Design Rationale

This brief was brought to you by "*JTR Industries Pty Ltd*"

Explain two key classes that you debated as a team, e.g. why was it decided to be made into a class? Why was it not appropriate to be a method.

Two key classes we debated were the separation of the 'Position' class from 'Piece'. In initial iterations, we had 'Position' as a private tuple attribute inside 'Piece', representing the coordinates of the piece on the board (accessible by a getter method). Intuitively, this made sense to us because to work with a piece, you would probably want to know its position on the board, and by having position as an attribute within a piece, there would be no mistaking which specific piece that position refers to.

After deeper discussion and early prototypes, it became clear to us that we were violating several design principles; principally the principle of separation of concerns.

We foresaw potential issues in the 'Piece' class handling not only its own visual representation, but also relatively complicated methods examining adjacent positions and its pieces. The 'Piece' class was obviously overloaded and needed to be broken down more atomically. We refined our definition of the 'Piece' class as being responsible for its own intrinsic properties (such as color and how it looks). We then extracted position's responsibilities into its own 'Position' class for it to handle solely, tracking a position on the board.

By doing so we also improved our design's quality in several ways. Immediately, the code's readability (and therefore presumed maintainability) was improved, by simple virtue of it being very clear whether methods and attributes were referencing the position or the piece itself. Encapsulation was also better adhered to, as the getters and setters for both 'Piece' and 'Position' were contained in their respective representations. Whilst it is outside of the assignment's scope, we also foresee a benefit in our design's improved flexibility, wherein a subclass of 'Piece' could be created to represent a new game piece (another variation/mod of Nine Men's Morris) without affecting the Position class.

Overall, our decision to separate the two classes hinged on the issue that our original 'Piece' class was overloaded in its responsibilities. And doing so yielded many benefits in making our design and prototype behaving in a more sound, object-oriented way.

Explain two key relationships in your class diagram, e.g. why is something an aggregation not a composition?

During Sprint 2, we identified and added a new class (outside of preliminary planning) which was 'PieceSet'. Our 'PieceSet' class represented a player's available Pieces to set at the start of the game (when the pieces are not yet placed on the board). The relationship between our 'PieceSet' and 'Piece' classes ended up being one of the key relationships in our design. And we felt there was an important nuance to recognize, in depicting this relationship as an aggregation relationship within our class diagram.

First, we can acknowledge the commonalities between the aggregation and composition relationship contexts, wherein the 'PieceSet' class is the 'whole' and the 'Piece' is the part which makes up the whole.

The key distinction for this relationship being one of aggregation rather than composition, is in the lifetime of the 'Piece' class and subsequently the strength of its relationship with the 'PieceSet' class. Within our 'Nine Men's Morris' board game, instances of the 'Piece' class exist within a 'PieceSet' instance. However, as the pieces are being set during 'phase 1', each 'Piece' instance is removed from the 'PieceSet' and placed on the board. This key interaction wherein a 'Piece' instance can be removed from the 'PieceSet' and placed elsewhere, means the 'Piece's instance lifetime is not conditionally tied to 'PieceSet', which is a key identifying factor of an aggregation relationship.

Furthermore, if we had interpreted this relationship as being a composition relationship and implemented it as such, our product would run into issues with inflexibility. Since a composition relationship implies that the 'Piece' cannot exist without the 'PieceSet', any changes to the 'PieceSet' would also affect the 'Piece'. This tight-coupling could potentially place unnecessary restrictions on the 'Piece' class's behavior and ultimately hinder its reusability and modularity in other areas of our design.

For these reasons, we ultimately decided to design and implement the relationship between 'PieceSet' and 'Piece' as aggregation.

One other key relationship is not so much a relationship, but more so a mechanism. We took inspiration from a similar trick used in the FIT2099; wherein State enumerations are globally accessible. This is extremely helpful for many implementations of classes as, we can easily perform conditional checks based on 'Status' enumerations, rather than checking for certain class types (which involves casting). This ultimately helped us implement things in a sounder object-oriented style.

Explain your decisions around inheritance, why did you decide to use (or not to use) it? Why is your decision justified in your design.

Inheritance was a key mechanism we leveraged in designing and implementing our game. It was particularly helpful for movement of pieces. For example: all our 'Action' classes (MovePieceAction, FlyAction, RemovePieceAction) make changes to a piece's position.

The key quality amongst these actions, which makes it a prime subject to be optimized by inheritance, is that they all share the common quality of taking in a 'Piece' and 'Position' class and moving it to some place on a 'Board'. They only differ in the actual implementation of the movement itself. This means that we were able to capture much of the common methods within a parent 'Action' class, which ultimately reduced our overall code repetition in its subclasses.

Having our specific actions be derived from a parent 'Action' class, also introduced polymorphic qualities to our piece movement design. This could be potentially helpful in the future if we were to add different movement actions, as our base code would automatically work with any new action (so long as its derived from the 'Action' class).

Thus, by using inheritance in our action and movement design, we were able to yield a more efficient, flexible, and maintainable product.

Explain how you arrived at two sets of cardinalities, e.g. why 0..1 and why not 1..2?

Our first cardinality set is between the 'Game' class and the 'Player' class. We extracted this cardinality from the brief's description of 'Nine Men's Morris's game rules, wherein two entities (hence 2 player instances) are required to play a match of 'Nine Men's Morris' (one game instance). Without two 'Player' instances (any other number than 2), the game cannot appropriately initialize the correct parameters to execute. Thus, our 1…2 cardinality between 'Game' and 'Player'.

Our second cardinality set is the one between ‘PieceSet’ and ‘Piece’. First, it is important to not mistake the ‘PieceSet’ class as containing all pieces in play. ‘PieceSet’ actually contains all pieces a player may have at the start of the game, and pieces are removed from ‘PieceSet’ and placed on the board. Therefore, according to the game rules, by the end of ‘phase 1’ all of a player’s pieces should be on the board (and in play) and therefore removed from ‘PieceSet’. Thus, by looking at the given game rules, we know that there must be two player instances (as established above), each of whom have their own piece set of nine pieces to play. In even more simple terms, there is 9 pieces for every piece-set at the start of the game and 0 pieces in piece-set after ‘phase 1’, hence the 0...9 cardinality.

Explain why you have applied a particular design pattern for your software architecture? Explain why you ruled out two other feasible alternatives.

In our project, we mostly use the State Design Pattern. In State Pattern, there are 3 main elements:

- Context: An object containing the state provides interface for others
- State: an interface that defines the function / action / behaviours an object in different situations
- Concrete State: The State implementation.

Each of them has their unique action / function / behaviour defined in the State interface. Comparing how State Pattern is defined with our project, we found lots of similarity. In our project, we have many action and behaviour that use the same variables and purpose, but all of them operate in different ways such as the MovePieceAction, FlyAction and RemovePieceAction. So that using State Pattern will not only help us maintain the functions easier but also make all of them consistent and accessible. The other 2 alternatives we ruled out are: observer pattern and strategy pattern. We did not want to use Observer Pattern because each object does not need change notification from another as they have their own state and responsibilities. For Strategy pattern, it has similar algorithm with State Pattern. However, State Pattern focuses more on object ‘s behaviour state which is suitable for 9 men Moris as it has 3 different phases.

Team 23: Nine Men's Morris, Sprint 2, Class Diagram