Assignment3 Report

Zhuo Wei
Master of Information Technology
University of Auckland
Auckland, New Zealand

Usename: zwei460@aucklanduni.ac.nz

Nickname: Pompeii Wei

I. Command Pattern

Command Pattern encapsulates a command into an object , also enables a request to be made to the object without knowing the receiver and operation of the request. Command pattern separates the requester from the executor. When we make changes or additions, the Command pattern can reduce the frequency of changing existing code. It is applicable for the Kalah which contains different commands.

In my implementation, there are 5 Commands responsible for moving the seeds, restarting the game, quitting the game, saving the game process, and loading the game process. They share a common interface - **Command**, Invoker-**GameController**, and Client - **GameController**. Command Interface defines the **execute()** for ConcreteCommand. GameController can do execute() without knowing the implementation details.

A. *Moving the seeds*:

ConcreteCommand - **MoveCommand**, Receiver - **Move**. When the GameController requests a move command (user input 1-6), the GameConroller creates a MoveCommand object and passes it to the Invoker-**GameController** (commandList). The MoveCommand object records the Receiver (Move) - the class that runs the move function(**makeMove()**). After that, **executeCommand ()** executes the commands in the commandList by calling the **execute()** of **Command**.

B. *Restarting game:*

ConcreteCommand - **NewGameCommand**, Receiver - **GameController**. When GameController requests to restart the game command (user input n), GameConroller creates a NewGameCommand object and passes it to Invoker - also **GameController** (commandList). NewGameCommand object contains the Receiver (GameController) - stores the class that runs the restart game function (**restartGame()**). After that, the commands in the commandList are executed by the **excuteCommand ()** by calling the **execute()** of **Command**.

C. *Exit game:*

ConcreteCommand - **QuitGameCommand**, Receiver - **GameController**. When the GameController requests an exit command (user input q), the GameConroller creates a QuitGameCommand object and passes it to the Invoker **GameController** (commandList). QuitGameCommand object records the Receiver (GameController) - the class that runs the exit game function is stored(**quitGame()**). After that, **excuteCommand ()** executes the command from the commandList by calling the **execute()** of **Command**.

D. *Save game :*

ConcreteCommand - **SaveGameCommand**, Receiver - **Save**. When GameController requests to archive game commands (user input s), GameConroller creates SaveGameCommand object and passes it object to Invoker - also **GameController** (commandList). SaveGameCommand object records the Receiver (Save) - it stores the actual running archive game function(**saveGame()**). After that, **excuteCommand ()** executes the commands from the commandList by calling the **execute()** of **Command**.

E. *Load game :*

ConcreteCommand - **LoadGameCommand**, Receiver - **Load**. When GameController requests to load a game command (user input l), GameConroller creates LoadGameCommand object and passes it to Invoker - also **GameController** (commandList). LoadGameCommand object records the Receiver (Load) of the command - it stores the class that runs the loaded game function(**loadGame()**). After that, **excuteCommand()** executes the commands in the commandList by calling the **execute()** of **Command**.

II. Memento Pattern

In my implementation, I use the memento pattern for save and load. The memento pattern allows storing and restoring the Internal State of an Object without violating encapsulation, which fits perfectly with the requirements of this assignment.

Memento - **Memento**, Originator - **Board**, CareTaker - **GameController**. When the user performs save and load operations on the game, the game needs to record the state of each player (state of houses and the store) and the order in which the game is played (which player's turn it is to play). Therefore, the **Memento** stores the two variables mentioned above, namely **players** and **playerNumberToMove**. **Board** is responsible for creating the memento that stores players and playerNumberToMove. It also uses the memento to restore the internal state. **GameController** is responsible for safe game data storage (storing memento) and does not take action on memento.

When the save operation is performed, **Board** stores the game records as a **Memento** object and puts them in **GameController** (**saveToMemento()**). When performing a load operation, **Board** takes the stored **Memento** object out of the **GameController** and restores it into the game (**restoreFromMemento()**).