CS2340 Group 9: Mirza Zuhayr, Benjamin Charest, Sushanth Malipati, Crystal Escoda, Connor Sempf, Zachary Weissman

SOLID and GRASP Write-Up Activity

GRASP:
In our implementation of the task management system, we made sure to follow several GRASP design principles in order to enhance the maintainability and extensibility of our code. Firstly, we made sure to design a system with low coupling in mind, to ensure that changes to the code can be easily made without having to go through the entire system. For example, we created the IndividualTaskRepository interface, which reduces the dependency of the service layer on the data access layer. We also created our system using techniques of polymorphism, as demonstrated by the Task class and the subclasses RecurringTask and UrgentTask, which override the abstract execute() method in Task, and using our IndividualTaskRepository interface. Using polymorphic techniques, we allow for flexible and interchangeable components within our system. Finally, we also make sure to implement the GRASP principle of high cohesion, in which our classes contain all related functionality and data. For instance, the Project class encapsulates all of the information and functions we need for a project in the task management system, making the class more manageable. In our case, we believe we have achieved functional cohesion, since for our classes all essential elements for a single function are grouped together in one place.

SOLID:
In our implementation of the task management system, we have incorporated and adhered to all the SOLID principles stated during lecture to ensure that our design and implementation is flexible, maintainable, and easy to extend. In this portion of the write up, we will be explaining three of the five SOLID principles implemented in our project. The first principle that we'll discuss is the Open/Closed principle which means our task is open for extension but closed for modification. For example: New types of tasks can be added by extending the Task class without modifying existing code. This allows for easy extension without the risk of breaking the existing functionality of the system. Our second principle is the Interface Segregation Principle, which states that we should design principles specific to client requirements. For example: the InterfaceTaskRespository interface focuses solely on persistent operations required by the data access layer, ensuring that classes are only dependent on the interfaces they use. This prevents any clients from being forced to implement methods they don't need and promotes a more cleaner/maintainable code. Finally, the dependency inversion principle ensures that high level modules don't depend on low level modules but on abstractions like the InterfaceTaskRespository. This reduces any coupling between

components, making it easier to modify and replace dependencies without affecting any other part of the system. When it depends on abstraction rather than concrete implementations, our system becomes more flexible to changes. By incorporating these SOLID principles into our design and implementation, we aim to build a maintainable system that can easily adapt to evolving requirements and future changes.