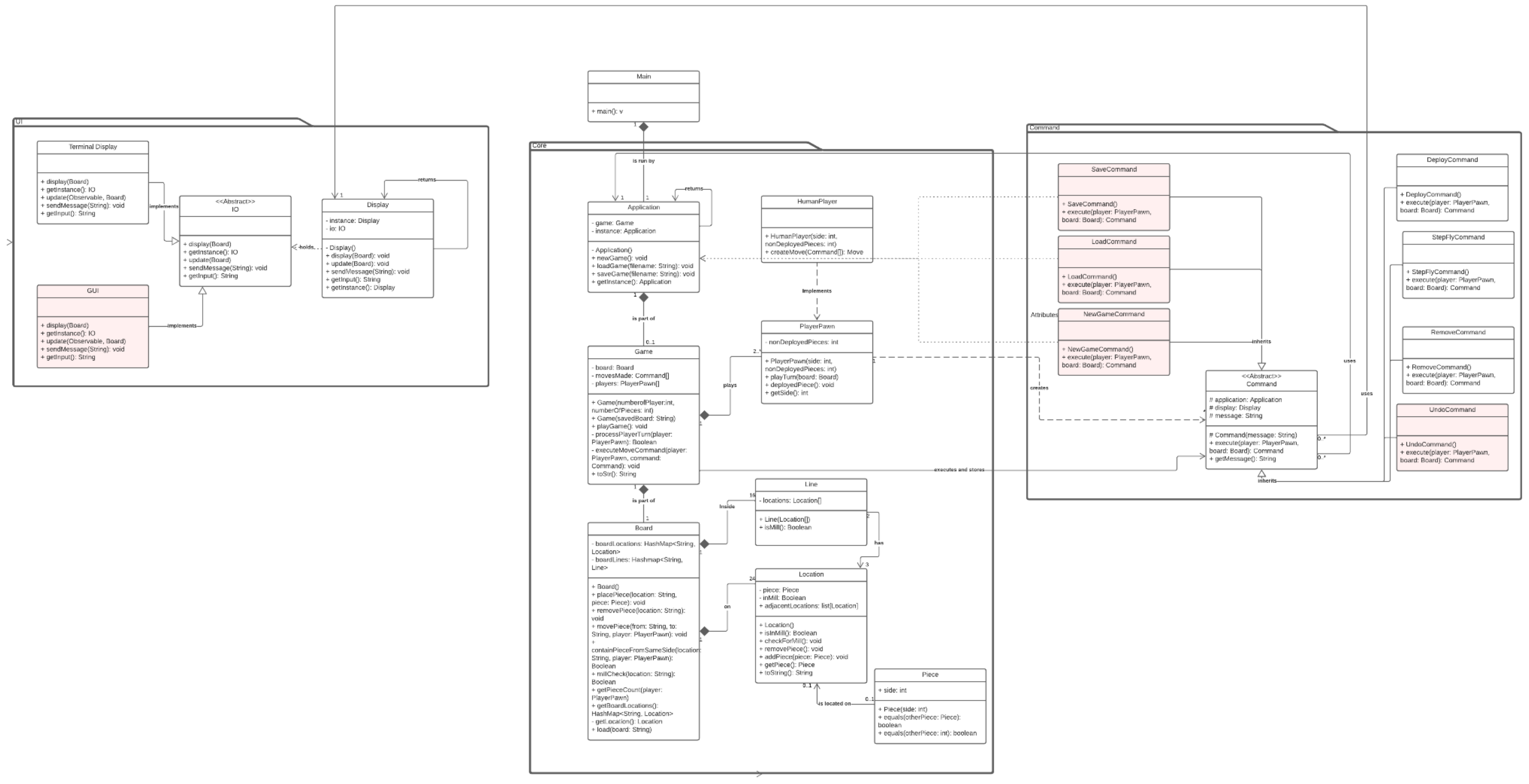
FIT3077 Sprint Two Architecture and Design Rationales Report

3 Man’s Mischief (Team 38)

Josiah Schuller, Caleb Ooi, Steven Pham

1 Class Diagram


UI
Terminal Display
+ display(Board)
+ getInstance(): IO
+ update(Observable, Board)
+ sendMessage(String): void
+ getInput(): String
implements
<<Abstract>>
IO
+ display(Board)
+ getInstance(): IO
+ update(Board)
+ sendMessage(String): void
+ getInput(): String
holds
Display
- instance: Display
- io: IO
- Display()
+ display(Board): void
+ update(Board): void
+ sendMessage(String): void
+ getInput(): String
+ getInstance(): Display
returns
1
GUI
+ display(Board)
+ getInstance(): IO
+ update(Observable, Board)
+ sendMessage(String): void
+ getInput(): String
implements
Main
+ main(): v
1
Core
is run by
1
1
returns
Application
- game: Game
- instance: Application
- Application()
+ newGame(): void
+ loadGame(filename: String): void
+ saveGame(filename: String): void
+ getInstance(): Application
1
is part of
0..1
HumanPlayer
+ HumanPlayer(side: int, nonDeployedPieces: int)
+ createMove(Command[]): Move
Implements
PlayerPawn
- nonDeployedPieces: int
+ PlayerPawn(side: int, nonDeployedPieces: int)
+ playTurn(board: Board)
+ deployedPiece(): void
+ getSide(): int
2..*
plays
1
1
Game
- board: Board
- movesMade: Command[]
- players: PlayerPawn[]
+ Game(numberofPlayer:int, numberOfPieces: int)
+ Game(savedBoard: String)
+ playGame(): void
- processPlayerTurn(player: PlayerPawn): Boolean
- executeMoveCommand(player: PlayerPawn, command: Command): void
+ toStr(): String
executes and stores
1
is part of
1
Board
- boardLocations: HashMap<String, Location>
- boardLines: Hashmap<String, Line>
+ Board()
+ placePiece(location: String, piece: Piece): void
+ removePiece(location: String): void
+ movePiece(from: String, to: String, player: PlayerPawn): void
+ containPieceFromSameSide(location: String, player: PlayerPawn): Boolean
+ millCheck(location: String): Boolean
+ getPieceCount(player: PlayerPawn)
+ getBoardLocations(): HashMap<String, Location>
- getLocation(): Location
+ load(board: String)
Inside
16
1
Line
- locations: Location[]
+ Line(Location[])
+ isMill(): Boolean
2
has
3
on
24
1
Location
- piece: Piece
- inMill: Boolean
+ adjacentLocations: list[Location]
+ Location()
+ isInMill(): Boolean
+ checkForMill(): void
+ removePiece(): void
+ addPiece(piece: Piece): void
+ getPiece(): Piece
+ toString(): String
0..1
is located on
0..1
Piece
+ side: int
+ Piece(side: int)
+ equals(otherPiece: Piece): boolean
+ equals(otherPiece: int): boolean
Command
SaveCommand
+ SaveCommand()
+ execute(player: PlayerPawn, board: Board): Command
LoadCommand
+ LoadCommand()
+ execute(player: PlayerPawn, board: Board): Command
Attributes
NewGameCommand
+ NewGameCommand()
+ execute(player: PlayerPawn, board: Board): Command
inherits
creates
<<Abstract>>
Command
# application: Application
# display: Display
# message: String
# Command(message: String)
+ execute(player: PlayerPawn, board: Board): Command
+ getMessage(): String
0..*
0..*
uses
uses
inherits
DeployCommand
+ DeployCommand()
+ execute(player: PlayerPawn, board: Board): Command
StepFlyCommand
+ StepFlyCommand()
+ execute(player: PlayerPawn, board: Board): Command
RemoveCommand
+ RemoveCommand()
+ execute(player: PlayerPawn, board: Board): Command
UndoCommand
+ UndoCommand()
+ execute(player: PlayerPawn, board: Board): Command

2 Design Rationales

2.1 Key Classes

Line

The Line class is responsible for keeping track of the location and the pieces relative to each other, as well as checking if a mill occurs. We decided to have a Line class instead of a unary relationship within the Location class because it reduces the code needed during the creation process. During the creation of the board, each Location would need to be injected with four other Locations for both horizontal and vertical relative positions, while the line implementation would only require the creation of 16 Line classes and the injection of each location twice.

One advantage of having a Line class is that when a piece is placed on the Location, the Location itself can verify if a mill occurs. However, we managed to simplify this process by using a hash map to store the Lines. The Lines can be easily retrieved by indexing the row and column of the modified Location.

Additionally, implementing the Line classes also decoupled the responsibility of checking a mill from the Location class, allowing for easy extension of other size boards.
One downside of using line is that it introduces additional complexity when a move occur. The game which is responsible for assigning moves to the player would need to query the line object every time a move occurs, which makes it more prone to bugs and human error.

PlayerPawn

The PlayerPawn class is an abstract class responsible for handling how a Player's turn executes during a round of Nine Men's Morris. We decided to use a PlayerPawn rather than just have a singular Player class as it allows for different players to have different implementations on how their turn plays out. This allows for future upgradability as we may have an AI player whose turn executes completely differently (such as generating the moves from a random generator) from a Human Player (who needs to interact with the I/O). Specific types of Players need to be initialised on Game creation (such as Player vs. Player or AI vs. Player) depending on the type of game that was selected.

It has the advantages of making the program more flexible in extension as new types of Players just need to inherit and implement the PlayerPawn then it can interact with the rest of the Game without needing to hardcode anything. And makes the code more reusable as player classes can be swapped out with different implementations while keeping the rest of the code intact whereas if it was a method then the rest of the system would need to be refactored whenever a player is modified or removed. It also makes the Class distinction more clear as Player logic and interaction with the I/O is not a responsibility of the Board or Game.

The PlayerPawn class was debated by the team on whether or not it should exist but it was ultimately decided that a PlayerPawn class allows for better upgradeability, for things such as AI players, and reusability as opposed to having just one Player class or relegating it to a method inside the Board class.

2.2 Key Relationships

Location and Board

Location has a composite relationship with Board. This means that the Board class is tightly coupled with the Location class, and modifying one may require changes to the other. While the Location class is responsible for keeping track of the placed pieces on the board, this implementation may make it harder to extend the code in the future. However, the composite relationship is required as it helps maintain the integrity of the Board and prevents new locations from being added to the Board.

Game and Command

Game and Command classes have an association relationship with each other instead of a dependency relationship because the relationship allows the game to keep track of what commands have been executed, enabling the undo functionality. However, having this coupling between the model (game class) and the controller (command class) can result in tight coupling, making the controller code less modular. To solve this problem, we implemented the command design pattern, which helps decouple the game and command class by providing a standard interface on how the commands should behave. Allowing the game to execute the command without having to know the implementation of the subclasses.

2.3 Inheritance

Inheritance is utilised in our object-oriented programming to reduce repeated code, enhance polymorphism and for extensibility.

Command

The classes SaveCommand, LoadCommand, NewGameCommand, DeployCommand, StepFlyCommand, RemoveCommand and UndoCommand all inherit from the Command class. We used inheritance here because all commands have the same attributes (application, io and message), so these attributes can be defined in the parent class (Command) and all the other classes can inherit these attributes. If we had used an interface instead of inheritance, the classes implementing the interface would all be repeating the same code to declare and instantiate the attributes.

Using inheritance is also essential for polymorphism. The PlayerPawn and HumanPlayer use a list of Command classes, however these Command classes could be any of the child classes that inherit from the Command class.

Inheritance was also used for extensibility, i.e. when we want to create a new type of command in the future. Because of inheritance, creating a new child class of Command would not cause us as developers to change other classes also.

IO

We decided to make the IO class that interacts with the game and the game logic an abstract class and have implementations such as the TerminalIO (and in the future the GUI class) inherit the IO class. This is useful for polymorphism as the rest of the game needs to interact with the IO and call upon methods inside the IO to properly interact with the user but by having the IO be an abstract class, we can have different implementations of the I/O such as different interpretations of a Terminal or different interpretations of a GUI allowing for flexibility in designing and creating the game. This is possible as inputs to the game are simple so a display would just need to mimic those inputs then it would integrate with the game automatically. This inheritance allows for flexibility and extensibility in how we design an interface for the user (create a new IO that inherits the IO abstract parent class) as otherwise hardcoded implementations may require the rest of the system to be updated.

2.4 Cardinalities

Two sets of cardinalities will be rationalised.

Application and Game

Firstly, the cardinalities of the relationship between Application and Game are “1” and “0..1” respectively, i.e. there is one Application for any number of Games. This is because when the application is first opened, no game has started - at this moment, there are zero Games for the one Application. When the user starts a new game, there is now one Game for the one Application. There cannot be more than one Game for each Application, because each time the user starts a new game or loads a game in, the previous game is overwritten.

Line and Location

The cardinalities of the relationship between Line and Location are “2” and “3” respectively, i.e. there are two Lines for each Location and three Locations for each Line. Each Location represents a place where a token can be placed. As described in section 2.1, the Line class represents each horizontal or vertical line at its entire span. As one can observe on a Nine Men’s Morris board, every horizontal/vertical line has three locations on it, and thus the cardinality of “3”. If you continue to look at the board, you will also notice that every location has two lines it intersects with - always one vertical line and one horizontal line - and thus the cardinality of “2”.

2.5 Design patterns

Display

Singleton

We decided to use a Singleton design pattern to coordinate the Display and its implementation with the I/O. This is because only one instance of the Display should exist at a time, having multiple displays may result in conflicts. This design pattern is also useful in reducing resource overhead as it prevents multiple instances from existing at once making the game run more efficiently. It also benefits in development as it makes it simple to work with the I/O. A singleton results in the classes being flexible as modifications to the I/O only need to be done to the Display singleton rather than having to refactor the rest of code if it had been hardcoded. A singleton prevents human error as references to the object do not need to be managed which could result in bugs.

In this particular instance, Singleton doesn't suffer from the normal disadvantages associated with it. Components don't know more than they otherwise would, they just interact with the I/O as necessary. And the Singleton hardly affects unit testing as the I/O wouldn't have been unit tested (it would have been system tested) and the inputs from the Display can be mocked easily as the inputs are very simple. Thus, we decided to go with the Singleton design pattern for the Display.

Feasible Alternatives

Observable

We decided against using the Observable design pattern for the Display implementation. While implementing the Observable pattern could decouple the UI interaction from the game, allowing for different UI implementations to be injected and registered in the Observable class, we chose not to use it. Our current implementation of the game only includes one display, and setting up the Observer, Observable interface, and registering method would increase the complexity of the code without providing significant benefits.
Additionally, our Display implementation handles user input, which may not work well in an Observable pattern since the Observable classes do not have direct access to the game.

Chain of responsibility

We decided not to use a chain of responsibility design pattern to coordinate the Display. This is because when there is a long chain of classes in a row handling Display, the structure becomes complex. This complexity will make bugs harder to track (because we do not know where in the chain the bug arises from). If one element of the chain breaks, then the whole chain will break. By letting various classes have access to the Display, we allow these classes a direct connection to the Display, making I/O handling easier and easier to track. The direct connection also allows for better performance, while a chain of responsibility would slow the Display communication down.