Software Architectural Design Document

COMP20050

HEXOUST PROJECT

By GROUP 13:

Cian Latchford - 23201433

Federica Fucetola - 23922982

Laura González Calleja - 23713479

RESEARCH AND INFORMATION ABOUT SOFTWARE ARCHITECTURAL DESIGN

What is Software Architectural Design?

It's the blueprint for how a system is structured, defining its components, their interactions, and overall behavior. According to IEEE, it's the process of organising hardware and software components to develop a complete computer system. Key elements include:

- **Components** (e.g., databases) perform essential functions.
- **Connectors** to enable communication and coordination between components.
- **Conditions** to determine how elements integrate.
- **Semantic models** to define general properties.

What are Different Software Architecture Styles and Characteristics?

1. **Blackboard or Repository Architecture**: Components work together by improving a shared data structure ("blackboard").
   - **Pros:** Centralised data, easy to modify and extend.
2. **Event-Driven Architecture**: Components react to events(changes in state or occurrence) asynchronously and independently.
   - **Pros:** Scalable, flexible, and ideal for real-time applications.
3. **Microservices or Services Architecture**: A system is broken into independent services each with a specific function.
   - **Pros:** Modular, scalable, and supports multiple technologies.
4. **Layered Architecture**: Organises the system into distinct layers. Each layer interacts with the one directly above or below it.
   - **Pros:** Modular, scalable, and supports multiple technologies.
5. **Hexagonal Architecture (Ports and Adapters):** Separates **core logic** from external systems (UI, databases, APIs) using **ports and adapters** for flexibility.
   - **Pros:** Easily adaptable, allows switching technologies without changing the core logic.

6. **Pipe-and-Filter Architecture:** Processes data in sequential steps, where each step (**filter**) transforms data before passing it to the next.

- **Pros:** Easy to modify steps, supports parallel processing, and improves maintainability.

What are the styles and patterns best suited for this project?

For implementing **HexOust** in Java, the **Model-View-Controller** (**MVC**) **pattern** is the best choice for structuring the game, while elements of the **Blackboard Architecture Style** help manage the game state efficiently.

1. Model-View-Controller (MVC) – Pattern

- **Model** (**Game Logic & Data Management**): Stores board state, player moves, and rules for capturing stones. It ensures the game follows its mechanics correctly.
- **View** (**User Interface**): Displays the board and game updates. This could be a **console-based**, **GUI**, or even a **web-based** interface.
- **Controller** (**User Input & Game Flow**): Handles player moves, validates inputs, and ensures turn-based gameplay runs smoothly.

How MVC applies to HOS:

- **Model:** The Board class maintains hexagonal cells, while Player and MoveProcessor handle placing stones and capturing mechanics.
- **View:** A BoardRenderer class updates the game display, whether in a console or GUI format.
- **Controller:** The GameController class takes player input, checks valid moves, updates the board, and manages turns.

2. Blackboard Architecture – Style

Blackboard Architecture helps centralise game state updates efficiently. The **blackboard** is the shared **game state**, ensuring every component (move validation, captures, win conditions) always works with the most up-to-date board.

How Blackboard Architecture applies to HOS

- **Blackboard** (**Game State - Model Layer**)**:** A GameState class holds board data, player actions, and captured stones.
- **Move Processor** (**Controller Layer**)**:** Reads from GameState, validates moves, and updates the board.
- **Capture Analyzer** (**Model Layer**)**:** Detects surrounded stones and removes them.
- **Win Condition Evaluator** (**Model Layer**)**:** Checks if a player has been completely ousted.

Why This Approach?

- **MVC ensures clarity, modularity, and flexibility** for different interfaces.
- **Blackboard Architecture keeps game state consistent and efficient** for game logic processing.
- **Avoids unnecessary complexity** (e.g., microservices architecture would be overkill for a single-player game).

What are the best tools for software architectural design?

1. Architectural Design & Diagramming

- **Draw.io**: Create clear **UML diagrams** (Sequence, Architectural Pattern/Style) to visualise game structure and interactions.

Why? These tools help visualise components, like the Board, Player, Move, and Group classes, before coding, ensuring a well-structured design.

2. Development & Implementation

- **IntelliJ IDEA**: Powerful **Java IDE** with built-in refactoring, debugging, and version control support.
- **GitHub:** For **version control and collaboration**, essential for Scrum teamwork.

- **JUnit**: To **test move validation, capturing logic, and game rules** early in development.

Why? These tools speed up development, enforce good coding practices, and support agile iterations in Scrum.

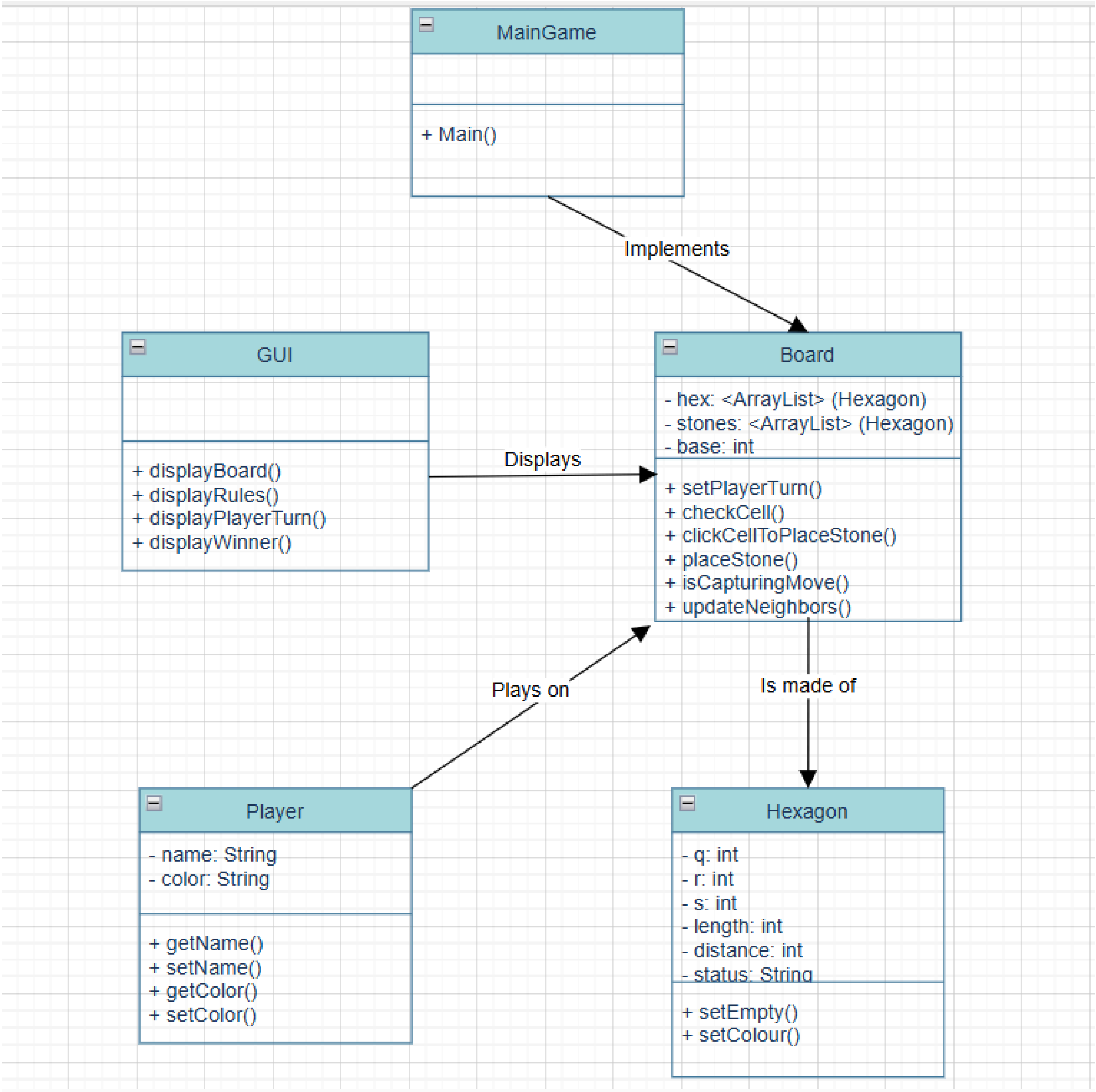
3. Agile & Collaboration

- **Trello**: **Managing sprints and tracking tasks**.
- **Whatsapp**: For **quick communication** between team members.
- **Shared Drive**: For organisation of **resources and documents.**
- **Google Docs:** For **documentation** through the whole project.

Why? Clear sprint planning and documentation

CLASS DIAGRAMS

The **class diagram** for HexOust provides the structure of the classes that are going to be implemented in this project. This is the first version of the diagram and, as a consequence, does not have all the attributes and methods that are going to be in a later version of the code. A class diagram is a type of **static structure diagram** that describes the **structure of a system**.



In this specific diagram we have five classes:

- **MainGame** is the class that will have the main function calling all the other classes and functionalities
- **GUI** is the graphical interface of the application, it will show the board made of hexagons
- **Board** is the class that implements all the moves made during a match
- **Hexagon** implements the hexagons that are needed to create a board
- **Player** is the class that store the name and the color of a player in a match

ACTIVITY/SEQUENCE DIAGRAMS

The **sequence diagrams** for HexOust provide a structured representation of how **game specifications** (**SR1–SR5**) are implemented using a **Model-View-Controller** (**MVC**) **approach**, with elements of **Blackboard Architecture** ensuring game state consistency. Below is a breakdown of how these diagrams implement the required functionalities.

SR1: Game Initialisation & Player Turn Display

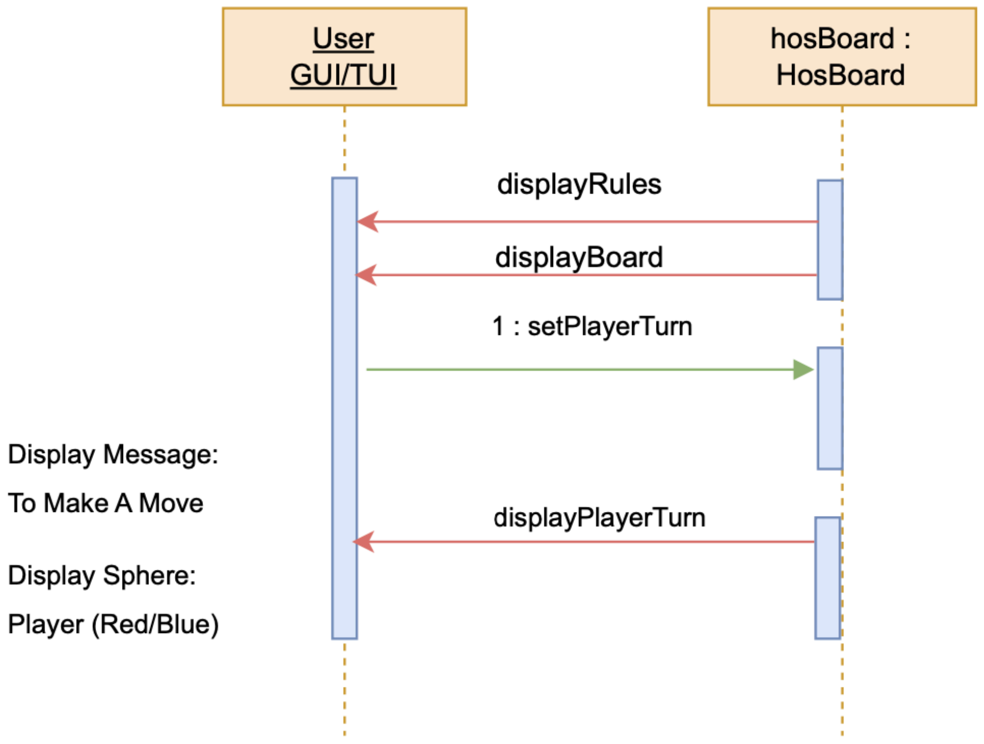
How It Works:

- When the game starts, the HosBoard initialises an **empty board**.
- The **RED player starts first**, indicated by a **RED sphere and a message** displaying "RED To Make A Move"
- The board is rendered, showing the available hexagons.

Diagram Implementation:

1. SetPlayerTurn is called in HosBoard to set the starting player.
2. DisplayPlayerTurn updates the GUI with a **RED sphere** and a message.
3. The board is displayed to the player (displayBoard).

For specification 1 (SR1, SR1.1, SR1.2, SR1.3)



SR2: Placing a Stone (Non-Capturing Move - NCP)

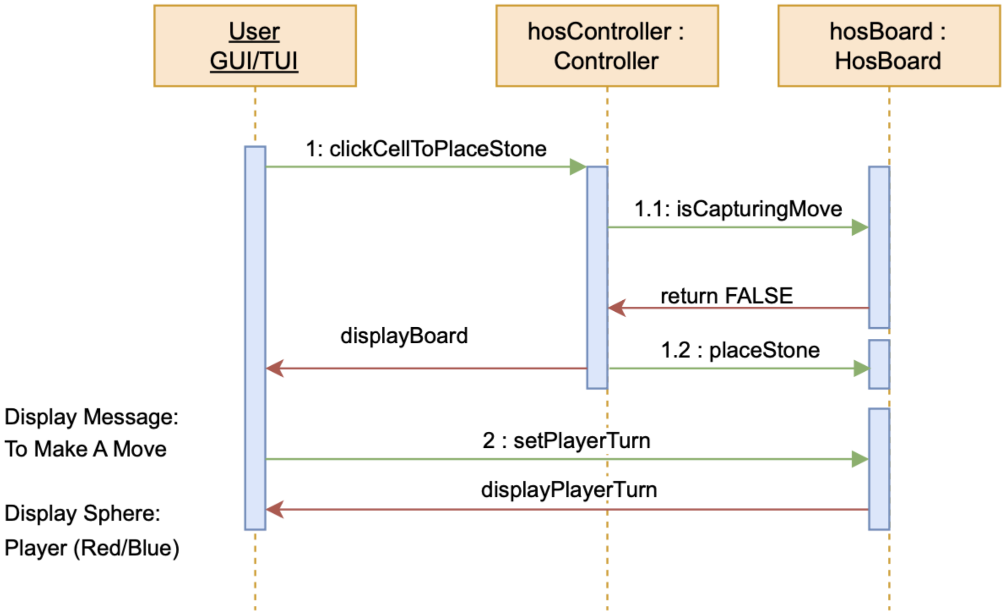
How It Works:

- A player clicks on an empty cell to place a stone.
- If the move is **valid and non-capturing**, the stone is placed, and the turn is passed to the opponent.
- A message and coloured sphere update to indicate the new player's turn.

Diagram Implementation:

1. clickCellToPlaceStone is triggered in HosController.
2. The isCapturingMove method checks if the move is capturing or non-capturing (returns **FALSE** for NCP).
3. placeStone updates the board with the player's stone.
4. setPlayerTurn updates turn ownership.
5. displayPlayerTurn changes the GUI to indicate the new player.

For specification 2 (SR2, SR2.1, SR2.2)



SR3: Capturing Move (CP) & Group Removal

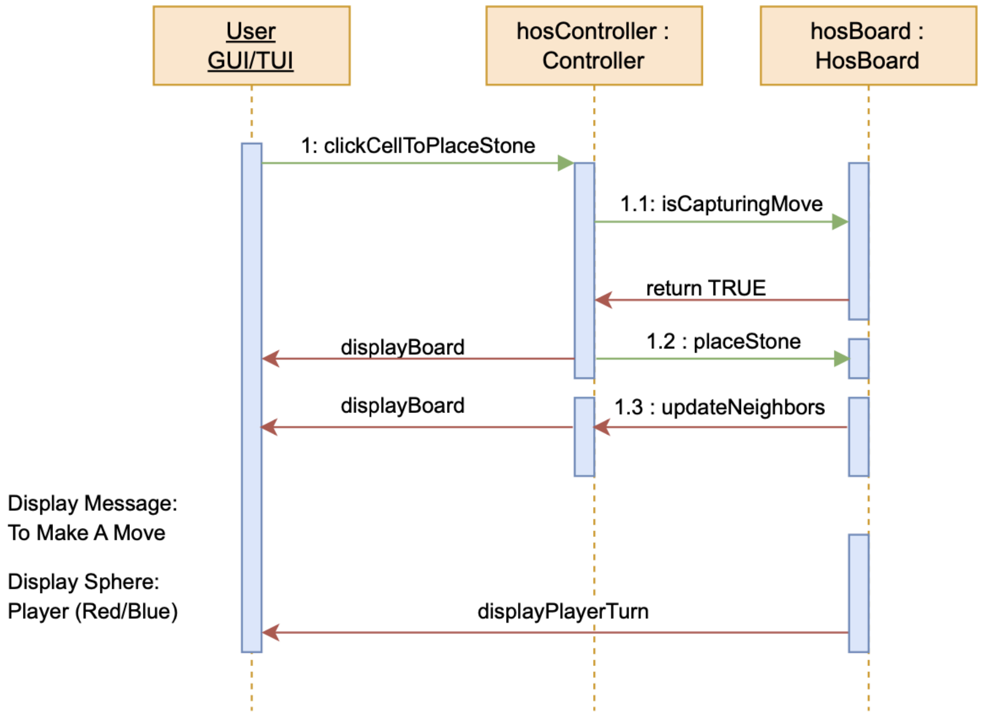
How It Works:

- A capturing move(CP) removes **all opponent's captured groups** from board.
- The turn remains with the **capturing player** after a successful capture.

Diagram Implementation:

1. clickCellToPlaceStone triggers the move.
2. isCapturingMove returns **TRUE**, indicating a capture.
3. placeStone updates the board.
4. updateNeighbors ensures surrounding groups are correctly updated.
5. displayPlayerTurn keeps the capturing player's turn.

For specification 3 (SR3, SR3.1, SR3.2)



SR4: Invalid Moves & Error Handling

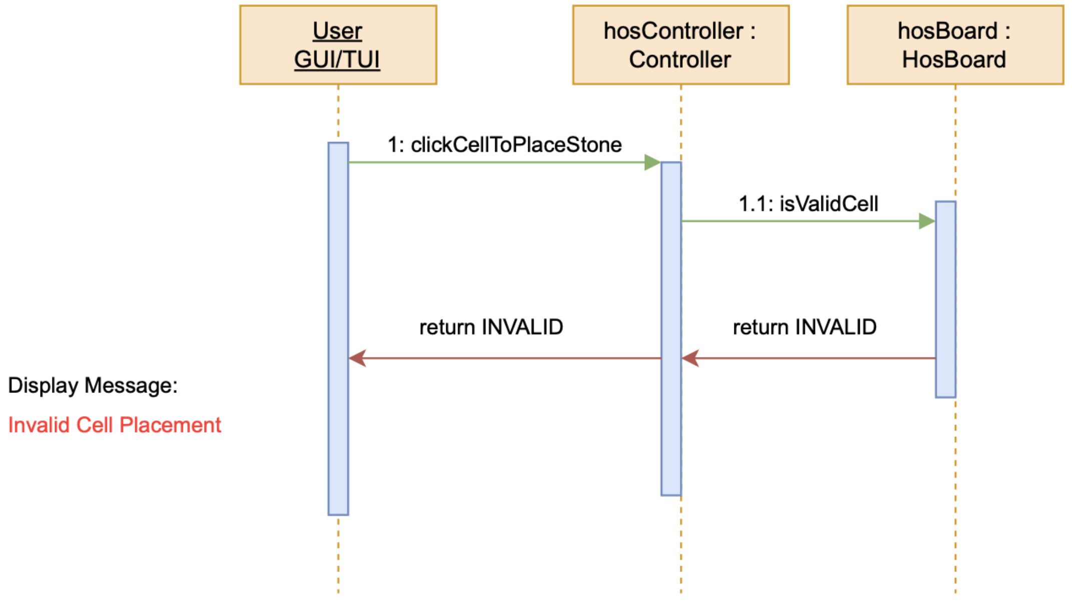
How It Works:

- If a player tries to place a stone in an **invalid cell**, an error message appears: "**Invalid Cell Placement.**"
- Optional enhancements are **hover indicators** showing a green tick (valid) or red cross (invalid).

Diagram Implementation:

1. clickCellToPlaceStone checks move validity.
2. isValidCell verifies the chosen cell.
3. If invalid, it returns **INVALID** and displays an **error message**.

For specification 4 (SR4, SR4.1, SR4E1, SR4E2)



SR5: Winning Condition Detection

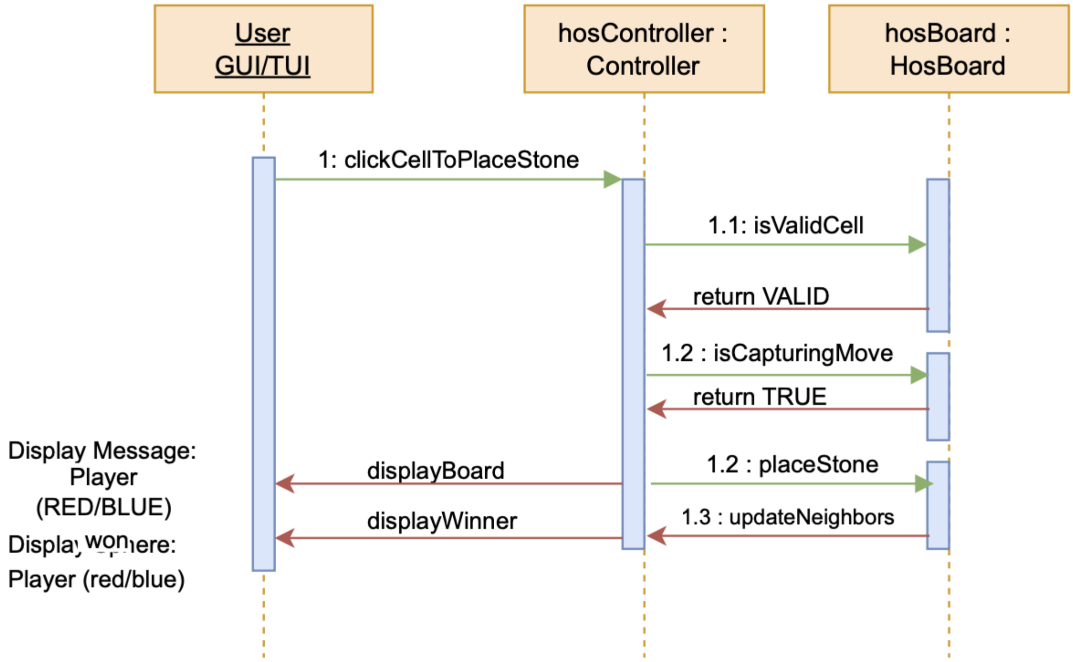
How It Works:

- If a player **removes all the opponent's stones**, a **winning message** appears.
- The winner is displayed, and no further moves are allowed.

Diagram Implementation:

1. clickCellToPlaceStone triggers the move.
2. isValidCell confirms placement validity.
3. isCapturingMove checks if the move results in the opponent's elimination.
4. updateNeighbors processes board changes.
5. displayWinner announces "**RED Wins**" or "**BLUE Wins.**"

For specification 5 (SR5, SR5.1)



How Blackboard Architecture Supports the Implementation

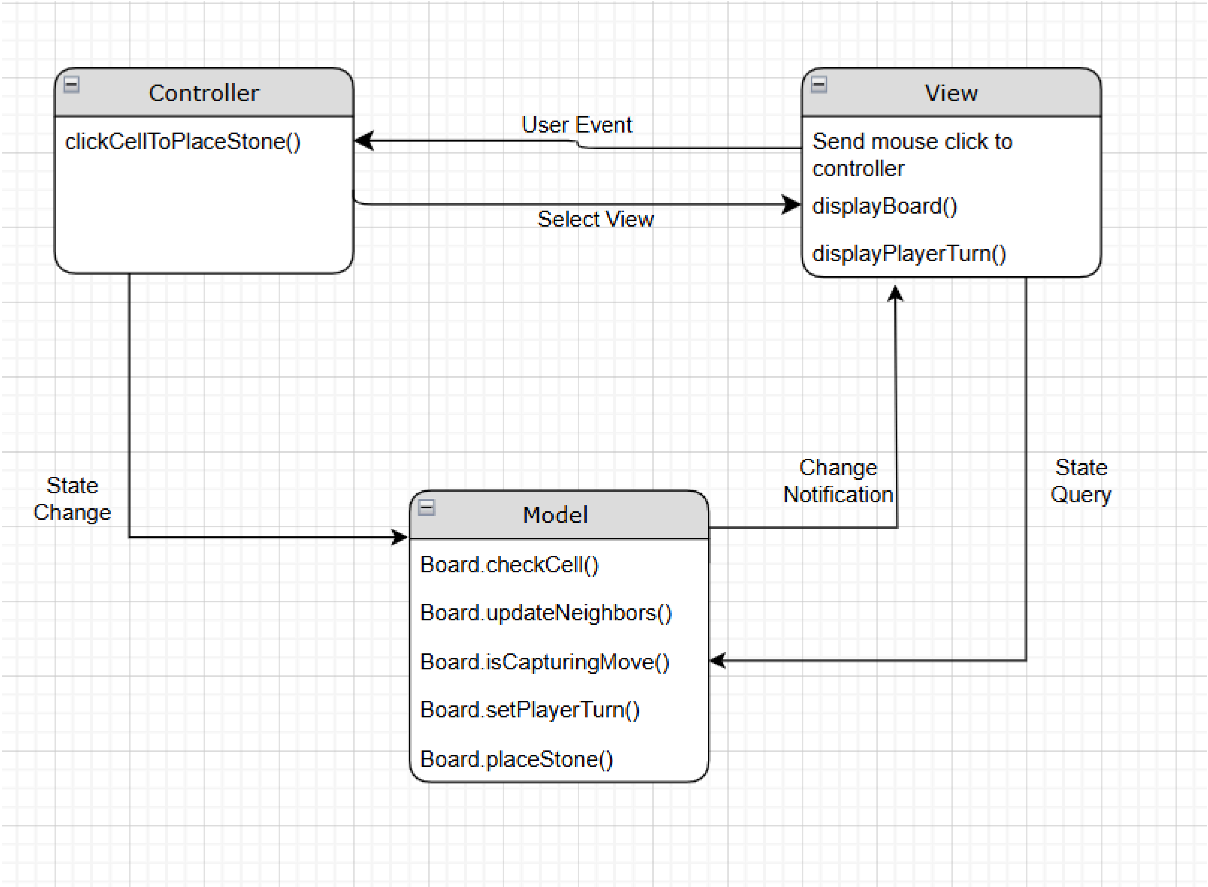
While MVC structures the **game flow**, the **Blackboard Architecture** ensures all components work with an up-to-date **central game state** (HosBoard).

How it integrates:

- The **blackboard** (**game state**) **stores the board, moves, and captured groups** in HosBoard.
- Ensures **real-time updates**, preventing inconsistencies between different parts of the game logic.

ARCHITECTURAL PATTERN/STYLE

The **architectural pattern** chosen for this project is the **Model-View-Controller** (**MVC**) approach. It is commonly used for **developing interfaces**. This approach splits the program into **three elements**, those elements being the model, view, and controller as stated in the name.



How the MVC architecture elements interact:

- The **model** is the **internal representation** of the information.
  - The model **receives state changes** from the **controller** in order to **update the internal data** of the program.
  - The model **sends out notifications** to the **view** for any **data that has changed** and needs to be updated and represented in the visual interface.

    - The model also **receives queries** from the **view** to **confirm the internal data** stored in the program.
- The **view** is the **interface** that presents and accepts information from the user.
    - The view **interacts with the controller through user events**, such as a mouse click.
    - The view also **displays the interface** that the user interacts with the program through.
- The **controller** is the **software that links** the two elements.
    - The controller **sends updates** to the **model** when it **receives a user event from the view**.
    - The controller can also **select the view** that the user is interacting with.

REFERENCES

- https://www.geeksforgeeks.org/software-engineering-architectural-design/
- https://www.geeksforgeeks.org/design-patterns-architecture/?ref=ml_lbp
- https://en.wikipedia.org/wiki/List_of_software_architecture_styles_and_patterns