Compile Instructions

Most group members used Netbeans 7.3 and JRE 1.7 to code this project, although the code should be IDE-independent. Please follow these instructions to get it to run:

1. Change directory (command cd) to the root of the download location and run **ant jar** to compile the project. This requires the Apache Ant utility
2. Double-click the SEng271.jar file under the ./dist folder.

Or open this folder directly in Netbeans.

Game Instructions

After launching the game:

1.) Choose a strategy from the drop down for each player
2.) Press play to start the game
3.) If all strategies are AI, they will start playing
4.) If player one is human
    1.) Click roll, you must roll before you can move
    2.) Click a pawn of your color. It should start flashing
    3.) If there is a valid move that pawn, the square it can move to will flash
    4.) To move the pawn click the flashing square
    5.) If the board is click anywhere other than the flashing square the current pawn will be deselected and must be clicked again before being moved.
5.) A game over window will appear when all players have reach their goal squares.

***Restart and New game implementations were not finished. Please Quit and Relaunch the game to play another match. Quit is working.**

Game Infrastructure

The classes modelled in milestone one have mostly remained the same. A few classes were added and the inheritance tree of others modified. These changes are detailed in the model change section.

While the classes modeling the game have stayed consistent the classes that model the interaction between players, the UI and AI added significant complexity to the design. These classes are not included in the class diagram. There are many additional classes that take care of animating and drawing game objects on the screen. These graphic classes mirror the model classes and hold a reference to the model they are responsible for drawing.

Our designed evolved to make heavy use of threading. This was required to support time based animations and asynchronous interaction.

The application makes use of two additional threads to accomplish this:

1.) Animator.java

This thread updates the state of animations, calculating their next position based on the amount of time that has elapsed since the last frame. The animation thread does not redraw anything. Instead it marks graphics as dirty and calls repaint(). Repaint does not immediately trigger a draw. Instead Swing adds a request to repaint to Swings EventDispatchThread. Eventually this thread will service the request and redraw.

When a redraw is called by the swing dispatch thread repainting is delegated to the Renderer2D.java class. This class holds a list of layers. Each layer contains a list of graphics that belong to the layer.

The renderer checks each layer for dirty graphics. Dirty graphics are those that have changed since the last frame. If a dirty graphic is found the renderer then checks to see if that graphic intersects with any other graphic. If an intersection is found, intersecting graphics are marked as dirty. Next the renderer gets the bounds of all dirty graphics and clears the areas using the graphics2d clearrect call. Dirty graphics are then redrawn to fill the cleared areas. Lastly the renderer composites all layers into one image and draws it to the screen. If no graphics are dirty in an iteration redraw is not called. Only dirty graphics are redrawn at each iteration.

2.) GameController.java

This thread responds to GameEvents. GameEvents are added to a BlockingQueue which is a member of the GameController class. The GameController then take()s these events off the other end of the queue and dispatches them to any handler that has registered with the GameController to receive events of that type. There are several types of game events including, roll events, turn events, move events etc. These events can arrive from any source. Currently they only arrive from AI or human players. The event based system was designed to allow the addition of networking and multiplayer in the future.

The handlers registered with the game controller typically implement simple logic that delegates calls to the GameLogic class defined in our model. A few handlers such as the BoardClickHandler implements more complex logic to determine which pawn is clicked and whether to make a move.

Other highlights

The game board is encoded as a 2D array. All player paths are generated from one path. This is the path from player ones home square to player ones last goal square. The path is defined as an array of vectors. Each vector represents the number of squares in the horizontal or vertical direction that are part of the path relative to the last square in the path. To get the paths for player 2,3 and 4 these path vectors are rotated by 90, 180 and 270 degrees.

Model Changes

Class Diagram

- GameController now called GameLogic
    - The GameController still exists but it's used to handle events for the game
    - GameLogic implements the logic of the game (what the GameController would have done before)
- Added a Path class that extends java.util.LinkedList<PathSegment>
    - Stores the Player's path around the board as well as their home squares
    - The home squares are not part of the main board path
- The Strategy class was renamed to AbstractStrategy
- GameEntity class created
    - Acts a a base class for other game entities - Pawn, Square, and Die all extend it
    - Holds the position of the entity on the board
    - Supports the property change listener (used to update the graphics)
- GetHomeSquare() method added to Path
- CanPass() method added to Square and Goal to check if the pawn is able to pass over that square
    - Used on the goal squares to prevent pawns from jumping over existing pawns

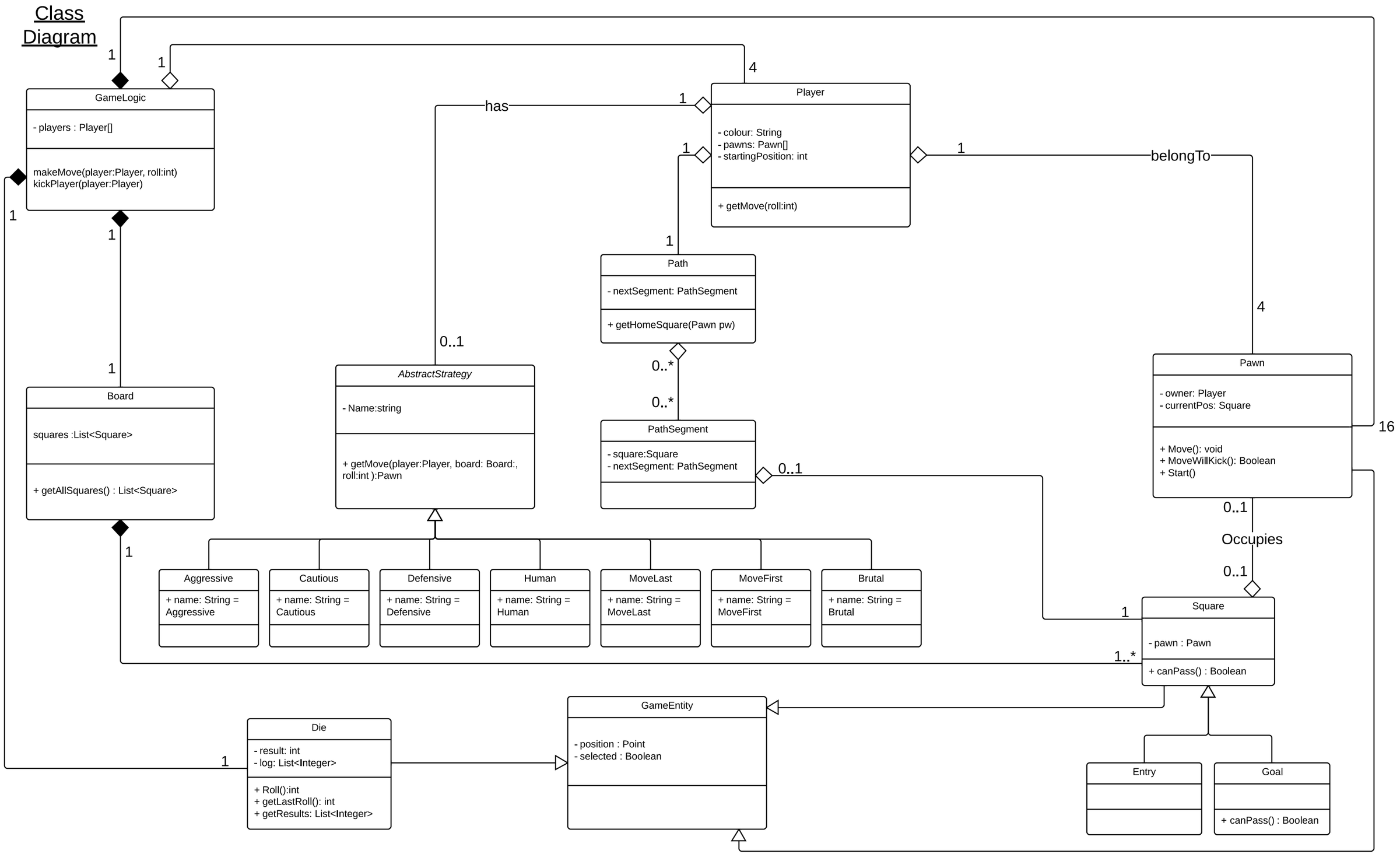
Use Case Diagram

- Computer player actor removed
    - Computer players are controlled from inside the game - they should not be considered an actor from an external source
- New Game and Restart use cases added

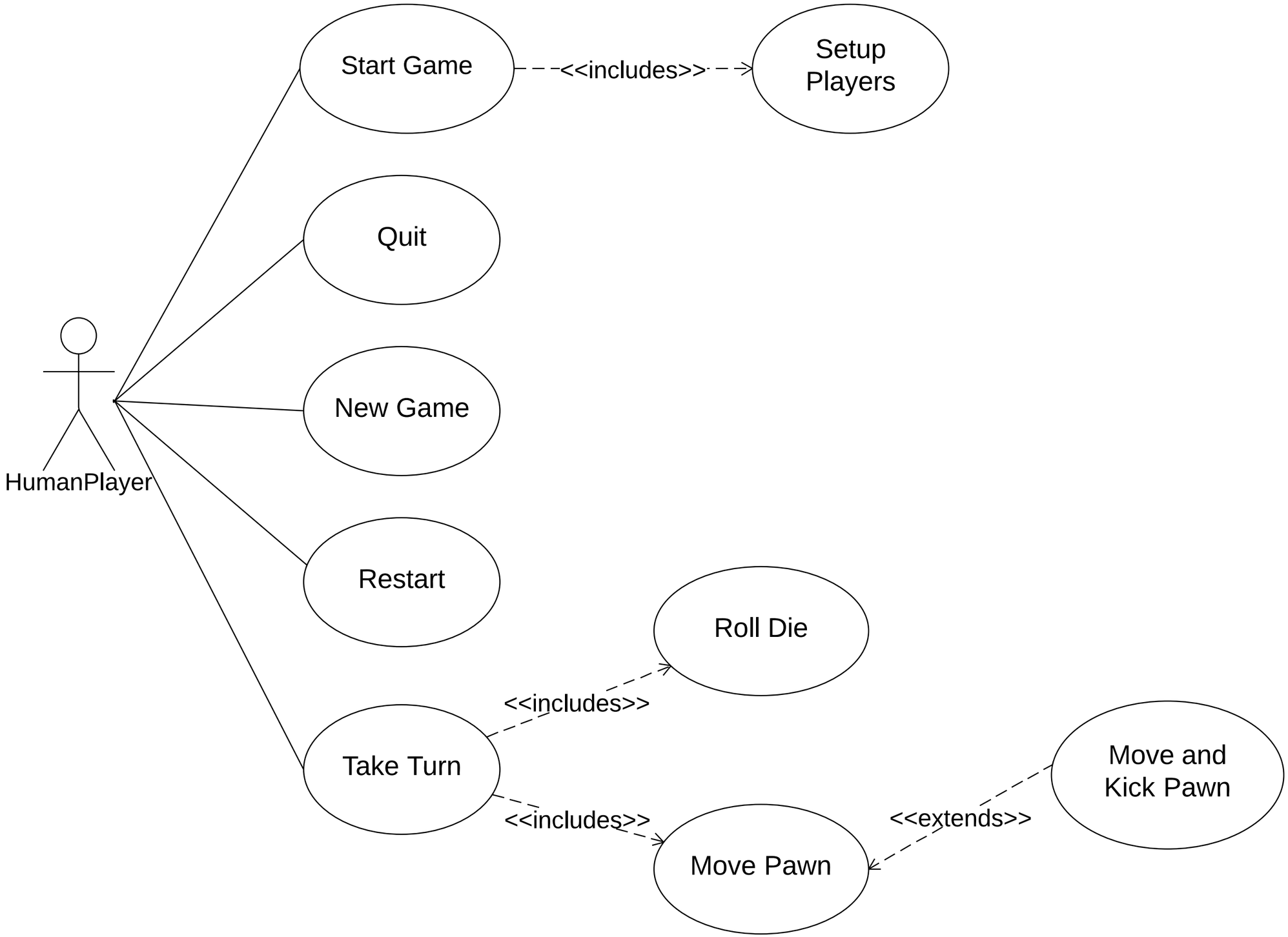
Sequence Diagram

- After implementing the game, we went with an event driven design
- To demonstrate the concept of events in the new sequence diagram, we've introduced "found-messages" that send messages to the main GameLogic from the GameController
- We've included the 3 main events that relate to board movement: RollEvent, MoveEvent, and TurnEvent
    - RollEvent signals that the player (whether human or computer) should roll the die
    - MoveEvent signals when the player should begin moving to the destination spot
    - TurnEvent signals to advance the turn to the next player

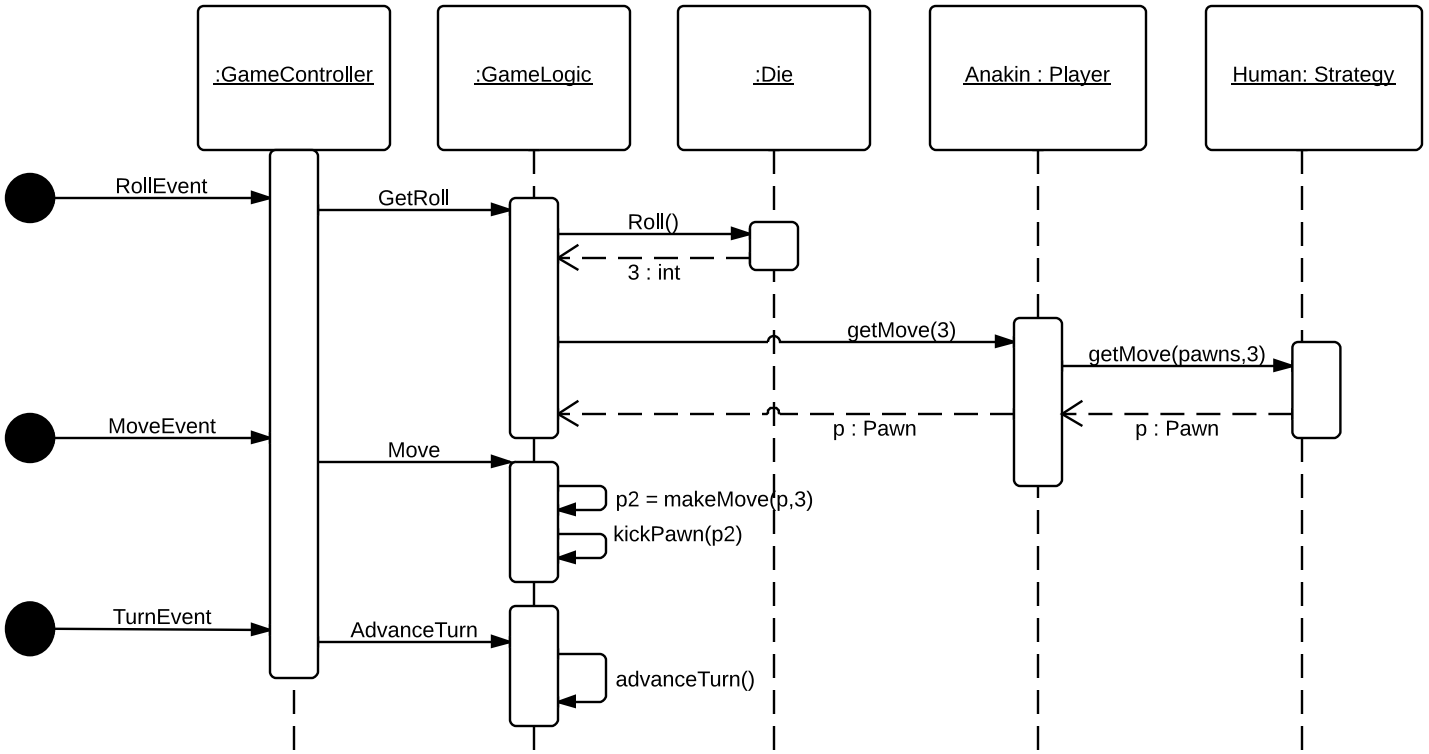
Class Diagram



Use Case Diagram



Sequence Diagram



Group 8 Milestone 2 Contributions

Group 8 is composed of Alastair Fehr, Greg Richardson, Bill Xiong, and Hiroki Yun. The group worked on the project asynchronously, coordinating over git and Google Docs. The fundamental game UI, infrastructure, and animations were implemented by Alastair. The Strategies were largely coded by Bill, and the various screens, game logic, and UI elements were done by Hiroki and Greg. The entire group contributed to the diagrams and write ups via LucidChart and Google Docs.