Implemented Patterns (4)

Template Pattern

The execution of moves between human, heuristic, and random can be generalized to follow the same pattern. They all need to 1) **generate the legal moves** in the current deep copy of the GameState, 2) select a worker, 3) select a move, 4) select a build move, and 5) execute these moves. However, the actual implementation of how each type of player selects a worker or selects a move, for example, is different. For example, the heuristic player doesn't need to get a worker since this is handled implicitly by choosing a move. We used a template pattern here to **improve code readability** and **simplify** the process of **updating any step** of the process for **any type of player.** The Player who holds the Strategy, also doesn't need to know what strategy he is holding. He just knows he can execute the strategy when the time comes.

Memento Pattern

The Memento pattern is **crucial** for implementing **undo and redo** functionality in the Santorini game. The Originator, Memento, and Caretaker classes work together to capture and restore the state of the game. The GameState class holds the deep copy of the game's state for cleaner code when passing a lot of props, and ConcreteMemento **encapsulates a snapshot** of this state. We further supply GameState to the board on every *run()* execution iteration in the main.py, which the Board receives **without direct access** since the player controls them. The originator further saves the memento snapshot on each iteration and repeats the process again.

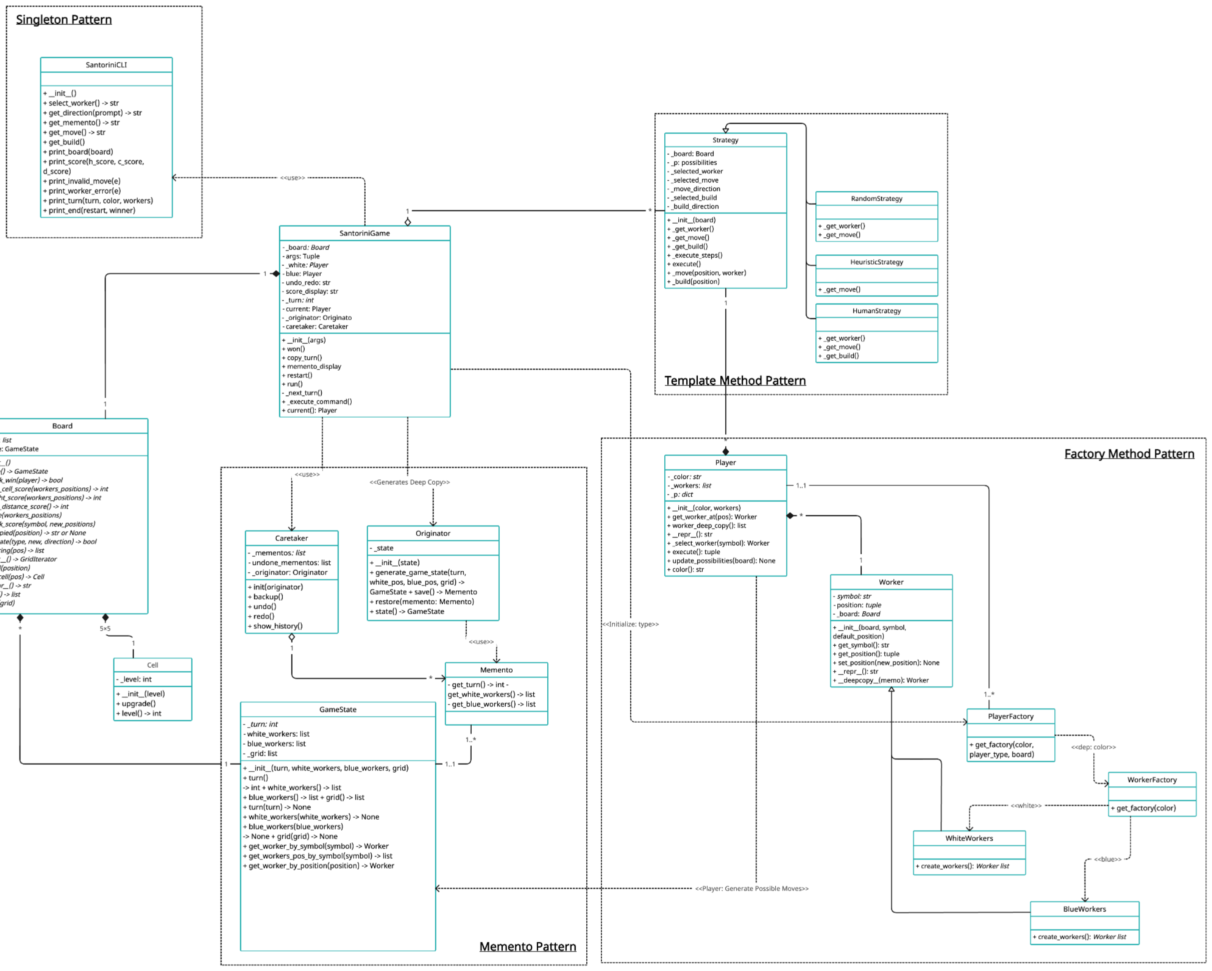
Singleton Pattern

We did significant work to ensure that the IO is handled within SantoriniCLI only. We chose to do this to centralize all code regarding IO so that any changes to IO will be much easier. In the process of doing this, we realized that our **SantoriniCLI is stateless** and would work very well as a singleton since having more than one instance of SantoriniCLI would be a waste of memory with no possible benefit in our case.

Factory Method

Correctly setting up the environment allows further easy access to players and their workers. We implement **nested** PlayerFactory and WorkerFactory methods that set up different game types based on human/heuristic/random/white(first arg)/blue(second arg) passed props. The PlayerFactory is an abstract creator providing an interface for creating players with different strategy templates (Template Method). Similarly, the WorkerFactory abstracts the creation of worker instances, and concrete subclasses like WhiteWorkerFactory and BlueWorkerFactory handle the instantiation of white and blue workers. Factory Method pattern **simplifies** the creation of objects, providing **flexibility for various game scenarios.**

Oliver Li & Anton Melnychuk | SantoriniGame OOP | PSet 6



Work General Overview

Oliver	**Anton**
Template Pattern	
Singleton Pattern	Factory Method
Project Refactoring	Memento Pattern
Memento Pattern	Project Draft SetUp
Fixing Code, Refactoring	Template Pattern
	Fixing Code, Refactoring

Distribution of the Work

All the code has been done collaboratively, with ongoing discussions about potential improvements. Everyone contributed equally. We used GitHub for code storage and LiveShare Visual Studio Code extension to access recent updates quickly.

Anton Melnychuk & Oliver Li