CPSC 327 HW 6
Julian Lee, Jeffrey Gong
Santorini Writeup

Work experience and lessons learned

Julian: The most interesting part of the project for me was figuring out how to structure the code and employ various design patterns. There were often various design patterns that could solve similar problems (e.g. we discussed structuring the different types of players using a factory vs. strategy pattern). Also, I practiced the process of refactoring code to achieve increasing amounts of encapsulation (e.g. only GamePosition and Board need to interact with Position in our final version). I had to undergo many rounds of refactoring after developing the UML: in the future, I think further contemplation of design decisions before beginning to code would have made the process faster

Jeffrey: The most rewarding aspect of the project was definitely implementing the Heuristic AI player. It was a super cool way to see all the OOP patterns put into use and see the different classes we had created tie together. I learned that starting with the UML and thinking deeply about what design patterns we could implement made the entire engineering process much smoother and faster than it would've been without those initial discussions. This was a great culminating project to tie in everything I've learned in this class from the 6 pillars of OOP, to design patterns, testing, etc.

Design Patterns Used

Command pattern

We used the command design pattern to represent moves as ExecuteMove objects. ExecuteMove objects have an execute method that performs the given move on the board (both moving the specified worker and building appropriately). This object also supports an undo method to restore the board to the prior state before this move. The command design pattern allowed us to easily implement undo/redo by maintaining the move history as a list of ExecuteMove objects.

Template Pattern

The template pattern was used to create a generic abstract TwoPlayerGame class supporting undo/redo functionality; this can be reused for future game classes beyond Santorini. The get_next_turn() method of TwoPlayerGame consists of the following potential steps: performing a move, undoing the last turn, and redoing the previous move. The _undo_step(), and _redo_step() methods from TwoPlayerGame are directly used by the Santorini class, and the Santorini class then overrides __init__(), get_next_turn() and _perform_move() to provide game-specific functionality.

Iterator Pattern

We created an object GamePositions containing a 5x5 collection of Position objects. This object is iterable: more specifically, a user can get an iterator for all valid positions neighboring a certain tile on the grid. All positions contained within the 5x5 board are considered valid positions. This assists the Board class for tasks such as testing the viability of neighboring build directions in the method worker_has_possible_move_and_build.

Factory Pattern

We use a PlayerFactory to determine whether each player is human, random, or heuristic. The HumanPlayer, HeuristicPlayer, and RandomPlayer classes all have the same methods and extend the same Player parent class. Therefore, after using the PlayerFactory class to determine the specific subclass that should be used, the Santorini game class can operate the same regardless of which player type is used.