My object model consists of 4 classes. There is a GameSystem class, Player class, Board class, and Worker class.

I utilized the Controller design pattern. The GameSystem class acts as a controller of the entire game; the methods in this class are able to be called by the user who plays the game, and it delegates the main computation steps to other classes. The GameSystem class has 7 fields: the *gameBoard* of type Board represents the game grid during gameplay, the *players* represent the list of Players in the game at this time, the *currPlayer* is the index of the current player in *players*, the *currWorker* is the index of the current player's current worker at that time of gameplay, the *numPlayers* represents the total number of players in the game (currently 2), and finally the 2 boolean values, *win*, which has value False up until the game state changes to reflect a player winning the game, and *lose*, which has value False up until the game state changes to reflect a player losing the game.

There are 4 main important methods in the GameSystem class. The method *move(..)* will attempt to move the current player's designated worker to the designated (x, y) position, by calling a similar method *playMove(..)* in the Board class. After every successful *move(..)*, the game state is checked to see if any player has won. If a player has won, it will be reflected in the stored boolean field *win*. After every unsuccessful *move(..)*, the game state is checked to see if any players have lost, which will be reflected in the stored boolean value, *lose*. Similarly, the method *build(...)* will attempt to move the current player's current worker (worker that just completed *move(..)*), by calling an analogous method *buildBlock(...)* in the Board class. Again, if the build was unsuccessful, the game state is checked to see if any players are "locked in", thereby losing the game, which is reflected in the *lose* boolean field. Then, the method *updatePlayer(..)* will update the stored value of *currPlayer* after every consecutive *move(..)* and *build(..)* to reflect that it is the other player's turn. The method *placeWorkers(..)* is for the beginning of the game, when both players need to set both their workers at their picked coordinates on the gameBoard.

The Board class consists of 2 fields: the *gameBoard* grid represented by a 2-dimensional integer array, and the *workerJustMoved*, which represents the Worker that just completed a *playMove(..)* successfully. The Board class holds the methods that were delegated from the GameSystem (controller) class. The method *initializeWorkers(..)* is called from the GameSystem's *placeWorkers(..)*. The 2 methods *playMove(..)* and *buildBlock(...)* are the methods responsible for actually moving or building during the game. Then, there are private helper methods to help check if a move/build is valid. For example, *isAdjacent* checks that the 2 (x, y) coordinates given are adjacent to each other. The method *isOccupied(..)* checks that the given (x, y) coordinates are occupied or not, and *inBounds* checks that the coordinates are all in bound of the 5x5 grid. Since the representation of the gameBoard is a 2-dimensional integer array, the heights of the "blocks"/"towers" are represented by the values in gameBoard[x][y], where a "dome" is represented by a value of 4 in gameBoard. The methods *getCurrentHeight(..)* and *addBlock(..)* are reflective of the integer array grid, for adding blocks/domes. Finally, again following the Controller heuristic, detecting winning or losing of the game from GameSystem, is by the methods in the Board class– *winGame(..), loseGameMove(..)* and *loseGameBuild(..)*. There is only 1 win detection method since a player can only win by moving onto a 3-level tower, but there are 2 lose detection methods since a player can lose by unsuccessfully completing a move or a build operation.

The Player class consists of the player's id, and the list of workers as fields. Then, there are methods to obtain the relevant fields.

The Worker class consists of a worker's id, and their position. A worker's position is represented by an integer array of size 2, where the x-coordinate is the first value and the y-coordinate is the second value. Then, there are methods to obtain a worker's position and set a worker's position.

## Extensibility

The goal was to make the Santorini game extensible, allowing the players to incorporate "God Cards" that change a part of the functionality of the game. My decision resulted in a design choice that closely resembled the template method pattern. The difference is that the template method pattern usually has an abstract class where some or all (abstract) methods in the template method are overwritten by the classes that extend it. My GameLogic class resembles the template method abstract class. In the beginning, the GameLogic class was declared abstract, and each successive GodCard class (eg. Minotaur) would extend it, and overwrite some methods pertaining to that god card (eg, isOccupied(..)). To make instantiating it more easily, I declared GameLogic to be a concrete class.

The GameLogic class holds the methods that are common across game play, with "placeholders" in the form of methods, that represent the changed functionality that comes with each respective god card. This serves as an advantage since all the core game logic methods are consolidated into a single class, thus, providing ease in switching out different play functionalities. Hence, results in high cohesion. On the other hand, this results in high coupling between the Board class and the GameLogic class, since much of the validity checking for moving or building requires knowledge of the current gameboard. For example, the GameLogic class contains the methods *playMove(..)* and *isValidMove(..)*, where the core logic of the "move" function is in those methods, and hence require *gameBoard* to be a parameter. Similarly, it contains the methods *buildBlock(..)* and *isValidBuild(..)*. Additionally, the GameLogic class contains other common gameplay functions, such as *winGame(..), winDefault(..), isOccupied(..), isAdjacent(..).*.etc. Then, each specific god card implementation would be its own concrete class, extending GameLogic, and overwriting the specific methods within the move, build, or win functionality. These overwritten methods reflect the specific changed functionality due to the respective god card. For example, the Minotaur class overwrote *isOccupied(..)*, that gets called within *isValidMove(..)*, since the Minotaur god card states a move into an occupied space is valid, if given requirements are met. The method *winDefault(..)* is called within *winGame(..)*, and will be overwritten for the god card Pan, since Pan has different winning rules.

Another reason for not declaring GameLogic to be abstract, is due to attempting to minimize "empty" code. If the methods to be switched out for changed functionality for all god cards are all declared to be abstract, then the god card classes that only change some functionality, will have "empty" implementations for the methods that remain the basic implementations. For example, since Minotaur changes the *move(..)* functionality, and Pan changes the *winGame(..)* functionality, Minotaur would require an "empty" implementation of *winGame(..)*.

Alternatively, the Strategy or Decorator design pattern could be used. In my opinion, the Strategy pattern would result in too much code duplication. It requires an interface, where methods with changed functionality would simply be implemented in each class. Since the interface cannot implement any methods, the common shared code checks (eg. for moving/building) would have to be included in each god card class' implementation, resulting in duplication.

## Design Pattern

As described above, there was no specific design pattern utilized, but the GameLogic class drew inspiration from the template method pattern. Rather than have the class be abstract, it was a concrete class, with all basic move/build/win functionality implemented. Then, each successive god card class would extend GameLogic, overwriting whichever method corresponded to their respective changed game functionalities. The reason for this design choice is to increase cohesion, and provide ease in switching out different play functionalities, but also to minimize "empty" code.