Design Document
CMPT 370

**Group C4**
Jack Huang
Brandon Jamieson
Ixabat Lahiji
Daniel Morris
Kevin Noonan

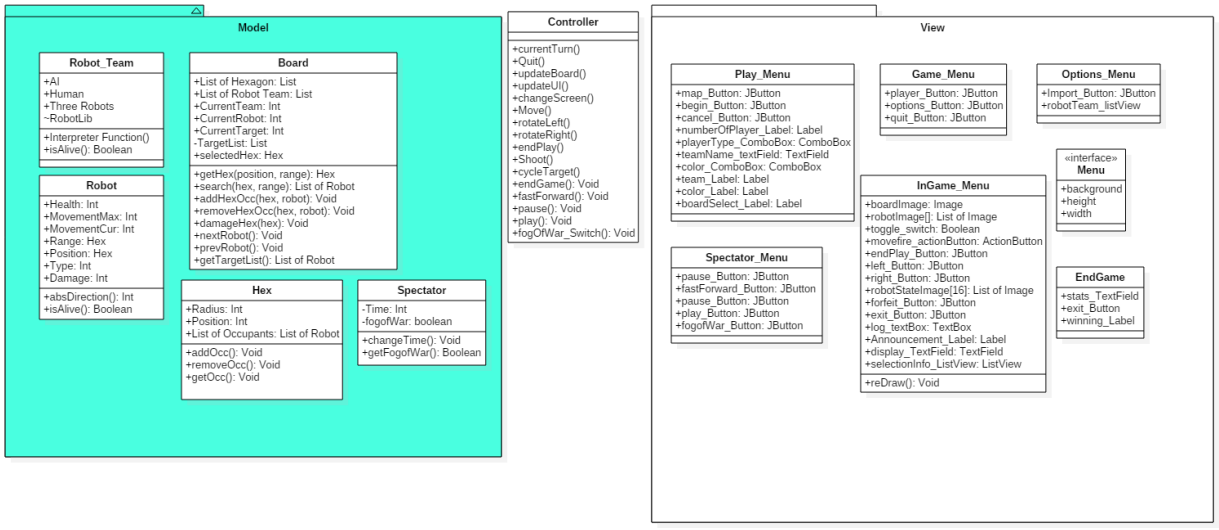
## Architectural Choice

Our team's design makes use of the model-view-controller architecture. We chose this architecture because it plays to our team's strengths, as we are all most familiar and experienced with it. One benefit of this is that model-view-controller cleanly separates two major components, the model and view. This allows us to develop the two separately without relying on each other, as well as providing the user with a clear, bounded interface to the system. On top of this, the architecture keeps all robot-handling confined to a single major component: the model.

The view will be event-driven, handling interactions with users and relaying information to and from the controller. It will serve as an interface into the system and will inform the controller of user-input. The controller will also be event-driven, handling interactions with actors by manipulating the model. It will receive and handle user-input from the view, as well as updating and fetching from the model as needed. The model itself will have aspects of both blackboard and database styles due to the components inside. The robot librarian will serve as a database to be accessed by the controller for operations involving the access of robot teams. The game's board will act as the blackboard, with the controller manipulating the state of it as gameplay progresses.

One architectural option we considered instead of model-view-controller was a modified three-layer architecture. The top-layer would have been event-driven and handled GUI interaction. The middle-layer would have been blackboard style and the bottom-layer database style. Intermediate information would be stored in the middle-layer for game use, as well as receive data from the bottom-layer. We reasoned against this architecture because it did not encapsulate the guts of the game to one major component and instead, divided it among two of the layers. Model-view-controller provided a clear division between the guts and the rest of the system so we agreed upon it as our overall architecture.

\/ UML STUFF GOES BELOW \/




Board

The board deals with managing moves and players. It will contain references to the robots on the board, references to the current team/robot in play, as well as positions and targeting.

- The getHex() function will return a Hex object, based on a distance and range that is passed into the function.
- The search() function will return a list of the Hexes that contain a robot on it, using a passed in Hex value as the origin, and moving outwards in a specified range.
- The addHex() function takes in a hex and a reference to a robot, and adds the referenced robot to the Hex.
- The removeHex() function takes in a hex and a reference to a robot, and removes the referenced robot from the Hex.
- The damageHex() function applies damage to the Hex value specified. Amount of damage applies is determined by the current robot's stats.
- The nextRobot() function changes the selected hex to the next robot that is available to be attacked by the current robot based on the current robot's range.
- The previousRobot() function changes the selected hex to the previous robot that is available to be attacked by the current robot based on the current robot's range.

The changes in the board will call an update to the view when necessary.

<u>Hex</u>

The hex class contains the info for a specific hexagon on the board. The radius and the position determine where the hexagon is on the board. The radius determines the amount of spaces away from the center of the board the hexagon is. The position determines which hexagon the specified hexagon is in that range starting from 0 to the right of the center then incrementing upwards clockwise. The list of occupants is a private variable in the hex will be a list of robots that are currently on that hex's position coordinates. The functions addOcc() and removeOcc() will change this list by either adding a robot to the list or deleting a robot from the list. The function getOcc() will return the list of robots currently on that hex.

Robot

The robot class contains the statistics about each individual robot on the robot team. The health keeps the health amount the robot has or has left. MovementMax and MovementCur keep track of the number of possible moves the robot has and the moves that it has left, respectively. The range field keeps track of how far the robot can shoot in terms of the furthest hexagonal board piece. The position field keeps track of the hexagonal board piece that the robot is currently placed on. The

Spectator

The spectator class contains the required information for when the spectator is active. (When there are no longer any human teams alive) The time field is an integer either 1, 2, or 4. This value determines the length of time in between plays. In between plays that the spectator is watching the delay time is the calculated by the default time divided by the time variable. The fogofWar boolean field will determine whether or not the UI displays the fog of war for the current robot's vision range.

- The changeFogofWar() function will change the fogofWar field.
- The changeTime() function will double the time field unless the time field is currently 4 then the time field will be set back to 1.
- The getFogofWar() function will return the fogofWar field.

We made several tentative specifications in our requirements document that we modified in our design document. Things we had previous classified as "should have" or "nice-to-have" included: operations such as clicking hexagons to move or attack, as well as having tanks visually stack on the same hex.

After much discussion, we decided that clicking hexagons to move and attack was not a feasible option for our design. Due to the nature of the robots and their individual orientations and relative coordinates, make click-based movement would be hard to implement. It would require some form of translation to an "absolute" board coordinate independent of the robot's relative coordinates, and then moving the robot based on this. We would also have to include functionality for if the user clicked multiple hexes away from the current robot, involving some sort of path-finding.

To avoid all these issues, we decided that movement and attacking could be performed via buttons. When moving, the user will rotate their tank using left and right buttons and move in

the current direction by pressing "Move". When attacking, the user will use the left and right buttons to cycle through potential hexes to shoot at, and will fire on the currently selected hex by pressing "Fire".

The notion of having tanks visually stack on the same hex was also modified in our design. Originally, tanks would stack and health bars would be displayed above them. We realized this may be unclear visually for the user, as health bars would end up stacked as well. Implementing some way of aligning health bars to avoid overlap while still providing clarity as to which robot they belong to would be a tough task to fulfill.

To combat this issue, our team decided to implement a "Hex Info" display panel. This panel will display the list of robots occupying the selected hex, as well as their health. This way, we can clearly notify the user of the tanks on any given hex without anything visually interfering with the game board itself. On top of this, we can simply update the info of one panel as opposed to updating the alignment of individual tanks. This also removes our need for a health bar for each robot.