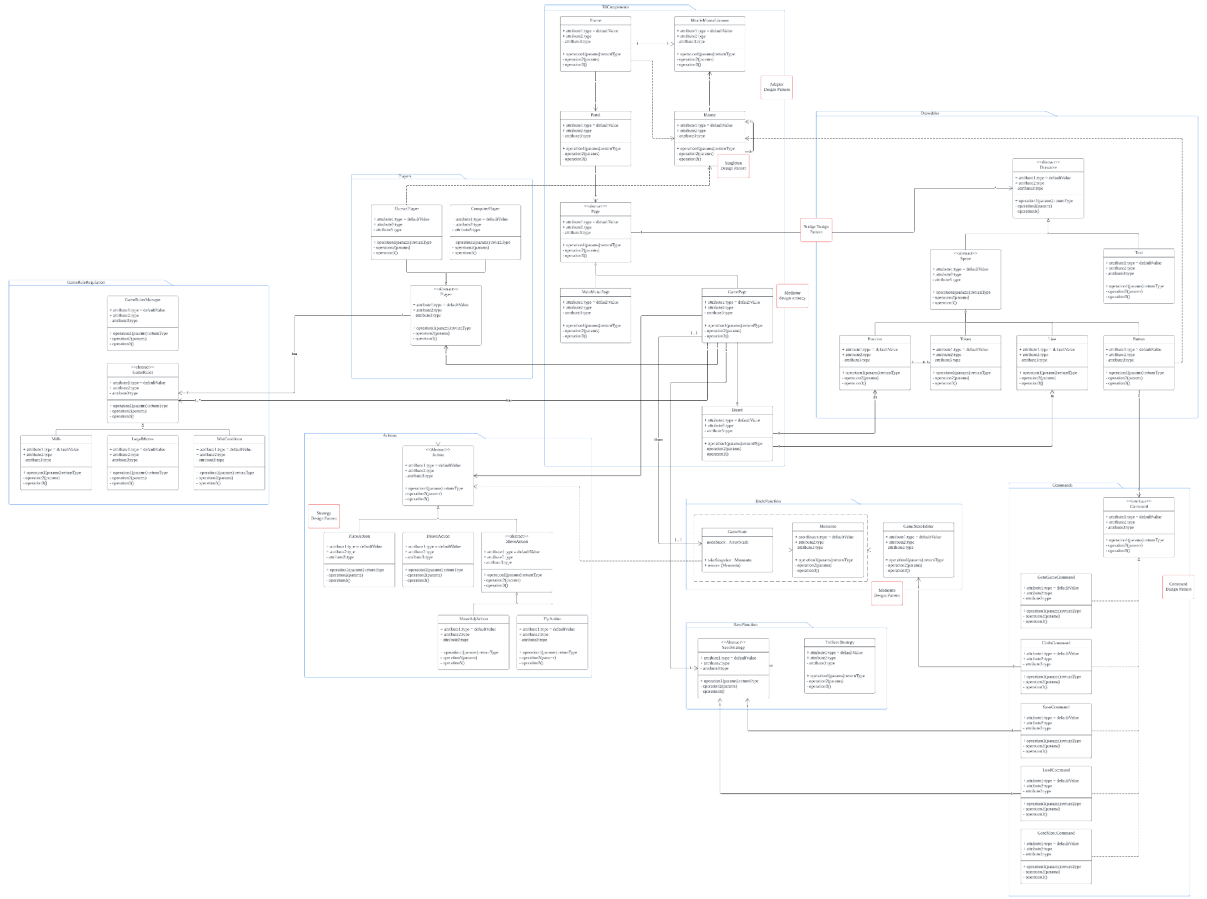
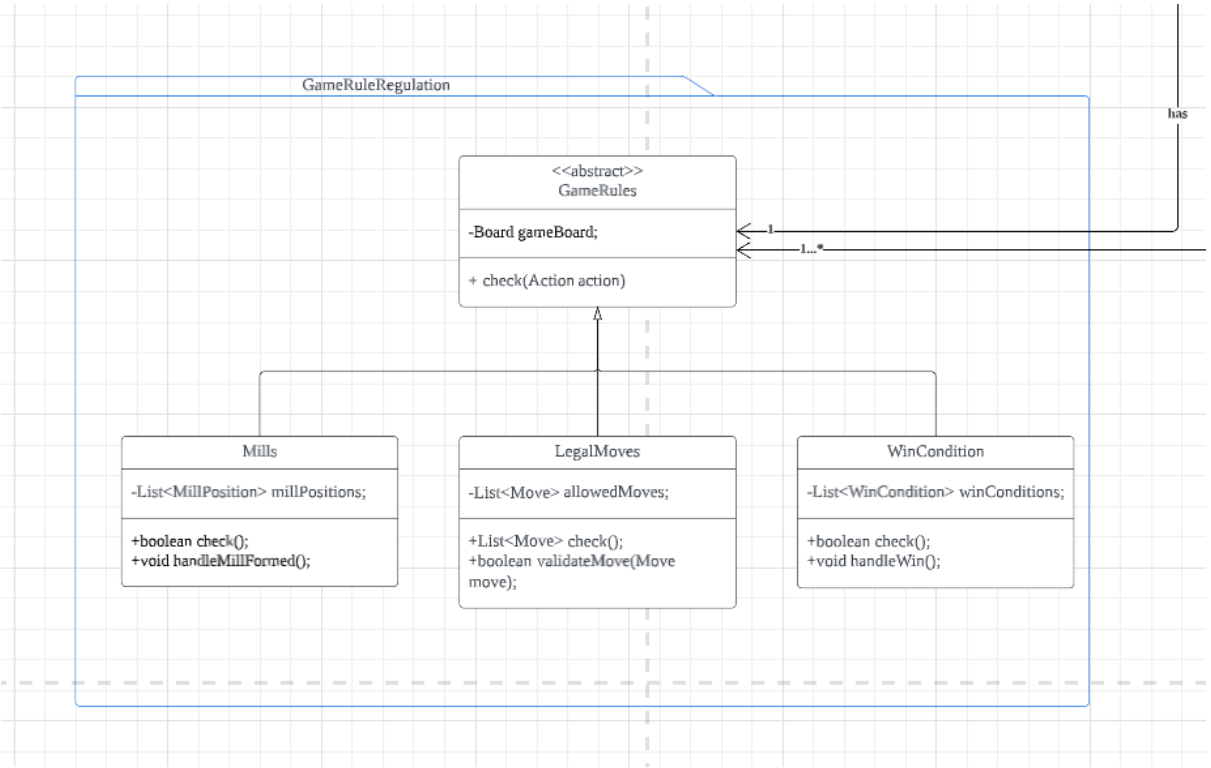
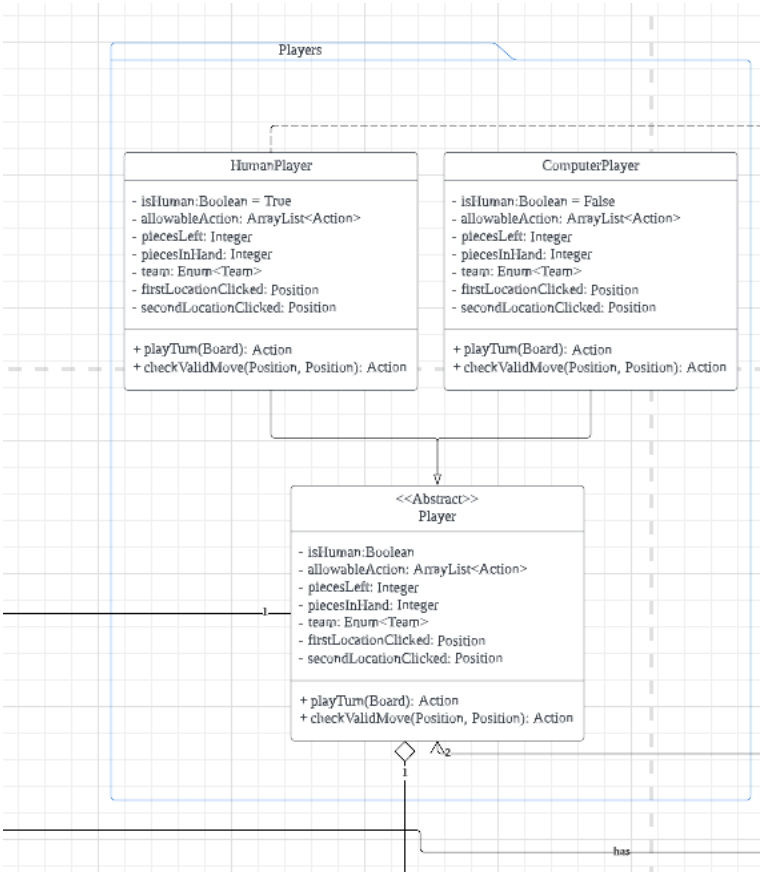
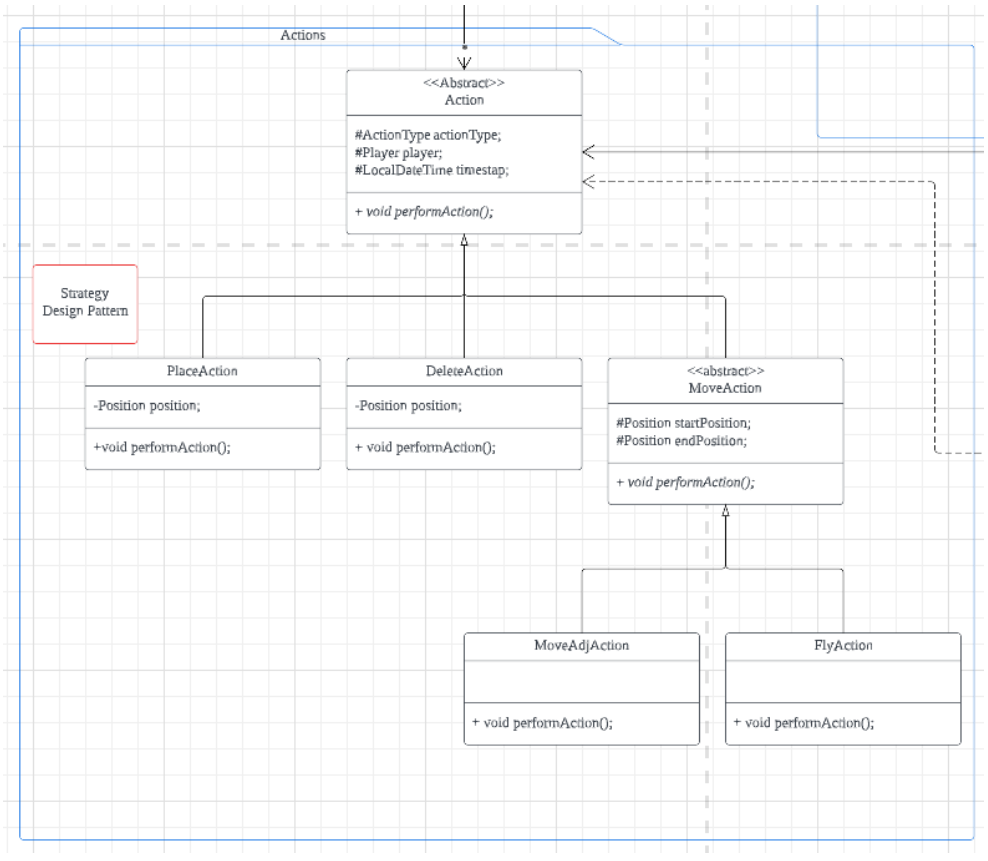
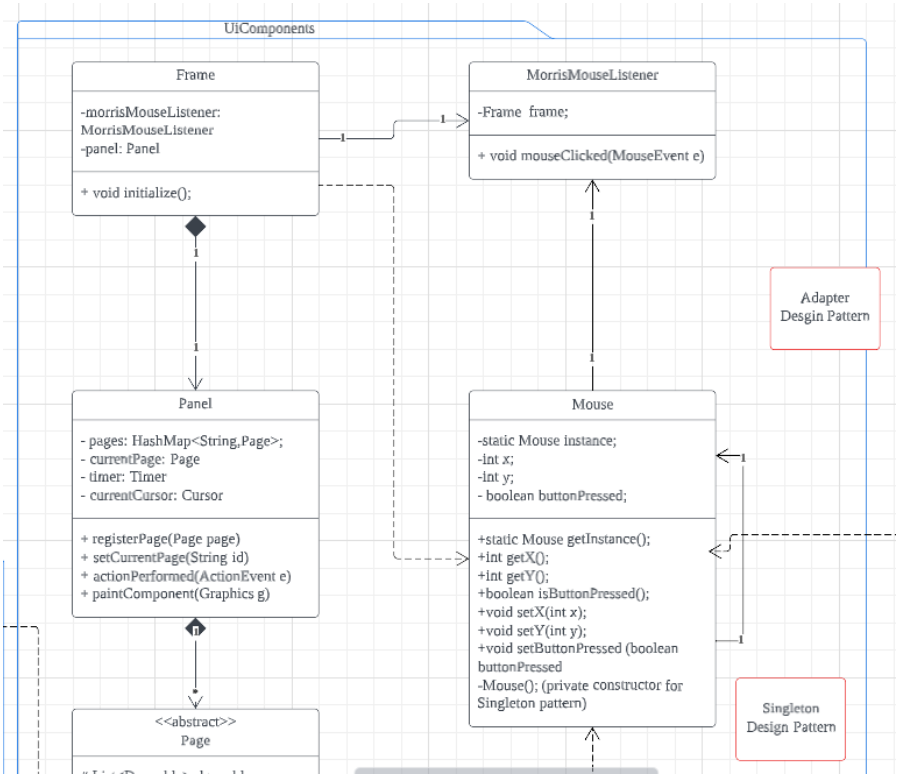
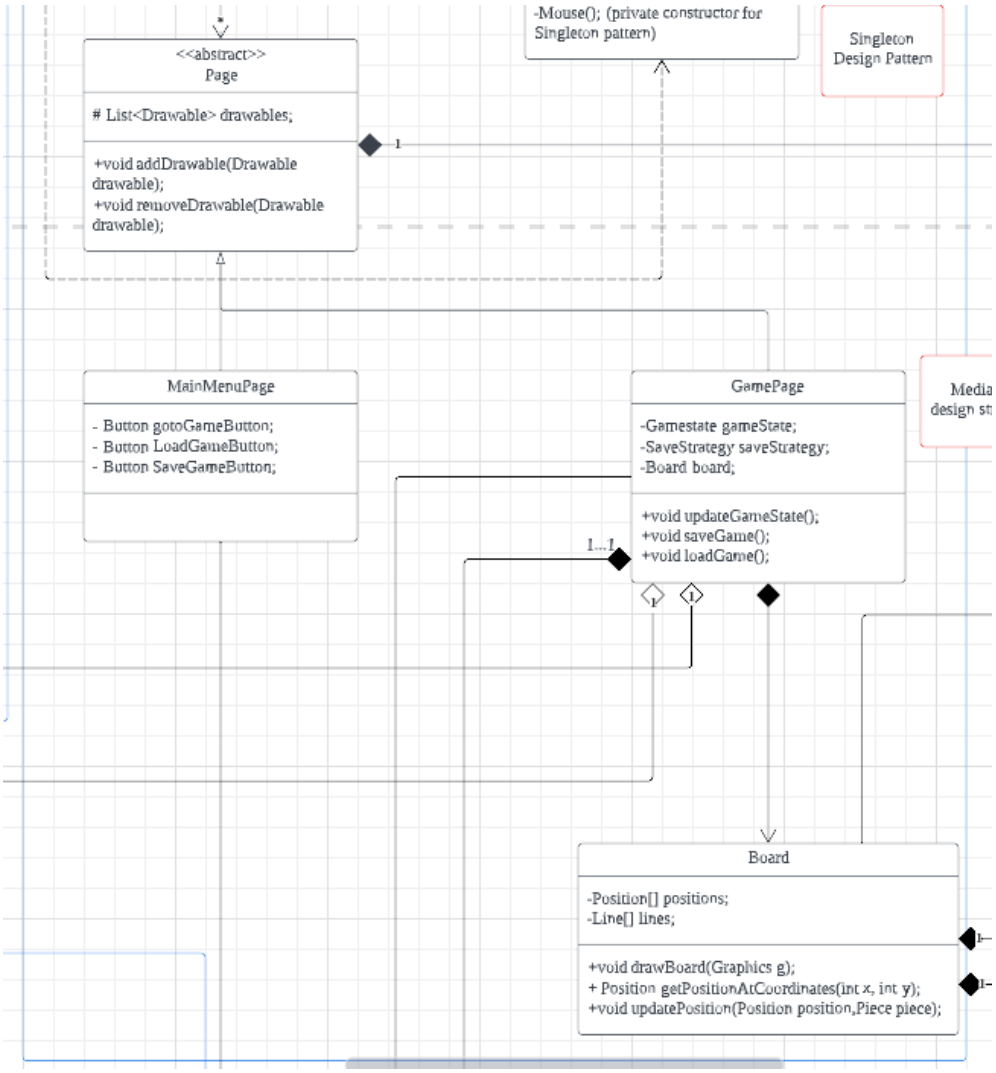


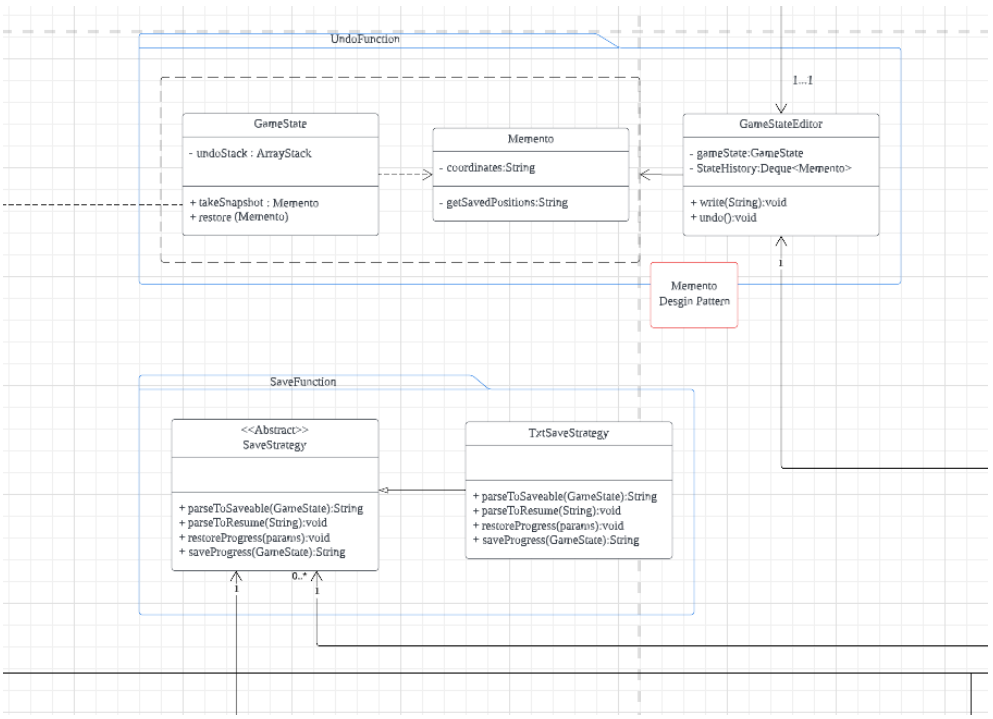

GameRuleRegulation
has
<<abstract>>
GameRules
-Board gameBoard;
+ check(Action action)
1
1...*
Mills
-List<MillPosition> millPositions;
+boolean check();
+void handleMillFormed();
LegalMoves
-List<Move> allowedMoves;
+List<Move> check();
+boolean validateMove(Move move);
WinCondition
-List<WinCondition> winConditions;
+boolean check();
+void handleWin();

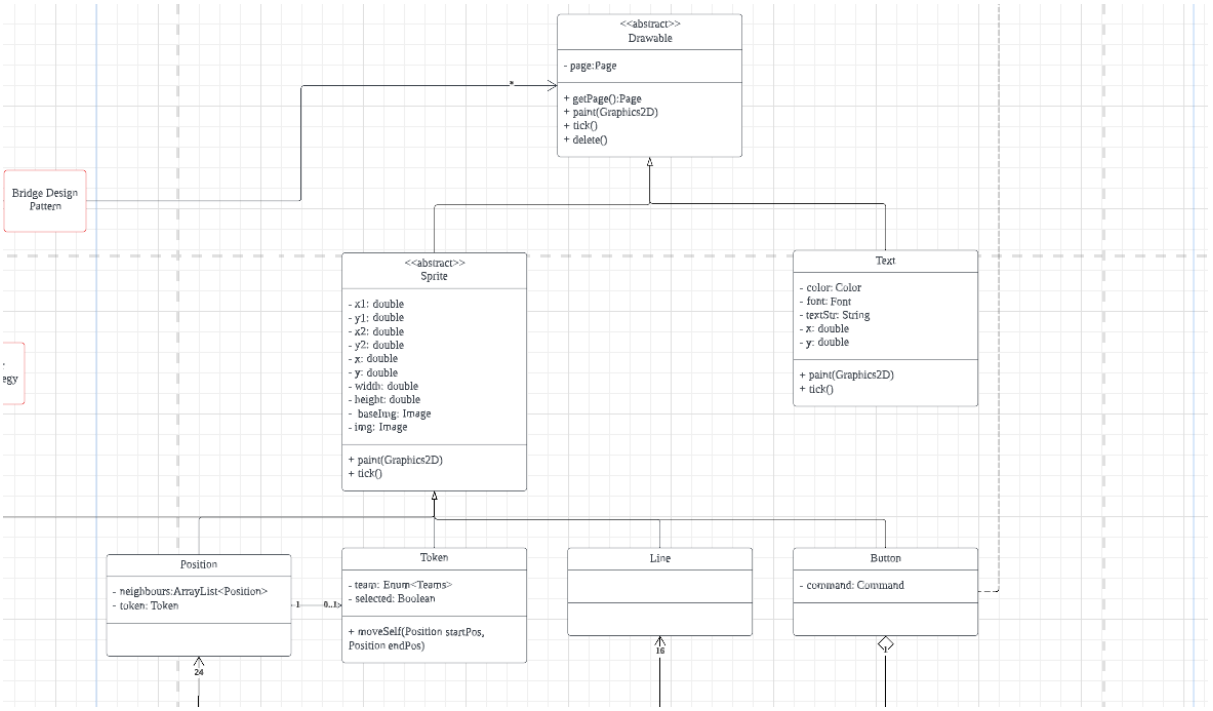

Players
HumanPlayer
- isHuman:Boolean = True
- allowableAction: ArrayList<Action>
- piecesLeft: Integer
- piecesInHand: Integer
- team: Enum<Team>
- firstLocationClicked: Position
- secondLocationClicked: Position
+ playTurn(Board): Action
+ checkValidMove(Position, Position): Action
ComputerPlayer
- isHuman:Boolean = False
- allowableAction: ArrayList<Action>
- piecesLeft: Integer
- piecesInHand: Integer
- team: Enum<Team>
- firstLocationClicked: Position
- secondLocationClicked: Position
+ playTurn(Board): Action
+ checkValidMove(Position, Position): Action
<<Abstract>>
Player
- isHuman:Boolean
- allowableAction: ArrayList<Action>
- piecesLeft: Integer
- piecesInHand: Integer
- team: Enum<Team>
- firstLocationClicked: Position
- secondLocationClicked: Position
+ playTurn(Board): Action
+ checkValidMove(Position, Position): Action
1
2
1
has

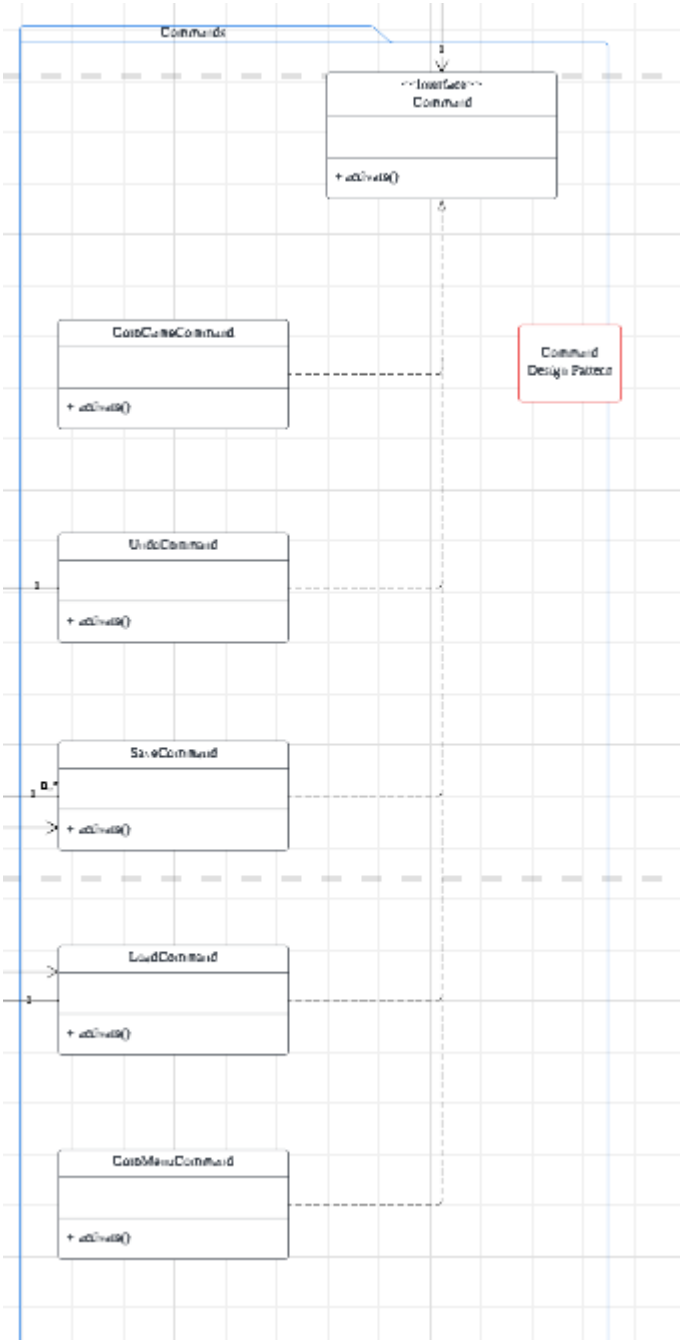

Actions
<<Abstract>>
Action
#ActionType actionType;
#Player player;
#LocalDateTime timestap;
+ void performAction();
Strategy
Design Pattern
PlaceAction
-Position position;
+void performAction();
DeleteAction
-Position position;
+ void performAction();
<<abstract>>
MoveAction
#Position startPosition;
#Position endPosition;
+ void performAction();
MoveAdjAction
+ void performAction();
FlyAction
+ void performAction();


UiComponents
Frame
-morrisMouseListener: MorrisMouseListener
-panel: Panel
+ void initialize();
MorrisMouseListener
-Frame frame;
+ void mouseClicked(MouseEvent e)
Adapter Desgin Pattern
Panel
- pages: HashMap<String,Page>;
- currentPage: Page
- timer: Timer
- currentCursor: Cursor
+ registerPage(Page page)
+ setCurrentPage(String id)
+ actionPerformed(ActionEvent e)
+ paintComponent(Graphics g)
Mouse
-static Mouse instance;
-int x;
-int y;
- boolean buttonPressed;
+static Mouse getInstance();
+int getX();
+int getY();
+boolean isButtonPressed();
+void setX(int x);
+void setY(int y);
+void setButtonPressed (boolean buttonPressed
-Mouse(); (private constructor for Singleton pattern)
Singleton Design Pattern
<<abstract>>
Page


-Mouse(); (private constructor for Singleton pattern)
Singleton Design Pattern
<<abstract>>
Page
List<Drawable> drawables;
+void addDrawable(Drawable drawable);
+void removeDrawable(Drawable drawable);
MainMenuPage
- Button gotoGameButton;
- Button LoadGameButton;
- Button SaveGameButton;
GamePage
-Gamestate gameState;
-SaveStrategy saveStrategy;
-Board board;
+void updateGameState();
+void saveGame();
+void loadGame();
Board
-Position[] positions;
-Line[] lines;
+void drawBoard(Graphics g);
+ Position getPositionAtCoordinates(int x, int y);
+void updatePosition(Position position,Piece piece);


UndoFunction
1...1
GameState
- undoStack : ArrayStack
+ takeSnapshot : Memento
+ restore (Memento)
Memento
- coordinates:String
- getSavedPositions:String
GameStateEditor
- gameState:GameState
- StateHistory:Deque<Memento>
+ write(String):void
+ undo():void
1
Memento
Desgin Pattern
SaveFunction
<<Abstract>>
SaveStrategy
+ parseToSaveable(GameState):String
+ parseToResume(String):void
+ restoreProgress(params):void
+ saveProgress(GameState):String
TxtSaveStrategy
+ parseToSaveable(GameState):String
+ parseToResume(String):void
+ restoreProgress(params):void
+ saveProgress(GameState):String
1
0..*
1


<<abstract>>
Drawable
- page:Page
+ getPage():Page
+ paint(Graphics2D)
+ tick()
+ delete()
Bridge Design
Pattern
<<abstract>>
Sprite
- x1: double
- y1: double
- x2: double
- y2: double
- x: double
- y: double
- width: double
- height: double
- baseImg: Image
- img: Image
+ paint(Graphics2D)
+ tick()
Text
- color: Color
- font: Font
- textStr: String
- x: double
- y: double
+ paint(Graphics2D)
+ tick()
Position
- neighbours:ArrayList<Position>
- token: Token
Token
- team: Enum<Teams>
- selected: Boolean
+ moveSelf(Position startPos,
Position endPos)
Line
Button
- command: Command
1
0..1
24
16



https://lucid.app/lucidchart/27de8e84-ffac-4296-8a15-8af769bc755a/edit?viewport_loc=1773%2C1912%2C2949%2C1416%2CyNnNF9y.RUGd&invitationId=inv_0671a230-e65d-416a-b124-591c7fd13cf0

Explain two key classes that you 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?

(1st one can be rule checker(s))
(2nd can be …)
Mill

The Mill class from the GameRuleRegulation package was one of the important classes that the team debated. The purpose of this class is to keep track of combinations of Positions that when occupied by a set of Tokens result in a Mill. We considered multiple approaches to representing the mill feature of the game, such as having a standalone class, having a list of three positions within the Board to represent the mill, and having a class that inherits from a shared class.

We considered using a list of three positions to represent a Mill, as it would be possible to loop through all three positions in order to determine whether a mill has occurred. Such an approach would reduce the complexity of the overall design by eliminating a class and integrating its functionality into the Board. However, we decided against this because we wanted to avoid adding excessive responsibilities to the Board class, and because we wanted to ensure that we could easily add additional features in the event that we need to add functions and attributes related to mills into an encapsulated class.

While we initially decided to have the Mill as a standalone class, we instead decided to have the Mill class inherit from the GameRules class. This is because we realized that the Mill class shares functionality with the LegalMoves and WinCondition classes, including the ability to validate actions based on the current status of the game. As such, having all three classes inherit from the GameRules class would help the design adhere to the Don't Repeat Yourself (DRY) principle and therefore improve code maintainability.

Command

The team also discussed the Command interface from the Commands package as a crucial class. The group first intended to make the different button effects, such as going to a different page or undoing an action, available as a subclass of the Button class. This solution was rejected in favour of the Command interface so that alternative methods, including keyboard shortcuts, could be used to access the button's effects. This choice was chosen to maintain the open-closed principle and streamline activation methods.

We furthermore needed to consider whether the Command interface would be better represented as an abstract class. While using an abstract class would be preferable if there were shared functionality between subclasses, we realized that this would be unlikely as their only shared functionality is that they can be activated by the same function. As such, representing Command as an interface is preferable.

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

1)

One key relationship in the class diagram is the composition from the Page class to the Drawable class. This relation is necessary as it allows for the Page to keep track of all the Drawable instances on it, allowing the page to draw the Drawable instances onto the screen, along with periodically calling each Drawable instance to run anything that needs to run every frame.

This relation is represented as a composition because the Drawable class is intrinsically tied to the Page class. This is because switching to another Page would result in the Drawable instances of the initial page being ignored until the initial page is used again.

2)

One other key relationship is the aggregation relationship between the **GamePage** class and the **GameRules** class. This relation is important as the **GamePage** is used to dictate the flow of the game, this means that when an essential element of checking and maintaining game rules such as checking for win conditions and checking for mill, the **GamePage** class would need to access the methods within the **GameRules** and even store an (or many) instance(s) of **GameRule** within itself and this justifies the association.

There is an aggregation between the relationship as **GamePage** can be said to have an (or more) instance(s) of **GameRule** class within itself but the **GameRule** class is not completely useless without the **GamePage** class and can still have function outside the **GamePage** class such as performing valid move checks in the **Player** classes. Hence, the relationship is an aggregation but not a composition.

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

(This can discuss inheritance of classes in GameRuleRegulation folder)

To lessen code duplication and encourage code reuse, the team chose to employ inheritance in the GameRuleRegulation package. To share similar capabilities, such as the validation of actions based on the game state, the Mills, LegalMoves, and WinCondition classes are all inherited from the GameRules class.
Instances when inheritance did not make sense, however, were avoided. For instance, the abstract Player class, the Player class, and the ComputerPlayer and HumanPlayer subclasses all share properties and functionality. The Drawable, Sprite, Text, Position, Token, and Line classes, however, did not inherit from each other because they each have unique functionalities and unwanted coupling would have resulted from inheritance.

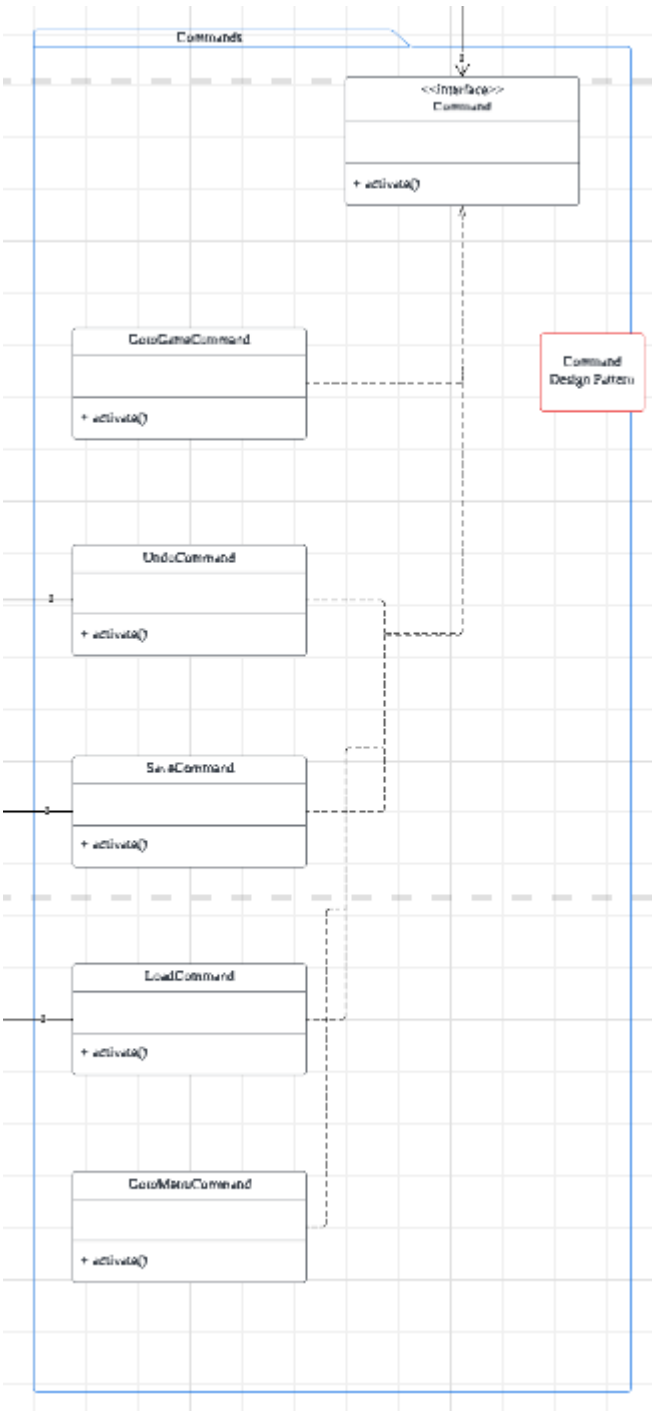
Explain how you arrived at two sets of cardinalities, e.g. why 0..1 and why not 1…2?

(1st one can be GamePage & GameRules<abstract>) [discuss violation of SRP]
(2nd one can be Player and GameRules<absrtact>)

One cardinality that we needed to consider was the relationship between the GamePage and the Player class, where we had to decide between a 1..2 link and a 1..* link. While the game is intended to support 2 players, we considered whether we wanted the design to have the potential to support an arbitrary number of players in order to improve extensibility. Furthermore, we discussed whether structuring the code to support many players would help simplify our implementation. We however realised that doing so would significantly complicate the structure of the game, as we would need to provide a dynamic system for setting the images of the game tokens, along with having to ensure that the program functions correctly with more or less than two players. As it is highly unlikely that supporting a variable number of players will be necessary, we decided to use a 1..2 link between GamePage and Player.

A 1..1 link exists between the Player class and the GameRules abstract class in the GameRuleRegulation package. We decided to connect GameRules to Player instead of its subclasses due to the fact that the game's rules will apply to both human and computer-controlled players equally. Furthermore, we decided that a 1..1 link is preferable to a 1..* link as it shouldn’t be possible for a player to abide by multiple sets of GameRules. As such, implementing the cardinality as such would be ideal.

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



In the above diagram is the implementation of the multiple buttons that will be used in the application to conduct multiple actions such as undoing, starting the game and so on. However when contemplating the design to follow to implement the buttons for the application, there were 3 methods that came to mind:

- One class **Button** and many methods
- Have multiple class implement a abstract **Button** class
- *Command Pattern*™

The first option was to implement one single **Button** class but with multiple methods, where each method corresponds to a specific function. However this became apparent that it would

violate the Single Responsibility Principle (SRP). This would make a god class out of the **Button** class and would cause many issues going down the line.

The second option was to implement one single **Button** abstract class while having all possible versions of the other buttons extend the **Button** abstract class. If the application was confirmed to be of small scale and the number of subclasses of the **Button** abstract class was fixed to a small number, this option would be feasible. However, this application is still in development and there may be more features that need to be added which may include newer buttons to provide an interface to access these newer features; it would be possible to go down the path of creating subclasses for each new button but this approach is deeply flawed. First, we would then have an enormous number of subclasses, and this would mean risking breaking the code in these subclasses each time we would modify the base **Button** class. Our GUI code will have become dependent on the volatile code of the business logic.

So our final option, and most effective option would be to implement the *Command Pattern* whereby a command interface would be created which would be implemented in the concrete command classes. So this would greatly decouple the dependency between the GUI code and the business logic. This would also mean that the button class would only be responsible for sending an encoded command to prompt some functionality instead of having multiple subclasses that extend themselves.