Justification for the Building Action in Santorini

1. Determining What is a Valid Build (Block or Dome)

The build action in Santorini is essential, as it influences the game's flow and ultimately determines if a player can win. The rules specify:

- Players can build either a **normal block** (levels 1, 2, or 3) or a **dome** on top of a level-3 tower.
- A build is only valid if:
    - The target cell is adjacent to the worker's current position.
    - The target cell is unoccupied (no workers or domes).
    - Domes can only be placed on top of level-3 towers, while normal blocks can be built on cells with towers of level 0, 1, or 2.

The implementation must check these conditions before allowing a build.

2. Performing the Build

The build action involves the following steps:

1. **Verify the worker's position** and ensure the build target is an adjacent unoccupied cell.
2. **Check the current level** of the target cell’s tower.
3. **Determine the valid build option** (either a block or dome) based on the current level.
4. **Update the target cell** by incrementing the tower level or adding a dome.

3. Responsibilities and Design Process

Object-Level Responsibilities:

- **Worker**: Initiates the build action by identifying the target cell.
- **Game**: Oversees the overall game flow, ensuring that the worker follows the rules.
- **Grid**: Holds the cells and verifies if the build action is valid by checking adjacency and occupation.
- **Cell**: Maintains the current state of the cell, including the tower's level and occupation status.
- **Tower**: Tracks the current height (number of blocks) and whether a dome is placed.

Method Responsibilities:

- **Worker::build(targetCell)**: Initiates the build process.
- **Game::validateBuild(worker, targetCell)**: Ensures the target cell is adjacent, unoccupied, and can support the chosen build.
- **Grid::isAdjacent(worker, targetCell)**: Checks adjacency.
- **Cell::canBuild()**: Verifies whether the target cell is unoccupied and can support a block or dome.
- **Tower::addBlock()**: Adds a block to the tower if it's below level 3.
- **Tower::addDome()**: Adds a dome if the tower is at level 3.

4. Design Process and Rationale

Single Responsibility Principle (SRP):

Each class has a clear responsibility. The **Worker** class is responsible for initiating actions (like movement and building), while **Game** ensures the rules are followed, and **Grid** verifies spatial constraints like adjacency. **Cell** manages its own state, and **Tower** manages tower construction.

- **Rationale**: This separation of concerns ensures that each class has a well-defined role and reduces complexity, making it easier to test and maintain.

Law of Demeter:

The worker does not directly manipulate the grid or the tower. Instead, it delegates validation to the game and uses high-level commands like `build()`. The worker interacts with the game to ensure that actions remain valid.

- **Rationale**: This minimizes coupling between classes, helping to ensure flexibility. If the game's rules change (e.g., additional constraints), only the game class needs to be updated without impacting the worker or grid classes.

Expert Principle:

Each class is the expert of its own data. The **Cell** is responsible for determining if it is occupied, and the **Tower** is responsible for tracking its height and whether a dome is in place. **Grid** knows the spatial layout of cells.

- **Rationale**: Each object should manage its own data. This ensures that decisions like whether a tower can be built or whether a cell is occupied are made by the objects with the necessary information.

5. Alternatives Considered and Trade-offs

Alternative 1: Assigning all validation to the **Worker** class.

- **Trade-off**: While this would simplify interactions (the worker does everything), it would violate the Single Responsibility Principle by overloading the worker with too many responsibilities. It would also make the code harder to maintain, as any rule changes would require updating the worker class, increasing the risk of introducing bugs.

Alternative 2: Managing tower levels and dome placement at the **Grid** level instead of having a **Tower** class.

- **Trade-off**: This would simplify the class structure but centralize too much functionality within the grid. This approach violates the Expert Principle since the grid doesn’t need to know the specifics of each tower. It would also reduce flexibility if, in the future, tower behavior becomes more complex.