**Evan Stagl:** We applied the Open-Closed Principle of SOLID when programming Tasks. Since many different types of Tasks may exist, each with different implementations of their common attributes, we decided to make Task an interface. This allows many different types of Tasks to exist while still allowing them to be interchangeable, and more types of Tasks can be created in the future without modifying the Task interface, making Task open to extension but closed to modification.

**Vincent Yang:** Additionally, the Creator pattern emphasizes separation between a system and the creation of its objects. Specifically, the factory method utilizes a method that can be overridden to create the object of a class. In this example, there are different types of TeamMembers with their own specific attributes which have their corresponding Factory implementation to create each object. The Factory class has an abstract method which can be overridden to provide logic to instantiate an object.

**Aveena Singh:** The Liskov Substitution Principle is a SOLID principle that states objects of a superclass must be able to be replaced by objects of its subclass without disrupting the functionality. The RecurringTask extends the BaseTask class and can work in place of BaseTask. RecurringTask additionally overrides the completeTask() method – changing the dueDate and status. This change, however, does not affect the functionality of the code.

**Gus Ivanowski:** Low coupling is a GRASP practice that aims to minimize the connections between entities in the design. In the case of this example, projects can interact with tasks and team members, so there is potential for coupling. To minimize this, methods in implementations of project interfaces that, for instance, remove a member from the project, must call upon a method from the member class instead of changing the member’s data internally, avoiding content coupling.

**Azman Hoque:** Interface Segregation is a SOLID principle that says to implement many smaller interfaces for classes to keep code clearer, especially for clients that don't need access to certain interfaces. In this specific example, The TeamMember class has 3 separate interfaces for its different methods: ContactInfo, ProjectMembership, and TaskCompletion. So, depending on what a given user needs, they can access one of those three smaller interfaces to do what they need. This keeps the code tidy while also reducing chances of coupling or unnecessary dependencies.