Sprint 2: Design Patterns

Team: Java Doctors

Team Members:
Alexander Thomas Hills Shea
Zoya Hassan
Jonathan Choi
Safa Al-Siaudi

Mentor: Manav Bhojak
CSC207H5: Software Design

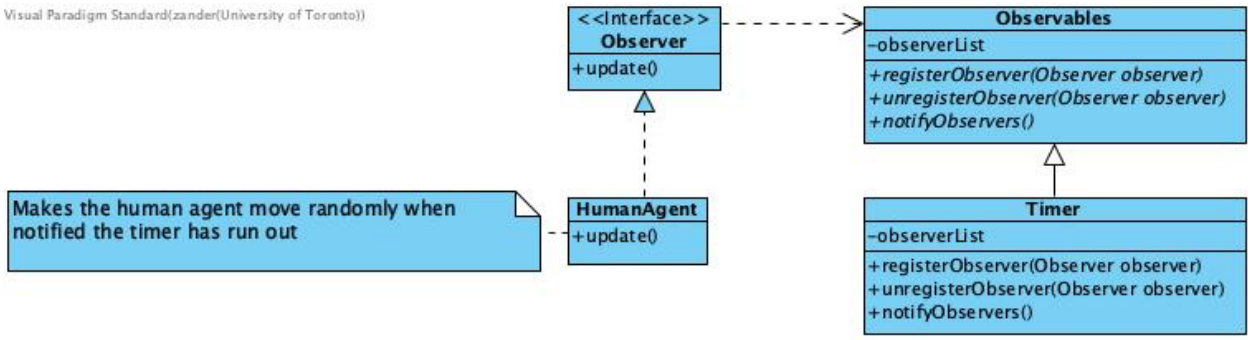
Table of Contents

Observer

Usage

The Human player is given an allocated time to make their move; if they do not, the HumanAgent is notified and a move is made for them. This feature uses the basic Observables/Observer class and interface while the human agent will be given a timer. Timers are not given to non-human agents since they may take some time and ruin their functionality.

UML Diagram

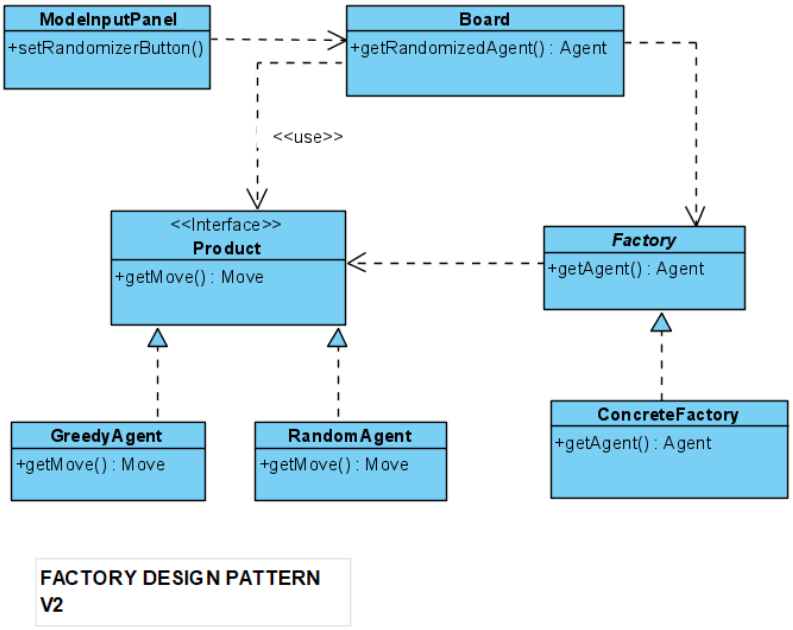


Factory

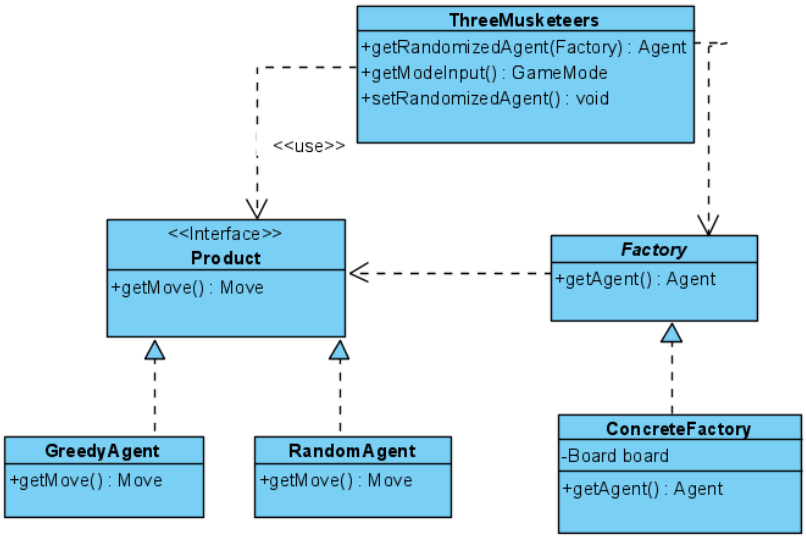
Usage

Set up the game state when the player selects modes, with the proper agents playing each side. This will also facilitate the implementation of a random mode feature, in which the human player is assigned to a random side opposing a random non-human agent. In this way, fewer decisions are needed from the user as this simplifies the situation of rather the humanAgent wants to play against a random or greedy agent.

UML Diagram



FACTORY DESIGN PATTERN
V2



Facade Pattern

Usage

The facade pattern is used in the implementation of the start window GUI. In this GUI, users are able to enter in their username and birth year, and press submit each time. This data is then stored in a .txt file. If they are a returning user, the GUI updates to show the number of games won/lost by that user. The user is then led to a window where they can press a button to start the game, and play on the console.

The addUsername method is called each time the user enters a username and clicks submit. Similarly, the addBirth method is called each time the user enters a birth year and clicks submit.

addUsername and addBirth both add text into a txt.file, and their code is very similar.

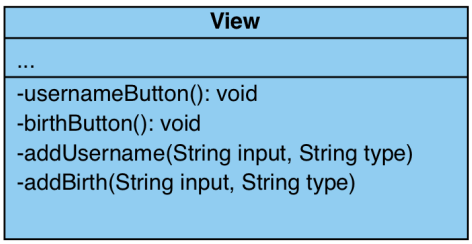
We can create a new method, addUserData, that generalizes addUsername and addBirth.

The addUserData method is called on by addUsername and addBirth, which now only execute that one call.

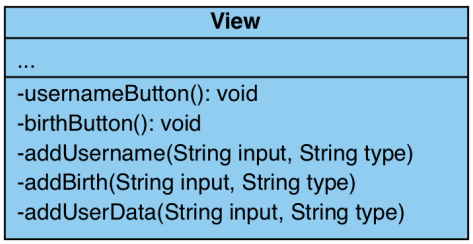
addUsername and addBirth now essentially hide away the complex parts of the code (creating a "facade"). It is now a lot easier to create more implementations of this code or to modify it as needed.

UML Diagram

without facade pattern



with facade pattern



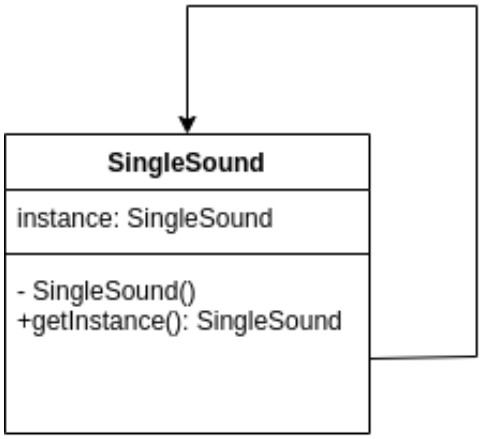
Singleton

Usage

Using this design pattern, users will have a more immersive experience while playing the Three Musketeers. Here, users will hear a sound as their piece moves, which resembles the experience of playing in real life. We will implement this by having a singleton class for sound effects and we will have a method that other classes can call to use the sound effect at a given point in the game.

A sound effect in the format of .wav is located in a folder within the assignment 3 directory. Originally, the soundEffect instance was implemented in Move() in the file Board.java and it played fine when players were a Human vs. Human or a Human vs. Random Agent. However, a GreedyAgent calls on Move() multiple times in its algorithm, so the audio kept replaying until the GreedyAgent made a decision. Instead, soundEffect is used in HumanAgent, RandomAgent, and GreedyAgent after the final move decision has been decided.

UML Diagram

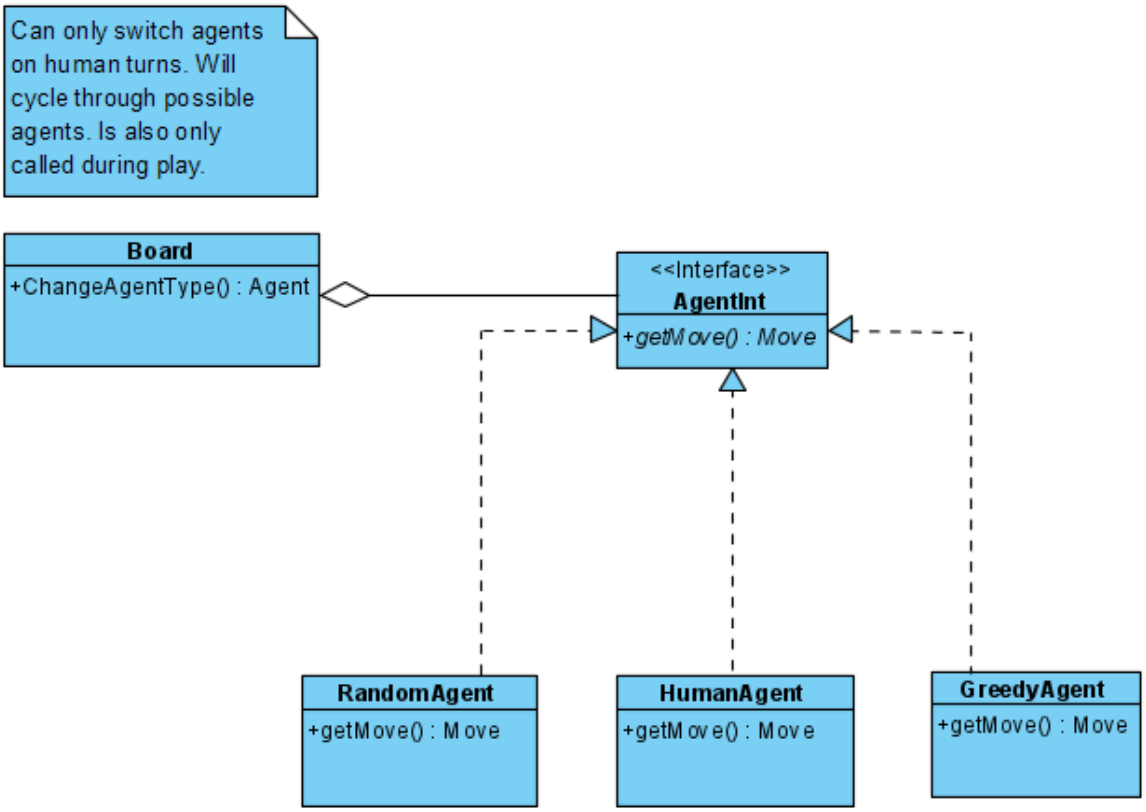


Strategy

Usage

The strategy pattern will be used to allow the human player to switch the game mode mid-game. If the user no longer wants to play against a random agent, they can switch modes at any point in the game to a greedy agent and vice-versa. There must always be at least one human player. This design pattern was used because it involves changing behaviour.

UML Diagram



Memento

Usage

To create a memento of the game state before undoing a move so that the game state can be stored and then reconstructed when the user redoes a move. The board will be the originator, a new BoardMemento class will store the game state, and the main game class ThreeMusketeers can be the caretaker. This solves the problem of game states being lost forever when a move is undone, allowing the implementation of a redo feature.

UML Diagram