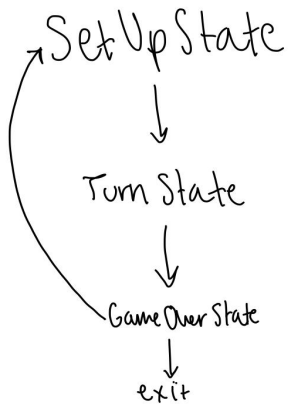
1. **State:**

We used a state pattern to control the overall flow of the game. GameManager is the context class that manages the game states and delegates actions to the current state. The GameManagerState is the abstract state base class. SetUpState, TurnState, and GameOverState are the concrete state classes that implement the specific behaviors associated with each game state. These classes work together, allowing the GameManager to change its behavior dynamically based on the current state and delegate the execution of certain actions to the corresponding state object. SetUpState initializes the game board with the specified types of players and certain settings enabled or disabled (redo/undo and score). TurnState checks if there is a winner, and if not, executes a turn, undo, or redo depending on the input. However, if there is a winner, the game state does not execute a turn and goes to GameOverState. Depending on if the player wants to play again the program either goes to the SetUpState again to start the game over or exits.



2. **Template:**

Our human, random & heuristic player objects were implemented to build turns based on their type. We used the template pattern for these 3 types of players in order to reduce the amount of repeated code needed to implement 3 different, but somewhat overlapping algorithms.
There was an overarching Player (abstract base) template class which HumanPlayer (concrete) and AIPlayer template inherit from. Then, AIRandomPlayer and AIHeuristicPlayer, both concrete classes, inherit from AIPlayer. The Player class implements building the valid potential turn list since all types of players do this the same way. The player class leaves the implementation of build_turn to each player type. Thus, HumanPlayer implements build_turn by asking for user input. Because AIRandomPlayer and AIHeuristicPlayer's build_turn overlaps, the AIPlayer template class implements build_turn and the way to select a build location. Then AIRandomPlayer and AIHeuristicPlayer implement select_player_and_placement. This allows for build_turn to be called in the game class on any type of player without specifically knowing the player's type.

3. Memento:

The memento pattern was used to capture the previous game states in order to implement undo/redo functionality, without revealing all the details of the game object. After each turn, a deep copy of the DoTurnCommand object is made, which contains both the turn object that was just executed and the board. The memento object is then added to the undo stack list; if a turn is undone, the state of the board is updated to the previous board and the memento object is added to the redo stack list. This allows the player to go back and forth with undos and redos. Our originator class is DoTurnCommand: this was chosen as opposed to the board class because a DoTurnCommand holds both the board and the turn, allowing for more flexibility. For example, you can also print a turn summary with each redo and undo. The Memento class is CommandSnapshot. The Caretaker class is Game, which is responsible for storing and managing the mementos in the undo and redo stacks, and has methods to restore the Board state in execute undo/execute redo using the mementos.

4. Command:

We used the command design pattern to capture building and doing a turn (executing a turn on the board) into stand alone objects that contain all the information about each request. This allowed us to encapsulate and decouple these actions in our application, while also supporting the undo/redo functionality by allowing us to create and store mementos of the do turn command object (as explained above). Thus, our Command class acts as the command abstract method. TurnBuilderCommand and DoTurnCommand are concrete command classes. In DoTurnCommand for example, its invoker is the Game class and its receiver is the Board class.