In the Santorini final implementation. I used many design patterns.

Factory Pattern

When generating God's object, the god's name/title/details are fixed, so it's better we use a function to wrap the object creation logic so that there is only one place we can create God.

Although there are only 10 gods, the factory pattern makes the god generation easy to extend. I noticed that there are extension versions "Advanced Version" and "Golden Fleece". If we decide to add those gods to the game, the factory pattern will be very useful. When choosing a different version of the game, we only need to override the factory to create a function to easily control the game version.

Strategy Pattern

I considered many ways to support various gods' power in the core game logic but I chose a strategy pattern. Because in the game design, no matter what the god power is, they mostly can be categorized into move strategy/build strategy/winning condition. So that sounds very suitable for the Strategy patterns. Strategy pattern focuses on the specific part of the algorithm. If we can scope the interaction of different gods into move/build/win. We can easily use the Strategy pattern to write three functions on move/build/and win.

My implementation is like. I created a God called "Man" with no god power, then implement all three functions.

And for each god with god power, we only need to override the method their god power focuses on. For example, Pan's power is only related to the winning conditions, so we override his winning method with 10 lines of code. Artemis has moving skills, and we can reuse the build/win part of code.

But strategy patterns cannot handle everything well. For example, Athena can "curse" her opponent for one turn. But Strategy patterns cannot stretch to our opponent's logic. Here I created a status variable called Buff. Buff is an object saying buff type, remaining turn, target player. At the end of each turn, the target sets the remaining turn - 1 until the buff expires. This temporarily solves the trouble brought by Athena. And when more and more god powers are added, we have to mix different design patterns to optimize the core logic.

State Pattern

Note that Santorini is a turn-based game. Off the top of my head is that we use state patterns to manage the turn-switch thing. I differentiate the states into PrepareState and RunState. PrepareState includes ChooseGod and InitWorker, that's the state of preparing the game. After players finished setting their workers, the state enters RunState, including BeforeMove, BeforeBuild, and Win. Normally, the state machine runs BeforeMove-(move)->BeforeBuild-(build)->(switch player)BeforeMove->BeforeBuild->...->Win(player). But the god power can influence the

order, e.g. Hermes can move->move->move->move->...->build. Prometheus can build->move->build.
For those thorny situations, I cooperate State Pattern and Strategy pattern to solve them by letting the gods' strategy returns the next state. Thus, the god themselves decide which next state to go to.
What’s more, the state pattern fits undo/redo functions very well. Because each state is a timeframe that could be perfectly taken a snapshot or recovered from. the undo/redo logic is to store a list of contexts, and a context is a snapshot containing all the data at this moment.

Adapter Pattern
I use an Adapter Pattern in serializing data at HTTP API. Since some of my models have a recursive reference, making them unable to be serialized automatically, I created an xxxDTO for each model. A model can simply be converted to its DTO version. The key is to stop the recursion at a certain depth. E.g. Context.player.worker.player.worker causes an infinite loop. But we can set WorkerDTO like workerDTO.playerId, stopping the reference at the worker level. Similarly, context will only be serializable if we set contextDTO.lastContextDto’s recursive reference stops at a certain level.

Template Pattern
We have 10 Strategies for ten Gods. It is a natural idea that we create an abstract template parent class to extract as much as what they have in common. So I created a “Man” without any god power as a template method. After “Man” implemented all build/move/win methods. All other gods can only override a minor part to achieve their god powers.