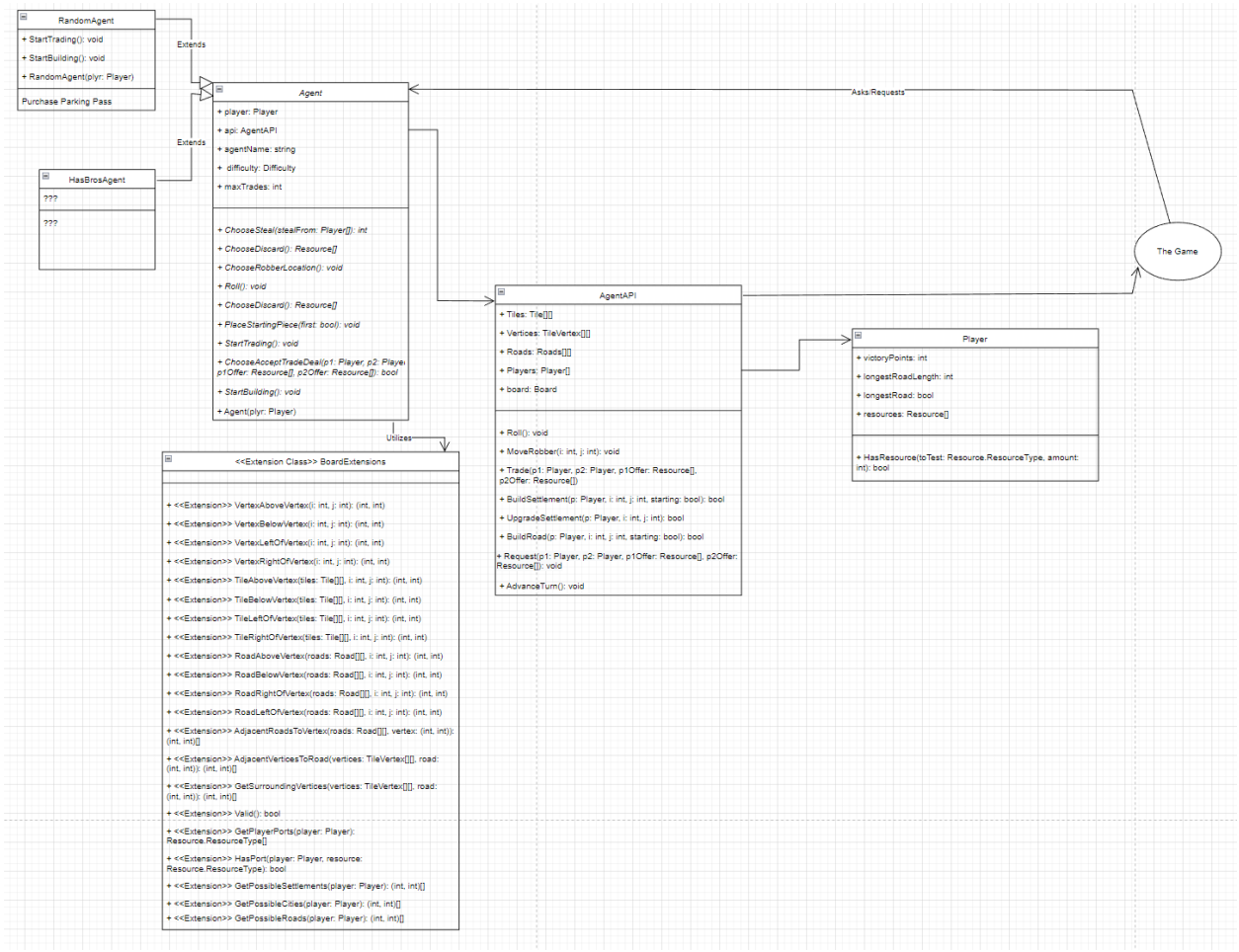
# Development Documentation for Catan AI

Team Has-Bros

# Architecture

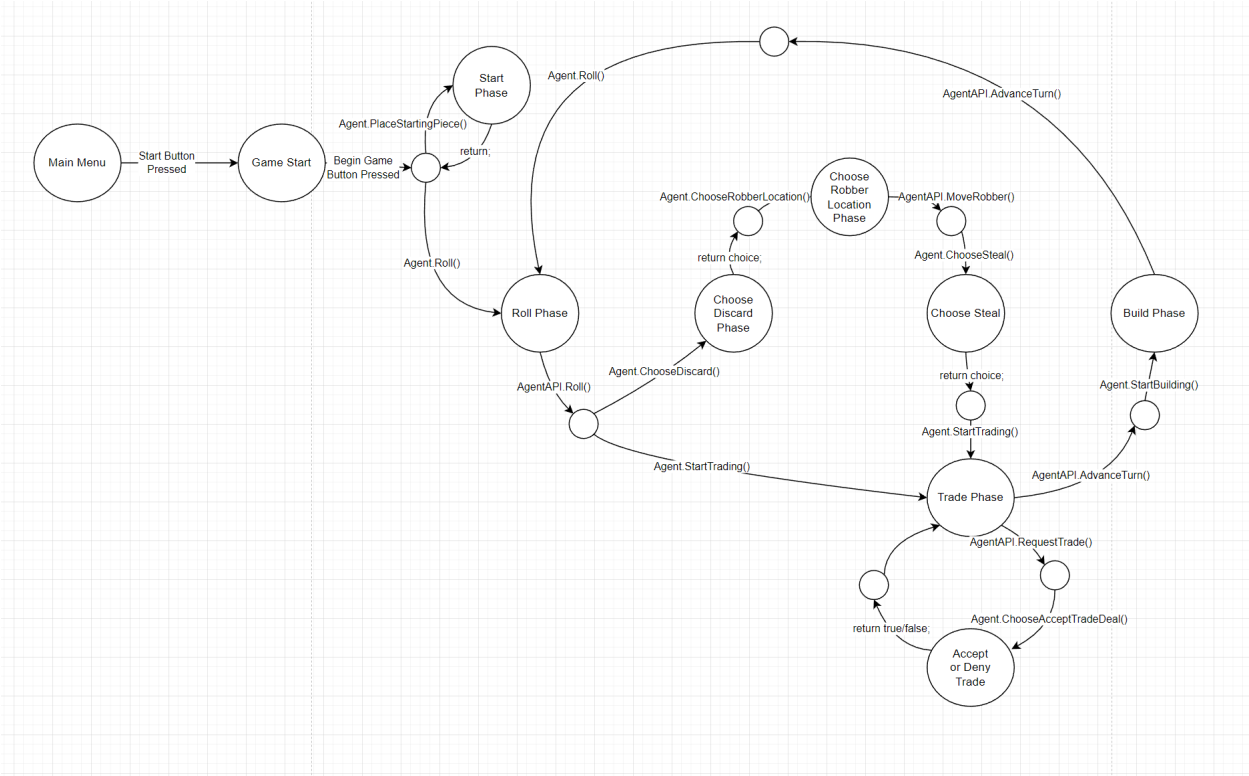
The AI portion of the game is outlined in the UML diagram below.




In our game, we have two agents: An agent that behaves randomly, **RandomAgent**, and the agent that we will be developing, the **HasBrosAgent**. These agents will **extend** the **Agent** class, and implement its functions. The **Derived Agents** section provides more information on how to do this.

# Flow of Control

The following diagram describes how the flow of control happens for our AI. The small circles represent the **Game** in control, and the large named circles represent the **AI** in control.




The game will end when a player reaches 10 victory points (VP).

Each AI action happens in three steps:

1) The game calls upon the AI to make a decision.
2) The AI makes said decision and takes the necessary action.
3) Control is transferred back to the game.

These three steps should be implemented in a class derived from **Agent**. Each of these functions is described in the **Agent** section.

# Agent

The **Agent** class contains 8 functions, each of which involves the agent making its decision on what action to take. In each derived agent (such as **RandomAgent** and **HasBrosAgent**), the creation of the AI involves filling out these 8 functions, referring to common or separate models that we create.

**Note:** See the **Agent** class and **RandomAgent** classes to view examples of the implementation of each function. These are stored in *scripts/AI**.***

## Roll()

Parameters: None
Returns: void

This function simply calls the AgentAPI's Roll() function.

*Dev Note: Unless you are doing anything fancy or strange, there is no need to override this.*

## ChooseRobberLocation()

Parameters: None
Returns: void

This function is called when a 7 is rolled, and the agent needs to choose the location of where the robber piece should be placed.

## ChooseDiscard()

Parameters: int **discardAmount;** - The number of resources the agent needs to discard.
Returns: Resource[] representing what resources to discard.

This function is called when the specified player has more than 7 cards, and a 7 is rolled.

## ChooseSteal()

Parameters: Player[] **stealFrom;** - A list of players who the agent has an option to steal from
Returns: int representing the index of which player the AI has chosen to steal from.

This function is called when a 7 is rolled, and the robber is placed on a tile next to several players, and the AI who rolled the 7 needs to choose which to steal from.

## PlaceStartingPiece()

Parameters: *(optional)* bool **first** = true; - A parameter representing if the piece placed is the first or second starting piece placed.
Returns: void

This function is called when the agent should place one of their starting pieces in the **start phase**.

*Dev Note: Make sure to include this in your override:*

```
if (first)
{
    api.board.DistributeResourcesFromVertex((i, j));
}
```

*i and j represent the index of what vertex you're distributing from, aka the vertex the AI placed their first starting piece on. If you don't have this, the AI players won't get their resources.*

# StartTrading()

Parameters: None
Returns: void

This is where trade phase is handled.
*Dev note: see **RandomAgent** for an example implementation.*

# ChooseAcceptTradeDeal()

Parameters:

- Player p1 - Player offering the deal
- Player p2 - Player receiving the deal
- Resource[] p1Offer - Offer from the player offering the deal
- Resource[] p2Offer - Offer from the player receiving the deal

Returns: bool representing whether or not the AI accepts the trade

This function is called when a player or AI offers a trade to an AI.

# StartBuilding()

Parameters: None
Returns: void

This is where build phase is handled.
*Dev note: see **RandomAgent** for an example implementation.*

# Derived Agents

Here, RandomAgent is inheriting Agent. This means that RandomAgent has all of the functions included on Agent, and can call/override them.

```
public class RandomAgent : Agent
```

All of the function in **Agent** are marked **virtual**. This means that you can **override** them. An example:

The function in **Agent:**

```
5 references
public virtual void StartTrading()
{
    api.AdvanceTurn();
}
```

All this function does is advance the turn as soon as the AI's trade phase starts. In **RandomAgent**, we rectify this by *overriding* this function.

```
4 references
public override void StartTrading()
{
    for (int i = 0; i < maxTrades; i++)
    {
        Player p = api.Players[UnityEngine.Random.Range(0, api.Players.Length)];
        if (p.resourceSum > 0 && p.playerIndex != player.playerIndex)
        {
            Resource[] senderOffer = new Resource[] { player.RandomResource() };
            Resource[] recieverOffer = new Resource[] { p.RandomResource() };
            Trader.Request(player, p, senderOffer, recieverOffer);
        }
    }

    base.StartTrading();
}
```

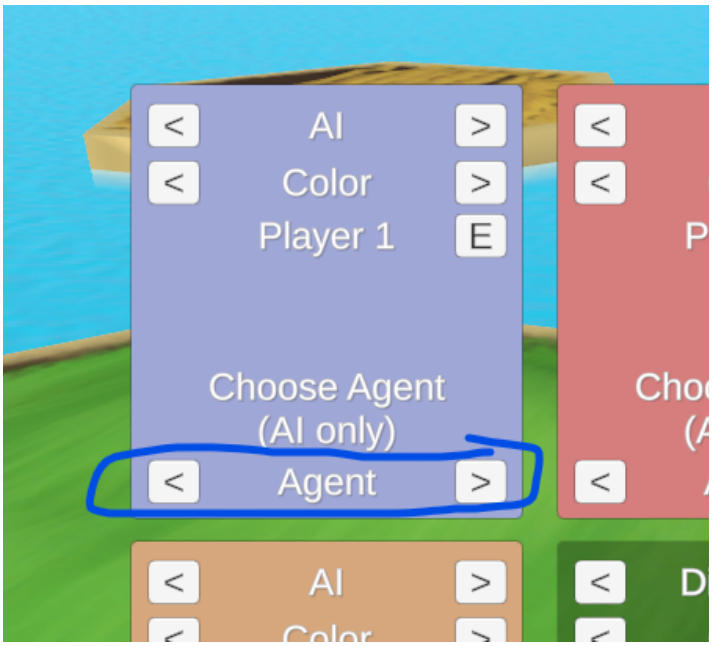
Note the keyword *override* in the function header. You'll notice that this function doesn't have "api.AdvanceTurn();". Doesn't this mean that the function won't advance the turn, and we'll be stuck in trade phase forever?

```
base.StartTrading();
```

No, because at the bottom of this function, the **base function** is called. This means that the **original, virtual method** is being called here. So…

```
base.StartTrading();
```
in the **child class** calls
```
public virtual void StartTrading()
{
    api.AdvanceTurn();
}
```
in the **parent class.**

In game, switch between what agent you're using on each player with this:




(In the start menu scene)

# AgentAPI

The attributes and functions under the **AgentAPI** class are used to interface with the game. These are what you can use in the **HasBrosAgent.**

## Attributes

Each attribute on the AgentAPI interfaces with the game, and is already set up. Attributes should be read, **but not written to.** There are API functions for properly writing data to the attributes.

It is up to you how you analyze the data. C# has for-loops and foreach-loops.

### Tile[][] Tiles

An array of arrays of type Tile. This is the list of every hexagonal tile on the board, with each row of tiles being a new array in the array.

### TileVertex[][] Vertices

An array of arrays of type TileVertex. This contains the hexagon vertices on the board.

### Road[][] Roads

An array of arrays of type Road. This contains the roads on the board.

### Player[] Players

An array of type Player. Each player contains all player functions.

## Roll()

Parameters: None
Returns: void

This function rolls the dice.

## MoveRobber()

Parameters: int i, int j representing the location to which the robber was moved.
Returns: void

This function is used by the AI to choose where to move the robber.

# Trade()

Parameters:
- Player p1 - Player offering the deal
- Player p2 - Player receiving the deal
- Resource[] p1Offer - Offer from the player offering the deal
- Resource[] p2Offer - Offer from the player receiving the deal

Returns: void

Forces a trade. The other person cannot say no, and the trade will be completed.

# RequestTrade()

Parameters:
- Player p1 - Player offering the deal
- Player p2 - Player receiving the deal
- Resource[] p1Offer - Offer from the player offering the deal
- Resource[] p2Offer - Offer from the player receiving the deal

Returns: bool representing if the trade was accepted or not.

Sends a request to another player for a trade, and returns whether the trade was successful.

# BuildSettlement()

Parameters:
- Player p: Player for whom the settlement is being built
- int i, j: ints representing the indices of the settlement to be built
- *Optional* bool starting = false: Boolean representing whether the settlement is being built during start phase

Returns: bool representing if the construction was successful or not.

Constructs a settlement at the specified location for the specified player.

*Dev note: If the construction fails for any reason, like if the specified settlement is invalid to build at, the function will return false. Make sure to account for this.*

# UpgradeSettlement()

Parameters:
- Player p: Player for whom the settlement is being upgraded
- int i, j: ints representing the indices of the road to be upgraded

Returns: bool representing if the construction was successful or not.

Upgrades a settlement at the specified location for the specified player

*Dev note: If the upgrade fails for any reason, like if the specified settlement is invalid to build at, the function will return false. Make sure to account for this.*

# BuildRoad()

Parameters:

- Player p: Player for whom the road is being built
- int i, j: ints representing the indices of the road to be built
- *Optional* bool starting = false: Boolean representing whether the road is being built during start phase

Returns: bool representing if the construction was successful or not.

Constructs a road at the specified location for the specified player.

*Dev note: If the construction fails for any reason, like if the specified road is invalid to build at, the function will return false. Make sure to account for this.*

# AdvanceTurn()

Parameters: None
Returns: void

Advances the game's turn.

# Player

*Dev note: Please do not edit the values on the player object. They are automatically calculated by the game, and don't need to be touched.* ***Consider them readonly****. Use the API functions instead.*

## Attributes

### int victoryPoints

Represents the amount of victory points a player has. Is calculated every time this player takes a turn.

### int longestRoadLength

Represents the length of the player's longest road. Is calculated for this player every build phase.

### bool longestRoad

Represents whether or not this player has the longest road

### Resource[] resources

An array of resources that the current player owns.

## HasResource

Parameters: Resource.ResourceType toTest, int amount.
Returns whether or not a player has equal to or greater than the amount of specified resources.

# Resource

## Attributes

### Resource.ResourceType type

Represents the type of resource.

### int amount

Represents the amount of the resource.

### enum ResourceType

An enum representing the resourceType. It is defined as: Any, Wool, Grain, Wood, Brick, Ore, None

# BoardExtensions

Extension methods work like regular methods, except they usually follow an object with a "." inbetween, rather than passing it in as a parameter. For example, on the **AgentAPI**, there's an array of **TileVertex** objects called **Vertices**. For example, to get a vertex above another, call:

(int, int) above = **Vertices**.VertexAboveVertex(i, j).

Here, **above** is assigned to what **VertexAboveVertex** returns, which is a tuple of ints representing the location of the vertex above the one specified by i and j.

## Vertex to Vertex

### Vertex…Vertex()

Four functions:

- Vertex**Above**Vertex(...)
- Vertex**Below**Vertex(...)
- Vertex**LeftOf**Vertex(...)
- Vertex**RightOf**Vertex(...)

Follow this format:

Vertex…Vertex(int i, int j)

Parameters:

- int i, j representing location of first vertex

Returns: (int, int) tuple representing the location of the found vertex.

*Dev note: Returns (-1, -1) if the vertex isn't valid. Please account for this.*

## Vertex to Tile

### Tile…Vertex()

Four functions:

- Tile**Above**Vertex(...)
- Tile**Below**Vertex(...)
- Tile**LeftOf**Vertex(...)
- Tile**RightOf**Vertex(...)

Follow this format:

Tile…Vertex(Tile[][] tiles, int i, int j)

Parameters:

- Tile[][] tiles: Pass in **Tiles** attribute from the **AgentAPI**
- int i, j representing location of first vertex

Returns: (int, int) tuple representing the location of the found tile.

*Dev note: Returns (-1, -1) if the tile isn't valid. Please account for this.*

# Vertex to Road

## Road…Vertex

Four functions:

- Road**Above**Vertex(...)
- Road**Below**Vertex(...)
- Road**LeftOf**Vertex(...)
- Road**RightOf**Vertex(...)

Follow this format:

Road…Vertex(Road[][] roads, int i, int j)

Parameters:

- Road[][] roads: Pass in **Roads** attribute from the **AgentAPI**
- int i, j representing location of first vertex

Returns: (int, int) tuple representing the location of the found road.

*Dev note: Returns (-1, -1) if the tile isn't valid. Please account for this.*

# Vertex to Road

## AdjacentRoadsToVertex()

Parameters:

- Road[][] roads - Pass in **Roads** from **AgentAPI**
- int i, j representing the location of the vertex

Returns: (int, int) tuple array representing the locations of the found roads

Ex: vertices.AdjacentRoadsToVertex(roads, i, j)

# Roads to Vertex

## AdjacentVerticesToRoad()

Parameters:

- TileVertex[][] vertices - Pass in **Vertices** from **AgentAPI**
- int i, j representing the location of the vertex

Returns: (int, int) tuple array representing the locations of the found roads

Ex: roads.AdjacentVerticesToRoad(vertices, i, j)

# Tile to Vertex

## GetSurroundingVertices()

Parameters:

- TileVertex[][] vertices - Pass in **Vertices** from **AgentAPI**
- int i, j representing the location of the tile

Returns: (int, int) tuple array representing the locations of the found vertices

Ex: tiles.GetSurroundingVertices(vertices, i, j)

# Valid()

Parameters: None
Returns: True if the (int, int) is NOT equal to (-1, -1), else false

Ex: (-1, -1).Valid() returns false, and (1, 4).Valid() returns true.

You might use this to check if there is a road above a vertex, or similar situation. You'd write:

```
if (vertices.RoadAboveVertex(roads, i, j).Valid())
{
        // Do some stuff
}
```

# GetPlayerPorts()

Parameters:

- Player player: The player of which we are checking the ports

Returns: Resource.ResourceType[] of which resources the player has a port on

Ex: vertices.GetPlayerPorts(player);

# HasPort()

Parameters:

- Player player: The player of which we are checking the ports
- Resource.ResourceType of which resource we are checking

Returns: Boolean representing if the specified player has a port of the specified resource

Ex: vertices.HasPort(player, Resource.ResourceType.Brick);

## GetPossibleSettlements()

Parameters:

- Player player: The player of which we are checking the possible settlements

Returns: (int, int)[] of all settlements where it is immediately possible for the player to build

Ex: (int, int) possibleSettlements = board.GetPossibleSettlements(Player)

## GetPossibleCities()

Parameters:

- Player player: The player of which we are checking the possible cities

Returns: (int, int)[] of all cities where it is immediately possible for the player to upgrade

Ex: (int, int) possibleCities = board.GetPossibleCities(Player)

## GetPossibleRoads()

Parameters:

- Player player: The player of which we are checking the possible roads

Returns: (int, int)[] of all roadswhere it is immediately possible for the player to build

Ex: (int, int) possibleRoads = board.GetPossibleRoads(Player)