1. 3 key classes:

a. Worker class

The Worker class is an essential part of the fundamental gameplay of Santorini. It is an individual piece (1 Player has access to 2 Workers) controlled by each Player. It is a fundamental unit in gameplay that moves across the board and performs building actions per turn to eventually win the game.

Creating Worker as a class is justified because:

- Each Worker holds state: its owner, canMove and currentCell are distinct attributes that persist and evolve across turns.
- It encapsulates behavior relevant only to the game piece: future methods such as getCurrentCell, or interaction with SpecialPower would logically belong here.
- The game enforces individual behavior rules per worker. For example, only a specific worker (not both) can act each turn (i.e only workers that moved in that turn), which would be hard to track if workers were just anonymous values in a list.

Thus, Worker is a domain object, not a utility holder making it appropriate and necessary as a class.

Alternative solution:

Hardcode movement and building logic into the Worker class, allowing it to validate and execute actions internally based on its position and surroundings.

b. Cell class

The Cell class represents a single tile on the Santorini game board. Each Cell holds its position (x, y), any structures built on it (Building and optionally a Dome), and may also be occupied by a Worker. The class extends JPanel, enabling it to serve as a visual and interactive element in the Swing GUI. This integration of game state and UI makes Cell a core component for both gameplay logic and interface rendering.

This class is necessary as a standalone object because it encapsulates multiple independent properties (position, occupancy, highlighting, and structural content), each of which evolves dynamically throughout the game. If this functionality were handled solely through methods or a generic board data structure, the design would become rigid, difficult to update, and hard to synchronize with the graphical interface. Additionally, since each cell must track its own state (occupied or not, highlighted or not, has dome or not), a class representation enables clean encapsulation and object-specific updates through repainting and logic validation.

c. TurnManager class

The TurnManager class is responsible for controlling the flow of the game. It handles the sequence of phases (movement, building, god powers), player switching, turn-based UI updates, and win/lose conditions. This class is essential because it acts as a coordinator to organize interactions between multiple classes such as Player, Worker, Cell, Board, and Action.

It is appropriate to implement TurnManager as a full class rather than distributing its logic across Player, Game, or Board, because it:

- Encapsulates the rules and sequencing logic that are essential to individual player or board behavior.
- Maintains stateful data across turns (current player, last moved worker, floor counters).
- Simplifies unit testing and debugging by centralizing turn logic.

If this logic were split into methods in other classes (Player.move() or Board.processTurn()), the result would be high coupling and reduced maintainability.

2. 3 General key relationships in our class diagram:

Board and Cell: Composition

This relationship is a composition because Cells are fully owned and controlled by the Board. They are created as part of the board's initialization and have no independent lifecycle outside of it. If the Board is deleted or reset, all its Cells are destroyed with it. This is a whole-part relationship: a Cell cannot exist meaningfully without a Board. Therefore, composition is appropriate. It expresses a strong lifecycle dependency.

Player and Worker: Aggregation

This relationship is an aggregation because while a Player owns 2 Workers during gameplay, the Workers are not tightly bound to the Player's lifecycle. For example, even if a player is removed from the game, the Workers might still exist momentarily during cleanup or win condition checks. They also maintain their own identity (currentCell, owner, etc) and interact independently with the board. Therefore, aggregation accurately models the "has-a" relationship without implying destruction of Workers when a Player is removed.

Action and Cell: Association
This is a simple association, not an aggregation or composition, because Action instances do not own or manage the Cell they affect. The Cell exists independently of the Action, and multiple actions can affect the same cell over the course of the game. The Action merely references a Cell as a target during execution (e.g., for moving or building). The association is temporary, contextual, and unidirectional, which makes aggregation or composition inappropriate.

3. Inheritance

The Cell class extends JPanel to integrate directly with the Java Swing framework. This inheritance is justified because Cell is fundamentally a visual component and must override the paintComponent(Graphics g) method to draw game elements like a Worker circle. Unlike domain-specific inheritance (between SpecialPower and its subclasses), this is a case of framework-driven inheritance: extending a GUI base class to gain rendering and event-handling capabilities. This avoids reinventing UI behavior and leverages polymorphism where it adds actual value, namely, in custom drawing and display updates.

But within the domain model itself, Cell does not use inheritance. There is no need for different types of cells (OceanCell, UsedCell) in the base Santorini game. it would make sense to create a superclass like GameTile or BoardElement if the system had different types of board tiles with distinct rules, such as impassable tiles, trap tiles, or power-up zones. However, in the base Santorini game, every cell on the board is functionally identical in structure and rules. All cells support the same operations: holding a worker, building up levels, domes, and tracking state like occupancy or highlighting. This eliminates the need for polymorphism, rendering inheritance both unnecessary and potentially confusing.

4. 2 sets of two cardinalities:

a. TurnManager and Worker: 0..*

Holds a list of the current player's workers for selection and highlighting.

Justification of 0 (Lower Bound):

At certain points in the game lifecycle (before the first turn starts, after game over), the TurnManager may not hold a reference to any workers. This assignment only happens when a new turn starts. Until that assignment, or if something goes wrong, the list could be empty or uninitialized.

Justification of * (Upper Bound):

In standard gameplay, the number of workers held is always 4 (because each player has exactly 2 workers). But to keep the model flexible and general, * is used to allow for:

- Potential future rules with more than 2 workers per player
- Code reuse across different modes (e.g., training simulations, AI testbeds)
- Cleaner UML, since hardcoding 2 in the cardinality would be too rigid for future extensions.

2. Game and TurnManager Cardinality: 0..1

The Game class creates the TurnManager and passes all required components (players, board, ui) to it. However, the TurnManager is not stored as a field inside Game — it's a local variable within startGame():

TurnManager gm = new TurnManager(players, board, ui);

Because the Game does not retain a persistent reference to the TurnManager, this is a transient association, not aggregation or composition. Since TurnManager is instantiated and used only inside the startGame() method, the Game holds no ongoing reference to it. If we were to extend the game logic (pause/resume, replays, turn history), we could store TurnManager as a field, which would change this cardinality to 1. But the TurnManager does not know about the Game class at all — there is no reference to Game inside TurnManager. Therefore, from TurnManager to Game, the cardinality is not applicable. This is a unidirectional association from Game to TurnManager.

5. Design Patterns

Mediator Pattern

The TurnManager class strongly applies the Mediator Pattern by acting as the central coordinator for all game components. Rather than allowing Player, Worker, Board, Cell, Action, and the UI to interact with each other directly, the TurnManager manages and sequences these interactions. For example, it determines which player's turn it is, highlights valid Cells for movement or building, passes control to Action objects for execution, updates the game state, checks win or loss conditions, and communicates relevant changes to the user interface. This centralized control flow reduces tight coupling across components, making the system easier to modify and extend. If a new god power or turn rule is introduced, it can be integrated by updating the TurnManager without needing to change the internal behavior of individual classes like Player or Board. By encapsulating the control logic in one place, the TurnManager enhances the modularity, maintainability, and clarity of the entire system, exemplifying the Mediator Pattern in practice.

6. Alternative Solutions

1. God power logics in TurnManager
   Hardcode god power logic directly into the TurnManager class using long if or switch statements that check the current player's god type before every move or build.

   But this method violates the Open/Closed Principle and breaks modularity of god powers, as they would be no longer reusable components.

2. The board and cell management.
   Represent the entire board as a 2D array of integers (e.g., int[][] boardHeights) and use integer values to track cell height, dome status, and worker presence.

   But why this was a bad solution is we lose object orientation by reducing Cell and Board to primitive types, we abandon the rich relationships and encapsulation that our current Cell, Worker, and Building classes provide.

3. Player class
   Make Player a static utility class with only global methods and variables, storing all players in a shared list and managing workers and god cards via static references.
   But this would have resulted in tight coupling where every part of the code would now depend on global Player state (e.g., Player.currentPlayer, Player.players[1]).