

# Table of Contents

[**Table of Contents 2**](#_2zj1nuzz5op)

[**Design Decisions 3**](#_7k81k5a19kiu)

[GameBoard GUI 3](#_loxqbilfkjxx)

[Action 4](#_p3uh7vc4v2eo)

[Game 5](#_uwo0julvzevj)

[Player 6](#_sjexazho2o1z)

[Intersection 7](#_l4wvrp43d8gs)

[Capability 8](#_vnw7pp7d4nv0)

[Capability 9](#_hiblx9afifcp)

[Token Type 10](#_7wptevabbbs4)

[Token Stack 11](#_1rqk8ooas3le)

[Token 12](#_oh7aen9awlfj)

[Game and GameBoardGui Relationship 13](#_ygo1hjarbmtt)

[TokenStack and Token Cardinality 14](#_ef0zmkofcx7y)

[MillObserver and Token Cardinality 14](#_1yd3w8a2jo72)

[**Appendix A: Design Rationale Guidelines 15**](#_3wv3rxyna035)

[Guidelines 15](#_5jwte7s68bb9)

[Template 15](#_ryqyumcn561e)

# Design Decisions

## GameBoard GUI

| **Issue:**  A basic GUI to create with Swing forms is a challenge, it might be easier for us to work with Swing with the game board as a background image rather than using Forms to place the swing components. | |
| --- | --- |
| **Alternative A:**  Using a static image background | **Alternative B:**  Creating the gameboard entirely using Swing GUI components in GUI Form. |
| **Advantages:**   * Simple and easier to code having the gameboard as just a background image to overlay more images and buttons on * A lot more stylistic variety in terms of hand drawing the board on a different platform * More freedom to conform the design of the board and its piece to what we’d like   **Disadvantages:**   * Having to hardcode to locations for each intersection as it needs to align with the image’s intersection points * It is limiting the ability of the user to resize the game window to their desired size. | **Advantages:**   * A non-static game board that can be resized to whichever size the user may prefer. The window can expand dynamically, making the board fit whichever size. * Access to multiple components in Swing to use on the empty form. * Easier to immediately drag and drop components without having to hard code locations on the window.   **Disadvantages:**   * Difficult for it to exact map to the structure of the Nine Men’s Morris game. * Cannot place components on top of each other, they can only align side by side. * Limited J components to use to join intersections together |
| **Decision:**  Alternative A, despite being restrictive to the user, was the decision the team ended up voting on. This form of the GUI allowed the Intersection class to be integrated easily with the buttons placed by the GUI via code instead of having to link an intersection manually to each button on the Swing Form alternative. This alternative also provided the team with a lot of variety in how the board was going to look and allowed more stylistic features such as colours, borders and token banks to be implemented that the Swing Form Alternative didn’t allow. The Image GUI version also allowed for the buttons to easily overlay on top of the board so that the user can click and place the token without it misaligning or going behind the board. | |

## Action

| **Issue:**  Having one class called the PlaceTokenClass with a parameter of what its initial intersection is, and where its destination location is would suffice for all moving token actions instead of having an Action interface and multiple action classes that inherit this. | |
| --- | --- |
| **Alternative A:**  Having one class called PlaceTokenClass that handles all the moving of the tokens | **Alternative B:**  Having an interface called Action that can be inherited by other concrete classes for each type of movement in the token |
| **Advantages:**   * Simple to implement as initially the movement of the tokens whether it's jumping, placing or moving, there is a very similar code that handles it * Would only need to call one class to do the work of moving the token and it only needs to take in the current location and destination location whether it be from intersection to intersection or token bank to intersection   **Disadvantages:**   * Risk of a god class when further implementing actions for the token to have * If a part of the code regarding one movement, i.e jumping, needs to be changed, it is at risk of altering the code that handles the placement and movement of the tokens as well * Does not adhere to the open/closed principle as the code that handles 3 different mechanisms is open for modification when it should only be open for extension | **Advantages:**   * Achieves abstraction as the game doesn’t need to rely on concrete classes which allows us to extend the classes that inherit it without having to refactor over and over again in the code * Adheres to open/closed principle as the concrete classes are closed for modification and you can extend the interface when expanding the actions to add more features later on if required like the removeTokenAction and more * Adheres to Liskov Substitution where whenever the Action class is called, any of the relevant Action extended classes like MoveToken or PlaceToken can replace it without having to rewrite the code to fit different concrete classes   **Disadvantages:**   * Requires multiple inherited classes to fulfil the abstraction whereas one class would suffice without having to write more code * Currently, the inherited classes don’t have too much of the difference aside from the execute function which also may be repeated and can cause unnecessary code. |
| **Decision:**  The team decided to implement alternative B and use an Action interface from which the PlaceToken, MoveToken, JumpToken and later, the RemoveToken Actions will inherit. Despite the code being very similar between the classes, it allows us to easily refactor and modify the code without the risk of bugs in the other token moving actions and if new functionality is added, the action interface will allow for easy extension of the code. Alternative A was discarded in this decision as it, despite being simpler, does not adhere to SOLID principles and ultimately will case bugs and refactoring issues if implemented. There is also the risk of not being able to extend the code when added other features to the token such as the RemoveTokenAction that has not yet been implemented in this sprint. | |

## 

## Game

| **Issue:**  Should we have one class called Game that manages both the game backend and the user interface or have a GameBoardGUI class as well as a Game class? | |
| --- | --- |
| **Alternative A:**  Having one Game class | **Alternative B:**  Having a Game class and a GameBoardGUI class |
| **Advantages:**   * Simple to implement as both the backend and front end can interact directly   **Disadvantages:**   * The class will become a god class that takes on too much responsibility and not be very modular or extensible * Changes cannot be made to just the front end or backend without refactoring the whole Game class * The god class will end up being extremely large with multiple pieces of data and methods being handled by one class. This can result in poor readability and maintainability of the Game class code. | **Advantages:**   * Each class now has only one responsibility, and refactoring just the front end or backend is now possible * Both classes will be appropriately sized, making them more readable and maintainable   **Disadvantages:**   * Introduces a dependency between GameBoardGUI and Game * Harder to implement as GameBoardGUI components have to interact with Game and depend on output from Game |
| **Decision:**  Although alternative A would have been easier to implement, the team decided to go with alternative B and implement a GameBoardGUI class for the frontend and a Game class to handle the gameboard backend. Alternative B adhered to OOP principles more closely, where each Class had a distinct single responsibility, and both classes separated are easier to refactor, maintain and extend. | |

## Player

| **Issue:**  Should we make an abstract player class and have players extend the class instead of having one player class | |
| --- | --- |
| **Alternative A:**  Abstract player class that BlackPlayer and WhitePlayer inherits | **Alternative B:**  One concrete player class that can be instantiated twice |
| **Advantages:**   * Can extend the class Player to create players with additional functionality whilst retaining required “Player” behaviours * Can refactor individual player behaviours without impacting the other players behaviours   **Disadvantages:**   * Since no significant difference exists between WhitePlayer and BlackPlayer, having two separate child classes will result in unnecessary code redundancy | **Advantages:**   * Since both players are identical and only differ by token type, having a single player class prevents unnecessary code redundancy.   **Disadvantages:**   * Cannot extend the functionality of either black or white player without affecting both players at once |
| **Decision:**  The team decided that although Alternative A provided the benefits of abstraction, Alternative B would pose to be more appropriate for the Nine Men's Morris game. Since both WhitePlayer and BlackPlayer are identical in behaviours and only differ based on token type, having separate classes that extend Player would just result in repeated code. Furthermore, it is expected that both players will always possess the exact same behaviours, thus having one class ensures that both players will always have equal abilities. | |

## 

## Intersection

| **Issue:**  Should there be a separate intersection class or should the GameBoardGUI just manage all the intersections itself? | |
| --- | --- |
| **Alternative A:**  GameBoardGUI handles intersections by itself | **Alternative B:**  Having a separate intersection |
| **Advantages:**   * GameBoardGUI has direct access to all the intersections and can thus   **Disadvantages:**   * GameBoardGUI has too many responsibilities, involving updating UI as well as handling user interaction with intersections * Prevents extensibility of intersections without refactoring the whole GameBoardUI. * More complex and less efficient as GameBoardUI has to check every individual intersection to see which one is selected. | **Advantages:**   * Adheres to the single responsibility principle as the GameBoardGUI does not need to handle both updating the UI as well as Intersection detection * Improves performance as every intersection does not need to be checked when clicked, rather the clicked intersection alerts the game that it has been pressed * Allows for easier extension of Intersection as it is its own separate class, meaning changes can be made without refactoring GameBoardGUI as a whole.   **Disadvantages:**   * Intersection ends up becoming a dependency in other classes such as Game and player, who utilise the Intersection in order to change its state |
| **Decision:**  Although alternative A allowed the GameBoardGUI class to have direct access to all intersections at once, the team decided to go with Alternative B due to the multitude of benefits it provides. Having a separate Intersection class allows for firstly, the single responsibility principle to be upheld, as the intersection is responsible for actions that are imposed upon itself, as opposed to the GameBoardGUI doing all the work. Furthermore performance is enhanced since the selected Intersection only needs to alert the game board when it has been pressed as opposed to checking all the intersections every single turn. As a result, these main advantages resulted in the alternative B being chosen. | |

## 

## Capability

| **Issue:**  Should capabilities be composed of Actions or should an overarching game class translate Capabilities into Actions? | |
| --- | --- |
| **Alternative A:**  Capabilities are simple enums handled by the Game class that are translated into Actions by a processPlayerActions method in Game. | **Alternative B:**  Capabilities are collections composed of Actions, and the Game class simply extracts the Actions from the Capability. |
| **Advantages:**   * Enums offer a simple way to track player states, allowing for easy development. * There are few Capabilities in the current game. An enum list is short, clean and easy to read for a simple game like 9MM. * Enums are immutable, reducing the probability of bugs in string comparisons.   **Disadvantages:**   * In this design, the Game class must match each Capability to a set of Actions. The method of performing this task could become bloated if more Capabilities are introduced. * The simplicity of enums could become a drawback if future Player states increased in complexity. | **Advantages:**   * Capabilities composed of Actions would make clear that, in practice, Capabilities are named sets of Actions that a Player can make each turn. * Composing Capabilities of Actions would allow for a simple method in the Game class that unpacks the Capability handed to it.   **Disadvantages:**   * Capabilities as collections shift complexity from the Game class to the Capability itself. It may be clearer to developers if they could determine the essential process of the game simply through Game, considering the simplicity of 9MM as a whole. * It is unlikely that a 9MM Capability would ever be composed of many Actions, reducing the need for control over the processPlayerActions method. |
| **Decision:**  In the current design, Capabilities are implemented as enums with a 13 line method in the Game class to translate Capabilities into Actions, each of which is executed during a Player’s turn. The primary driver of the use of enums is their simplicity, increasing readability considering the simplicity of the underlying 9MM game. Enums in particular are attractive as immutable data types that are less prone to bugs. However, method bloat may become more significant as future requirements are implemented, and discussion is ongoing as to whether to switch Capabilities to an alternative method. | |

## Capability

| **Issue:**  Should capabilities be composed of Actions or should an overarching game class translate Capabilities into Actions? | |
| --- | --- |
| **Alternative A:**  Capabilities are simple enums handled by the Game class that are translated into Actions by a processPlayerActions method in Game. | **Alternative B:**  Capabilities are collections composed of Actions, and the Game class simply extracts the Actions from the Capability. |
| **Advantages:**   * Enums offer a simple way to track player states, allowing for easy development. * There are few Capabilities in the current game. An enum list is short, clean and easy to read for a simple game like 9MM. * Enums are immutable, reducing the probability of bugs in string comparisons.   **Disadvantages:**   * In this design, the Game class must match each Capability to a set of Actions. The method performing this task could become bloated if more Capabilities are introduced. * The simplicity of enums could become a drawback if future Player states increased in complexity. | **Advantages:**   * Capabilities composed of Actions would make clear that, in practice, Capabilities are named sets of Actions that a Player can make each turn. * Composing Capabilities of Actions would allow for a simple method in the Game class that unpacks the Capability handed to it.   **Disadvantages:**   * Capabilities as collections shift complexity from the Game class to the Capability itself. It may be clearer to developers if they could determine the essential process of the game simply through Game, considering the simplicity of 9MM as a whole. * It is unlikely that a 9MM Capability would ever be composed of many Actions, reducing the need for control over the processPlayerActions method. |
| **Decision:**  In the current design, Capabilities are implemented as enums using a 13 line method in the Game class to translate Capabilities into Actions, each of which is executed during a Player’s turn. The primary driver of the use of enums is their simplicity, increasing readability considering the simplicity of the underlying 9MM game. Enums in particular are attractive as immutable data types that are less prone to bugs.  In contrast, refactoring Capabilities into being collections of Actions suggests a paradigm shift in the code towards composition over inheritance and dependency injections. Method bloat may become more significant as future requirements are implemented, which could make the code base less readable and promote such an approach. Dependency injection might also allow for easier unit testing for a more robust game.  The desire to get the application off the ground before introducing complexity drove the current enum approach as the team’s choice for sprint 2. The team continues to examine the value of either approach and may consider Capabilities as collections in future sprints. | |

## 

## Token Type

| **Issue:**  Should we use an enum class for Token Type or should we simply just use strings instead | |
| --- | --- |
| **Alternative A:**  Use an enum class called Token Type | **Alternative B:**  Use Strings |
| **Advantages:**   * Reduces chances of bugs forming from spelling mistakes * Provides explicit value to specified constant values, making code more maintainable and readable   **Disadvantages:**   * More work required to make an enum class and to ensure enum is used in every relevant area that requires the enum | **Advantages:**   * Much easier to implement as string has to just be entered within code   **Disadvantages:**   * Bugs can form if the string is misspelt * Readability and maintainability decrease with strings as the relevant strings can be harder to identify in the code base. |
| **Decision:**  Apart from the fact Alternative B is easier to implement, Alternative A was the clear choice here, as enums provide a multitude of benefits for our constant values. Using an enum class would minimise human made errors in comparison to string as well as provide explicit mean to each constant, making these constants easier to identify within the codebase. As a result, Alternative B was selected. | |

## 

## Token Stack

| **Issue:**  How should we reduce code duplication in classes that are responsible for providing tokens to other objects, like a player’s TokenBank and an Intersection? | |
| --- | --- |
| **Alternative A:**  Create a TokenStack superclass that the TokenBank and Intersection inherit from, providing shared methods for adding and removing tokens. | **Alternative B:**  Implement separate methods for adding and removing Tokens to and from the TokenBank and Intersections. |
| **Advantages:**   * Reduces code duplication, making for more readable code by simplifying methods for both subclasses of the TokenStack. * Only a slight extension on the included Java Stack utility class, making for clear and easy-to-understand code.   **Disadvantages:**   * Intersections already inherit from JButtons, meaning multiple inheritance would be required to have Intersections also inherit from a TokenStack. | **Advantages:**   * Clearer; it is not immediately intuitive that Intersections are stacks. * Intersection Token getter and setter methods also involve changing the frontend image of an Intersection, meaning the methods for getting and setting are somewhat different.   **Disadvantages:**   * Results in code duplication and messier code, ignoring DRY. |
| **Decision:**  A TokenStack superclass was introduced in order to improve the readability of the codebase, with the small variation on a regular Stack of having a maximum height so as to stop Players from placing piles of tokens on Intersections. The issue of multiple inheritance in Intersections was solved by instead providing a TokenStack to an Intersection as an attribute. The TokenStack methods were called alongside image change methods, making for simple, readable getter and setter methods while maintaining the inheritance of Intersections from JButtons. | |

## Token

| **Issue:**  How can we build the most minimal object possible to keep track of Token data, including which player a Token belongs to? | |
| --- | --- |
| **Alternative A:**  Create a Token object composed of a TokenType, which is shared with a Player who is also composed of a TokenType, and compare the two to determine who a Token belongs to. | **Alternative B:**  Create a Token object that has a shared inheritance chain with the Player; both the Token and the Player inherit from some ‘Type’ class that determines their allegiance. |
| **Advantages:**   * In the opinion of a team, a more intuitive solution. * Allows for the TokenType to be shared to any extended objects that might need to be aware of the object’s allegiance (for example, Blocking objects that stop opponents from moving into an Intersection without actually counting as a Token for that Player) * Relatively easy to write tests compared to an inheritance based solution.   **Disadvantages:**   * Giving the wrong TokenType to a Player or Token might result in a somewhat subtle bug, especially as the number of objects that use the TokenType increases. * Token objects themselves might be simpler in terms of lines of code using inheritance. | **Advantages:**   * Future objects that use the same Type could simply also inherit from Type.   **Disadvantages:**   * Any changes to the overarching Type object could have downstream effects on subclasses that might result in messier code overall. * Not necessarily as representative of the physical game as composition, in which Players choose a Type by selecting a colour of Token. This may make it less immediately understandable to developers. |
| **Decision:**  Composition was selected over inheritance in this relatively niche case as a matter of increased readability for developers with the potential for easier testing in the future should unit tests be developed. It was also felt that downstream effects of changes in inheritance changes could introduce significant technical debt which was not desired. Despite some very slight overhead from the Constructor code because of the use of composition, the Token class remains extremely lightweight, which was the desired outcome considering the simplicity of the object in practice. Switching the object from a regular class to a record is a consideration that the team is taking on in the future as we explore the use cases for records in Java to hold data, considering that the Token is, in essence, a data class. | |

## Game and GameBoardGui Relationship

| **Issue:**  Should the game have a dependency relationship with the GameBoardGui where the game gets used as an attribute to build the game, or should the gameboard use a composition relationship where the gameboard cannot exist without the game? | |
| --- | --- |
| **Alternative A:**  Using dependency relationship between game and gameboard | **Alternative B:**  Using a composition relationship between game and gameboard |
| **Advantages:**   * The GameBoard sets up the Intersections class hence knowing what game it's playing through the Game class allows for the Intersections to pass information through between the classes * The GameBoard and Game having known about each other will also allow it to pass information between the classes and allow the game to use the gameboard GUI’s methods to make the board add/remove the token images * Separates the Game from becoming a god class as the GameBoard handles the GUI functionality on its own without having to duplicate the code or to have too much functionality handled by the game   **Disadvantages:**   * May create a loop with the intersection class which does not adhere to proper object-orientated design * Cannot extend the GameBoard class as it is purely dependent on the Game Class | **Advantages:**   * Allows to extend the GameBoard for future uses * Adheres to object-oriented principles as the Game composes of the gameboard and the gameboard code will not be repeated if ever used again   **Disadvantages:**   * Game class has too much to handle and will become a god class which violates the good design principles * The GameBoard is a GUI so it is redundant and unnecessary to have another class edit its features and extend its methods to create just one game * For the purpose of this game, there is only one nine men's morris game being created, hence there is no need for an extension. |
| **Decision:**  Alternative A was the decision the team went with as this takes the load off of the Gmae Class from having too much responsibility. In this structure of the code, the GameBoardGUI is a Swing-implemented class that creates the front-end part of the game, hence having another class through composition inherit its features is redundant. The composition alternative B was discarded as the Game class becoming a god class will violate the SOLID principles and hence should not be implemented just for one instance of the game with no current intentions of being extended. | |

## TokenStack and Token Cardinality

TokenStack has 0 to 9 tokens, and this represents how during the gameplay, as the player places a token down, the number of tokens the player possesses drops from 9 to 0. Therefore it was only logical to assign a cardinality of 0 to 9, as a player at any time can have up to 9 tokens at most, and 0 if all tokens have been placed down

## MillObserver and Token Cardinality

A MillObserver can have 0 to 3 tokens at any time. The main goal of the MillObserver class is to detect the formation of a mill for an assigned set of 3 tokens. Rather than having a single MillObserver class looking at all tokens at once, and effectively having a 2 to 9 token relationship, each MillObserver can observe 0 to 3 tokens, whereas the player places down more tokens in the designated mill, the tokens observed go from 0 up to a maximum of 3.

# Appendix A: Design Rationale Guidelines

## Guidelines

# 

## 

## Template

| **Issue:** | |
| --- | --- |
| **Alternative A:** | **Alternative B:** |
| **Advantages:**  **Disadvantages:** | **Advantages:**  **Disadvantages:** |
| **Decision:** | |

## 