Design Changes:

ColorablePiece:

ColorablePiece ensures that a piece's appearance/colour is stored within themselves. In the previous iteration of Santorini, the color of tiles was handled within the GameBoardPanel. However, the GameBoardPanel had too much responsibility, not only having to construct the board, but also having to handle the colours of each tile. That approach also did not allow for much extensibility, as if we needed to add a new type of piece, we would also need to edit the code within GameBoardPanel. With the addition of ColorablePiece I ensure that GameBoardPanel adheres to the single responsibility principle. This also meets the Liskov substitution principle as ColorablePiece can be used where piece has also been used instead.

GameController (Potential Rework):

GameController was a class I considered reworking, as it has many responsibilities. This includes not only selecting actions for the worker to complete, but also some game logic like detecting win conditions, and generating actions for the player. Not only does this violate the single responsibility principle, but also the open close principle, as if I want to add a new type of action, I need to alter the existing methods of GameController for it to work. Whilst I could have distributed some of these functionalities to other classes such as the game or player classes, due to the scope of the project and time constraints, I decided it was more beneficial to focus on polishing the core mechanics of the game, especially since the current implementation of game controller did hinder the addition of the extensions I chose to implement.

New Classes:

Zeus:

Zeus was implemented using the template methods pattern that was already established. With this pattern, functionality such as displaying the god's name during popups were easy to implement and ensure that existing methods do not have to be altered when adding new gods in the future (open close principle).

PlayerTimer:

PlayerTimer was implemented to update the player time that gets displayed to the screen. PlayerTimer utilises the observer pattern as it needs to implement ActionListener. This allows it to observe timer events and count down whenever an event gets triggered. Because I want it to count down every second, it will be triggered every 1000 milliseconds. A potential design pattern that could have been implemented is the template method pattern. For this to work I could have made PlayerTimer an abstract class and construct methods that all timer classes could follow. This would allow for the addition of different types of timers, for example a timer that goes at 1.5x speed. However, for simplicity I decided to not implement it that way, as player time countdowns should remain the same for all players and potential game modes.

TimerPanel:

The design pattern I utilised in the TimerPanel was the iterator pattern to iterate through each player's timer. Whilst I did not formally use an iterator, the nextPlayer function worked in a similar way to traverse the list of timers in order. The façade pattern was also used, as TimerPanel acts as a centralised place to handle some of the logic for the timers e.g. switching between players. Instead of holding the count down timers in a single data structure, an alternative approach would be to only allow for 2 timers to be activated, for player 1 and player 2. Whilst this approach would have been suitable, since I was not implementing extensions that alter the number of players, this also would prevent the possibility for the timer working for more players at once in the future.

Self-Directed Extension Rationale:

I chose to use helpful as my human value of choice. The value of being helpful promotes qualities such as collaboration, and a positive and supportive environment. From a societal context, being helpful can strengthen communities, and lead to a diverse range of people coming together. Without helpfulness, people would be very segmented, and we would progress slower as a society.

This idea is manifested within my extension of having 'helpful tokens'. This allows for inactive workers to build for the current active worker when the player consumes a helpful token. This conveys the theme of being helpful, as workers are now able to assist your team beyond their usual capabilities. For players to experience this, before the start of each build phase, players are given the opportunity to use a helpful token. If they select yes, building is now the responsibility of the currently inactive worker. This value of helpfulness is especially illustrated when you have chances to block your opponent from winning, without having to sacrifice the position of either of your workers.

The main logic that had to be changed within the game was just to check if each player has a helpful token at the start of each turn, as well as generating build actions for a player's currently inactive worker.