- Design Rationales (explaining the 'Whys')
  - Explain three key *classes* (not interfaces) that you have included in your design and provide your reasons why you decided to create these classes. Why was it not appropriate for them to be methods?
  - Explain three key relationships in your class diagram, for example, why is something an aggregation not a composition?
  - Explain your decisions around inheritance, why did you decide to use (or not use) it? Why is your decision justified in your design?
  - Explain how you arrived at two sets of cardinalities, for example, why 0..1 and why not 1…2?
  - If you have used any *Design Patterns* in your design, explain why and where you have applied them. If you have not used any Design Patterns, list at least 3 known Design Patterns and justify why they were not appropriate to be used in your design..

Classes:
Action Classes:
The action class exists to store events, both possible events that can happen and events that have already occurred. The upside of creating an action class is that you can then pass events around, modify them when needed, and apply particular ones when your heart desires. If the action class didn't exist, then every time a possible action had to be taken, you would need to call large if check within the player class and find the particular actions that were possible. This would require constant modification anytime you wanted to change particular action behaviour or add new actions, and would require code duplication if you wanted a way for actions to be taken outside of the player class. Creating the action class (and all of its child classes) allows for a seamless, extensible, and easily adaptable way of processing player inputs and updates.

Space Class:
The space class exists to represent a single tile within the board of the game. The original plan was to make the board hold a 3D array, with the x, y, and internal array of pieces stacked upwards. However, this made the board class particularly bloated and messy whenever you wanted to scan through to access particular pieces at each position, as well as be able to hold the worker separately from the array at each position. The space class helps abstract out a lot of the necessary behaviour that only a particular tile is concerned about. It's significantly easier to separate the responsibilities of the overall board managing the relationship between spaces, and the space managing its internal data.

History Class:
The history class was originally a series of methods and variables held by the player class. However, there were a couple issues with it's original implementation. Firstly, we needed the god class to be able to access the history, so that it could be informed by what mechanisms were available to it. Secondly, the presence of history in the player class put it at risk of being too much of a god class, as the amount of processing it was doing was far too great. Thirdly, we wanted a more lightweight (yet coupled) way to interact and pass around data from turn to turn. All of these benefits weren't possible anymore if we left the history as a method and arrays within the player class.

Relationships:
Why is board an aggregation of spaces?
Board is an aggregation of spaces and not composed of spaces because of the necessary relationship between spaces and board. Board purely exists as a collection of spaces, along with methods necessary for managing all of the spaces. However, in theory the spaces can exist independently of the board, as a single point holding pieces.

Why does space hold an association with worker and pieces separately?
Workers are a unique type of piece, and as such since there can only be one per space, and they get called frequently with methods that can't be found in their super class, they have are held separately from the rest of the pieces. The other pieces exist in the space class in their abstracted form, since their superclass is all that's needed to get their important values. This just makes it easy to interact with the worker, while simultaneously allowing for a degree of polymorphism with it. The alternative is to place worker in the stack of pieces in space, but this would mean either having to scan through all of the pieces and get their null moves or breaking open close principle in checking for workers.

Why does the game have a dependency on gods?
The game class functions as the centralised controller of all of the logic in the game, and as such organises things that the decentralised players need coordination for. This includes distributing gods, since there is only 1 copy of each god in play, and also certain restricted matchups of gods. As such, the central game class generates the god classes in play and distributes them to the players, making sure there are no double ups, and as such holds a dependency but not association. This is much more preferable to the alternative of a decentralised approach, in which players would need to pass around the gods that they have chosen and are remaining, requiring the players to know who's turn it was next, if a god class has been chosen, polling other players to see if their god card is banned, it's just simpler to use a centralised system.

Inheritance:
Inheritance is a key part of our design. While we wanted to prioritise composition over inheritance where we could, there were instances where it was clear that inheritance was the superior way of proceeding. For example, using inheritance with the action class allows us to utilise double dispatch, so that each different action could process the board in a different way. Similarly with the god class, using inheritance and overriding base methods was easier than passing a god commands to apply, since each god had widely different behaviours from each other, and there aren't ever any exchanging of parts of god abilities, so it's better to force all aspects of the god abilities to be conjoined.

Cardinalities:
Worker cardinalities: 0..2

In Santorini, the most workers a player can have is 2. This is a limit of the board game, as you cannot pull out more worker pieces from the aether. This is the justification for the upper bound of 2, since a player will never own more than 2. As for the lower bound, there exists a god named Bia, who has the ability to delete workers from the board. Bia has the capability to remove 1 or both of the workers a player owns, so therefore the range of workers a player could own at any point in time is between 0 and 2.

Board to spaces cardinalities: 9..*
At an absolute minimum, we decided that the smallest shape a board should be in Santorini is 3x3. This is enough space for players to be able to build just enough towers, and to be able to climb up them. As such, since any smaller than 9 tiles would result in a non-functional game, we set 9 spaces as the lower bounds. Since we have implemented the board in a way to be a dynamic size, we can have in theory an infinite size of board, thus the upper bounds is many.

Design Patterns:
Worker Factory - Factory Pattern
There's a lot of data that the worker holds that is dependent on external conditions. For example, the colour of the worker, the players id number, the gender of the worker, different graphics for different workers, etc. Instead of trying to construct the worker object by pooling all of this information separately, we instead make a factory class that manages the construction. The benefit of a factory class like this is that it can handle the logic for how many workers need to be constructed, what gender of the worker, what colour, etc. This helps abstract all of the messy implementation, so that whenever we need to create a worker in the code, we don't need to duplicate all of the tricky logic, and just use a seamless interface.

Action Class (And inherited classes) - Command Pattern (and double dispatch)
The action class is a prime example of the command pattern, as each action is a possible event that can trigger on the board to modify it. The exact method that the modification is triggered utilises double dispatch across its inherited classes. For example, the build class takes in the board class, and calls on it the build piece function. Keeping the functions encapsulated in an object means that we place them into a list to track their order, which can allow for an undo function down the line, and also gives a succinct list to the player of options that they can take. The alternative would require some hacky functional programming of passing around structs of data that would then need to be processed by the player onto the board, which would require more dependencies.

Action Class (And inherited classes) and Player / God Class - Builder Pattern
The action class is also an example of the builder pattern. Basically, although the worker generates all of the move and build actions, it doesn't know the history of other workers, or whether the player's god class activates. That's where the builder pattern comes into play. The player and god class can modify aspects of the action, like whether or not it ends the turn, and whether or not it causes the player to win, allowing for flexibility in how actions function but also separation between making and modifying the possible actions. The alternative is for the

object to obsess about all of the relevant information at construction, which is impractical since then the worker would need knowledge of everything.

Design Changes:
One of the concerns when we started implementing the player class was that it was becoming too much of a god class. It was generating a lot of actions, and handling a lot of the logic to do with those actions. So we made the decision to move all of the action generating functions into the worker class, since it is the worker that generates these actions. This helps better distribute the workload, and reduces the over complication of the player class, which now just handles minor modifications to the actions and move selection.

Another minor change was creating a dedicated history class to hold the data and methods for managing the players move histories. This is mostly to hide some of the messy coding required for checking and managing histories, and to reduce the responsibilities of the player class.