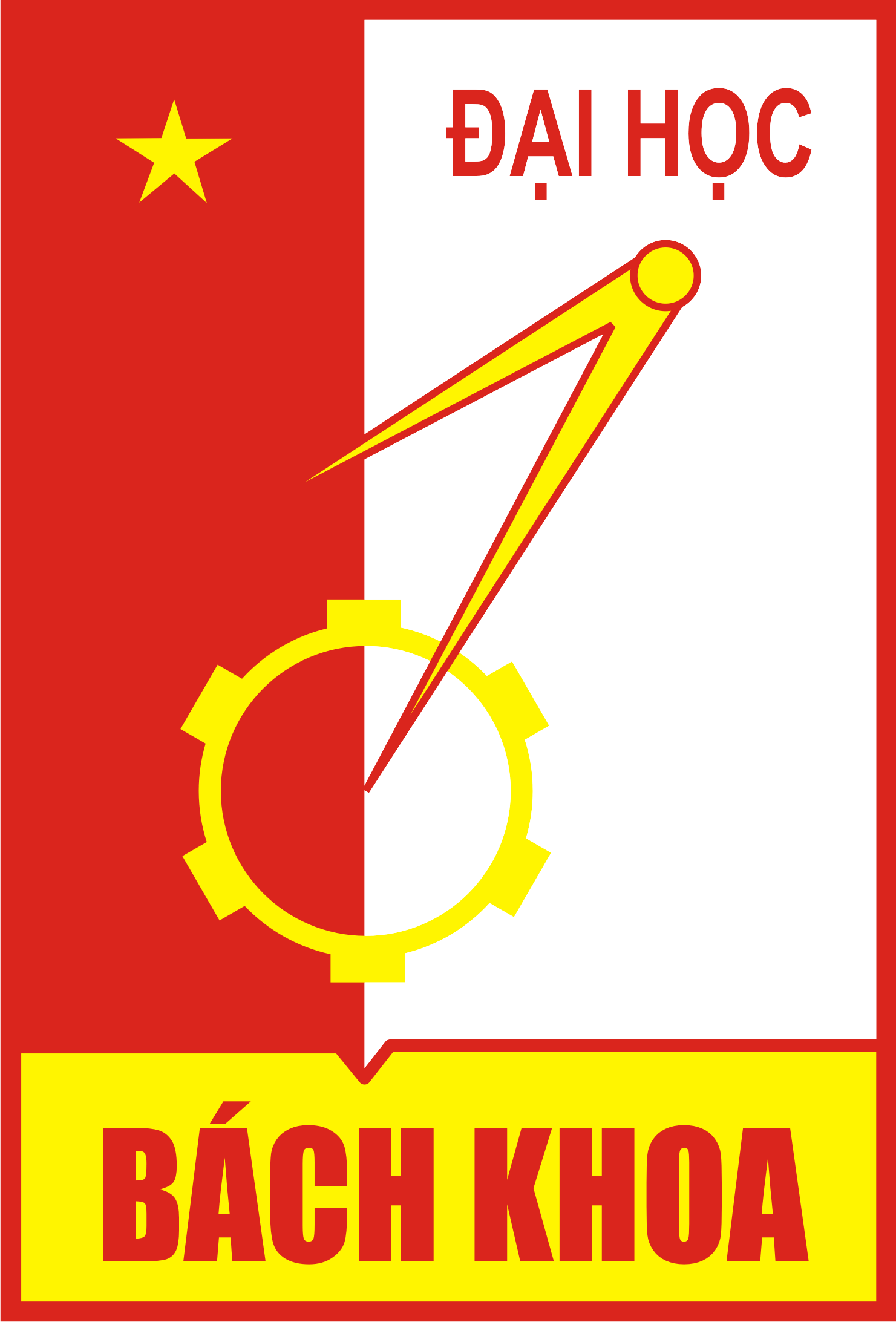
Đại học Bách Khoa Hà Nội – Trường CNTT&TT



**BÁO CÁO MINI-PROJECT LẬP TRÌNH HƯỚNG ĐỐI TƯỢNG**

**HEDSPI – K66**

|  |  |
| --- | --- |
| Mã học phần: | **IT3103** |
| Mã lớp học: | **143577** |
| Giảng viên: | **Nguyễn Thị Thu Trang** |
| Nhóm số: | **38** |

**Mục lục**

[I. Thành viên và phân công nhiệm vụ 2](#_Toc155373205)

[II. Mô tả đề tài 2](#_Toc155373206)

[1. Yêu cầu ứng dụng 2](#_Toc155373207)

[2. Biểu đồ Use case 3](#_Toc155373208)

[III. Thiết kế ứng dụng 4](#_Toc155373209)

[1. Sơ đồ lớp chung 4](#_Toc155373210)

[2. Sơ đồ lớp chi tiết cho từng package 5](#_Toc155373211)

[ Model 5](#_Toc155373212)

[ View 6](#_Toc155373213)

[ Controller 7](#_Toc155373214)

[3. Giải thích thiết kế 7](#_Toc155373215)

[ Liên kết giữa các package 7](#_Toc155373216)

[ Liên kết giữa các lớp 8](#_Toc155373217)

[ Một số phương thức quan trọng của các lớp 10](#_Toc155373218)

# Thành viên và phân công nhiệm vụ

* + - * Các thành viên trong nhóm đảm nhận một số công việc khác nhau của project.
      * Tiến độ công việc được cập nhật lên Github để những thành viên khác cùng xét duyệt.
      * Quá trình thiết kế, phát triển đề tài được cả nhóm bàn bạc và thực hiện qua group chat và các buổi họp trên Teams.
      * Bảng phân công

|  |  |  |
| --- | --- | --- |
| Thành viên | MSSV | Nhiệm vụ đảm nhận |
| Nhữ Xuân Vinh | 20215170 | - Code phần giao diện người dùng của ứng dụng  - Kiểm thử, sửa lỗi |
| Hoàng Nguyên Vũ | 20215171 | - Thiết kế kiến trúc ứng dụng  - Code phần back-end (Model) của ứng dụng  - Code phần giao diện bàn cờ |
| Hoàng Sĩ Vương | 20215172 | - Làm báo cáo  - Làm silde |
| Lê Đàm Quân | 2023T051 | - Code phần giao diện người dùng của ứng dụng  - Kiểm thử, sửa lỗi |

# Mô tả đề tài

## Yêu cầu ứng dụng

Trong project này bạn sẽ thiết kế một ứng dụng cho phép 2 người chơi trò chơi dân gian Cờ gánh.

* + - * GUI: bạn được tự do thiết kế giao diện người dùng tuy nhiên project này mục đích là thiết kế một chương trình sử dụng OOP nên không cần quá tập trung vào phần giao diện.
      * Thiết kế: Ứng dụng cần có các tính năng sau:
* Trên màn hình chính:
* Bắt đầu: Bắt đầu trò chơi. Để thuận tiện thì bạn không cần phải tạo các độ khó khác nhau của trò chơi.
* Thoát: Thoát ứng dụng. Hãy xác nhận người dùng nếu họ thực sự muốn thoát trò chơi.
* Trong trò chơi:
* Bàn cờ: Bàn cờ là một mặt phẳng được chia thành 16 ô vuông với các đường kẻ dọc, ngang và chéo biểu thị cho các con đường mà các quân cờ được phép đi. Bàn cờ có 25 giao điểm từ các đường kẻ trên bàn cờ, đó chính là 25 vị trí để đặt các quân cờ.
* Trong mỗi lượt, ứng dụng phải chỉ rõ lượt hiện tại là của người chơi nào. Người chơi đến lượt sẽ chọn một trong các quân cờ của mình và một điểm hợp lệ để đặt quân cờ đó. Quân cờ sẽ bị đổi màu sang màu của đối thủ nếu bị ăn.
* Trò chơi kết thúc khi chỉ còn một màu quân trên bàn cờ.
* Để đơn giản thì bạn không cần thiết kế bot để chơi với người.
  + - * Luật chơi Cờ Gánh: [Cờ Gánh](https://thuthuatchoi.com/huong-dan-cach-choi-co-ganh.html)

## Biểu đồ Use case

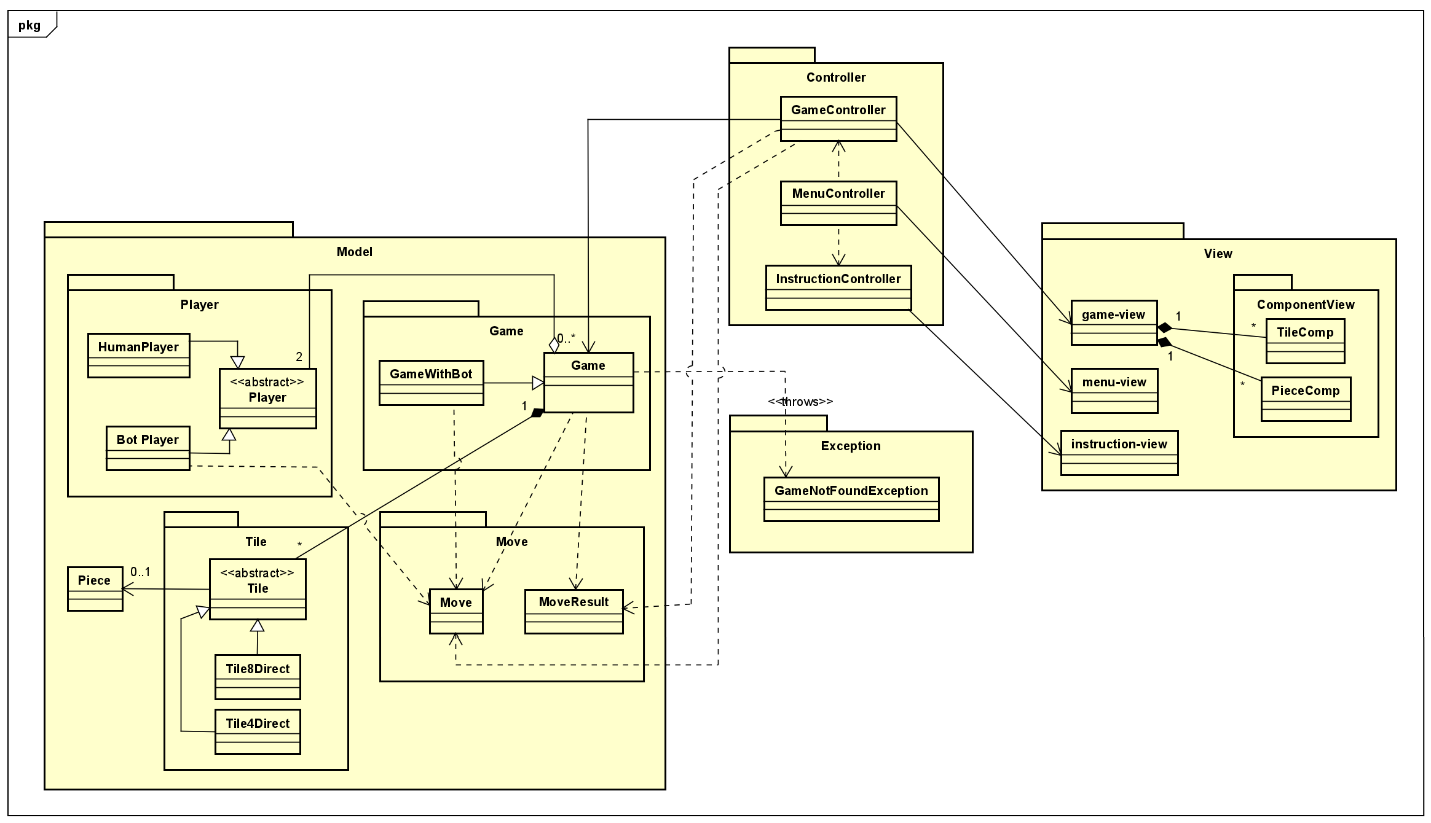
A diagram of a game

Description automatically generated

* + - * Bắt đầu trò chơi mới: người dùng có thể chọn chơi với người (2 người chơi) hoặc chơi với bot. Người dùng được yêu cầu nhập tên và chọn giới hạn thời gian. Nếu chơi với bot thì được yêu cầu chọn độ khó của bot.
      * Tiếp tục ván cờ cũ: tiếp tục ván cờ đã được lưu. Nếu không có ván cờ nào được lưu thì một thông báo lỗi được hiển thị.
      * Thực hiện nước đi: người dùng thực hiện nước đi với một quân cờ. Đó có thể là một nước ăn quân của đối phương. Sau khi nước đi được thực hiện thì bàn cờ được cập nhật và lượt đi được đổi.
      * Xem hướng dẫn: người dùng chọn xem hướng dẫn, một màn hình hướng dẫn được hiển thị.

# Thiết kế ứng dụng

## Sơ đồ lớp chung



## Sơ đồ lớp chi tiết cho từng package

### Model

A screenshot of a computer screen

Description automatically generated

### View

A screenshot of a computer

Description automatically generated

### Controller

A screenshot of a computer

Description automatically generated

## Giải thích thiết kế

### Liên kết giữa các package

* Ứng dụng được phát triển sử dụng kiến trúc MVC (Model – View – Controller) bao gồm 3 package:
  + Model: chứa các lớp lưu trữ dữ liệu và logic. Nó bao gồm các lớp và phương thức thực hiện các tác vụ như tính toán logic, thêm hoặc sử dữ liệu,…
  + View: chịu trách nhiệm hiển thị thông tin cho người dùng. Các dữ liệu được cung cấp bởi Model sẽ được thể hiện lên người dùng bởi thành phần View.
  + Controller: đóng vai trò trung tâm, nắm bắt và điều phối giữa Model và View. Khi người dùng tương tác với giao diện, các yêu cầu sẽ được gửi đến Controller. Controller sau đó xử lý sơ bộ dữ liệu này và chuyển tiếp cho Model. Model, chứa logic nghiệp vụ, sẽ tiếp tục xử lý và trả dữ liệu đã qua xử lý trở lại cho Controller. Cuối cùng, Controller sẽ sử dụng View để hiển thị dữ liệu đã qua xử lý cho người dùng, hoàn thành quá trình tương tác.
  + Ví dụ trong ứng dụng: khi người dùng thực hiện di chuyển một quân cờ trên giao diện (View), yêu cầu đó sẽ được Controller nắm bắt (thông qua các event handler). Controller nhận di chuyển đó, chuyển nó thành dữ liệu nước đi ứng với thao tác của người dùng và gửi cho Model để xử lý nước đi với các logic. Sau khi Model xử lí xong nước đi, nó sẽ gửi lại một kết quả nước đi đến cho Controller, Controlelr nhận kết quả đó và hiển thị lên giao diện.

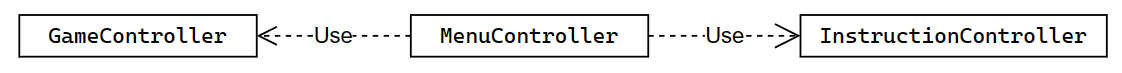
A diagram of a computer system

Description automatically generated

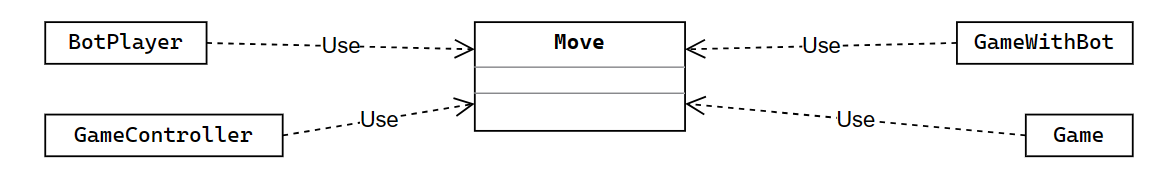
→ Sử dụng kiến trúc MVC mang lại lợi ích lớn trong việc phân tách trách nhiệm giữa các thành phần. Trong MVC, Model, View và Controller đều có trách nhiệm riêng biệt từ đó giúp mã nguồn được tổ chức tốt, dễ dàng bảo trì và mở rộng.

### Liên kết giữa các lớp

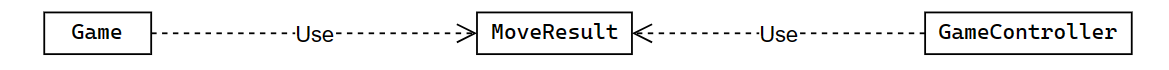
* **Dependency:**



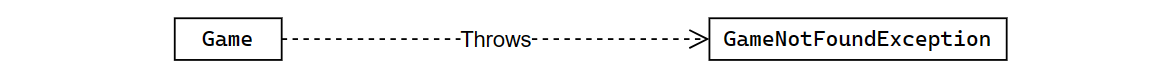
**menu-view** là màn hình khởi đầu của ứng dụng, ở đó có các lựa chọn để chuyển hướng đến **game-view** hoặc **instruction-view**. Khi người dùng lựa chọn chuyển hướng thì **MenuController** sẽ khởi tạo một đối tượng **GameController** hoặc **InstructionController** tương ứng.



**Move** là một lớp record để lưu thông tin của một nước đi được thực hiện bởi cả người chơi và bot. Các lớp sử dụng các đối tượng của lớp **Move** để trao đổi và xử lý dữ liệu với nhau.

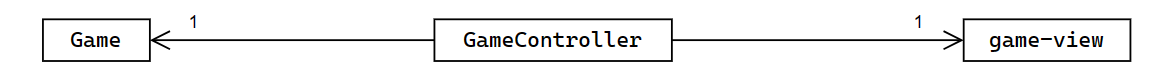


**MoveResult** là một lớp record để lưu thông tin về kết quả của một nước đi sau khi được xử lí bởi các logic trong lớp **Game**, kết quả này được gửi đến **GameController** để thể hiện lên **game-view**.

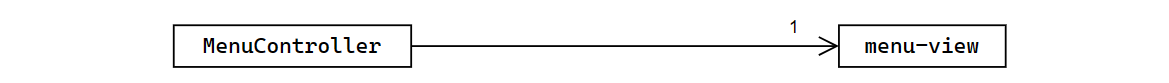


Phương thức **loadGame()** (được gọi khi người dùng chọn Continue) của lớp **Game** sẽ throw một đối tượng **GameNotFoundException** nếu ứng dụng không tìm thấy file lưu game (**game\_state.txt**) hoặc file lưu game trống.

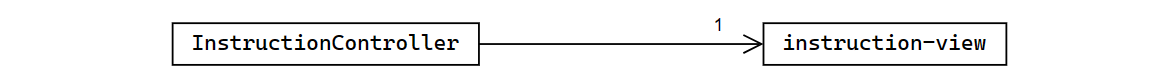
* **Association:**



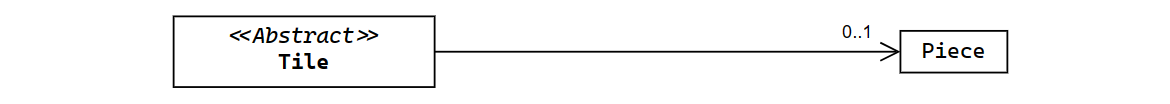
**GameController** là cầu nối giữa hai lớp **Game** và **game-view.**



**MenuController** điều khiển **menu-view**.

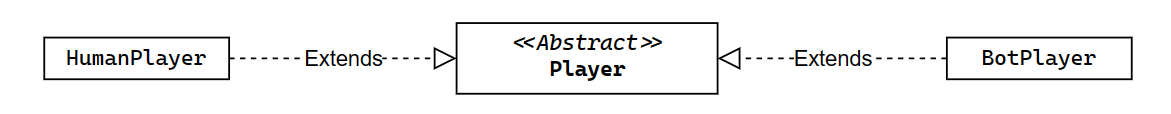


**InstructionController** điều khiển **instruction-view**.

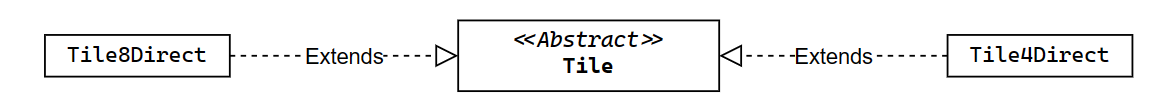


Mỗi một **Tile** có thể có 1 **Piece** nằm trong nó hoặc không có **Piece** nào.

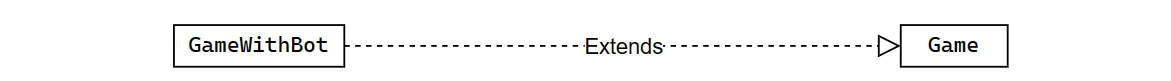
* **Generalization:**



**Player** là một lớp abstract, **HumanPlayer** và **BotPlayer** là hai lớp kế thừa lớp **Player**. Hai lớp này có thêm các phương thức và thuộc tính khác nhau phục vụ cho hoạt động của người chơi và bot.

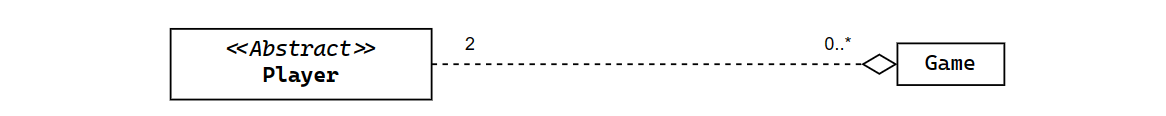


**Tile8Direct** và **Tile4Direct** là 2 loại ô xuất hiện xen kẽ trên bàn cờ. **Tile8Direct** là những ô có kết nối với 8 ô xung quanh, **Tile4Direct** là những ô có kết nối với 4 ô xung quanh (trong bàn cờ).



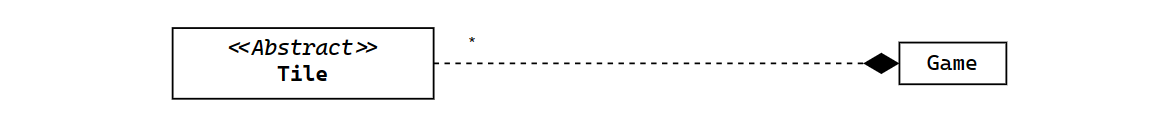
**GameWithBot** là lớp kế thừa **Game** và được khởi tạo khi người dùng chọn chơi với bot. Lớp này cung cấp một số phương thức nhiều hơn **Game** để phục vụ cho hoạt động của bot (sản sinh nước đi, hoàn tác nước đi,…).

* **Aggregation:**

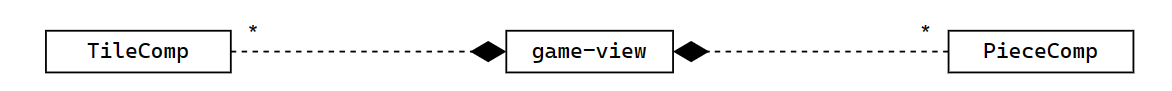


Một **Game** sẽ có 2 **Player** và mỗi **Player** có thể thuộc từ không đến nhiều **Game**.

* **Composition:**



Mỗi **Game** sẽ sở hữu nhiều **Tile** (cụ thể là 25 **Tile**) phục vụ cho việc xử lí các logic của trò chơi.



**TileComp** và **PieceComp** là thành phần view thể hiện cho một ô và một quân trên bàn cờ. Các đối tượng của hai lớp này sẽ được thêm vào **game-view** và được điều khiển bởi **ControllerView**, tạo nên bàn cờ hiển thị cho người dùng.

### Một số phương thức quan trọng của các lớp

* **Move**:

public record Move(Tile fromTile, Tile toTile) {  
}

Một nước đi bao gồm **fromTile** – là ô khởi đầu của nước đi, hiện đang chứa Piece của nước đi, **toTile** – ô đích đến của nước đi, hiện đang không chứa Piece.

* **MoveResult**:

public record MoveResult(boolean isValidMove, ArrayList<Piece> capturedPieces){

}

**boolean isValidMove** – thể hiện nước đi có hợp lệ không, nếu nước đi không hợp lệ thì nó sẽ bị huỷ.

**ArrayList<Piece> capturedPieces** – một danh sách các quân cờ của đối thủ bị ăn khi thực hiện bước đi này.

* **Game:**
  + **Thuộc tính:**

protected Tile[][] board;  
private final Player player1;  
private final Player player2;  
private Player currentPlayer;  
private final int timeLimit;

* + **private void initBoard()**: khởi tạo bàn cờ model. Phương thức này thêm các đối tượng **Tile** vào thuộc tính **board** của **Game**, khởi tạo **Piece** cho các **Tile** đó theo như sắp xếp ban đầu của bàn cờ.
  + **public MoveResult processMove(Move move)**: nhận vào một **Move**, xử lý **Move** đó và trả về kết quả bằng một đối tượng **MoveResult**.

Trong phương thức **processMove()**, sau khi xác nhận nước đi là một nước đi hợp lệ, sẽ lần lượt gọi các phương thức **getCarriedPieces()** và **getSurroundedPieces()** để lấy các quân cờ bị gánh và bị vây bởi nước đi đó.

* + **private ArrayList<Piece> getCarriedPieces** **(Tile toTile)**: phương thức này nhận vào ô đích đến, sau đó nó duyệt qua các cặp ô đối xứng trong các ô liên kết với ô đích đến và tìm các quân bị gánh. Các quân bị gánh được đảo bên trả về trong một danh sách các **Piece**.
  + **public ArrayList<Piece> getSurroundedPieces()**: phương thức này duyệt qua toàn bộ board, với mỗi ô trên bàn cờ có quân khác màu với người chơi hiện tại thì nó gọi một thuật toán flood fill để tìm tất cả các **Piece** hình thành một nhóm (các **Piece** ở cùng bên và liên kết với nhau), kiểm tra nhóm đó có bị vây hay không và trả về một **boolean**. Nếu hàm này tìm thấy một nhóm bị vây thì đảo bên tất cả các quân trong nhóm đó và trả về danh sách các quân đó.
  + **public void switchPlayer()**: thực hiện chuyển đổi vai trò của hai người chơi sau mỗi nước đi
  + **public boolean isGameOver()**: trả về kết quả thể hiện game đã kết thúc chưa bằng việc kiểm tra số quân mỗi người chơi đang sở hữu, nếu có một người chơi sở hữu tất cả các quân thì người đó chiến thắng và game kết thúc.
  + **public void saveGame()**: sử dụng tính năng Serialization của Java để lưu đối tượng **Game** hiện tại vào file **game\_state.txt** để có thể tải lại và chơi tiếp. Serialization trong Java là cơ chế chuyển đổi trạng thái của một đối tượng (giá trị các thuộc tính trong object) thành một chuỗi byte sao cho chuỗi byte này có thể chuyển đổi ngược lại thành một đối tượng.
  + **public static Game loadGame() throws GameNotFoundException**: đây là một phương thức static trả về một đối tượng **Game**. Phương thức này thực hiện Deserialization để chuyển đổi ngược lại dữ liệu trong file **game\_state.txt** và trả về đối tượng Game đó. Phương thức throw một đối tượng **GameNotFoundException** khi file **game\_state.txt** là rỗng hoặc không tồn tại.
* **GameController**
  + **Thuộc tính:**

private final Game game; *// model reference*  
private Tile currentTile;  
private final Group tileCompGroup = new Group();  
private final Group pieceCompGroup = new Group();  
private final TileComp[][] viewBoard = new TileComp[*WIDTH*][*HEIGHT*];  
private final Map<Piece, PieceComp> pieceMap = new HashMap<>(); *// this map maps Piece with the corresponding PieceComp*   
private final ChangeListener<Number> timeLeftListener = (observable, oldValue, newValue) -> {  
 if (newValue.intValue() <= 0) {  
 *// Timeout* switchPlayer();  
 }  
};  
private final ExecutorService executor = Executors.*newSingleThreadExecutor*(); *// this executor is used to run the botMoveTask*

* + **public void initialize()**: phương thức khởi tạo của **GameController**, nó gọi đến phương thức **initBoard()** để khởi tạo bàn cờ view.
  + **public void initViewBoard()**: Phương thức này lấy **modelBoard** từ đối tượng **game** sau đó thực hiện khởi tạo các **TileComp** và **PieceComp** tương ứng với các **Tile** và **Piece** thuộc **modelBoard** cho **viewBoard**. Sau đó phương thức này vẽ các đường kẻ trên bàn cờ và đánh số các ô trên bàn cờ.
  + **private PieceComp makePieceComp(boolean side, int row, int col)**: phương thức này khởi tạo một **PieceComp** mới và trả về **PieceComp** đó. **PieceComp** mới được thêm các action handler cho thao tác nhấp chuột (khi người chơi bắt đầu nước đi), thao tác kéo chuột (khi người chơi chọn ô muốn đi) và thao tác thả chuột (khi người chơi kết thúc nước đi → gọi **processMove()** của **Game** để thực xử lí nước đi) sau đó thực hiện các logic tương ứng.
  + **private void switchPlayer()**: sau mỗi nước đi, kiểm tra xem game đã kết thúc chưa nếu có thì gọi đến **endGame()** nếu không thì thực hiện chuyển đổi vai trò của người chơi, thể hiện lượt chơi lên view, thực hiện đếm tổng thời gian và kiểm tra thời gian còn lại của mỗi người chơi (khi hết thời gian thì tự động đổi lượt chơi). Nếu lượt chơi hiện tại là của bot thì gọi đến **botMakeMove()** để thực hiện nước đi của bot.
  + **private void botMakeMove()**: tạo một **Task** mới - **botMoveTask** để thực hiện việc tính toán và ra nước đi của bot, thực thi **Task** này bằng **executor** giúp nó được thực hiện trên một Thread khác tách biệt với JavaFX Application Thread, mục đích là tránh giao diện người dùng bị chặn đứng trong quá trình bot tính toán nước đi. Khi **botMoveTask** hoàn thành thì thực hiện nước đi đó và cập nhật bàn cờ theo đó.
  + **private void endGame()**: vô hiệu hoá toàn bộ quân cờ, thể hiện người chiến thắng một bảng tổng sắp các thông số của hai người chơi lên view.
  + **public void onBtnExitClick(ActionEvent actionEvent)**: xử lý sự kiện khi người chơi bấm nút Exit, nếu game chưa kết thúc thì hỏi người chơi có muốn lưu game trước khi thoát, nếu game đã kết thúc thì xác nhận thoát.
* **Player**
  + **Thuộc tính:**

private final String name;  
private final boolean side; *// true: red, false: blue*private int totalPiece;  
protected int totalTime = 0; *// in milliseconds*

* **HumanPlayer**
  + **Thuộc tính:**

private final int timeLimit; *// in seconds*  
private transient IntegerProperty timeLeft;   
private transient Timeline timeline;

* + **private void initTimeline()**: khởi tạo **timeLeft** (là một thuộc tính **IntegerProperty**) và **timeline** (là một thuộc tính **Timeline**). **timeLeft** được khởi tạo giá trị bằng timeLimit. **timeline** được khởi tạo để tăng **totalTime** lên và giảm **timeLeft** xuống mỗi 1 milli giây, phục vụ cho việc đếm tổng thời gian và thời gian còn lại của người chơi. Đối tượng **timeLeftListener** thuộc **GameController** sẽ được gán cho **timeLeft** của người chơi mỗi khi đến lượt người chơi đó, đối tượng này sẽ lắng nghe mỗi khi giá trị của **timeLeft** thay đổi, khi **timeLeft** bằng 0 nghĩa là người chơi đã hết thời gian.
* **BotPlayer**
  + **Thuộc tính:**

private final int botLevel; *// the botLevel is the depth of the minimax algorithm*private long startTime;

* + **public Move getBestMove(GameWithBot game)**: tính toán nước đi có lợi nhất cho bot và trả về nước đi đó. Phương thức này gọi đến **game.generateMoves()** để lấy tất cả các nước đi khả thi. Sau đó với mỗi nước đi thu được nó gọi **game.makeMove(move)** để mô phỏng nước đi đó và gọi hàm **minimax()** để tính toán độ lợi nếu nước đi đó được thực hiện. Sau đó phương thức gọi **game.undoMove(move, moveResult)** để hoàn tác nước đi trước đó và cuối cùng trả về nước đi có độ lợi lớn nhất để thực thi.
  + **private int minimax(GameWithBot game, int depth, int alpha, int beta, boolean maximizingPlayer)**: đây là thuật toán minimax kết hợp alpha-beta pruning dùng để mô phỏng và tính toán nước đi có lợi nhất trong một trò chơi hai người chơi. Đó là một hàm đệ quy mô phỏng việc chơi trò chơi ở một độ sâu **depth** nhất định (**depth** của hàm phụ thuộc vào level của bot) và đánh giá trạng thái trò chơi. Nếu trò chơi kết thúc hoặc đạt đến độ sâu tối đa, nó sẽ đánh giá bàn cờ hiện tại. Nếu đó là lượt của bot (maximizing player), nó sẽ cố gắng tối đa hóa điểm số. Nếu đến lượt người chơi (minimizing player), nó sẽ cố gắng tối thiểu hoá điểm số.
  + **private int evaluateBoard(Tile[][] board)**: nhận vào bàn cờ hiện tại và trả về tổng giá trị của bàn cờ. Phương thức này duyệt qua tất cả các ô trên bàn cờ, nếu ô có quân thì trừ hoặc cộng một giá trị tuỳ theo bên của quân đó. Giá trị dương nghĩa là bàn cờ có lợi cho bot còn giá trị âm nghĩa là bàn cờ có lợi cho người chơi.
  + **private final int[][] favourablePosition**: đây là một ma trận đánh dấu các ô trên bàn cờ có độ lợi lớn hơn dựa trên một số chiến thuật của trò chơi. Ví dụ, chiếm được các ô ở trung tâm sẽ có lợi ích lớn, ngược lại các quân ở góc bàn cờ sẽ dễ bị vây hơn.

Hai lớp **HumanPlayer** và **BotPlayer** đều override hai phương thức abstract của **Player** là **playTimer()** và **pauseTimer()**, đây là hai phương thức dùng để tiếp tục và dừng bộ đếm thời gian của đối tượng. Với **HumanPlayer** thì dừng và tiếp tục **timeline**, với **BotPlayer** thì dùng chênh lệch thời gian hệ thống để tính thời gian bot dùng để tính toán nước đi.

* **GameWithBot**
  + **public ArrayList<Move> generateMoves()**: trả về một danh sách tất cả các nước đi có thể đi trong lượt hiện tại. Phương thức này duyệt qua tất cả các ô trên bàn cờ, nếu ô có quân và quân này là quân của người chơi hiện tại thì thêm các nước đi hợp lệ của quân đó vào danh sách.
  + **public MoveResult makeMove(Move move)**: thực hiện một nước đi và đổi lượt người chơi, trả về một MoveResult.
  + **public void undoMove(Move move, MoveResult moveResult)**: nhận vào một Move và một MoveResult, hoàn tác trạng thái của bàn cờ và của hai bên về trước khi thực hiện move.
* **Tile**
  + **Thuộc tính:**

private Piece piece; *// each tile may have a piece or not*private final int row;  
private final int col;

* + **public abstract ArrayList<Tile> getConnectedTiles(Tile[][] board)**: đây là một phương thức abstract nhận vào **board** (ma trận các **Tile**) và trả về danh sách các **Tile** liên kết với **Tile** hiện tại. Phương thức này sẽ được override bởi các lớp **Tile8Direct** và **Tile4Direct**, ở các phương thức override đó sẽ trả về danh sách ô liên kết tương ứng bao gồm 8 hoặc 4 **Tile**.
  + **public ArrayList<Tile> getAvailableMoves(Tile[][] board)**: phương thức này nhận vào **board** và trả về các ô hợp lệ được phép đi vào từ ô hiện tại. Ô hợp lệ phải nằm trong danh sách các ô liên kết và không có quân nào ở trong đó.