Justification for the Santorini Game Design

The Santorini implementation excels in extensibility and modularity through the strategic use of design principles and patterns. This section builds a robust argument for how the system achieves the requirements for **Milestone 6b**, addressing the criteria systematically.

How the Implemented Design Allows Additional (Optional) Actions

The design enables God cards like Demeter and Hephaestus to introduce **optional actions** seamlessly. This flexibility is achieved through the use of the **Strategy Pattern** and well-defined separation of responsibilities.

Key Design Mechanisms

1. **Strategy Pattern**
   - The `GodStrategy` interface serves as the blueprint for implementing God-specific behaviors.
   - Subclasses like `DemeterGodStrategy` and `HephaestusGodStrategy` encapsulate the logic for additional actions.

```
public interface GodStrategy {
    boolean build(Game game, Worker worker, int x, int y) throws Exception;
    void nextPhase(Game game);
}
```

   - `build()` and `nextPhase()` are overridden in specific strategies to enable optional actions.

2. **Dynamic Delegation**
   - The `Game` class delegates control to the `GodStrategy` for each player's actions, ensuring extensibility.

```
public boolean build(int buildX, int buildY) throws Exception {
    if (selectedWorker == null) {
        throw new Exception("No worker has been moved this turn.");
    }
    boolean buildSuccess = currentPlayer.getGodStrategy().build(this,
selectedWorker, buildX, buildY);
    currentPlayer.getGodStrategy().nextPhase(this);
    return buildSuccess;
}
```

   - The `Game` class does not include hardcoded logic for specific God cards. Instead, it relies on the `GodStrategy` to determine the behavior.

3. **State Management**

- Strategies can maintain their internal state to track optional actions, like additional builds.

```
public class DemeterGodStrategy implements GodStrategy {
    private boolean extraBuildUsed = false;

    @Override
    public boolean build(Game game, Worker worker, int x, int y) throws
Exception {
        if (!extraBuildUsed) {
            // Logic for optional second build
            extraBuildUsed = true;
            return true;
        }
        return false;
    }
}
```

How This Eliminates Hardcoding

- No `if` statements in `Game` or `Board` to check for specific God cards.
- Adding a new God card involves creating a new `GodStrategy` subclass without modifying existing classes.

Example: DemeterGodStrategy

1. Implements `build()` to allow an optional second build:

```
@Override
public boolean build(Game game, Worker worker, int x, int y) throws
Exception {
    boolean success = game.defaultBuild(worker, x, y);
    if (success && !extraBuildUsed) {
        extraBuildUsed = true; // Allow a second build
    }
    return success;
}
```

2. Adjusts the game flow using `nextPhase()`:

```
@Override
public void nextPhase(Game game) {
    if (extraBuildUsed) {
        game.setCurrentPhase(Game.GamePhase.END_TURN);
    } else {
        game.setCurrentPhase(Game.GamePhase.BUILD); // Allow another build
    }
}
```

Benefits of the Design

- **Modular Extension:** Adding a new God card with unique optional actions is isolated to the new strategy class.
- **UI Compatibility:** The frontend adapts dynamically based on the state (`strategyState`) exposed by the backend.
- **Seamless Integration:** Existing game flow remains unaffected, ensuring robust scalability.

Why This Works

By encapsulating optional actions in `GodStrategy` subclasses and delegating decisions to these strategies, the design adheres to the **Open-Closed Principle (OCP)**, ensuring flexibility without compromising the integrity of the core classes.

How the Implemented Design Allows God Cards to Change Winning Rules

The design supports customizable winning rules for God cards like Pan through the use of the **Strategy Pattern** and dynamic delegation of responsibilities. Each God card can override the default victory conditions, ensuring flexibility and extensibility.

Key Design Mechanisms

1. **Custom Victory Logic in `GodStrategy`**
    - The `GodStrategy` interface defines a `checkVictory()` method, enabling each strategy to implement its own victory conditions.

```
public interface GodStrategy {
    boolean checkVictory(Game game, Worker worker);
}
```

    - This method is overridden in specific God card strategies to provide custom logic for winning.

2. **Dynamic Delegation to `GodStrategy`**
    - The `Game` class delegates victory checking to the active player's `GodStrategy`. This ensures that the game logic remains flexible and does not need hardcoded checks for specific God cards.

```
public boolean checkVictory() throws Exception {
    for (Worker worker : currentPlayer.getWorkers()) {
        if (currentPlayer.getGodStrategy().checkVictory(this, worker)) {
            System.out.println(currentPlayer.getName() + " wins!");
            gameEnded = true;
            winner = currentPlayer.getName();
            return true;
        }
    }
```

```
        return false;
    }
```

3. **Default and Custom Behavior**
    - The base `GodStrategy` provides standard victory rules (e.g., reaching the third tower level). Specific strategies like `PanGodStrategy` override this method to add unique conditions.

```
public class DefaultGodStrategy implements GodStrategy {
    @Override
    public boolean checkVictory(Game game, Worker worker) {
        return game.defaultCheckVictory(worker);
    }
}
```

```
public class PanGodStrategy implements GodStrategy {
    @Override
    public boolean checkVictory(Game game, Worker worker) {
        // Default win condition: reaching level 3
        if (game.defaultCheckVictory(worker)) {
            return true;
        }
        // Pan's additional condition: moving down two or more levels
        int currentHeight = game.getBoard().getTowerHeight(worker.getX(),
worker.getY());
        int previousHeight = game.getPreviousHeight(worker);
        return previousHeight - currentHeight >= 2;
    }
}
```

How This Eliminates Hardcoding

- The `Game` class does not check for specific God cards or victory conditions using `if` statements.
- New God cards can be added by implementing a subclass of `GodStrategy` and overriding `checkVictory()`.

Example: PanGodStrategy

1. Implements `checkVictory()` to add a custom rule:

```
@Override
public boolean checkVictory(Game game, Worker worker) {
    if (game.defaultCheckVictory(worker)) {
        return true;
    }
    int currentHeight = game.getBoard().getTowerHeight(worker.getX(),
worker.getY());
```

```
    int previousHeight = game.getPreviousHeight(worker);
    return previousHeight - currentHeight >= 2; // Pan wins by moving down
two levels
}
```

2. This logic is seamlessly integrated into the game through dynamic delegation in the `Game` class.

Benefits of the Design

- **Encapsulation of Rules:** Victory conditions are encapsulated in `GodStrategy` subclasses, isolating them from core game logic.
- **Modular and Extensible:** Adding a new God card with custom victory conditions requires implementing a new strategy without modifying existing classes.
- **Adherence to OCP:** The design is open to extension (new God cards) but closed to modification (core classes remain unchanged).

Why This Works

By delegating victory logic to `GodStrategy`, the design ensures:

1. Flexibility to define unique winning rules for each God card.
2. Scalability to accommodate future expansions without altering the core game logic.
3. Separation of concerns, maintaining a clean and modular architecture.

How the Implemented Design Allows God Cards to Move Other Workers

The design enables God cards like Minotaur to move other workers as part of the current player's actions by leveraging the **Strategy Pattern** and reusable utility methods in the `Board` class. This approach ensures flexibility and avoids hardcoding logic in the core classes.

Key Design Mechanisms

1. **Custom Movement Logic in `GodStrategy`**
    - The `GodStrategy` interface defines a `move()` method, allowing each strategy to implement custom movement rules, including moving opponent workers.

```
public interface GodStrategy {
    boolean move(Game game, Worker worker, int x, int y) throws Exception;
}
```

    - Specific strategies, such as `MinotaurGodStrategy`, override this method to add logic for moving other workers.

2. **Reusable Board Methods**
    - The `Board` class provides utility methods to check and execute complex moves:
        - `isOccupied(int x, int y)`: Checks if a cell is occupied by a worker or dome.

        - `swapWorkers(Worker worker1, Worker worker2)`: Moves workers between two positions on the board.
    - These methods ensure consistency and reusability across strategies.

```
public boolean swapWorkers(Worker worker1, Worker worker2) {
    int x1 = worker1.getX();
    int y1 = worker1.getY();
    int x2 = worker2.getX();
    int y2 = worker2.getY();

    workers[x1][y1] = worker2;
    workers[x2][y2] = worker1;

    worker1.setPosition(x2, y2);
    worker2.setPosition(x1, y1);

    return true;
}
```

3. **Dynamic Delegation in `Game`**
    - The `Game` class delegates movement actions to the current player's `GodStrategy`.
    - This eliminates the need to hardcode specific rules for God cards.

```
public boolean moveWorker(int workerIndex, int moveX, int moveY) throws
Exception {
    selectedWorker = currentPlayer.getWorkers().get(workerIndex);
    if (selectedWorker == null) {
        throw new Exception("Invalid worker selection.");
    }
    boolean moveSuccess = currentPlayer.getGodStrategy().move(this,
selectedWorker, moveX, moveY);
    currentPlayer.getGodStrategy().nextPhase(this);
    return moveSuccess;
}
```

How This Eliminates Hardcoding

- The `Game` class does not check for specific God cards or movement rules using `if` statements.
- Adding a new God card with custom movement logic involves creating a new `GodStrategy` subclass without altering the `Game` or `Board` classes.

Example: MinotaurGodStrategy

1. Implements `move()` to add custom logic for moving opponent workers:

```
public class MinotaurGodStrategy implements GodStrategy {
    @Override
    public boolean move(Game game, Worker worker, int x, int y) throws
Exception {
        Board board = game.getBoard();
        // Check if target cell is occupied by an opponent worker
        Worker targetWorker = board.getWorkerAt(x, y);
        if (targetWorker != null && targetWorker.getOwner() !=
worker.getOwner()) {
            // Calculate the pushed position
            int dx = x - worker.getX();
            int dy = y - worker.getY();
            int pushedX = x + dx;
            int pushedY = y + dy;

            // Validate pushed position
            if (board.isWithinBounds(pushedX, pushedY) &&
!board.isOccupied(pushedX, pushedY)) {
                // Execute the move and push
                board.swapWorkers(worker, targetWorker);
                board.moveWorker(x, y, pushedX, pushedY);
                return true;
            }
        }
        return false; // Default to invalid move
    }
}
```

2. This logic is seamlessly integrated into the game without requiring changes to the `Game` class.

Benefits of the Design

- **Encapsulation of Rules:** Movement rules are encapsulated in `GodStrategy` subclasses, isolating them from core game logic.
- **Reusability:** Common board operations (e.g., swapping workers) are implemented in the `Board` class, avoiding duplication.
- **Modular and Extensible:** Adding new God cards with unique movement rules requires implementing a new strategy, not modifying existing code.

Why This Works

By delegating movement logic to `GodStrategy` and utilizing reusable board methods, the design ensures:

1. Flexibility to define unique movement rules for each God card.
2. Consistency and reliability through centralized board operations.
3. Scalability for future expansions without altering the core architecture.

How the Implemented Design Makes Reasonable Decisions About Responsibility Assignment, Avoids Unnecessary Coupling, and Maintains High Cohesion

The Santorini game design ensures that responsibilities are appropriately distributed across its models. The architecture adheres to key software engineering principles like **Single Responsibility Principle (SRP)** and **Separation of Concerns**, resulting in a modular, maintainable, and extensible codebase.

Key Design Principles and Their Implementation

1. **Clear Responsibility Assignment**
    - Each class in the design has a focused responsibility, ensuring that logic is not spread across multiple components.

Examples:

- `Game` **Class:**
    - Manages the overall state and flow of the game, such as phases (`PLACEMENT`, `MOVE`, `BUILD`) and player turns.

```
public class Game {
    private GamePhase currentPhase;
    private Player currentPlayer;

    public boolean placeWorker(int x, int y) throws Exception {
        // Handles worker placement logic
    }
}
```

- `GodStrategy` **Interface:**
    - Encapsulates God-specific rules and behaviors, such as additional actions, movement rules, and victory conditions.

```
public interface GodStrategy {
    boolean move(Game game, Worker worker, int x, int y) throws
Exception;
    boolean checkVictory(Game game, Worker worker);
}
```

- `Board` **Class:**
    - Handles all grid-related operations, such as managing tower heights, worker positions, and cell interactions.

```
public class Board {
    public boolean isOccupied(int x, int y);
    public boolean build(int x, int y);
    public boolean moveWorker(int fromX, int fromY, int toX, int toY);
}
```

2. **Avoidance of Unnecessary Coupling**

   - **Decoupling Through Interfaces:**

     - The `Game` class interacts with `GodStrategy` via its interface, enabling flexibility and extensibility.

```
public boolean moveWorker(int workerIndex, int moveX, int moveY) throws
Exception {
    Worker selectedWorker =
currentPlayer.getWorkers().get(workerIndex);
    boolean moveSuccess = currentPlayer.getGodStrategy().move(this,
selectedWorker, moveX, moveY);
    return moveSuccess;
}
```

   - **Reusability in the `Board` Class:**

     - Common operations like checking cell occupancy or swapping workers are centralized in the `Board` class, avoiding duplication across strategies.

```
public boolean swapWorkers(Worker worker1, Worker worker2) {
    // Handles worker swapping logic for special moves
}
```

3. **High Cohesion**

   - Classes and methods are tightly focused on a single responsibility, making them easier to understand, maintain, and extend.

Examples:

   - **`Player` Class:**
     - Manages player-specific information, such as their workers and assigned God strategy.

```
public class Player {
    private List<Worker> workers;
    private GodStrategy godStrategy;

    public void addWorker(Worker worker);
    public GodStrategy getGodStrategy();
}
```

   - **`Cell` Class:**
     - Represents a single cell on the board, managing its height, occupancy, and selectability.

```
public class Cell {
    private int height;
    private Worker worker;

    public boolean increaseHeight();
    public boolean setWorker(Worker worker);
}
```

Why This Design Works

Reasonable Responsibility Assignment

- Responsibilities are logically grouped:
  - `Game` handles game flow and player interactions.
  - `Board` focuses on grid-related operations.
  - `GodStrategy` defines God-specific rules and behaviors.

Low Coupling

- The `Game` class interacts with other components through well-defined interfaces (`GodStrategy`, `Board`), reducing dependencies and making the system easier to extend.

High Cohesion

- Each class handles a focused set of tasks, such as managing the grid (`Board`) or defining custom rules (`GodStrategy`), ensuring maintainability and clarity.

Code Example: Adding a New God Card

To implement a new God card, the developer only needs to create a new `GodStrategy` subclass. The existing system remains untouched.

Example: HermesGodStrategy

```
public class HermesGodStrategy implements GodStrategy {
    @Override
    public boolean move(Game game, Worker worker, int x, int y) throws Exception {
        // Custom movement logic for Hermes
    }

    @Override
    public boolean checkVictory(Game game, Worker worker) {
        return game.defaultCheckVictory(worker);
    }
}
```

Conclusion

The design's use of clear responsibility assignment, decoupling through interfaces, and cohesive classes ensures a robust architecture that is:

1. Easy to understand and maintain.
2. Open to extension while being closed to modification.
3. Scalable for future enhancements and additional God cards.

Design Justification: Engagement with Design Principles, Tradeoffs, and Patterns

The Santorini implementation demonstrates a thoughtful application of design principles and patterns to balance extensibility, maintainability, and clarity. This section evaluates the design choices, considers alternatives, and justifies the use of specific design patterns.

Design Principles and Patterns Used

1. **Strategy Pattern**
    - Used to encapsulate God card behaviors in the `GodStrategy` interface and its subclasses.
    - **Why It Works:**
        - Facilitates the addition of new God cards without modifying the core logic in `Game` or `Board`.
        - Supports the **Open-Closed Principle (OCP)**: The system is open to extension but closed to modification.
    - **Tradeoff:**
        - Requires creating a new class for each God card, which can increase the number of classes but ensures modularity.

2. **Single Responsibility Principle (SRP)**
    - Each class has a clearly defined responsibility, such as:
        - `Game`: Manages game state and player interactions.
        - `Board`: Handles grid-related operations.
        - `GodStrategy`: Encapsulates God-specific rules.
    - **Why It Works:**
        - Reduces interdependencies, making the system easier to understand and maintain.
    - **Tradeoff:**
        - Requires careful coordination between classes, such as `Game` delegating tasks to `GodStrategy` and `Board`.

3. **Separation of Concerns**
    - Each component (e.g., `Game`, `Board`, `GodStrategy`) focuses on a distinct area of functionality.
    - **Why It Works:**
        - Simplifies testing and debugging by isolating responsibilities.
    - **Tradeoff:**
        - Increased need for integration logic, such as `Game` orchestrating between `GodStrategy` and `Board`.

4. **Encapsulation**

- Internal states, like `strategyState` in `GodStrategy`, are hidden from other classes.
- **Why It Works:**
  - Protects the integrity of each class by preventing direct access to its internal state.

Discussion of Design Alternatives

Alternative 1: Hardcoding God Card Logic

- **Approach:** Implement specific God card logic directly in the `Game` class.
- **Advantages:**
  - Simplifies the initial implementation.
  - Reduces the number of classes.
- **Disadvantages:**
  - Violates **SRP** by overloading `Game` with God-specific logic.
  - Makes the system brittle and harder to extend (e.g., adding a new God card requires modifying `Game`).
  - **Tradeoff:** Simplicity in the short term versus scalability in the long term.

Alternative 2: Use a Rules Engine

- **Approach:** Replace `GodStrategy` with a rules engine to define behaviors dynamically.
- **Advantages:**
  - Centralizes logic for all God cards.
  - Allows non-programmers to define or modify rules.
- **Disadvantages:**
  - Adds complexity to the system.
  - Reduces the clarity of the codebase for developers.
  - **Tradeoff:** Flexibility for defining rules versus increased complexity.

Chosen Design: Strategy Pattern

- Encapsulates God-specific logic in dedicated strategy classes.
- Balances clarity, scalability, and maintainability.

Design Patterns Justification

1. **Why the Strategy Pattern?**
   - Enables seamless extension for new God cards.
   - Decouples core game logic from God-specific rules.
   - Avoids code duplication by delegating common logic to the `GodStrategy` interface.
2. **Why Not Use a Factory Pattern?**
   - A factory pattern could be used to instantiate `GodStrategy` objects, but it is unnecessary in this context because the logic for creating strategies is straightforward.
3. **Why Not Use Singleton for `Game`?**

- While a singleton might ensure a single instance of `Game`, it could restrict testing and parallel games. Dependency injection is preferred for managing the `Game` instance.

4. **Why Not Use Observer Pattern?**
    - The observer pattern might be used to notify the UI of game state changes. However, the existing architecture with direct updates and state management suffices.

Engagement with Tradeoffs

The design reflects thoughtful tradeoffs between flexibility, simplicity, and maintainability:

- **Flexibility vs. Simplicity:** The Strategy Pattern adds complexity but ensures extensibility.
- **Encapsulation vs. Integration:** Responsibilities are encapsulated within classes like `GodStrategy`, requiring `Game` to integrate components effectively.

Conclusion

The chosen design:

1. Uses **Strategy Pattern** to encapsulate God-specific logic.
2. Adheres to principles like **SRP** and **Separation of Concerns**.
3. Balances tradeoffs between scalability, maintainability, and complexity.

This design ensures that the system is robust, extensible, and easy to maintain, addressing the requirements of Milestone 6b effectively.

Improving the Justifications: Alternatives, Principles, and Tradeoffs

The following sections provide an enhanced discussion of design alternatives, engaging with core design principles and tradeoffs to strengthen the justifications for the Santorini game design.

1. Allow Additional (Optional) Actions (e.g., Demeter and Hephaestus)

Alternative Designs

- **Alternative 1: Hardcoded Logic in `Game`**
    - Actions like additional builds could be hardcoded in the `Game` class using conditional statements.
    - **Tradeoff:** Simpler initial implementation but violates **SRP** and makes the system harder to maintain and extend.

- **Alternative 2: Flag-Based State Management**
    - Use flags in the `Game` class to track optional actions for specific God cards.
    - **Tradeoff:** Increased state complexity in `Game` and tighter coupling between game logic and God-specific behaviors.

Chosen Design: Strategy Pattern

- Encapsulates God-specific rules in subclasses of `GodStrategy`, enabling modular and extensible logic.

- **Tradeoff:** Increased number of classes but ensures scalability and maintainability.

Why This Design Works

- Avoids code duplication by centralizing logic in strategies.
- Keeps `Game` focused on state management, adhering to **SRP**.

---

2. Change Winning Rules (e.g., Pan)

Alternative Designs

- **Alternative 1: Centralized Winning Logic in `Game`**
  - All victory conditions are implemented directly in `Game`.
  - **Tradeoff:** Centralization simplifies logic initially but creates a bottleneck for extensibility.
- **Alternative 2: Use a VictoryEvaluator Class**
  - Create a separate class to evaluate victory conditions, which queries game state and player strategies.
  - **Tradeoff:** Decouples logic but introduces another layer of abstraction, potentially overengineering the solution.

Chosen Design: Strategy Pattern

- Each `GodStrategy` subclass defines its own victory conditions via `checkVictory()`.
- **Tradeoff:** Localizes complexity within strategies, making the system easier to extend but requiring careful testing of new strategies.

Why This Design Works

- Delegates responsibility to strategies, ensuring **low coupling** and **high cohesion**.
- Adding new God cards does not affect core logic.

---

3. Move Other Workers (e.g., Minotaur)

Alternative Designs

- **Alternative 1: Add Special Cases to `Board`**
  - Implement movement rules for specific God cards directly in `Board`.
  - **Tradeoff:** Violates **SRP** by overloading `Board` with strategy-specific logic.
- **Alternative 2: Movement Rules in `Game`**
  - Handle all movement rules in the `Game` class.
  - **Tradeoff:** Centralizes logic but tightly couples `Game` to specific God cards.

Chosen Design: Delegation to `GodStrategy`

- Movement logic is handled by `GodStrategy` subclasses, with support from reusable `Board` methods.
- **Tradeoff:** Requires additional strategy classes but ensures modularity.

Why This Design Works

- Encapsulates movement rules in strategies, adhering to **SRP**.
- Reuses board utilities (`isOccupied()`, `swapWorkers()`) to maintain consistency.

---

4. Responsibility Assignment, Coupling, and Cohesion

Alternative Designs

- **Alternative 1: Monolithic Design**
  - Implement all logic in a single class, such as `Game`.
  - **Tradeoff:** Simplifies initial implementation but creates a tightly coupled and brittle system.
- **Alternative 2: Use Multiple Small Classes**
  - Divide responsibilities into many small, highly specialized classes.
  - **Tradeoff:** Improves modularity but increases the complexity of interactions and integration.

Chosen Design: Layered and Modular Architecture

- Core responsibilities are divided across `Game`, `Board`, `GodStrategy`, and other focused classes.
- **Tradeoff:** Balances modularity and integration complexity.

Why This Design Works

- Achieves **low coupling** by delegating responsibilities through interfaces like `GodStrategy`.
- Ensures **high cohesion** by grouping related logic within each class.

Conclusion

These improved justifications highlight the thoughtful application of design principles and tradeoffs, reinforcing the modularity, scalability, and maintainability of the Santorini game implementation.