Participating team members: Samantha Mau, Kyle Sanquist, Jackson Li, Jiayi Liang, Riley Oliver, Braden Flournoy

Information Expert

This principle states that each class holds the information it is responsible for. Through this, the responsibility for handling specific pieces of evidence is assigned to the class with the most knowledge to handle that data. Applying this principle helps with low coupling by relying less on other classes.Within our code, only the Project class knows of its tasks and team members, and the Task class knows about its own title, due date, etc.

Controller

Controllers delegate other objects' work, coordinate the activity, and don't do work itself. The use of Controllers can increase the potential to reuse interfaces and reason about the state of a use case. Controllers can be seen implemented through the Project class, since it is responsible for coordination behaviors between tasks and team members.

Low Coupling

Low coupling assigns responsibilities so that classes can interact through defined methods, keeping dependencies to a minimum. If classes were highly interconnected, it would be hard to maintain and understand. A change in one place can cause a ripple effect and make it difficult to see where others have been affected. We used low couplings in our Task class, as it doesn’t know the internals of other classes such as Project.

High Cohesion

Cohesion informally measures how the operations of a software element are related to each other. High cohesion classes focus on a well-defined role, leading to high internal

consistency. Low cohesion classes make it difficult to reuse, maintain, and comprehend. An example of high cohesion is Task handling task logic and TeamMember handling team member logic.

Open/Closed Principle

The open/closed principle states that software entities should be open for extensions but closed for modifications. This can be accomplished through implementing interfaces. The principle also allows code to develop and change over time. The Task class is open for extension by introducing child classes such as RecurringTask and UrgentTask to represent different types of tasks, but Task is closed for modification.

Interface Segregation Principle

The Interface Segregation Principle states that there shouldn’t be large multipurpose interfaces, client’s shouldn’t depend on interfaces they don’t use, and classes should depend on others through the smallest possible interface. These can be accomplished through many client-specific interfaces rather than one general-purpose interface. In our code, each task can have its own implementation of execute() without forcing unrelated tasks to implement methods they don't need.