Design Pattern and Modularity Report

Yu Chen Dong
ydon461
SOFTENG 701

I. Command Implementation

In my assignment 3 design, the game logic was mainly handled by the GameController class. This included anything the game should check for upon a player's move, such as the sowing of seeds and the conditional capture or score adding. Likewise, the GameIO class controlled the IO for the game. When refactoring my design to implement the command design pattern, I created two separate groups of the command design pattern – one for the logic, and one for output execution.

I changed the existing GameController class to be the client component that is responsible for instantiating the logic command objects and initiating their execution at the right place and time during a round of play. I created a new 'commands' package where I placed the new logic command interface, the concrete commands, and the invoker. The command interface declares an execute method to run commands. The CaptureCommand and SowCommand objects implement this interface to perform their respective actions. Both take a board and a list of players in their constructors so that they can act on the appropriate objects during the game. The implementations of execute act upon the receiver objects, which are the pits and the players for these instances. Seeds are added or cleared from the pits, and score is added to players appropriately. These concrete commands are executed through the CommandController class, which acts as the invoker – the object takes the two arguments upon instantiation which are the instantiated CaptureCommand and SowCommand. The client, GameController, uses an instance of CommandController to execute the concrete commands when required.

The IO handling was also changed to send commands through another invoker – the IOCommandController class. Instead of directly accessing IO methods from the Kalah class, the Kalah class now acts as the client which instantiates the commands and the invoker. I created another interface for the IO commands – this was because the logic command interface's execute method required some arguments to be passed, while this execute method did not. All the information needed by the IO methods could be extracted from objects passed to the constructor. For each output message (board, error message, etc) I implemented a new concrete class based on this interface.

II. Command and Modularity

From my experience, using the command design pattern improved modularity in most areas and reduced it in others, according to the definitions mentioned in lectures. However, I did feel that command improved my design overall.

By introducing command classes to handle different events in the game, looser coupling is achieved as the different command objects can be used to execute specific tasks instead of directly accessing methods from one class. I found that by allowing data to be shared by initialising objects in the constructors of the command classes, I did not need to pass around as many arguments. Looser coupling can be an indicator of modularity, as indicated in the lectures. By having the commands be based on the same abstract interface, the same invoker instance is able to be reused for different commands. This increases the reusability of components in the project, as the new interfaces I implemented are reused for each command, as opposed to the single instance of GameController I used before. Additionally, the interfaces add abstraction to the project, which is good for modularity as it improves maintainability. Previously, I did not have interfaces which outlined the functionalities of each class, instead having concrete GameController and GameIO objects handle everything. The command design pattern also encourages encapsulation and information hiding – the invokers (CommandController for logic and IOCommandController for IO) deliver the command requests from the clients (GameController and Kalah) to the receivers (GameIO, Pit, Player), which keeps the sender and receiver apart. This isolation also increases maintainability as it is easier to identify issues, as each command component only has one purpose.

The one downside I experienced from implementing the command design pattern is having to create many new files to handle all the commands. While the act of separating the code into cohesive classes improves maintainability, I think that having too many files could create an inverse effect, as it gets more difficult trying to navigate through the many files. While refactoring, I created 12 new classes including the interfaces, invokers, and concrete commands. I found that it became a bit difficult keeping track of all the files, and instantiating the invokers required a very long line of code to pass all the commands to the constructors.

In general, I believe that the command design pattern would improve modularity as seen from the effects described from my experience for my Kalah game. Although it may be inefficient to implement this pattern for large-scale software applications for handling multiple tasks due to file complexity, it does not change the fact that the pattern has positive effects on the modularity of the design. Specifically, the design pattern improves maintainability by isolating functionalities, improves abstraction and reusability by using interfaces, and promotes data hiding and encapsulation with the client-invoker-receiver structure.