# AJETH

**4stones**

# Software Design Document

|  |  |
| --- | --- |
| **Class name: AI** | |
| **Brief description:** Acts as a computer object with methods to execute moves. | |
| int difficulty | **Attribute Description** |
| This is a declaration of an integer to be used through the class as a means of determining which move method to use. |
| **Program Description Language** |
| public int difficulty |
| int makeMove(); | **Method Description** |
| A method to execute one of the various move methods based on the difficulty selected. |
| **Program Description Language** |
| int AI::makeMove(){  switch(\_difficulty){  case 1: //Easy AI  minAlgorithm();  break;  case 2: //Medium AI  return randomMove();  break;  case 3: //Hard AI  smartAI();  break;  }  } |
| int randomMove(); | **Method Description** |
| A method that generates a number then checks if that position on the board is available, if available place piece at that location. If not available, generate new number and try again. |
| **Program Description Language** |
| int randomMove(){  int random\_integer = random number  while (spot isn’t available){  random\_integer = new random number  }  board->placePiece(random\_integer, -1);  return random\_integer;  } |
| int smartAI(); | **Method Description** |
| First checks if it’s possible to win or lose.Try to find a fork;Try to get a viable 3-in-a-row;Try to prevent a fork;If all else fails fall back to medium AI. |
| **Program Description Language** |
| int maxAlgorithm(){  Grid tempboard;  For (int i … board->\_gridLegth) clone the board;  For (int j … board->\_gridLength^2){  If (can win) return position and place piece;  }  For (int j … board->\_gridLength^2){  If (will lose) return position and place piece;  }  for (int x = -1; x < = 1; x++){  if (abs(x) == 1){  if(fork is found){  placePiece(location, -1)  return location;  }  }else{  if (found a viable 3-in-a-row){  placePiece(location, -1);  }  if (last move on board is AI) return move;  }  }  //Fall back to medium AI to block  int medium = mediumAi(true);  board->placePiece(medium, -1);  return medium;  } |
| int mediumAi(); | **Method Description** |
| Attempts to block the opponent and/or place a stone in the best possible line and cell. |
| **Program Description Language** |
| Int mediumAi(){  insert complex algorithm here;  return location  } |

|  |  |
| --- | --- |
| **Class name: Grid** | |
| **Brief description:** Acts as a 2D-array based grid with methods to bridge the GUI 1D-array grid with the 2D-array grid | |
| QList<QList<int>> grid; | **Attribute Description** |
| 2D-array of integers that serves as a grid |
| **Program Description Language** |
| public QList<QList<int>> grid; |
| int gridLength; | **Attribute Description** |
| Length of the grid |
| **Program Description Language** |
| public int gridLength; |
| int lastMove; | **Attribute Description** |
| Stores the last move made on the board (stored as 1D coordinate) |
| **Program Description Language** |
| public int lastMove; |
| int lastMoveX; | **Attribute Description** |
| Stores the last move made on the board by Player 1 (stored as 1D coordinate) |
| **Program Description Language** |
| public int lastMoveX; |
| int lastMoveO; | **Attribute Description** |
| Stores the last move made on the board by Player 2 (stored as 1D coordinate) |
| **Program Description Language** |
| public int lastMoveO; |
| int countToWin; | **Attribute Description** |
| The amount of stones needed to win a game |
| **Program Description Language** |
| public int countToWin; |
| int firstPlayer; | **Attribute Description** |
| The value of the player who plays first |
| **Program Description Language** |
| public int firstPlayer; |
| void placePiece(const int index, const int player); | **Method Description** |
| Takes a 1D index and player number to place stone on grid. |
| **Program Description Language** |
| void placePiece(const int index, const int player){  lastMove = index;  lastMoveX or lastMoveO = index;  grid[indexToRow(index)][indexToColumn(index)] = player;  } |
| int indexToRow(int index); | **Method Description** |
| Converts 1D index to 2D row of index |
| **Program Description Language** |
| Int indexToRow(int index){  return index / gridLength;  } |
| int indexToColumn(int index); | **Method Description** |
| Converts 1D index to 2D column of index |
| **Program Description Language** |
| Int indexToColumn(int index){  return index % gridLength;  } |
| int coordinateToIndex(int row, int column); | **Method Description** |
| Converts 2D coordinates to 1D coordinate |
| **Program Description Language** |
| int coordinateToIndex(int row, int column){  return ((row \* gridLength) + (column \* gridLength));  } |
| int valueFromIndex(int index); | **Method Description** |
| Returns the value located at a coordinate given a 1D index |
| **Program Description Language** |
| int valueFromIndex(int index){  return grid[indexToRow(index)][indexToColumn(index)];  } |
| bool checkWin(int player); | **Method Description** |
| Checks to see if player has won |
| **Program Description Language** |
| bool checkWin(int player){  bool isWin = false;  for/while (start to finish for direction){  count player stones  }  if (count == needed to win) isWin = true;  return isWin;  } |
| public bool isFilled(); | **Method Description** |
| Returns if the board is filled |
| **Program Description Language** |
| public bool isFilled{  int count = 0;  for (space != 0) count++;  return (count >= 25);  } |
| void boardReset(); | **Method Description** |
| Resets the grid; |
| **Program Description Language** |
| void isFilled{  \_lastMove = 0;  \_moveCount = 0;  for (all spaces) space = 0;  } |

|  |  |
| --- | --- |
| **Class name: Main** | |
| **Brief description:** This is the main class. It initializes the GUI, players, grid, etc. | |
| Player\* player; | **Attribute Description** |
| Player object for player 1 |
| **Program Description Language** |
| public Player\* player; |
| AI\* computer; | **Attribute Description** |
| AI object for player 2 |
| **Program Description Language** |
| public AI\* computer; |
| int main(int argc, char \*argv[]); | **Method Description** |
| Main method, initializes entire application. |
| **Program Description Language** |
| int main(int argc, char \*argv[]){register types for GUI;create application engine;init board, player, computer;launch application;} |

|  |  |
| --- | --- |
| **Class name: Player** | |
| **Brief description:** Holds all player data | |
| int wins; | **Attribute Description** |
| Number of wins |
| **Program Description Language** |
| int wins; |
| int draws; | **Attribute Description** |
| Number of draws |
| **Program Description Language** |
| int draws; |
| int losses; | **Attribute Description** |
| Number of losses |
| **Program Description Language** |
| int losses; |
| int losses; | **Attribute Description** |
| Number of losses |
| **Program Description Language** |
| int losses; |
| QString username; | **Attribute Description** |
| Name of player |
| **Program Description Language** |
| public QString username; |
| QString objectID; | **Attribute Description** |
| objectID of the player in Parse |
| **Program Description Language** |
| QString objectID; |
| QString session; | **Attribute Description** |
| Session token of the player in Parse |
| **Program Description Language** |
| QString session; |
| void loginPlayer(QString username); | **Method Description** |
| Login player |
| **Program Description Language** |
| void loginPlayer(QString username){\_username = username;ParseHelper \*p;p->login(username);} |
| void registerPlayer(QString username); | **Attribute Description** |
| Registers the player |
| **Program Description Language** |
| void registerPlayer(QString username){\_username = username;ParseHelper \*p;p->login(username);} |
| void updatePlayer(int wins, int losses, int draws); | **Attribute Description** |
| Update the player |
| **Program Description Language** |
| void updatePlayer(int wins, int losses, int draws){  \_wins += wins;  \_losses += losses;  \_draws += draws;  ParseHelper \*p = new ParseHelper(this);  p->updatePlayer(objectID, session, \_wins, \_losses, \_draws);  } |
| void logout(); | **Attribute Description** |
| Logs out the player |
| **Program Description Language** |
| void logout(){  \_username = "";  objectID = "";  session = "";  \_wins = 0;  \_losses = 0;  \_draws = 0;  } |
| void deletePlayer(); | **Attribute Description** |
| Deletes the player |
| **Program Description Language** |
| void deletePlayer(){  ParseHelper \*p = new ParseHelper(this);  p->deletePlayer(objectID, session);  logout();  } |
| void handleLogin(bool exists, QNetworkReply \*reply); | **Attribute Description** |
| Handle ParseHelper response from login |
| **Program Description Language** |
| void handleLogin(bool exists, QNetworkReply\* reply){  if (exists){  QJsonObject json = QJsonDocument::fromJson(reply->readAll()).object();  \_username = json.value("username").toString();  objectID = json.value("objectId").toString();  session = json.value("sessionToken").toString();  \_wins = json.value("win").toInt();  \_losses = json.value("loss").toInt();  \_draws = json.value("draw").toInt();  emit loginFinished();  }else{  \_username = "";  emit loginFailed();  }  } |
| void handleRegister(bool exists, QNetworkReply \*reply); | **Attribute Description** |
| Handle ParseHelper response from registration |
| **Program Description Language** |
| void handleRegister(bool exists, QNetworkReply\* reply){  if (exists){  QJsonObject json = QJsonDocument::fromJson(reply->readAll()).object();  objectID = json.value("objectId").toString();  session = json.value("sessionToken").toString();  \_wins = 0;  \_losses = 0;  \_draws = 0;  emit registerFinished();  }else{  \_username = "";  emit registerFailed();  }  } |