Santori Game Design Specification:

In order to appropriately model the game Santori according to the given rules we will need to build a few separate parts and allow them to interact with each other. Firstly a *Client* will be needed that will be the main interface between the players and the game board to allow them to view and interact with their pieces. Any communication between the *Clients* and the *Game Server* should go over a JSON communication protocol either on STDIN or over a socket, depending on if we want to support remote functionality. The *Game Server* will be running at a known process handle or address and port where a client can connect to and send information. At the same time the *Game Server* is responsible for relaying information between clients and the *Game Boards* they are playing on. A *Game Board* will handle the individual turns of the game, showing the *Clients* what the board looks like, and deciding if the game has ended along with who the victor is. It will also be responsible for reporting back to the *Client* whether their chosen move is valid or not. All communication from the *Game Board* to the client will be relayed through the *Game Server* as the *Game Board* and *Clients* do not have a direct connection. The *Game Board* will have a collection of *Cells* where each *Cell* can be accessed from a coordinate on the board. These *Cells* will be responsible for tracking the state of each location on the board i.e whether a player is present, whose player is present and the height, if any of a building on the *Cell*

To make sure communication between the *Client* and *Game Server* are successful the there must be a few actions taken when a *Client* connects and starts playing a game. When a *Client* initiates a connection to a *Game Server* it must provide some identifying string that the *Game Server* will ensure is unique, if not it will request a new one. When the *Game Server* creates a *Game Board* instance for two *Clients* to play on it generates a unique identifier for that *Game Board* and tells that to *Clients*. The *Game Server* will maintain a mapping of *Client* and *Game Board* ids to *Game Board* instances. All subsequent communication intended for the *Game Board* must include the *Client's* unique identifier and the the *Game Boards* unique Identifier.

Client:
- cid: *CID*
  - Client ID
- sid: *SID*
  - Session ID from Game Server
- get_action(): JSON
  - Get *Input* from player and parse into JSON
- send_msg(JSON): void
  - Send JSON to Game Server
- recv_msg(): JSON
  - Receive JSON from Game Server

Game Server:
- dict[(*CID*, *SID*)] = Game Board
  - Stores collection of on-going games
- recv_msg(): JSON
  - Checks or incoming message
- do_turn(*CID*, *SID*, *Action*, (*N*, *N*), (*N*, *N*)): boolean
  - Tries to do a turn, returns success of that action
- report_winner(*CID*, *SID*): void
  - Sends a win message to *CID*

Game Board:
- dict[(N, *N*)] = Cell
  - Collection of cells that represent the board
- turn_count: int
  - Keeps track how many turns have occured
- turn_phase: *TurnPhase*
  - Which *TurnPhase* is it
- __is_valid_move((*N*, *N*), (*N*, *N*)): boolean
  - Checks that a move from p1 to p2 is valid
- __is_valid_build((*N*, *N*), (*N*, *N*)): boolean
  - Checks that a worker at p1 can build at p2
- __build((*N*, *N*)): void
  - Adds 1 to height of cell at p1
- __move((*N*, *N*), (*N*, *N*)): void
  - Moves player from p1 to p2
- is_over(): void
  - Checks if a game over condition has been met
- get_current_player(): *CID*
  - Gets the current players *CID*
- list_of_players: [*CID*, …]
  - Keeps track of *CIDs* that are playing on this board
- do_turn(*Action*, (*N*, *N*), (*N*, *N*)): boolean
  - Does the give *Action* targeting the 2 given points

Cell:
- height: int < 4
  - Height of the cell
- worker: string
  - *CID* of player whose worker is on the cell | None

Terms:
- *CID: Client ID*
- *SID: Session ID*
- *N:* [0, 5]
- *Input: coded against player Input*
- *Action: "MOVE" | "BUILD"*
- *TurnPhase: "MOVE" | "BUILD"*