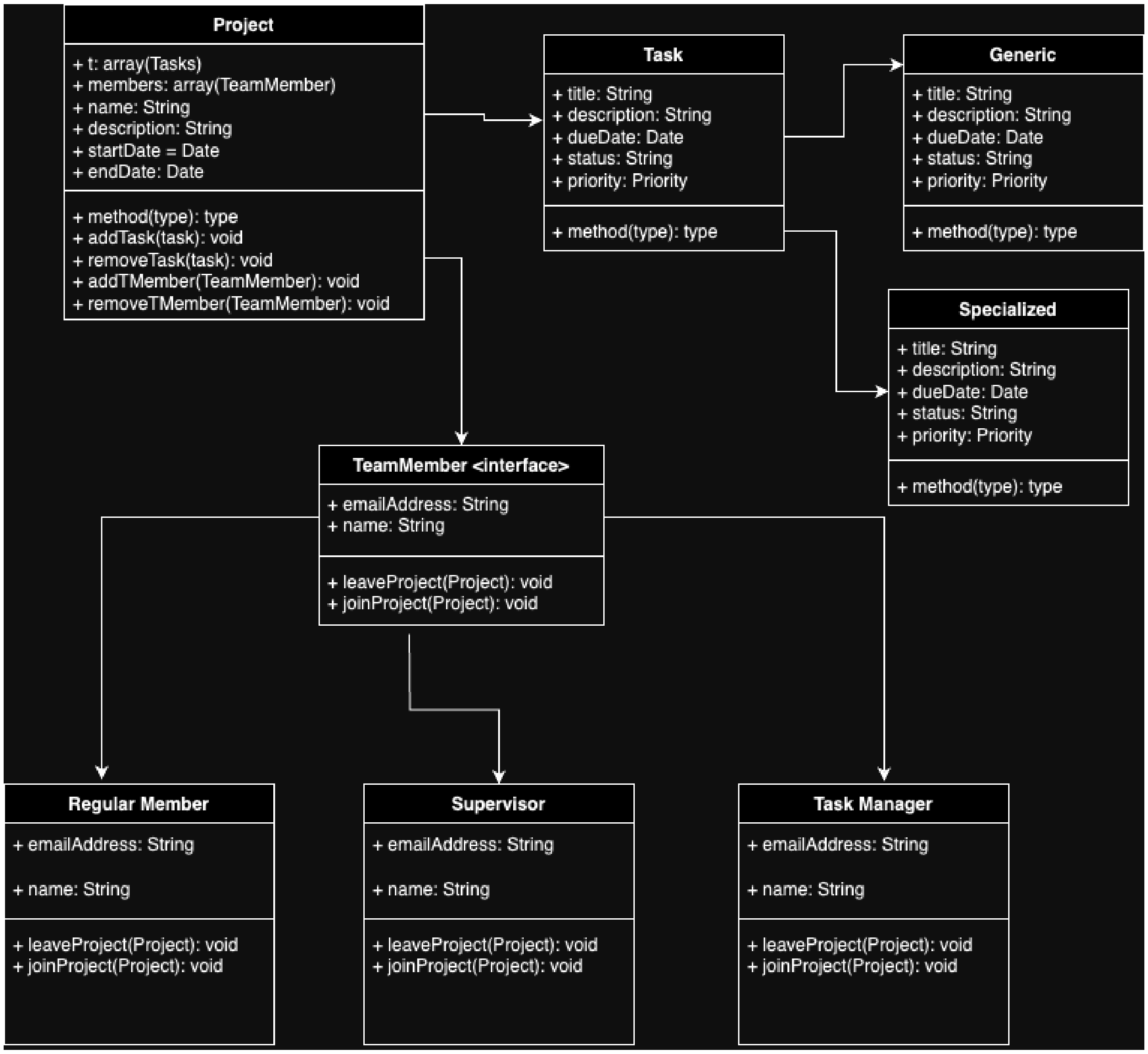
Low Coupling - Coupling is a measure of how strongly different elements are connected. Our design and implementation utilizes low coupling by keeping dependencies to a minimum. One sign of this is the fact that there are no bidirectional relationships between the classes. Also, the TeamMember interface makes it so that the Project class is not tightly coupled with the regular member and supervisor implementations.

High Cohesion - Cohesion is a measure of how functionally related the components of software elements are to each other. Our implementation maintains high cohesion because the methods in each class are all related. The team member interface has the methods of joining and leaving a project. These are related methods that act towards the functionality of a team member. Thus, the element is highly cohesive.

Open/Closed Principle (OCP) - For the open/closed principle to be maintained, the existing code should not be modified but rather should be expanded upon. By doing this, the chance that the system will break will be lessened. Our implementation follows OCP by implementing interfaces. Since the TeamMember interface is created, when new roles need to be added, they can simply be extended from the interface instead of changing the class itself.

Single Responsibility Principle (SRP) - According to the single responsibility principle, every class should only be responsible for one thing. The classes in our implementation match this principle as the functionality of methods in each class are specific to one responsibility. For example, the methods used in the TeamMember interface work only to join or leave a project. This is a specific responsibility that the class works to do.

Liskov Substitution Principle (LSP) - If this principle is followed, then objects of a superclass should be replaceable with objects of its subclasses without breaking the application. This principle exists in our implementation as seen with our interface. A team member object can be replaced with an object of a specific role such as regular member or task manager as they are capable of doing the same thing.


Project
+ t: array(Tasks)
+ members: array(TeamMember)
+ name: String
+ description: String
+ startDate = Date
+ endDate: Date
+ method(type): type
+ addTask(task): void
+ removeTask(task): void
+ addTMember(TeamMember): void
+ removeTMember(TeamMember): void
Task
+ title: String
+ description: String
+ dueDate: Date
+ status: String
+ priority: Priority
+ method(type): type
Generic
+ title: String
+ description: String
+ dueDate: Date
+ status: String
+ priority: Priority
+ method(type): type
Specialized
+ title: String
+ description: String
+ dueDate: Date
+ status: String
+ priority: Priority
+ method(type): type
TeamMember <interface>
+ emailAddress: String
+ name: String
+ leaveProject(Project): void
+ joinProject(Project): void
Regular Member
+ emailAddress: String
+ name: String
+ leaveProject(Project): void
+ joinProject(Project): void
Supervisor
+ emailAddress: String
+ name: String
+ leaveProject(Project): void
+ joinProject(Project): void
Task Manager
+ emailAddress: String
+ name: String
+ leaveProject(Project): void
+ joinProject(Project): void