Team Members: Shreya Mahesh, Sritha Bhupatiraju,
Japleen Kaur, Srihasa Penchikala, Trisha Nittala, Archita Manasvi

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. Single Responsibility Principle (SRP)

The Single Responsibility Principle (SRP) ensures that each class has one clearly defined responsibility. One instance in our code where we adhered to this principle is the Task class which is solely responsible for handling task-related information and functionality, such as setting and retrieving the task's title, due date, status, and priority. It does not handle any unrelated concerns such as project management or team member assignment. Similarly, the Manager class focuses only on managing projects, specifically overseeing them. It does not manage tasks or team members directly. This separation of responsibilities aligns with SRP, as each class has a single reason to change, making our code more modular and easier to maintain. If project management logic changes, it will only impact the Manager class without affecting the Task class, and vice versa.

2. Open-Closed Principle (OCP)

Our code also follows the Open-Closed Principle (OCP), which states that classes should be designed to be open for extension but closed for modification. For example, the Task class in our implementation provides core functionality for handling tasks, but we extend this functionality with the RecurringTask class, thereby adding the ability to handle recurring tasks without modifying the base Task class. By using inheritance, we can introduce new types of tasks, such as RecurringTask, while preserving the original behavior of the Task class. This ensures that we can add new features or types of tasks in the future by extending existing classes, without altering the existing code, reducing the risk of introducing errors.

3. Liskov Substitution Principle (LSP)

Our implementation adheres to the Liskov Substitution Principle (LSP) by ensuring that subclasses can be used interchangeably with their parent classes without affecting the correctness of the program. For example, the RecurringTask class extends the Task class, and since it implements the ITask interface, it can be used wherever a Task is expected. In our Project class, we can add instances of RecurringTask alongside

regular tasks without any issues, since RecurringTask behaves as a subtype of Task. This allows us to treat different types of tasks uniformly, ensuring the system's correctness and flexibility in handling task extensions.

4. Interface Segregation Principle (ISP)

Our code applies the Interface Segregation Principle (ISP) from SOLID, which suggests that clients should not be forced to depend on interfaces they do not use. Instead, smaller, more specific interfaces are preferred. In our design, the ITask interface defines a clear and concise set of methods that are relevant to all types of tasks, such as getTitle(), getDueDate(), getStatus(), and assignMember(). This interface ensures that both Task and RecurringTask classes provide only the task-related behavior necessary for their functionality, without forcing unnecessary methods upon the implementers. By keeping the interface focused on task-specific responsibilities, we prevent the Project class or any other consumer of tasks from having to rely on methods they do not need or use.

5. Dependency Inversion Principle (DIP)

We applied the Dependency Inversion Principle (DIP) by ensuring that our high-level modules depend on abstractions rather than concrete implementations. In our code, the Project class does not rely on a specific task implementation but interacts with tasks through the ITask interface. This allows us to easily substitute different task implementations, such as RecurringTask or other future task types, without changing the Project class. The flexibility provided by depending on the ITask abstraction, rather than directly coupling with specific implementations, makes our code more adaptable to future changes or new functionality. This separation of concerns follows the DIP by reducing dependencies between classes and encouraging abstraction.

6. Controller (GRASP)

The Controller pattern from GRASP is evident in our code, specifically within the Main class, which functions as a controller by managing high-level interactions and events in the system. The Main class is responsible for creating a project, assigning tasks, adding team members, and updating task statuses, but it delegates the detailed functionality to other classes like Task, Project, and Manager. This separation of responsibilities allows the Main class to serve as a controller, handling system inputs and coordinating interactions without becoming overly complex. By acting as a coordinator between the different classes, the Main class follows the Controller pattern, ensuring that the rest of the system remains decoupled from input management and higher-level orchestration.

7. Low Coupling (GRASP)

We applied the Low Coupling principle from GRASP by designing our classes to have minimal dependencies on each other, which increases the flexibility and maintainability of our system. For example, the Project class does not directly manage the specifics of tasks or team members but instead interacts with them through its methods like addTask, removeTask, addMember, and removeMember. This approach minimizes the dependencies between Project and the concrete implementations of tasks and team members. By loosely coupling these classes, we ensure that changes to one class, such as modifying the task implementation, do not ripple through the entire system, making our code easier to maintain and extend over time.