Participants: Andrew Marc Pilarczyk, Cole Daniel Fitzhenry, Hawthorne Frazier Brown, Kaston Keith Carr, Keith Holden Burroughs, Srihitha Jagarlamudi

Single Responsibility Principle –

The single responsibility principle states that each class should have a single, distinct function and should not be diluted with multiple functionalities. This goes hand in hand with high cohesion, and is reflected in our classes, as each has its distinct purpose. Tasks, and their corresponding sub classes are each held in their own classes. Likewise, team members are all held in their own classes, as is the project. This separation of functionality between classes such that each only has a single responsibility fulfills the requirements of the Single Responsibility Principle.

Open-Closed Principle -

This code demonstrates the open/closed principle through the use of interfaces to define Tasks and TeamMembers. Through these interfaces, new types of tasks and team members can be created without changing the tasks and team members that already have been created. This fits with this principle, as the code is set up to be open for extension and implementation of these new features, while being closed for modification of the existing task types and team member types.

Liskov Substitution Principle –

For both situations where inheritance is used (Tasks and TeamMembers), the subclasses are substitutable as the super classes. Task types such as recurring and high priority have all the functionality of a basic task, having titles, descriptions, due dates, statuses, and priorities, and the ability to mark as in progress, complete and change the priority. Likewise, the TeamLeader can be directly substituted as a TeamMember, while also having the additional functionality beyond that of a member.

Interface Segregation –

Interface Segregation is essentially when no client will be forced to depend on methods that are not necessary or not in use. Rather, it is more important to create smaller and more specific interfaces, rather than a single larger interface for all usage. The code demonstrates this principle through the use of smaller interfaces such as Task and Specialization interfaces, and interfaces that can define Tasks and TeamMembers(for example, HighPriorityTask and RecurringTask implementing the Task interface). This adheres to the principle as there are smaller and specific interfaces that are created to fit the description of the requirements/various components that are necessary for the project. By doing this, the code is more adaptable and tailored to the necessary guidelines of the project, while remaining manageable.

Low Coupling –

Low coupling is when modules act independently, with changes in one of the modules having a low impact on other modules. Typically, low coupling occurs when a module interacts with other modules through simple interfaces without concern for how the other modules are implemented. Within our code, low coupling is exhibited by the use of the Specialization and TaskInterface interfaces, which provide methods for assigning and removing roles. Overall, low coupling is utilized by separating the requirements and utilizing interfaces that allow certain classes to interact with each other with methods and not through editing of the code directly.

High Cohesion –

High cohesion refers to how closely related and focused the responsibilities of each component of our code are. Each class in our code exhibits high cohesion, as each class only deals with methods that are necessary and related to the responsibilities of said class. For example, TeamMember has methods to join and leave projects but does not have methods to add or remove Tasks which belong to the Project class. By doing this the code is easier to comprehend, test, and extend. This makes the code scalable and resilient so that it is easy to add changes without causing many problems.