Design Rationales

Explain two key classes that have 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?

- Save/Load class: We were unsure whether saving/loading would be an Action taken by a player or a method belonging to the Game or Application. In the end, we decided that the Application would be responsible for saving/loading the game as it would allow a Game to be closed and an existing Game opened, with a new set of players. This makes more sense than a Game trying to close itself, or a Player trying to load a Game that it isn't a part of.
- Action manager. As there are various actions users can make while playing the game, it's more efficient to utilise an action manager class that will take the user interaction from the interface and translate it to the appropriate action that can be understood and performed by the game. Although the same goal can be achieved easier by using the methods, the methods will have to be implemented in a suitable class such as the Game main class so it won't create any unnecessary dependencies between the classes. However it will cause the game class to become more complex, hence hard to test and maintain, so it a good abstraction practice to have another class that manage the action

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

- The Board class has a composition with the Location class. This means that the Board class is composed of 24 locations as shown in the cardinality of the class diagram. When a Board is created, all 24 locations will be created alongside the Board as well. The same can be said when a Board is removed, the locations will be removed as well. The reason we decided to make this relationship a composition is because the lifecycle of Location is under the control of the Board class. If this relationship were to be an aggregation instead, a Location will be able to exist without

Game aggregates Token

- As the game aggregates a Token, it's easier to support the basic functionality of the game such as save, load, undo as well as making the win condition. The game agge

Game aggregates Player

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

- We decided to use inheritance with an abstract Action class which would form the basis for all actions within our system. Each action is similar in that they are called when something needs to happen, and calling it creates that change that the action is designed for. This way, all actions have a common interface for executing their actions when they need to be performed.

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

- The cardinality between a Player class and the Token class is 2..9 and not 0..9. Each player starts with 9 tokens, which is the maximum number of tokens each player can have during the game. As the tokens get removed by an opponent, the number of tokens the player has decreases. The minimum number of tokens a player can have is 2 instead of 0 because the game ends when a player has less than 3 tokens. This means that the game ends when a player has 2 tokens left, resulting in the cardinality from one player class to the token class to be 2..9 instead of 0..9 as stated in our previous domain model.
- The cardinality between a location class and itself is 2..4. This is because every location will be neighbours to 2 to 4 other locations. Although most locations will have only 2 neighbours. There are 4 locations that will each have 4 neighbours
- An application runs 0..1 games instead of 1 game. The game client could be running (the GUI open) but no games are played. The user can only play a single game at a time, so the maximum cardinality is one.
- The game is acted on by 0..* actions instead of 1 action. There are many actions throughout the game, or there could be no actions if the players don’t do anything. Overall, there are multiple action objects which act on the game throughout the game, so the cardinality is 0..* instead of 1 since only one action object isn’t executed.

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

- See the rationale for the modified Memento pattern

Moves and Actions

Omitting the Move Class

In the initial domain model, the Move entity was included to show that a player could do one or many things on a given turn. However, this entity isn't very useful in the implementation, since it only serves as a container for actions, and doesn't do anything itself. Additionally, the exact number and type of actions made in a move are determined dynamically and are unknown beforehand, meaning a Move object couldn't be completely initialised before its execution. The Move entity was useful in the domain model to describe how the game is played, but isn't useful in the class diagram since it introduces complications without providing much benefit. Therefore, there will be no Move class in the implementation.

Removing the SetupAction

The SetupAction entity was included in the domain model since in the real world a player must do something to prepare a game for play. There is also the LoadAction entity, which is also where the player prepares a previous game to be resumed. Essentially, these two actions are the same thing; they prepare a game for play. Therefore, the SetupAction will be removed.

Considering Undoing as Movement

In the domain model, the action of undoing a move was not considered to be an action that affects the game board. This is not true since undoing an action clearly moves a token back on the board or back to its previous position on the board. The action of undoing can be considered as similar to the other standard gameplay actions, since they all result in the movement of one token on the board. The BoardAction entity will be converted to a MoveAction class, so it's clear that this class of actions specifically moves one token on the board, adds a token to the board, or removes one token from the board, rather than affecting the board in some indistinct way.

Using a Modified Memento Pattern

To implement the undoing of actions in the game, there needs to be a way of saving and restoring previous states of the game. The Memento design pattern aims to solve this problem, however there is an issue with this pattern which is why it won't exactly be followed. This design pattern requires that a separate caretaker class is used to store the history of game states, and also instruct the game class when to make and restore snapshots. In the advanced requirement where games must be saved and loaded, the entire game history must also be saved and loaded. If the history is stored in a separate class, this process is more complicated. Additionally, there is unnecessary coupling between the game history class and the game class.

The responsibilities of the caretaker class are to store the game history, request when the game should snapshot itself, and request when the game should restore a previous state. The game history can be stored in the game itself, and the mementoes themselves won't again store the game's history at that state. There are two options for assigning the remaining responsibilities.

The game class could determine when it should snapshot itself. A snapshot should be taken whenever the game state changes, and since the game has unrestricted access to its internal state, it can easily know when its state has changed. This would reduce coupling between any other class which needs to determine when a snapshot should be taken, since they wouldn't need to check the game state.

The action classes could request the game to snapshot itself. This way there is no need to check the game state to see if it has changed, since it is already known that an action will affect the game state and require a new snapshot. This is a simpler implementation than the above option, since the action class only tells the game class to take a snapshot, while in the above option the game class needs to check its own state first to see if a snapshot is required.

In both cases, the game only needs to revert its state when an undo action is executed. This means the undo action can take the responsibility of requesting the game to revert its state. Since the game stores its own history, it can fetch the previous state by itself, and the undo action doesn't need to figure out what state the game should be reverted to.

All the responsibilities of the caretaker are assigned in these two options, so the solution with the caretaker won't be taken. The second of these options will have a simpler implementation when determining when to make snapshots, so it will be taken.

Implementing Loading and Saving as Methods over Classes

In the initial domain model, the loading and saving of games were modelled as action entities, meaning they are things that the player does to play the game. It is more accurate that the application user takes the actions rather than the player, since the game must be paused before loading or saving. The player isn't playing when the game is paused, and instead the user is interacting with the application. When the application is first run, a game must be loaded before play begins, which means that loading and saving can't be considered actions taken during the game.

Classifying loading and saving as actions simplified the domain model, however poses some challenges in the implementation.
Each game will run by executing a game loop until play stops. In each iteration the game will fetch an action from the player whose turn it is and then execute it. This means the entirety of the execution of an action exists in the context of the game loop. If an action were to load a new game, it would need to first end the current game, which would mean terminating the action before it can load the new game.
If an action were to save the state of the game, it would be tightly coupled to the Application class, since the application stores all saved games. It's a much better assignment of responsibilities if the application is responsible for loading and saving games, since the Application is the top-level class which manages games.

Application Class and Singleton Pattern

The domain model only modelled the gameplay of the Nine Men's Morris game, however this is only a part of the overall program. There is also the user interface which must be managed in addition to the game being played. This means there needs to be a top-level Application class which is responsible for setting up the user interface as well as setting up the game to be played. The Application objects represent an instance of the game client, and since there is only one game client, there should only be one Application instance. This is

exactly what the Singleton pattern accomplishes, so it will be implemented to ensure only a single Application object exists. If the Singleton pattern isn't used, it's possible that there are multiple Application objects existing at the same time, which means multiple games are running at the same time, which shouldn't occur.

Game Display

As the game progresses further into the development, each element of the game needs to be displayed on the screen. The game has draw classes that handle the display for each element. These draw classes extend the JComponent and JPanel as a way of abstraction. As the draw class has the relation with the object that it draws and the drawing information is from the drawn object, the update inside the object will also update the drawing image. The Display game will handle the mouse input event and send it to the ActionManager where it will decide which action is it before it can be executed.