Extension Justification for God Cards in Santorini Game

Justification: In the Santorini game, we introduced the concept of god cards to enhance the gameplay and provide unique abilities to the players. The goal was to design an extensible mechanism that allows for the easy addition of new god cards without modifying the existing codebase. To achieve this, we employed several design principles, heuristics, and made conscious design choices.

1. Design Principles and Heuristics:
   a. Low Coupling: We designed the god card system to have low coupling, meaning that each module depends on as few other modules as possible. This is achieved by creating an interface called `GodCard` that defines the common behavior and methods for all god cards. Each specific god card (Demeter, Hephaestus, Minotaur, and Pan) implements this interface and provides its own implementation of the methods. This allows for a clear separation of concerns, as the god card implementations are independent of each other and can be changed or added without affecting the rest of the codebase. Low coupling enhances understandability, reduces the cost of change, and promotes reusability.
   b. Law of Demeter: We applied the Law of Demeter principle, which states that each module should have only limited knowledge about others and should only interact with closely related units. In our design, the `Player` class holds a reference to the selected god card through the `godCard` field of type `GodCard`. The `Player` class interacts with the god card through the interface methods, such as `godCard.canMove()`, `godCard.canBuild()`, `godCard.afterMove()`, etc. This promotes loose coupling and allows for easy switching of god cards without exposing unnecessary details.
   c. High Cohesion (Single Responsibility Principle): Each component in our design has a small set of closely-related responsibilities. The `GodCard` interface defines the common behavior and methods for all god cards, ensuring that each god card implementation has a specific and focused responsibility. The god card classes (`DemeterGodCard`, `HephaestusGodCard`, `MinotaurGodCard`, and `PanGodCard`) encapsulate the specific behaviors and rules of each god card, adhering to the Single Responsibility Principle. This high cohesion enhances code maintainability, readability, and promotes reusability.
2. Design Patterns:
   a. Strategy Pattern: We used the Strategy pattern to encapsulate the different behaviors and algorithms of god cards. Each god card class represents a specific strategy for modifying the game rules. The `GodCard` interface defines the common methods that each god card must implement, such as `canMove()`, `canBuild()`, `afterMove()`, `afterBuild()`, etc. The `Player` class holds a reference to the selected god card, and the game logic interacts with the god card through the interface methods. This allows for easy switching of god cards and adding new god cards without affecting the game logic.
   b. Template Method Pattern: Within each god card class, we used the Template Method pattern to define the common structure of the god card behavior while allowing subclasses to override specific steps. For example, the `canMove()` and `canBuild()` methods in the `GodCard` interface provide a template for checking if a move or build action is allowed. Each god card subclass can override these methods to add its own specific rules or conditions. This promotes code reuse and maintains a consistent structure across different god cards.

3. Alternatives Considered and Tradeoffs:
   a. Inheritance vs. Composition: Instead of using an interface and composition, we could have used inheritance to create a base `GodCard` class and subclasses for each specific god card. However, we chose composition over inheritance to achieve greater flexibility and avoid the limitations of a rigid inheritance hierarchy. With composition, we can easily add new god cards without modifying the existing class hierarchy and avoid the complexity of deep inheritance chains.
   b. Enum vs. Classes: We could have used an enum to represent different god cards instead of separate classes. However, using classes allows for more flexibility and extensibility. Each god card class can have its own state and behavior, and we can easily add new methods or properties specific to each god card. Enums are more suitable for fixed sets of values, whereas classes provide more flexibility for evolving and adding new functionality.
4. Code References:
   a. `GodCard` interface: The `GodCard` interface defines the common methods that all god cards must implement. It includes methods like `canMove()`, `canBuild()`, `afterMove()`, `afterBuild()`, `isSecondBuildAllowed()`, `hasBuiltOnce()`, `resetBuildState()`, `isSpecialMove()`, and `checkWinCondition()`. This interface serves as the contract for all god card implementations.
   b. God Card classes: The specific god card classes (`DemeterGodCard`, `HephaestusGodCard`, `MinotaurGodCard`, and `PanGodCard`) implement the `GodCard` interface and provide their own implementation of the methods. For example, the `DemeterGodCard` class overrides the `canBuild()` and `afterBuild()` methods to allow for a second build on a different cell. The `HephaestusGodCard` class overrides the `canBuild()` method to allow for building a second block on top of the first block. The `MinotaurGodCard` class overrides the `isSpecialMove()` method to enable the special move of pushing an opponent's worker. The `PanGodCard` class overrides the `checkWinCondition()` method to check for the win condition of moving down two or more levels.
   c. `Player` class: The `Player` class holds a reference to the selected god card through the `godCard` field of type `GodCard`. This allows the player to be associated with any god card implementation. The `Player` class interacts with the god card through the interface methods, such as `godCard.canMove()`, `godCard.canBuild()`, `godCard.afterMove()`, etc. This promotes loose coupling and allows for easy switching of god cards.
   d. Game logic: The game logic in the `GameController` class and other related classes interacts with the god cards through the `GodCard` interface. For example, in the `playerMove()` method of the `GameController` class, the `godCard.afterMove()` method is called after a successful move. Similarly, in the `playerBuild()` method, the `godCard.canBuild()` and `godCard.afterBuild()` methods are called to check if a build action is allowed and perform any necessary actions after the build. This allows the game logic to work with any god card implementation without knowing the specific type.

Conclusion: By applying the principles of low coupling, the Law of Demeter, and high cohesion (Single Responsibility Principle), along with the Strategy and Template Method patterns, we have designed an extensible mechanism for god cards in the Santorini game. The use of an interface (`GodCard`) and composition allows for easy addition of new god cards without

modifying the existing codebase. The god card classes encapsulate the specific behaviors and rules of each god card, while the game logic interacts with them through the interface methods. This design promotes flexibility, maintainability, and extensibility, enabling the game to evolve and incorporate new god cards with minimal impact on the existing code structure. The design choices made, such as favoring composition over inheritance and using classes instead of enums, contribute to the overall extensibility and adaptability of the god card mechanism.