- Explain why you have designed the architecture for the advanced requirement(s) the way you have.

  Our selected advanced features for the game include a player vs. bot mode, save/load functionality, and the ability to undo turns. In the bot mode implementation, the abstract player class contains the "generateAction" method for generating actions for both the player and the bot during their turns. As the player action comes from the listener when the player interacts with the UI, the generateAction will be blank for the player. In the bot class it will firstly design which type of action it can be allowed to perform and then call the methods to perform it. The bot class consists of private methods, which will fall into 2 types of goals which are to handle the specific action and selected target. Handling the action methods (place, move, jump and remove) are the basic methods that translate and perform the bot's target to actions for the game. Selected target methods (token and location) are created at random (as the requirement) at the moment. However as we separate these methods, the bot can improve further in the intelligence as more smart decisions when choosing the target can be made and increase the hardness of vs bot mode. For the save/load and undo feature we are using the same string variable to store all of the information of the turn, and all the tokens in the game. As all the other information such as score, whose turn it is is handled by the game and other class.This will help us easily translate it to the save file and we save it. Each token will be called to handle the change of the itself when the load/undo is used. By using this way we can have the same save file for both play with bot and play with player mode as well as support needed objects that need to be saved in the future if needed. The other way we can do it is have a save file class that will store all the information from the game class which will be mostly useful and waste the save file size because most of them can be calculated when the file is loaded.

  The memento pattern was used to implement the undoing functionality.
  Add a memento class (GameMemento) to represent the gamestate. Memento class will hold data related to the game's state which includes turns, and the two token banks of both players. The originator class which is responsible for creating the memento and restoring the game state from a memento will be our existing class, Game. Finally we will have a caretaker (GameCaretaker) class which will be storing and tracking multiple memento states and be able to return a memento to the originator (Game) whenever needed. There are certainly pros and cons for using the memento pattern.
  - Pros:
    - Being able to produce snapshots or mementos of the game state without violating the encapsulation.
    - Simplifying GamePanel's code by allowing for the caretaker class to maintain the history of the game state.
  - Cons:
    - If snapshots of mementos are created too often, there will be high memory usage in the RAM which will lead to a slower performance.
    - Caretakers should track the lifecycle of the gamestate to be able to trash collect properly.

- Explain why you have revised the architecture, if you have revised it. (What has changed would have been shown in the revised class diagram. This one is about why it changed).

  One of the design changes was shifting responsibility from the GamePanel class to the core game functionality classes such as Board, Location, and Token. The reason was that it became clear that the GamePanel class had an anti-pattern of being a very large class, and was taking responsibility for the core game functionality such as checking if a mill was formed. Initially, the GamePanel class's responsibility was to draw the game on the screen, not to run the game. This misassigned responsibility was also causing a lot of code duplication between GamePanel and other classes such as Board and Location. Its large size and misassigned responsibilities were the reason functionality was shifted from the GamePanel to other game classes.

- Explain when your advanced feature was finalised (e.g. it is the same as we decided from sprint one; or we changed it in Sprint 3) and how easy/difficult it was to implement. e.g. was it easy to implement due to good design practice/pattern(s) that you have applied in 1 the earlier Sprints (provide evidence)? Or was it difficult (such that you needed to rewrite the majority of the code) for the advanced feature?

  Luckily in terms of advanced features our design from the sprint one is still applied here as the player has a dependency on the game. We can just use the abstraction to allow the different performance that can be allowed for player and bot. After that we will only need to work with the bot class to generate different actions for each situation. Another way that this one can be implemented is using the bot independently with the player called in the game class. However this will made the game mode become more harder to implemented and the game class become more complicated comparing to using simple abstraction as in our way

  One change that we made is that instead of having 2 different classes for save, load and undo feature we only implement the GameCareTake which will handle all of them at once. This decision will be implemented in our original way and realise that we can reuse most of the line of code in each class for the other, therefore we combine all of the class into one which will have methods to handle each feature.

Options for design patterns to use:

- Command for actions functionality
- Memento for undoing functionality
- Observer pattern in active MVC

Design alternative 1:
Instead of having a few listeners for the entire game which has to find out exactly what happened, break each of the game parts (location, token, etc) into its own ui component (e.g. JComponent). Add listeners for each component. This way when a listener is called it's already known exactly what component/token has been pressed and no additional logic is needed in a single large listener.

Keep action classes. This way there is decoupling of the actions from the controller classes. Adding new actions can be done by adding new classes, so OCP is not broken.
Will need to modify listener processing though. Maybe a new listener class for each action?

Design alternative 2:
Having a separate listener for each action decouples the event processing for each action from one listener class. OCP is followed. This will mean multiple listeners are notified for the same event. Each listener only has to check if its event has occurred, otherwise it discards the event. This means each listener class is simpler.

Design alternative 3:
The command pattern with the actions functionality will decouple the event listeners in the view package from the game classes in the model package. This also allows different events to reuse actions functionality. For example, clicking tokens or dragging them to move them generates the same actions.

Design alternative 4:
Using multiple observer patterns in the active MVC pattern. If there is one publisher and one subscriber for the entire game, the subscriber must figure out exactly what has been updated in the game. If there are multiple observer patterns, then within each it is already known what has been updated. For example, there is a subscriber interface for token updates and one for location updates. Each ui class for the token and location subscribes to each, and then when they are updated they don’t need to figure out whether a token or location has been updated