Justification for Handling State in Santorini

Santorini involves a few key elements whose state must be stored during the game: the players, the current player, the workers' locations, the towers, and determining the winner. The following is a breakdown of where and how each type of game state is stored, along with justifications based on design principles, goals, and trade-offs.

1. Players

- **State to Store**: Player identity (color or ID), the two workers controlled by the player.
- **Where It's Stored**:
  - Stored in the **Player** class, which holds information about the two workers and the player's color or identifier.
- **Justification**:
  - **Single Responsibility Principle (SRP)**: The `Player` class is responsible for the state and behavior of a player. By storing the player's workers within the `Player` class, the responsibility for controlling those workers is encapsulated within the object that directly relates to the player, promoting cohesion.
  - **Expert Principle**: The `Player` class is the best suited to hold the state of the player's workers and their positions, as it directly relates to the player's functionality (i.e., controlling two workers).
- **Alternative Considered**:
  - Storing player and worker states within the `Game` object.
  - **Trade-off**: This would violate SRP, overloading the `Game` class with responsibilities for both game flow and player state. It would also make code harder to extend if new player-specific functionality is added later, such as special powers or player strategies.

2. Current Player

- **State to Store**: The player whose turn it is (either Player A or Player B).
- **Where It's Stored**:
  - Stored in the **Game** class. The `Game` class keeps track of which player's turn it is and ensures that turns alternate properly.
- **Justification**:
  - **SRP**: The `Game` class should be responsible for managing the flow of the game. Managing whose turn it is fits into this responsibility.
  - **Low Coupling**: This state should be in the `Game` class because the current player status affects the entire game flow. Storing it in the `Player` class would couple turn management with player logic, making it harder to manage turn sequencing across multiple players.

- **Alternative Considered**:
    - Store the current player as a global variable.
    - **Trade-off**: This would decrease encapsulation and increase potential for side effects and hard-to-trace bugs.

3. Worker Locations

- **State to Store**: The positions of the two workers (x, y coordinates) for each player on the grid.
- **Where It's Stored**:
    - Stored in the **Worker** class, where each `Worker` object tracks its current position on the grid.
    - The **Grid** class maintains an overall map of the board and knows which workers occupy which cells.
- **Justification**:
    - **Encapsulation**: The `Worker` object should be responsible for storing its own state, as this keeps the responsibilities focused within the object that "owns" the position.
    - **Information Expert**: The `Worker` object knows best about its own position, making it the most appropriate place to store the worker's coordinates. Meanwhile, the `Grid` class should maintain overall knowledge of which cells are occupied.
- **Alternative Considered**:
    - Storing worker locations in the `Game` or `Grid` class alone.
    - **Trade-off**: This would violate the Expert Principle, as the `Game` or `Grid` would need to know too much about individual worker positions, leading to tight coupling between classes.

4. Towers

- **State to Store**: The current height of the tower at each cell (0 to 3), whether or not a dome has been placed on top.
- **Where It's Stored**:
    - Stored in the **Cell** class. Each `Cell` maintains the state of the tower on that cell (number of blocks and whether a dome exists).
- **Justification**:
    - **Expert Principle**: The `Cell` is the natural expert about what is built on top of it. Since each cell can contain either a worker or a tower (or both at different times), the `Cell` object should track the tower's height and whether a dome has been placed.
    - **Cohesion**: By assigning tower state to the `Cell`, we maintain high cohesion, as each cell is responsible for managing everything about its content, including workers and towers.
- **Alternative Considered**:
    - Storing tower data in a separate `Tower` class or directly in the `Grid`.

- **Trade-off**: While this might centralize tower data, it would increase complexity in managing the cell's state. The grid itself shouldn't need to store the individual state of each cell's tower, as it would violate SRP by mixing spatial layout management with the detailed state of individual components.

5. Winner

- **State to Store**: The identity of the player who wins the game (if any).
- **Where It's Stored**:
  - Stored in the **Game** class. The `Game` object is responsible for managing the overall flow and end conditions of the game.
- **Justification**:
  - **SRP**: The game should manage the win condition, as it handles the rules of the game. Storing the winner in the `Game` class ensures that the logic for declaring a winner remains centralized and consistent.
  - **Low Coupling**: By keeping win logic within the game object, we avoid tying the determination of the winner to specific players or workers.
- **Alternative Considered**:
  - Storing the winner state in the `Player` class.
  - **Trade-off**: This would require players to have knowledge of game-wide rules, which is undesirable. The `Player` class should not be responsible for determining when the game has ended.

Design Goals and Principles Referenced

1. **Single Responsibility Principle (SRP)**: Each class has a single, focused responsibility. This ensures modularity and makes each class easier to modify and extend.
2. **Expert Principle**: State should be stored with the object that has the most relevant knowledge and responsibility over that state. This avoids unnecessary coupling between classes.
3. **Low Coupling**: By keeping objects focused on their responsibilities and not exposing internal state unnecessarily, we maintain loose coupling, which enhances flexibility and maintainability.
4. **Cohesion**: Grouping related behavior and state within the same class increases cohesion, making the design easier to reason about and more resilient to changes.