Justification for Building Action of Demeter Worker

Understanding Building Action Requirements

The building action of Demeter in Santorini is subject to several key rules:

- The game state must be 'BUILD' or 'SECOND_BUILD'.
- The player action must be 'BUILD'.
- The Demeter worker must move before it can perform the build.
- For the first build, the Demeter worker can only build on an adjacent square.
- The Demeter worker cannot build on squares that other workers occupy.
- The Demeter worker can either add a level to a tower whose level is smaller than three or place a dome on a three-level tower.
- The Demeter worker may build one additional time, but not on the same space.
- The Demeter worker can skip the optional second build.
- After the first build, the game phase turns to 'SECOND_BUILD'. After the second build, the turn switches to another player. The game state turns to 'MOVE', and player action turns to 'MOVE'.

Given these rules, the system needs to:

1. Validate the build action.
2. Check the target square and execute the build action.
3. Update the game state and player action accordingly.

Responsibility Assignment

Game Class

- **Responsibilities**: Manages high-level game state and enforces game rules.
- **Methods**:
    - **buildBlock(buildPosition)**: call Board class to perform build action
    - **validateBuildPreconditions()**: check if the build meets pre-conditions
    - **isNotPlayerTurn(worker.getOwnerID()):** Check if it's the player's turn
    - **isNotCurrentAction(PlayerAction.*BUILD*)**: Check if it's the build action
    - **performBuild(buildPosition)**: actual method to perform build
    - **postBuildActions(buildPosition)**: method to deal with post-build actions
    - **switchTurn()**: switch turn to another player:
- **Justification**: The **Game** class acts as the coordinator for the overall game flow, making it the logical place to enforce the sequence of play. This adheres to the **Single Responsibility Principle** by keeping the game state management centralized and follows the **Principle of Least Knowledge** by not exposing the intricacies of movement and building rules to other classes.

Board Class

- **Responsibilities**: Manages the spatial layout of the game, including squares and their adjacency.
- **Methods**:
    - **buildAt(worker, buildPosition)**: perform build action on buildPosition by worker
    - **isBuildLegal (worker, from, to)**: Determines if the build is legal
    - **isOutOfBounds(position)**: Check if the target position is out of bounds
    - **isTargetOccupied(position)**: Check if the target square is occupied
    - **isBuildTargetAdjacentAndLegal(from, to)**: Check if the build position is adjacent to the worker's current position, and the target square and does not have a dome
- **Justification**: The **Board** manages the spatial aspects of the game, making it best suited for validating position-based rules. This design choice supports **high cohesion** by grouping related spatial validation logic within a single class.

Square Class

- **Responsibilities**: Represents a single square on the game board and manages its occupancy and construction status.
- **Methods**:
    - **getBuildingLevel()**: call Tower class to get building levels
    - **hasDome()**: check if there is a tower on the square and whether it has a dome
    - **buildBlock()**: call Tower class to perform build action
    - **placeDome()**: call Tower class to perform dome placing action
- **Justification**: Each **Square** is responsible for the state of its segment of the board, including any structures on it. Having **Square** manage these details encapsulates the logic related to building structures, adhering to the **Encapsulation** principle.

Tower Class

- **Responsibilities**: Manages the state of a tower (levels and dome presence) within a square.
- **Methods**:
    - **hasDome()**: check if the tower has a dome
    - **buildLevel()**: Adds a level to the tower.
    - **placeDome()**: Places a dome on the tower.
- **Justification**: The **Tower** class encapsulates the structure's state and behavior, making it the information expert on its condition. This approach aligns with the **Information Expert** design principle, as the **Tower** knows necessary to modify its state according to the rules.

Demeter Class

- **Responsibilities**: Responsible for a particular aspect or rule of the game related to building structures.
- **Methods**:
    - **activateEffect()**: assign values to Demeter class's field, lastBuildPosition, and hasBuiltOnce.

    - **modifyBuildValidation(worker, buildPosition, board)**: Modify the build validation rules based on its special ability. If it is called during the first build, it validates the build as standard. Else if it is called during the second build, it ensures the second build position is not the same as the first build position.
    - **postBuildExecution(game, worker, buildPosition)**: Perform value changed after the first build and the second build of Demeter worker. If it is called after the first build, the method records buildPosition as lastBuildPosition, and assign hasBuiltOnce as true. Else if the method is called after the worker skips the second build or after the second build, the method resets fields of Demeter and calls the game to switch turn.
- **Justification**: The **Demeter** class encapsulates the unique abilities of the Demeter god card in the Santorini game, adhering to the Single Responsibility and Expert principles. It maintains the state and logic for Demeter's special build action—allowing an additional build, but not on the same space. This class serves as the sole authority on the rules and conditions of Demeter's powers, ensuring that all related decisions are localized, which simplifies maintenance and testing. By isolating this functionality, the **Demeter** class enhances the modularity of the game's codebase, making it easier to manage and extend.

Design Principles and Heuristics Considered

- **Single Responsibility Principle**: Ensuring that each class has a single, clear purpose.
- **Encapsulation**: Keeping data and methods that manipulate the data within the same class.
- **Principle of Least Knowledge**: Minimizing interactions between objects to only those necessary.
- **Information Expert**: Assigning responsibilities to the class that has the necessary information.

Alternatives Considered

- **Centralizing Building Logic in the Game Class**: Initially considered to simplify interactions but was ultimately rejected due to the potential for bloating the **Game** class and reducing modularity.
- **Direct Tower Manipulation**: Allowing external classes to directly modify **Tower** states, bypassing **Square**. This was avoided to maintain encapsulation and ensure that all modifications go through appropriate validations.

Design Process

1. Identify building constraints
    a. A worker can only build after a move
    b. A worker can only build on an adjacent square
    c. A worker cannot build on squares that other workers occupy
    d. A worker can either add a level to a tower whose level is smaller than three or place a dome on a three-level tower.
    e. If it is the second build, the worker cannot build on the same position as the first build.

2. Gathering detailed requirements
3. Defining criteria for the solution
4. Generating implementation alternatives
    a. Enforce building rules directly in the **Game** class by maintaining a state machine for player actions.
    b. Utilize the **Board** class for spatial validations (like adjacency) and delegate the sequence of play rules to the **Game** class.
    c. Introduce a **RuleEngine** class to abstract the validation of game rules, used by the **Game** class.
5. Evaluating alternatives
    a. **A** simplifies the enforcement logic at the cost of bloating the **Game** class.
    b. **B** separates concerns but may lead to fragmented rule validation logic.
    c. **C** centralizes rule enforcement, potentially improving maintainability but adding complexity.
6. Making the decision
    a. Choosing **B** for its balance of concern separation and direct rule enforcement. The **Game** class maintains the sequence of play, ensuring a worker has moved before allowing a build action. The **Board** and **Square** classes handle spatial validations like adjacency and level building restrictions.
7. Implementation
    a. Enhance the **Game** class with state tracking for each player's turn phase (Move or Build).
    b. Implement adjacency checking in the **Board** class, which verifies if a build position is adjacent to the worker's current position.
    c. Update **Square** and **Tower** to handle structural changes, ensuring the action adheres to the rules (e.g., only adding a dome on a three-level tower).

Object-level interaction diagram

Participants:

- **Player**: The player initiating the build action.
- **Worker**: The worker performs build action.
- **Game**: Coordinates the high-level game flow and rules.
- **Board**: Manages the game's spatial layout and squares.
- **Square**: Represents a single square on the board, managing occupancy and building status.
- **Tower**: Manages the structure's levels and dome.
- **Demeter**: God Card class to perform special ability.

Sequence of Interactions:

1. **Player Initiates Build**:
    - Interaction: The player decides to build on a specific square adjacent to their worker's position and signals this intent, likely through a user interface action which is then relayed to the game logic.
    - **Player -> Game: buildBlock(buildPosition)**
2. **Game Validates Player Turn and Action**:
    - Check if it's currently the player's turn and if the build action is allowed at this stage.
        - **Game: validateBuildPreconditions()**
            1. Game: currentWorker!=null
            2. Game: gamePhase == GamePhase.BUILD || gamePhase == GamePhase.SECOND_BUILD
            3. **Game: isNotPlayerTurn(worker.getOwnerID())**
            4. **Game: isNotCurrentAction(PlayerAction.BUILD)**
3. **Game Requests God Card to Validate Build** (if validation passes):
    - If the player's turn and action are valid, the Game firstly get god card to validate special ability, then instructs the Board to execute the building action.
        - **Game: performBuild(buildPosition)**
            1. godCard = godCards.get(currentPlayer)
            2. **Game->Demeter:modifyBuildValidation(currentWorker, buildPosition, board)**
            3. **Demeter->Board: isBuildLegal(worker, worker.getPosition(), buildPosition)**
4. **Board Verifies Build Legality**:
    - Within the **Board** class, several internal checks are performed sequentially to validate the build action:
        - **Board -> Board: isOutOfBounds(buildPosition)** to ensure the target position is within valid boundaries.
        - **Board -> Board: isTargetOccupied(buildPosition)** to ensure the target square is not occupied by another worker.
        - **Board -> Board: isBuildAdjacentAndLegal(from, to)** combines the above validations with an additional check to ensure the target position is adjacent and legally permissible for a build.

5. **Game Requests Board to Perform Build Action:**
    - **Game->Board: buildAt(currentWorker, buildPosition)**
    - If all validations pass, the **Board** proceeds to instruct the relevant **Square** to execute the build action, which could be adding a level or placing a dome based on the tower's current state:
        - **Board: executeBuildAction(worker, buildPosition)**
        - **Board -> Square: buildBlock()** or **Board -> Square: placeDome()**
6. **Square Updates Tower** (if the build is legal):
    - If the build action is deemed legal, the Square tells its Tower to either add a level or place a dome, based on the worker's action and the current tower state.
        - **Square -> Tower: buildLevel()** or **Square -> Tower: placeDome()**
7. **Tower State Modification:**
    - The **Tower** object updates its state accordingly. If adding a level, **levels** are incremented unless it's already at maximum height (3) and does not have a dome. If placing a dome, **hasDome** is set to true provided the tower is at the correct height.
8. **Game Do Post Build Actions**:
    - After the **Square** and **Tower** update successfully, a confirmation is sent back to the **Game** to finalize the build action and update the game state.
    - **Game: postBuildActions(buildPosition)**
    - **Game->Demeter: postBuildExecution(this, currentWorker, buildPosition)**
    - Demeter class should check if the build is the first or second, to perform different executions.
        - If hasBuiltOnce==false, **Demeter: firstBuildAction(buildPosition, game, worker)**. This method will reassign fields and send a new game phase back to the Game class, gamePhase=GamePhase.SECOND_BUILD.
        - Else if hasBuiltOnce==true, **Demeter: endBuildPhase(game, worker)**. This method resets fields in Demeter, and asks the Game class to do **switchTurn().**