FIT3077
Software Architecture & Design

Composite

Design Rationale

Undoing of Moves

Our goal is to incorporate functionality that allows moves in our game to be undone, until the point where there are no more available moves.

Initially, our thought process was to undo moves in the most direct sense: for a PlaceAction, we would carry out a RemoveAction; and for a MoveAction, we would swap the source and destination Positions and re-execute it. However, upon closer examination, this methodology was tightly coupled, creating undesirable complexities.

In this approach, we would need to explicitly know that an Action is a PlaceAction to instantiate a corresponding RemoveAction, forming a tight coupling between the two actions. This type of coupling is considered disadvantageous as it binds us directly to a specific class. If we were to add more action types in the future, we would need to continually update this explicit case analysis to accommodate each new type, which would significantly hinder scalability and violate the principle of loose coupling.

To resolve this issue, we propose an alternative approach. Each move is represented as a triplet of (Piece, Source, Destination). Given that each Move implements the Command Pattern and is executed on the Board, we can leverage this to enable the Move to know how to undo itself. In essence, the Move just needs to reverse its direction: if it originally moved a piece from Position A to B, it would, in undoing, move it back from B to A. This can be easily achieved by swapping the source and destination Positions of the Move and then allowing it to execute itself on the Board.

We encapsulate this logic within an UndoAction. This UndoAction is exposed to the ViewController, serving as a functionality provided by the game model. The ViewModel does not need to be aware of the specifics of how the game model implements undos. It simply needs to create an instance of the UndoAction and execute it on the Game instance, which then handles the undo internally.

This alternative approach, while more sophisticated, brings about several benefits:

1. **Reduced Coupling**: This design removes the tight coupling between different types of Actions, promoting a more modular design which is easier to maintain and extend.

2. **Scalability**: As each Move knows how to undo itself, adding new types of moves in the future won't require us to modify the undo functionality. This ensures that our design is scalable and future-proof.

3. **Encapsulation**: By encapsulating the undo logic within the UndoAction, we keep the complexity within the game model, making the design easier to comprehend and modify.

The implementation of move undoing in our game was not as straightforward as we initially anticipated. While we had planned for a stack of Moves played in the game from the early

stages of development, we had overlooked the necessity of transitioning player phases between moves. A simplistic reversal of moves using the above-discussed pattern wouldn't effectively revert the changes in the player phases as required.

Addressing this issue necessitated a modification to the PlayerPhase. As previously explained, we engineered the game engine with the flexibility to accommodate non-linear transitions in the PlayerPhase. Using the State Pattern, we could make the PlayerPhase transition similar to a Finite Automaton (FA). This implies that based on a specific condition, it can transition from one phase to another, such as from the PlacePhase to MovePhase. We simply had to introduce a path from phase B back to phase A, for instance, from MovePhase back to PlacePhase. Consider a scenario where the player has no more pieces left; they should transition from the PlacePhase to MovePhase. Conversely, if a player is in the MovePhase but still has pieces remaining, they should transition back to the PlacePhase. This mechanism enables us to reuse the existing pattern in executing Moves. Following each Move execution, we allow each player to attempt the transition to the subsequent phase. Similarly, upon undoing each Move, they would attempt to transition between phases.

This above modification was the only significant change necessary to enable the undoing of moves in our game engine. This change does not even equate to an architectural level modification, as it only pertains to the logic of PlayerPhase and demands only minimal method-level changes to our existing design.

Implementing this feature is easy, for the aforementioned reasons.

Saving / Loading of the Game

The incorporation of save and load functionalities in our game involves strategic decision-making regarding two main aspects: (1) identifying the data to be stored, and (2) selecting the most appropriate format for data storage.

Initially, we decided to preserve the most recent Board configuration and a chronological list of executed Moves. The rationale behind this decision was that these two pieces of data are sufficient for inferring all other game states, thereby establishing an efficient and effective methodology for storing and retrieving game states.

After deciding what to store, our focus shifted to how to store it. We sought a format that was lightweight and human-readable. Our initial proposal was to create a custom grammar for serialization, requiring us to develop dedicated serializer and deserializer. However, this approach had significant downsides, such as the need to alter the grammar (and subsequently, the serializer and deserializer) every time we sought to include additional game information in the save file.

To overcome this, we turned to more conventional formats for serialization and deserialization: JSON and XML. We opted for JSON due to its superior readability and compactness. To facilitate our JSON data-binding, we leveraged the robust capabilities of the Jackson library. While this decision seems specific, it doesn't preclude the addition of more file formats in the future.

In the early stages, our design involved using dedicated Serializer and Deserializer instances, which were held by the SaveAction and LoadAction. However, upon implementation, we identified a potential violation of the Open-Closed Principle (OCP). Tying SaveAction and LoadAction to specific Serializer/Deserializer types would mean modifying SaveAction/LoadAction logic when expanding to more file formats in the future.

In response, we iteratively refined our design and added an additional layer of abstraction. Taking serialization as an example (although this applies symmetrically to deserialization), we introduced a new entity: SerializerFactory. This factory pattern allows us to decouple the serialization process from the specific Serializer implementations, enhancing flexibility and extensibility.

The SerializerFactory is instantiated by the ViewController, which is aware of the file formats the game supports. It registers all supported Serializers, which currently includes the JsonSerializer, but can easily be expanded to include other Serializer types, like an XmlSerializer.

When the user specifies the save file path and format through a file dialog, the file format (extension) is extracted and passed to the SerializerFactory. This factory then creates the appropriate Serializer to handle the specific file format, since each Serializer knows the formats it can process (through Serializer.getSupportedFileFormat()). The Serializer instance, coupled with the file path, is then passed to the SaveAction. The SaveAction need

not be aware of the serialization specifics; its role is merely to request the Serializer to perform its function. This separation of concerns reinforces our design's modularity and scalability. The same principle applies to the DeserializerFactory and LoadAction in the deserialization process, except the Game class requiring a new method for loading from a GameState (see below).

This refined design offers several key advantages:

1. **Scalability**: By leveraging a registration-based factory pattern for Serializers and Deserializers, our design is equipped for future expansion. Adding support for more file formats won't necessitate changes to SaveAction/LoadAction logic (adherence to the OCP).

2. **Flexibility**: With the Registration-based Factory Pattern, Serializer/Deserializer types are not hardcoded but can be registered dynamically, providing more flexibility and making the system more adaptable to change.

3. **Reduced Coupling**: The registration-based Factory Pattern helps to further reduce coupling in the system by removing the need for explicit references to concrete Serializer/Deserializer classes in the ViewController.

As we have outlined, the most substantial change is the addition of an abstraction layer, SerializerFactory/DeserializerFactory, which helps us fix our design mistakes in the first sprint and better adhere to the Single Responsibility Principle (SRP). Notably, we did not significantly modify the logic of existing classes to implement this functionality; instead, we just added new layers of abstraction.

To help with serialization and deserialization, we also created a GameState class that holds a tuple (Board, Moves), which the Serializers and Deserializers use. The Serializers and Deserializers recursively serialize and deserialize the Board and Moves. A new method is introduced in the Game class to allow loading from a GameState.

For better file management, we moved the Serializers/Deserializers and their related classes to a separate package, named edu.monash.game.io.

Implementation of this feature, for the reasons mentioned above, was very straightforward and easy.