InSight: RIVIER ACADEMIC JOURNAL, VOLUME 7, NUMBER 2, FALL 2011

CLUE COMPUTER GAME

Douglas Selent '11G[*]
M.S. Program in Computer Science, Rivier College

1: Summary

The purpose of Clue computer game project was to create a computer version of the popular board game Clue. The existing version is not free and is also not that good. The AI, board presentations, and game mechanics of the existing version were all bad. The Clue game for this project improves on all of those areas with most of the emphasis put into the artificial intelligence of computer players in the game.

The Clue computer game will have most features of any computer game. It offers simple and intuitive user interfaces for the user. The game includes a soundtrack as well as roughly 40 other sounds to enhance the game. Several images were also used for a better look to the game. A single player mode and multiplayer mode are also available to play the game in.

The game incorporates several areas and concepts in computer science. The multiplayer mode uses multithreaded client-server architecture with message passing over a TCP/IP network connection. The concept of synchronization is used in the multiplayer game to ensure all players remain in the same game state. A state machine is used in both the single player and multiplayer modes for keeping the state of the game. The state machine operates with the use of timers to keep the timing more realistic. Much of the ease of the game movement comes from the complex recursive algorithms used to calculate movement spots and paths to those spots. The AI also uses complex and lengthy algorithms to make good decisions. Several UML diagrams were used in the documentation of the program to represent how it works. Class diagrams, package diagrams, use case diagrams, sequence diagrams, and several other diagrams were used to represent different aspects of the program.

The actual game consists of over 20k lines of java code spread out over 40 classes. There is about 900MB worth of sound files for the game. This is because the sounds are all uncompressed to keep the game more portable without the use of additional libraries to play sound. The game also has over 25 images that can be displayed for different graphics settings. The game is able to run on any computer with a java runtime environment that is somewhat up to date.

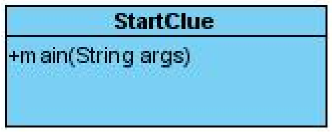
2: Architectural Design

2.1: Introduction

The architectural design consists of all the classes and their relationship with each other. To make things more visible each class is shown separately with their data members and operations. All GUI related data members are not included. Basic set() and get() methods are also not included, but less basic ones are. All handlers, table models, and table renderers are also not shown. A full class diagram is shown but excludes the information shown in he individual class diagrams. This was done for visibility purposes. Package groupings are also shown in separate diagrams without the relationships because the full class diagram shows the relationships.

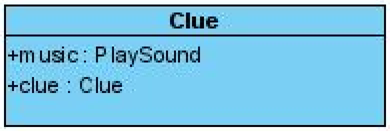
2.2: Classes

1. StartClue



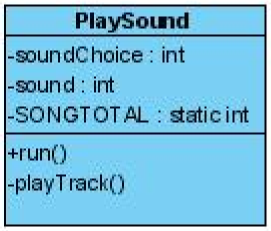
The StartClue class is the driver class for the program. It provides the main entry point and makes the Clue class.

2. Clue



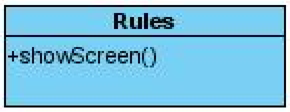
The Clue class provides the main screen GUI for the Clue game. It passes an instance of itself to all other GUI classes for reference. It holds a PlaySound object so the soundtrack can be toggled on or off. When the exit to main screen option is chosen from other screens this GUI Clue class will become visible again. The Clue class GUI provides buttons to get to the Rules screen, MPSetupScreen, and the SinglePlayerSetup screen.

3. PlaySound



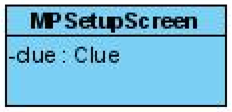
The PlaySound class is the class that plays all the sounds in the game. All sounds are created as a new anonymous thread, which plays the specified sound then ends. It can be accessed by any class. For the soundtrack to be toggled on or off it must be referenced through the Clue class. The static integer SONGTOTAL holds the total number of songs in the sound track. This does not include all the sounds, but just the music for the game. The sound integer variable is set by the Clue class to determine whether or not the game music should play. The soundChoice variable specifies what sound should play.

4. Rules



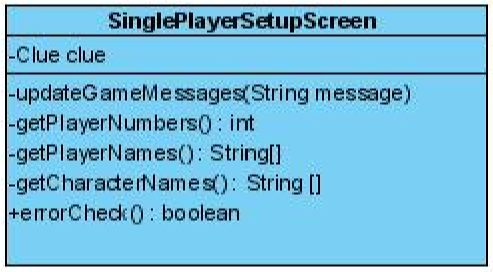
The rules screen provides a GUI for displaying the game rules as well as some additional notes. It can be accessed from any main screen with a rules menu option. None of the classes hold a Rules object. Instead a new Rules class is created anonymously every time the rules option is chosen. The instance of the Rules class will disappear when the rules screen is closed. The Rules class basically reads in a special text file with the information. The text file is special because it has special tags for font and size similar to HTML language. The Rules class identifies these tags and makes the text a difference font, size, or color based on the tag. This provides a much nicer look to the rules screen.

5. MPSetupScreen



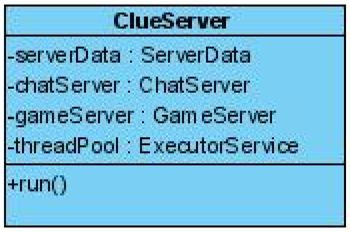
The MPSetupScreen is an intermediate popup dialog box between the Clue main screen and the MultiplayerSetupScreen GUI. The Clue main screen is still active and visible when this class GUI becomes active. This class proves a small GUI for choosing to host or join a multiplayer game. It also provides a text field to enter the IP address of the host when choosing to join a game. If the host option is chosen, the MPSetupScreen class creates an instance of the ClueServer class as a separate thread. It also creates instances of GameClient1 and ChatClient1 classes as separate threads. Then the Clue class GUI is set to invisible and the MPSetupScreen class GUI is disposed to prepare for the MultiPlayerSetupScreen class GUI to appear as the next main screen. The only difference between selecting the host option and the join option is that the host option creates the server and connects to itself, whereas the join option does not create an instance of ClueServer and uses the IP address entered in the text field to connect to someone else’s game. If the connection process is unsuccessful an error message will be generated by another one of the classes involved in the connection process. After this message is acknowledged the Clue main screen will become visible again. If the connection process is successful another one of the classes in the connection process will create the MultiPlayerSetupScreen class with the new active GUI.

6. SinglePlayerSetupScreen


SinglePlayerSetupScreen
-Clue clue
-updateGameMessages(String message)
-getPlayerNumbers() : int
-getPlayerNames() : String[]
-getCharacterNames() : String []
+errorCheck() : boolean

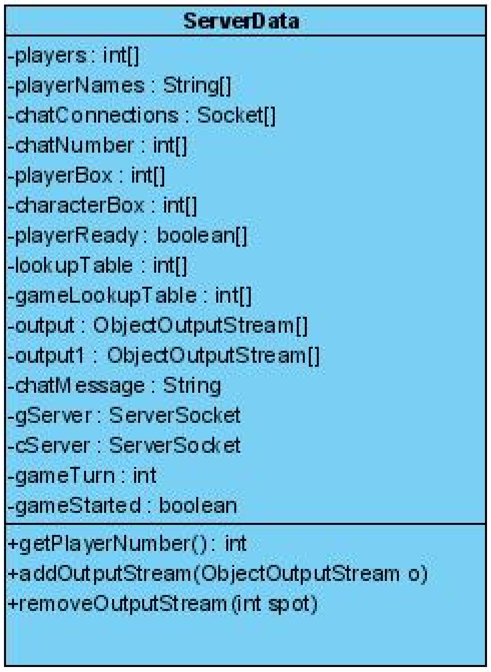
After clicking the single player button from the Clue class GUI, the Clue class GUI is set invisible and the SinglePlayerSetupScreen class is created. The GUI for the SinglePlayerSetupScreen class is set to visible. The SinglePlayerSetupScreen class provides the GUI to setup the game before it starts. This includes the options to choose player names, characters, and the number of players. It also provides a StyledDocument object for displaying important messages. When the start button is clicked the method errorCheck() will check for any errors. If there are errors the error message will be added to the StyledDocument object and an error sound will play. If there are no errors the SinglePlayerSetupScreen class will create an instance of the SinglePlayerGameBoard class and the SinglePlayerSetupScreen class GUI will dispose itself.

7. ClueServer



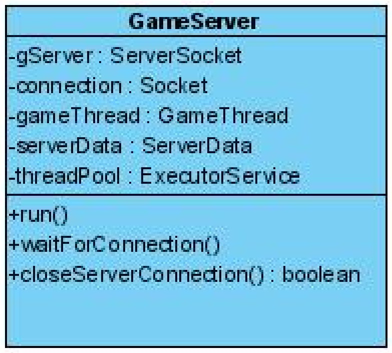
The ClueServer class is the starting class for creating the server mechanics. The class itself does not really do much. It creates a GameServer and ChatServer object and a ServerData object to pass to the game server and chat server. Both the game and chat server run as new separate threads. Once the ClueServer class has created the game server and chat server it is done.

8. ServerData



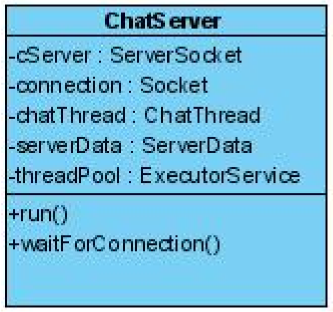
The ServerData class holds all the important information for the network connections of the game. It also holds some important game data. All the server threads have access to a single ServerData object so the data can be shared among every player's server threads. All the methods are synchronized so that data integrity is maintained. It holds an array of all the output streams to all the connected players. This allows for one player to send a message to all the other players by using the ServerData object to get all the output streams. Then a simple loop is done to send the same message to all the streams. Important game data such as if a player is ready or not is held in the playerReady array. Also a characterBox array and a playerBox array hold the array of what characters and players are active. This required because when a new player joins the game the state of the setup screen needs to be transmitted. Several things may have changed since the start of the setup screen by previously connected players. All these arrays hold the state of the setup screen. This way when a new player joins, the setup screen displays the correct and current information. A lookup table is also required for the players. Since players may join the game in a different order than the player spot they are given a lookup table is used to translate the player's connection position to their game player number. When a player joins they are assigned a player number from the getPlayerNumber() method. Their output stream is also added to the array of output streams by the addOutputStream() method. Likewise when a player leaves or disconnects their output stream is removed by the removeOutputStream() method and the tables are updated accordingly.

9. GameServer



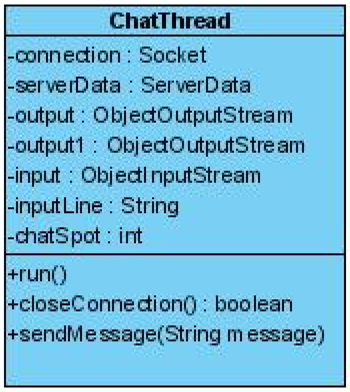
The GameServer class listens for incoming game connections. Every time it receives a new connection it creates a new GameThread object in a separate thread to handle that game connection. It passes the same instance of ServerData to all the connections.

10. ChatServer



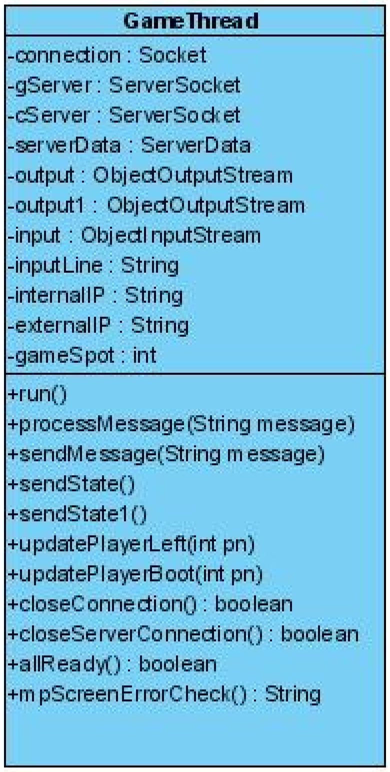
The ChatServer is basically the same as the GameServer except for chat connections instead of game connections. It listens for incoming chat connections and creates a new ChatThread instance with the ServerData object for each connection.

11. ChatThread


ChatThread
-connection : Socket
-serverData : ServerData
-output : ObjectOutputStream
-output1 : ObjectOutputStream
-input : ObjectInputStream
-inputLine : String
-chatSpot : int
+run()
+closeConnection() : boolean
+sendMessage(String message)

The ChatThread class handles all the chat connections. The chat design is simple so the ChatThread is simple. It listens for any input and echoes that input message back out to all the players connected. It uses the ServerData object to get all the output streams connected to the chat. If an exception is thrown while sending a message that loop counter value is used as the spot to remove the chat output stream from the ServerData list. It assumes that that chat connection has been lost and removes it from the list.

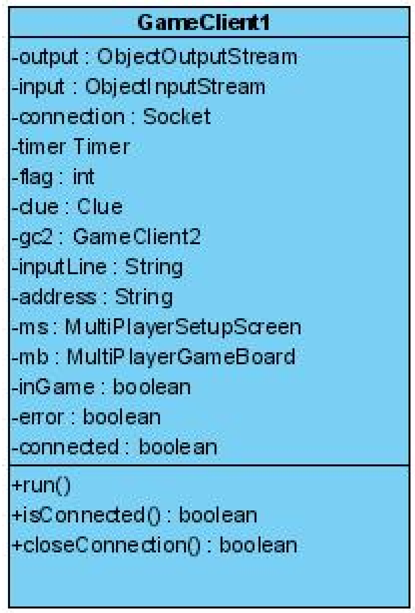
12. GameThread


GameThread
-connection : Socket
-gServer : ServerSocket
-cServer : ServerSocket
-serverData : ServerData
-output : ObjectOutputStream
-output1 : ObjectOutputStream
-input : ObjectInputStream
-inputLine : String
-internalIP : String
-externalIP : String
-gameSpot : int
+run()
+processMessage(String message)
+sendMessage(String message)
+sendState()
+sendState1()
+updatePlayerLeft(int pn)
+updatePlayerBoot(int pn)
+closeConnection() : boolean
+closeServerConnection() : boolean
+allReady() : boolean
+mpScreenErrorCheck() : String

Upon creating the GameThread class, a connection to an external website is made. This is done because it is the only way the hosting player can retrieve his/her external IP address. The host needs this incase the hosting player want to give his/her IP address to a friend to join the game. The GameThread then reads back the contents of the web page, which returns the external IP address of the computer. The

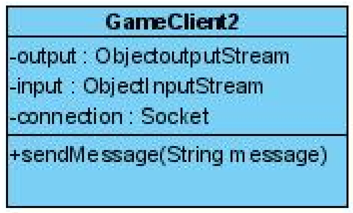
GameThread then runs its main loop, which listens for input. Depending on what the input is some server side processing is done and messages are sent back to one or all the connected players. The function processMessage() analyzes the message that is received and calls the appropriate function based on the message. Some functions include sendState(), allready(), and updatePlayerLeft(). The sendState() function sends the current state of the setup screen data for all the players to have current data displayed. The allready() function is called when all the players in the game are ready to continue. The updatePlayerLeft() function handles when a player has left the game. This includes updating all the affected data values in the ServerData class and sending a message back to all the other players.

13. GameClient1



The GameClient1 class is the class that receives messages from the server. It then passes the message along to either the MultiPlayerGameBoard class or the MultiPlayerSetupScreen class to process the message. It contains an instance of each class, ms and mb, so it can call the appropriate method with the message as the parameter. It also has an instance of the GameClient2 class since it creates it to run in another thread.

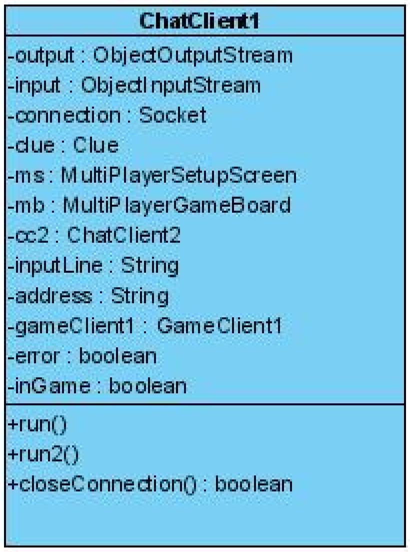
14. GameClient2


GameClient2
-output : ObjectoutputStream
-input : ObjectInputStream
-connection : Socket
+sendMessage(String message)

The GameClient2 class is the class that sends messages to the server. Both the MultiPlayerSetupScreen and the MultiPlayerGameBoard class have an instance of this class since they both have to send messages. When an event occurs in either of the two classes they call the sendMessage() method of the GameClient2 class with the appropriate message as the parameter.

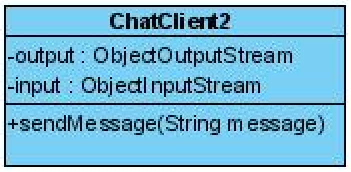
Additional classes also have the same instance of the GameClient2 class. These include the DisproveScreen and the MakeSuggestion classes. Both those classes also need to send messages to the server for what cards were chosen.

15. ChatClient1


ChatClient1
-output : ObjectOutputStream
-input : ObjectInputStream
-connection : Socket
-clue : Clue
-ms : MultiPlayerSetupScreen
-mb : MultiPlayerGameBoard
-cc2 : ChatClient2
-inputLine : String
-address : String
-gameClient1 : GameClient1
-error : boolean
-inGame : boolean
+run()
+run2()
+closeConnection() : boolean

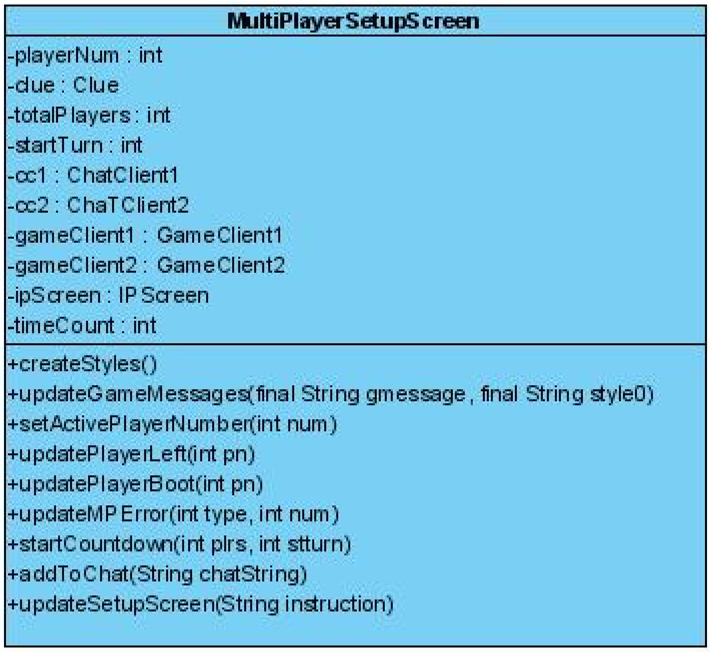
The ChatClient1 class is similar to the GameClient1 class. The difference is that it handles chat messages and not game messages. The ChatClient1 class receives chat messages from the server and passes them along. It also has an instance of the MultiPlayerGameBoard and the MultiPlayerSetupScreen class to pass the message to. The appropriate class handles the chat message by updating the screen.

16. ChatClient2


ChatClient2
-output : ObjectOutputStream
-input : ObjectInputStream
+sendMessage(String message)

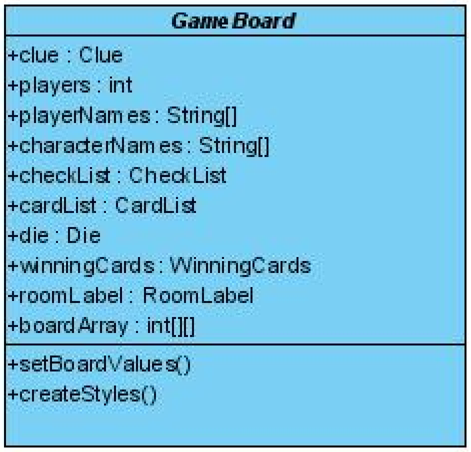
The ChatClient2 class is similar to the GameClient2 class. It sends chat messages to the server. It has the method sendMessage() that takes the message String as the parameter. Both the MultiPlayerSetupScreen and the MultiPlayerGameBoard class have the same instance of the ChatClient2 object in order to send messages.

17. MultiPlayerSetupScreen


MultiPlayerSetupScreen
-playerNum : int
-clue : Clue
-totalPlayers : int
-startTurn : int
-cc1 : ChatClient1
-cc2 : ChaTClient2
-gameClient1 : GameClient1
-gameClient2 : GameClient2
-ipScreen : IPScreen
-timeCount : int
+createStyles()
+updateGameMessages(final String gmessage, final String style0)
+setActivePlayerNumber(int num)
+updatePlayerLeft(int pn)
+updatePlayerBoot(int pn)
+updateMPError(int type, int num)
+startCountdown(int plrs, int stturn)
+addToChat(String chatString)
+updateSetupScreen(String instruction)

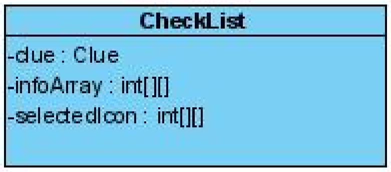
The MultiPlayerSetup screen is one of the main GUI classes in the game; hence why the class name has screen in it. It is the screen that is brought up after connecting to a MultiPlayerGame. Aside from the MPSetupScreen class it is the next screen that will display after the main screen when choosing a multiplayer game. It is similar to the SinglePlayerSetupScreen class in terms of the user interface. There are some slight differences but it is essentially the same. The big difference is the connection part of the class. The SinglePlayerSetupScreen allows direct GUI manipulation. The MultiPlayerSetupScreen class GUI is only updated by messages. When a player chooses a character a message is sent to the server via the GameClient2 class and echoed back to all players. Then the message received by the GameClient1 class and is passed back to the MultiPlayerSetupScreen class for processing. Finally the GUI is updated accordingly based on the message that was received. This way a network error will be quickly detected because a player would not be able to update his/her own screen. The MultiPlayerSetupScreen class also has instances of the ChatClient classes for chat message that work the same way game messages do. The class also has some special methods to handle certain methods. One of these methods is the updatePlayerBoot() method. This method handles the message of when a player is booted from the game. The method needs to do several things. The method must inform all the players who was booted and update the GUI accordingly. The method also must boot the specified player from the game by terminating the connections for that player and sending that player back to the main screen. Other methods listed also have additional processing involved after being called based on the message received.

18. GameBoard



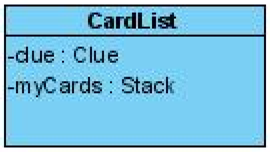
GameBoard is an abstract class that is extended by both the SinglePlayerGameBoard class and the MultiPlayerGameBoard class. Since both the SinglePlayerGameBoard and MultiPlayerGameBoard classes had so much in common the GameBoard class was created. The boards for a single player game and a multiplayer game are the same; therefore the GameBoard class can handle both. The GameBoard class sets up the initial array to display the game board. It also sets up the text styles to be used by the game messages that are displayed on the screen. Other GUI components are also initialized from the GameBoard class. These include the Die class, CardList class, CheckList class, and the WinningCards class. The player names and their characters are also setup in the GameBoard class.

19. CheckList



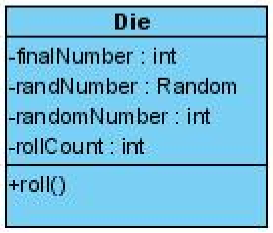
The CheckList class is basically a GUI of the checklist for the game. It contains the array infoArray[][] that contains the list information for what was marked. It also contains the selectedIcon variable for which symbol to mark the checklist with. It functions as a separate from the game so both can be visible at once. The checklist can be minimized without being closed so it can be open while the game runs.

20. CardList



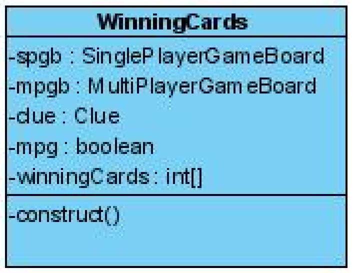
The CardList class is the class for the GUI that displays what cards a player has. It contains a stack of the player's cards. The images of the cards are displayed on a 3x3 grid on the screen. Likewise the names of the card are displayed underneath all the images. The CardList GUI also runs as a separate window so it can be open and minimized while the game runs. Typically after the checklist is marked there is no need for the CardList screen.

21. Die



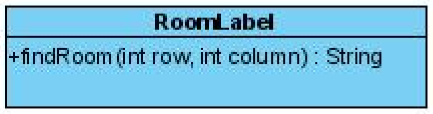
The Die class is a simple one. It calculates a random number for the die value. The number is displayed after a series of random images of other values to simulate the die actually rolling. The SinglePlayerGameBoard or the MultiPlayeGameBoard classes will call this classes' roll() method whenever the die is rolled. The roll() method will also show the screen when it starts and hide the screen when the die animation is finished. The finalNumber variable will hold the value of the current die roll. The rollCount variable will count the number of die images displayed. Once the counter is has reached ten the final die image is displayed. This means that die images cycle through ten times before stopping on the final value. The images shown before the final value are random numbers to simulate the roll.

22. WinningCards



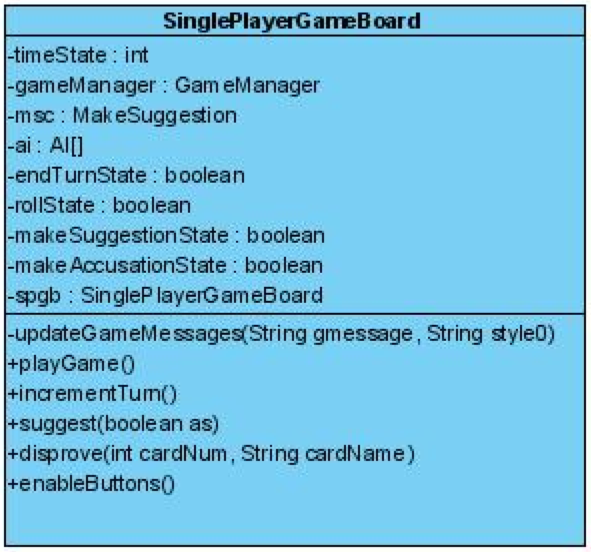
The WinningCards class is also another GUI class used for displaying the winning cards. This class is created at the beginning of the game when the winning cards are initialized, although the GUI is not shown until an accusation is made at the end of the game. It contains an instance of either the SinglePlayerGameBoard class or the MultiPlayerGameBoard class depending on whether or not the game is single player or mutlplayer. The screen displays more options in a single player game such as letting the computer continue, exit to the main screen, or restart the game. In a multiplayer game these options are not available directly from the screen because there are multiple players that could be affected. A multiplayer game still has these options available but only from the hosts menu bar. The winning cards are stored in the winningCards[] array and images of the winning cards are displayed on the screen.

23. RoomLabel



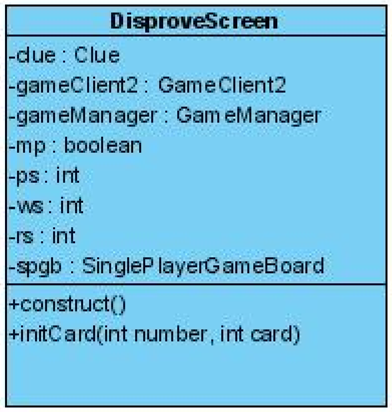
The RoomLabel class is a class that displays the label of the room selected. Since the GameBoard may not properly display the room names for all display settings, the room label can be brought up to clearly display the room. Upon right-clicking on the game board the method findRoom() will be called for the RoomLabel class. If the spot clicked on the board was a room the RoomLabel class popup window will display the room name near where the player clicked.

24. SinglePlayerGameBoard


SinglePlayerGameBoard
-timeState : int
-gameManager : GameManager
-msc : MakeSuggestion
-ai : AI[]
-endTurnState : boolean
-rollState : boolean
-makeSuggestionState : boolean
-makeAccusationState : boolean
-spgb : SinglePlayerGameBoard
-updateGameMessages(String gmessage, String style0)
+playGame()
+incrementTurn()
+suggest(boolean as)
+disprove(int cardNum, String cardName)
+enableButtons()

The SinglePlayerGameBoard class extends from the GameBoard class. It inherits all the methods and variables from the GameBoard class. In addition it has some of its own class instances. These include an instance of the AI class, MakeSuggestion class, GameManager class, and an instance of itself. The instance of itself is used for passing to other classes that may need to communicate back with itself. The AI class is an array of the AI players used for the computer turns. The GameManager class holds the information related to the game such as the board positions and methods to calculate movement to display to the player. The SinglePlayerGameBoard class keeps the state machine for the game. This is done for the entire game and an individual player's turn. The turn state is kept with Boolean variables rollState, endTurnState, makeSuggestionState, and makeAccusationState. These four states represent whether or not the respective buttons will be enabled for the player during the player's turn. The enableButtons() method is called, which goes through all the states to enable or disable the buttons. The state machine for the entire game is done with the help of the timeState variable. The timeState variable keeps the state for anyone's turn. When timer events occur the timeState variable determines what happens. The variable is set to different numbers throughout the game to keep the game running properly. In addition there are a few other methods this class in order to play the game. This includes playGame(), incrementTurn(), suggest(), and disprove(). The playGame() method is the start of a player's turn, which initializes what buttons should be enabled for the player. It also handles the AI players and their turns by starting the timer. The other methods do as they are named. The suggest() method is used to make a suggestion and the disprove() method is used to disprove a suggestion. All AI player data is updated accordingly to the information gained.

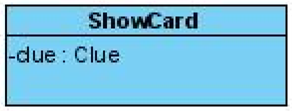
25. DisproveScreen



DisproveScreen
-clue : Clue -gameClient2 : GameClient2 -gameManager : GameManager -mp : boolean -ps : int -ws : int -rs : int -spgb : SinglePlayerGameBoard
+construct() +initCard(int number, int card)

The DisproveScreen class is for disproving suggestions. Players will see this class GUI when it is their turn to disprove a suggestion. The three integers named ps, ws, and rs hold the value of the person, weapon, and room card to disprove, which are retrieved from the GameManager class. If the player has any of those cards the image of the card will appear on the GUI and will be selectable by the player. The method initCard() will display all the cards that a player can disprove with on the GUI. Depending on whether the game is single player or multiplayer the class must hold objects of each class. If the game is single player then the DisproveScreen class will hold a SingelPlayerGameBoard object, otherwise it will hold a GameClient2 object. This is because the disprove mechanics work differently for single player and multiplayer games. In a multiplayer game the card chosen to disprove with is sent as a message to the GameClient2 class, which will send it to the server. In a single player game the card to disprove with can directly call the SinglePlayerGameBoard disprove() method. In addition the Boolean variable mp determines whether or not the game is single player or multiplayer. This allows the correct path of the code to function. Once a card is chosen the DisproveScreen GUI will disappear and the class disposes of itself.

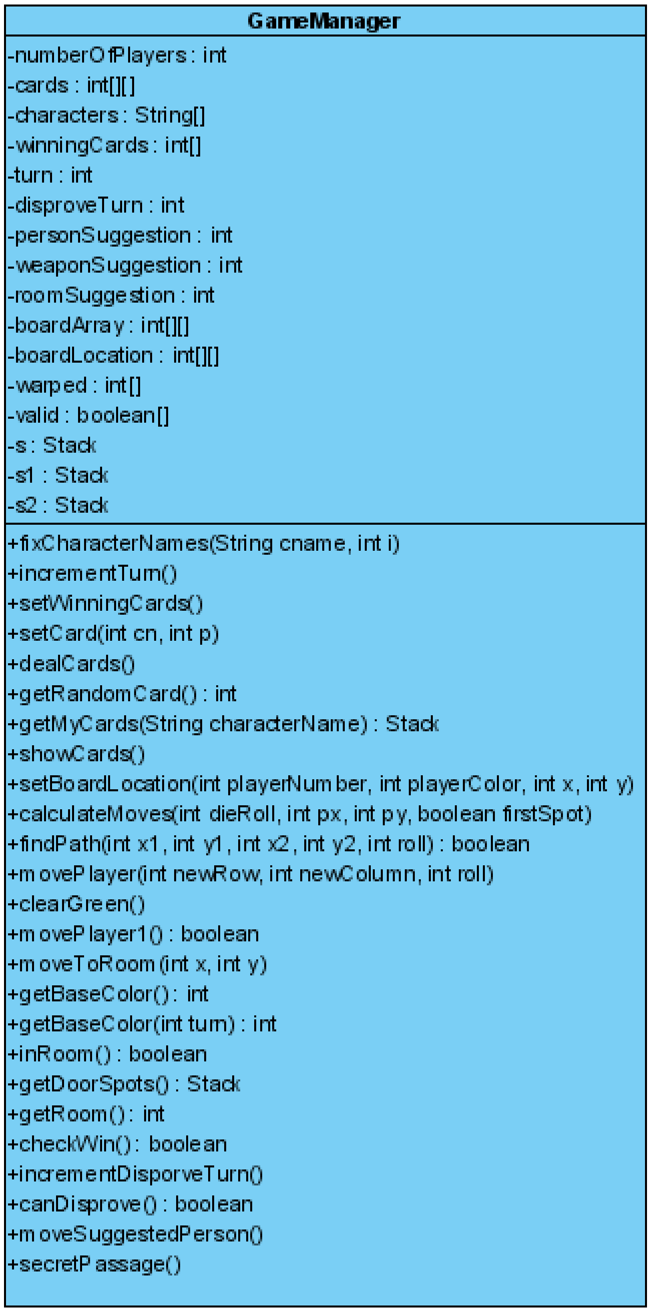
26. ShowCard



ShowCard
-clue : Clue

The ShowCard class is the class that displays the card that was chosen to disprove the suggestion to the player. It basically just displays an image of the card until the player closes the window.

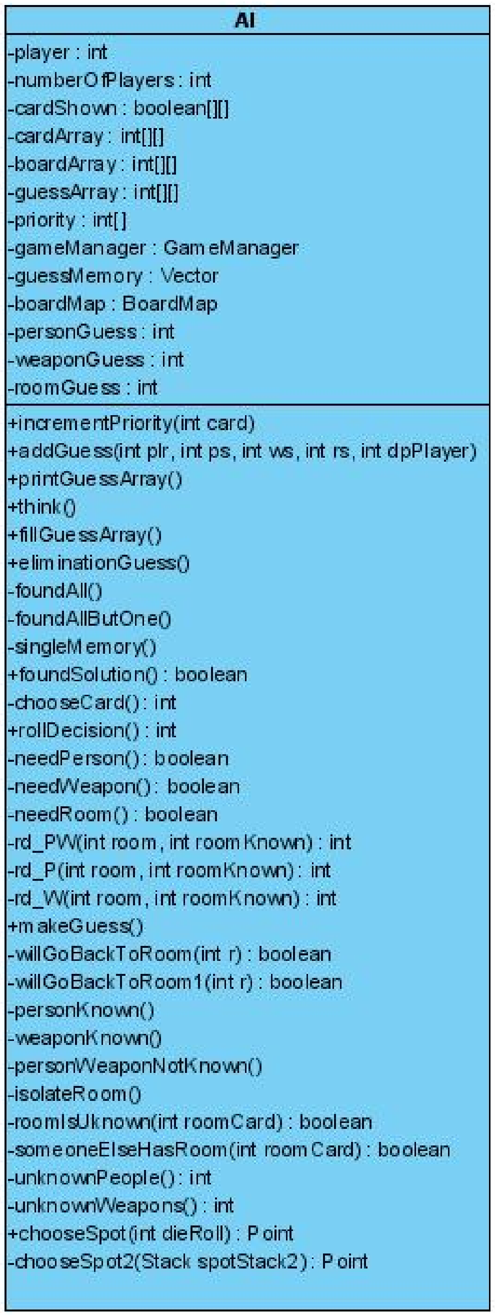
27. GameManager



```
GameManager
-numberOfPlayers : int
-cards : int[][]
-characters : String[]
-winningCards : int[]
-turn : int
-disproveTurn : int
-personSuggestion : int
-weaponSuggestion : int
-roomSuggestion : int
-boardArray : int[][]
-boardLocation : int[][]
-warped : int[]
-valid : boolean[]
-s : Stack
-s1 : Stack
-s2 : Stack
+fixCharacterNames(String cname, int i)
+incrementTurn()
+setWinningCards()
+setCard(int cn, int p)
+dealCards()
+getRandomCard() : int
+getMyCards(String characterName) : Stack
+showCards()
+setBoardLocation(int playerNumber, int playerColor, int x, int y)
+calculateMoves(int dieRoll, int px, int py, boolean firstSpot)
+findPath(int x1, int y1, int x2, int y2, int roll) : boolean
+movePlayer(int newRow, int newColumn, int roll)
+clearGreen()
+movePlayer1() : boolean
+moveToRoom(int x, int y)
+getBaseColor() : int
+getBaseColor(int turn) : int
+inRoom() : boolean
+getDoorSpots() : Stack
+getRoom() : int
+checkWin() : boolean
+incrementDisporveTurn()
+canDisprove() : boolean
+moveSuggestedPerson()
+secretPassage()
```

The GameManager class holds all the information related to playing the game. This includes all positions of all the players on the board and whose turn it is. The GameManager class also has a second function, which is to manage the GameBoard. The GameManager has methods such as calculateMoves() and findPath(), which will calculate all the possible moves a player has based on the die roll and find the path to the selected spot respectively. Stacks s, s1, and s2 are used in these calculations. The GameManager also keeps track of all the cards in the game as well as the current suggestion. The GameManager class keeps track of whose turn it is, whose disprove turn it is, and whether or not a player is valid or has been called into the room. All the private data members reflect all the data that is stored. Several of the functions deal with the movement of the character pieces in the game, such as secretPassage(), moveSuggestedPerson(), and movePlayer(). Other helping methods include getBaseColor(), and getDoorSpots(). All the data and methods combined accomplish all the tasks in managing tall of the game data. The board locations and movement, cards, turn information, player information, and current suggestion are all managed in the GameManager class.

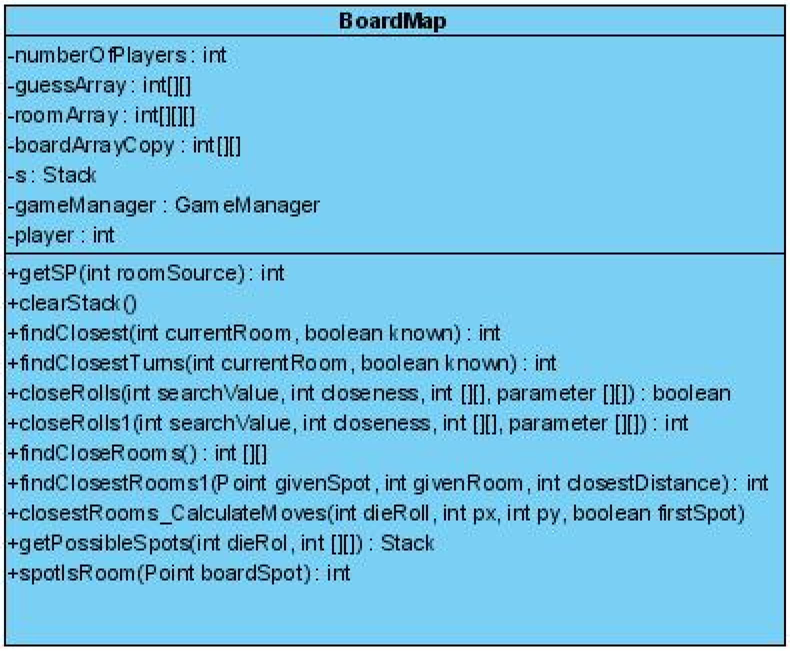
28. AI



```
AI
-player : int
-numberOfPlayers : int
-cardShown : boolean[][]
-cardArray : int[][]
-boardArray : int[][]
-guessArray : int[][]
-priority : int[]
-gameManager : GameManager
-guessMemory : Vector
-boardMap : BoardMap
-personGuess : int
-weaponGuess : int
-roomGuess : int
+incrementPriority(int card)
+addGuess(int plr, int ps, int ws, int rs, int dpPlayer)
+printGuessArray()
+think()
+fillGuessArray()
+eliminationGuess()
-foundAll()
-foundAllButOne()
-singleMemory()
+foundSolution() : boolean
-chooseCard() : int
+rollDecision() : int
-needPerson() : boolean
-needWeapon() : boolean
-needRoom() : boolean
-rd_PW(int room, int roomKnown) : int
-rd_P(int room, int roomKnown) : int
-rd_W(int room, int roomKnown) : int
+makeGuess()
-willGoBackToRoom(int r) : boolean
-willGoBackToRoom1(int r) : boolean
-personKnown()
-weaponKnown()
-personWeaponNotKnown()
-isolateRoom()
-roomIsUknown(int roomCard) : boolean
-someoneElseHasRoom(int roomCard) : boolean
-unknownPeople() : int
-unknownWeapons() : int
+chooseSpot(int dieRoll) : Point
-chooseSpot2(Stack spotStack2) : Point
```

The AI class is used by the SinglePlayerGameBoard and MultiPlayerGameBoard classes to interact with the AI players. All the important functions and variables are described in the AI section. When appropriate, the SinglePlayerGameBoard and MultiPlayerGameBoard classes will call the correct methods of the AI class. The method called is dependent on the state of the game. Once the AI class returns to the calling class the game will continue and handle the value returned to it by the AI class.

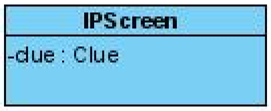
29. BoardMap



```
BoardMap
-numberOfPlayers : int
-guessArray : int[][]
-roomArray : int[][][]
-boardArrayCopy : int[][]
-s : Stack
-gameManager : GameManager
-player : int
+getSP(int roomSource) : int
+clearStack()
+findClosest(int currentRoom, boolean known) : int
+findClosestTurns(int currentRoom, boolean known) : int
+closeRolls(int searchValue, int closeness, int [][], parameter [][]) : boolean
+closeRolls1(int searchValue, int closeness, int [][], parameter [][]) : int
+findCloseRooms() : int [][]
+findClosestRooms1(Point givenSpot, int givenRoom, int closestDistance) : int
+closestRooms_CalculateMoves(int dieRoll, int px, int py, boolean firstSpot)
+getPossibleSpots(int dieRol, int [][]) : Stack
+spotIsRoom(Point boardSpot) : int
```

The BoardMap class has the purpose of helping the AI class when it comes to movement decisions. The AI class was not powerful enough on its own to calculate movement decisions. For this reason the BoardMasp class was created. The AI class deals with all the card information where the BoardMap class handles all the board location information. The guessArray[][] is passed from the AI class to the BoardMap class because the movement decisions also involve the array of card information. Also a copy of the actual board information is held in boardArrayCopy[][][]. This is because all the thinking the AI does should not actually change the actual board information. Therefore a copy of the information must be passed along to the BoardMap class every time. The BoardMap class has methods which calculate closest rooms from the current board location by roll and by room. An array called roomArray[][][] stores all the distance from one room to another in the form of turns to get there. The BoardMap class also provides methods to find the closest distance from an arbitrary spot to a desired room by roll. Both the methods to find the distance from a spot to a room and to find the closest room to another room can be used together by the AI class to find more complicated and long distance paths to desired rooms. Important BoardMap algorithms are explained in the Other Algorithm's section of the game. An overview of how the AI class uses the BoardMap functions is explained in the AI section on movement.

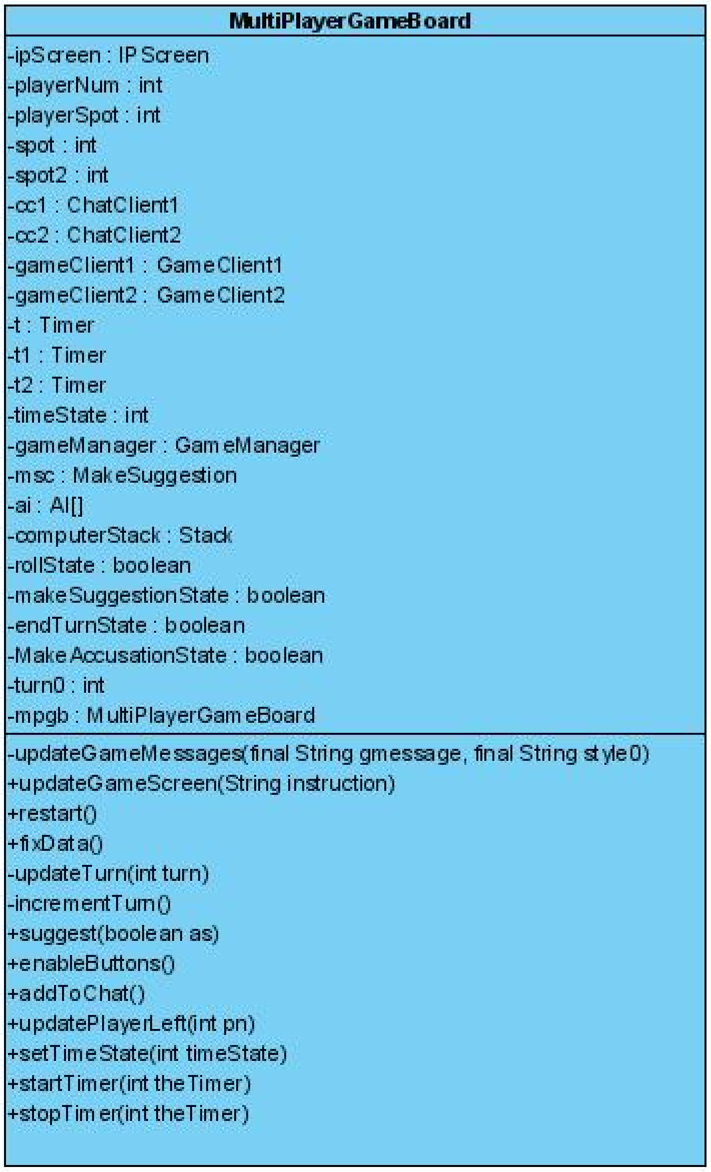
30. IPScreen



```
IPScreen
-clue : Clue
```

The IPScreen class is the class that displays the host's IP addresses. This class screen is accessible from both the MultiPlayerSetupScreen class and the MultiPlayerGameBoard class as a popup window.

31. MultiPlayerGameBoard



```
MultiPlayerGameBoard
-ipScreen : IPScreen
-playerNum : int
-playerSpot : int
-spot : int
-spot2 : int
-cc1 : ChatClient1
-cc2 : ChatClient2
-gameClient1 : GameClient1
-gameClient2 : GameClient2
-t : Timer
-t1 : Timer
-t2 : Timer
-timeState : int
-gameManager : GameManager
-msc : MakeSuggestion
-ai : AI[]
-computerStack : Stack
-rollState : boolean
-makeSuggestionState : boolean
-endTurnState : boolean
-MakeAccusationState : boolean
-turn0 : int
-mpgb : MultiPlayerGameBoard
-updateGameMessages(final String gmessage, final String style0)
+updateGameScreen(String instruction)
+restart()
+fixData()
-updateTurn(int turn)
-incrementTurn()
+suggest(boolean as)
+enableButtons()
+addToChat()
+updatePlayerLeft(int pn)
+setTimeState(int timeState)
+startTimer(int theTimer)
+stopTimer(int theTimer)
```

The MultiPlayerGameBoard class is a GUI class for the multiplayer game. After the countdown from the MultiplayerSetupScreen class has completed the MultiplayerSetup screen class GUI is disposed and the MultiPlayerGameBoard class is created. The MultiPlayerGameBoard has everything the SinglePlayerGameBoard class has. The major addition is that everything is run by messages and the connection process. Messages for nearly every game action are sent to the server and echoes back to all the clients, which then process the message. The fixData() method fixes all the card data upon the start of the game, since each player deals their won cards. All the cards must be the same so player 1 sends his data to all the other players who then correct their data. This involves the GameManager class to fix the card data. The methods setTimeState(), startTimer(), and stopTimer() handle the timers in the game.

Unlike a single player game the multiplayer game can have inputs from two sources. One source is the individual player’s timer events, which can also generate more timer events. Another source is the messages that the player can receive from the server. In order to keep synchronization the timers must only be accessed by one source at a time. Only one timer should be active at a time and the timeState variable should not change while a timer is active. The methods setTimeState(), startTimer(), and stopTimer() are all synchronized to ensure the time state is correct in almost all cases. The MultiPlayerGameBoard class essentially combines the client-server functionality with the SinglePlayerGameBoard functionality. The main difference is that the state machine is done with messages as well as timers, where a single player game only has timers.

2.3: Full Class Diagram

The full class diagram is shown below. Lines were not drawn from several classes to the PlaySound class to make the diagram simpler. The only line drawn is from the Clue class, which has a PlaySound object for the soundtrack. Several other classes have access to create new PlaySound objects, but they are only created anonymously for a sound to play one time. The Rules class was also not included. The Rules class is also created anonymously from several classes with a GUI. To make the diagram less messy the Rules class was not included.

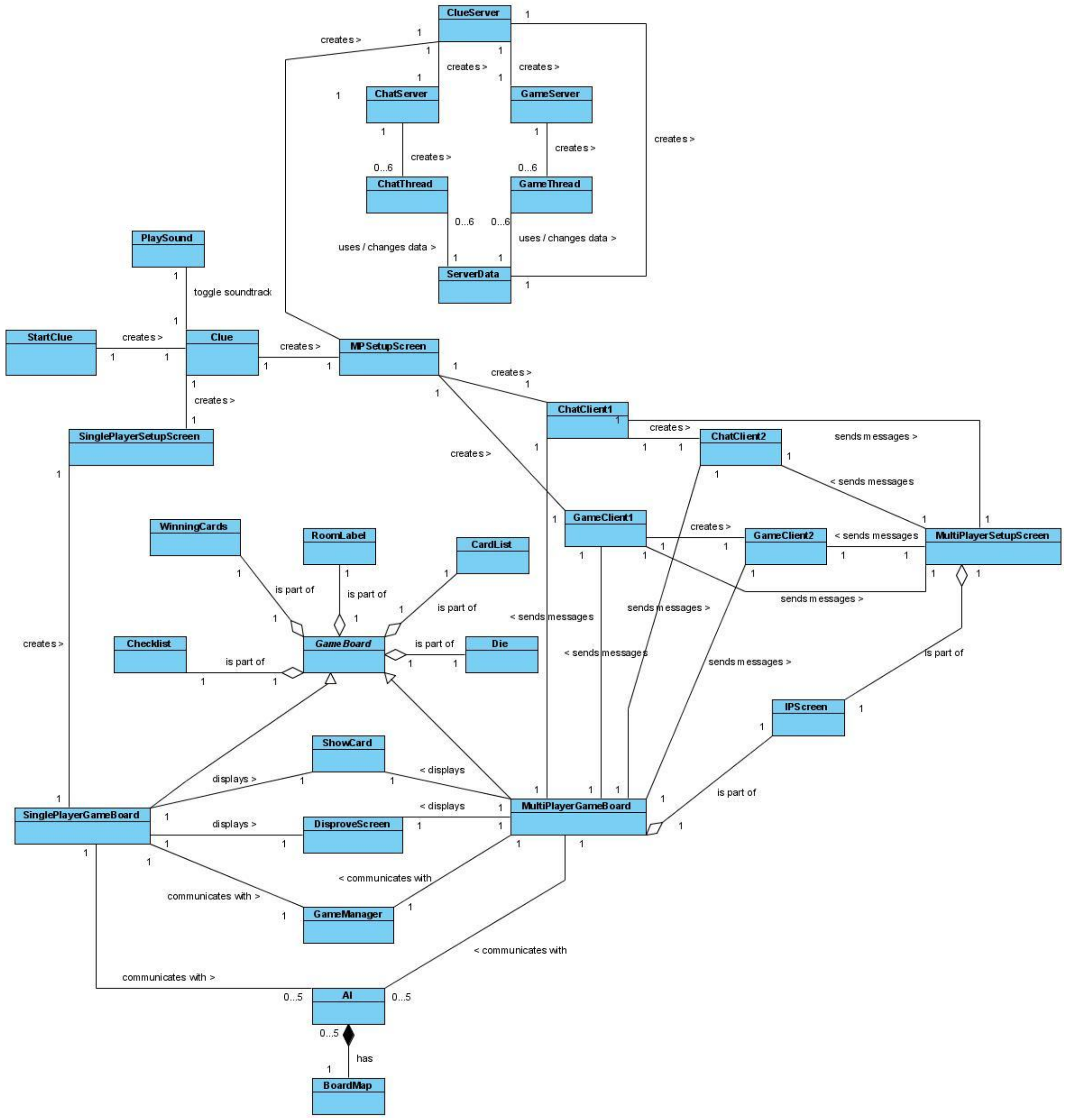


Diagram 2.3.1: Full Class Diagram

2.4: Packages

2.4.1: Introduction

The game can be broken up into five packages. The actual code was not done with packages for simplicity purposes. Using packages makes the documentation easier to see but makes the program harder to code. When using packages in java, all the class files need to be in the appropriate class folder.

This makes the program structure much more complicated. It also makes the game more inconvenient to compile. Therefore the actual program does not use packages, but they are included in the documentation if they were to actually exist.

1. Server Package

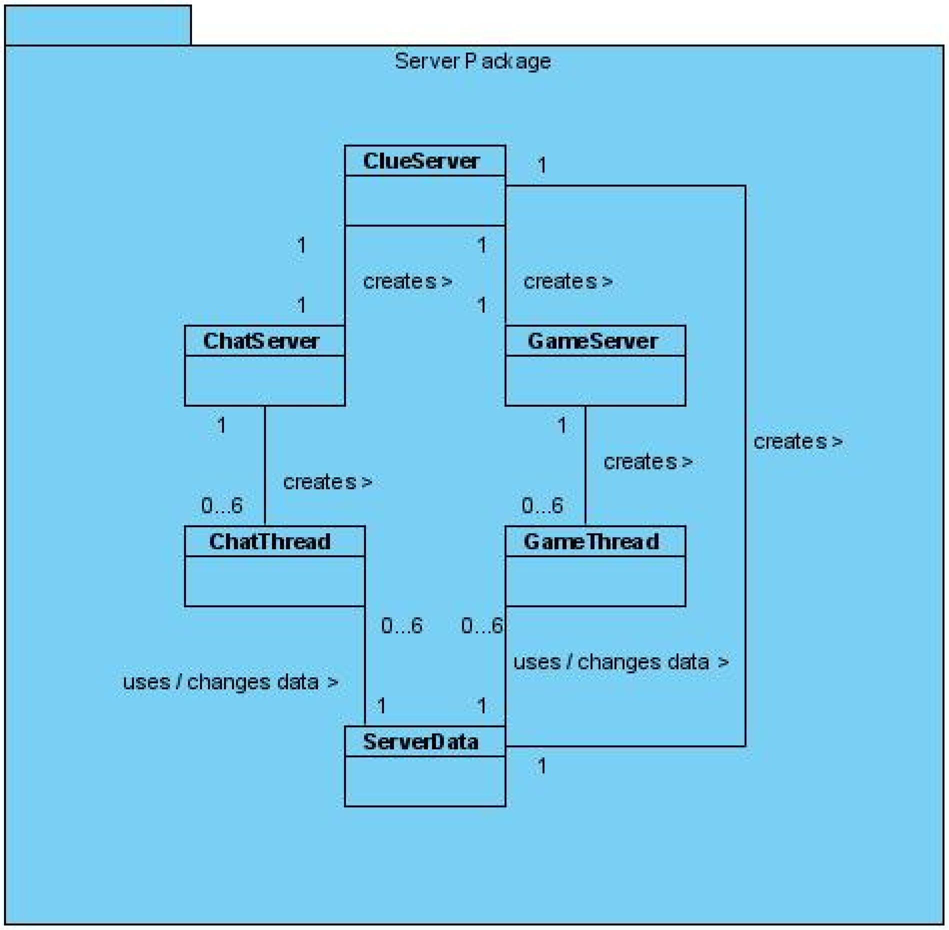


Diagram 2.4.2: Server Package

The Server package is made up of all the classes that make up the server. This includes the ClueServer as the main starting point for the server classes. The ClueServer creates both a GameServer and a ChatServer class that each has their own thread. All threads have access to the ServerData class that stores all the data for the server.

2. Clue Startup Package

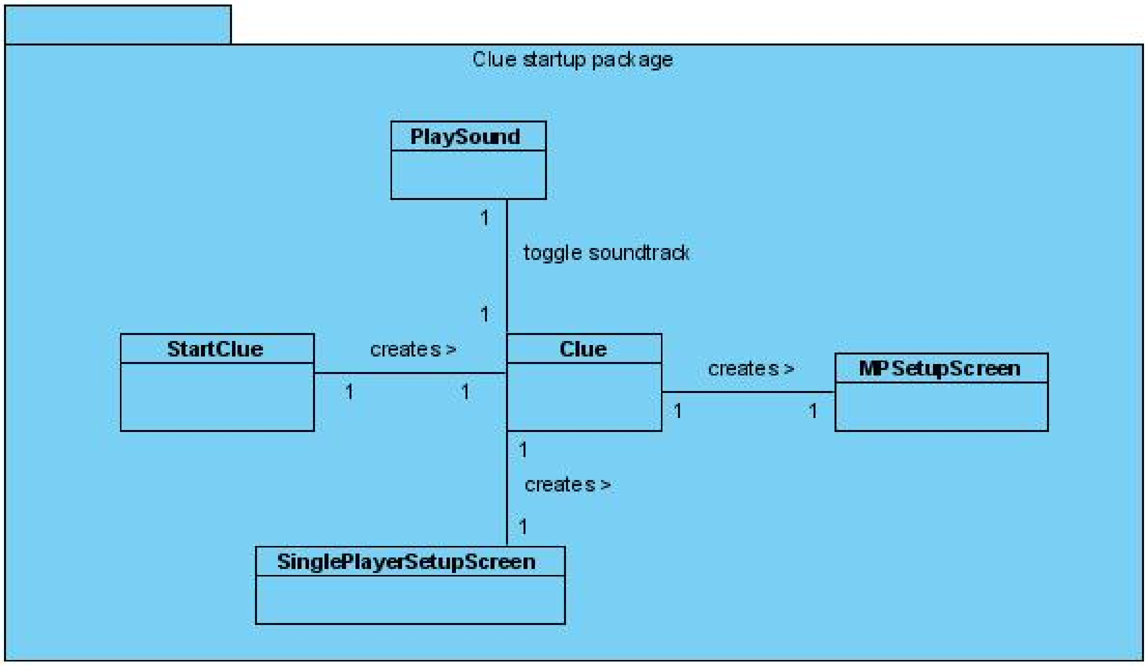


Diagram 2.4.3: Clue Startup Package

The Clue Startup package consists of all the classes that are initially created when the program starts. StartClue is where the main entry point to the program and creates the Clue class. The Clue class is where the main screen is initialized and displayed. Classes that the Clue class directly communicates with are also included in the startup package.

3. AI Package

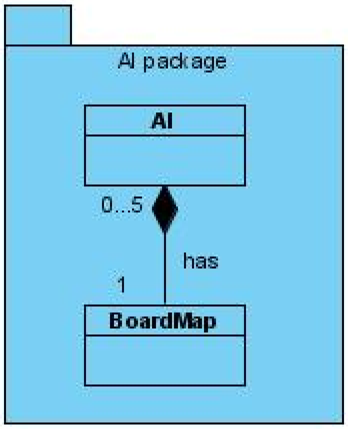


Diagram 2.4.4: AI Package

The AI package is simple because the AI is essentially separate from the rest of the game. The AI package includes the AI class where most of the AI processing is done. In addition the AI package includes the BoardMap class, which works closely with the AI class for movement decisions. There can be up to five instances of the AI class; one for each computer player.

4. Client Package

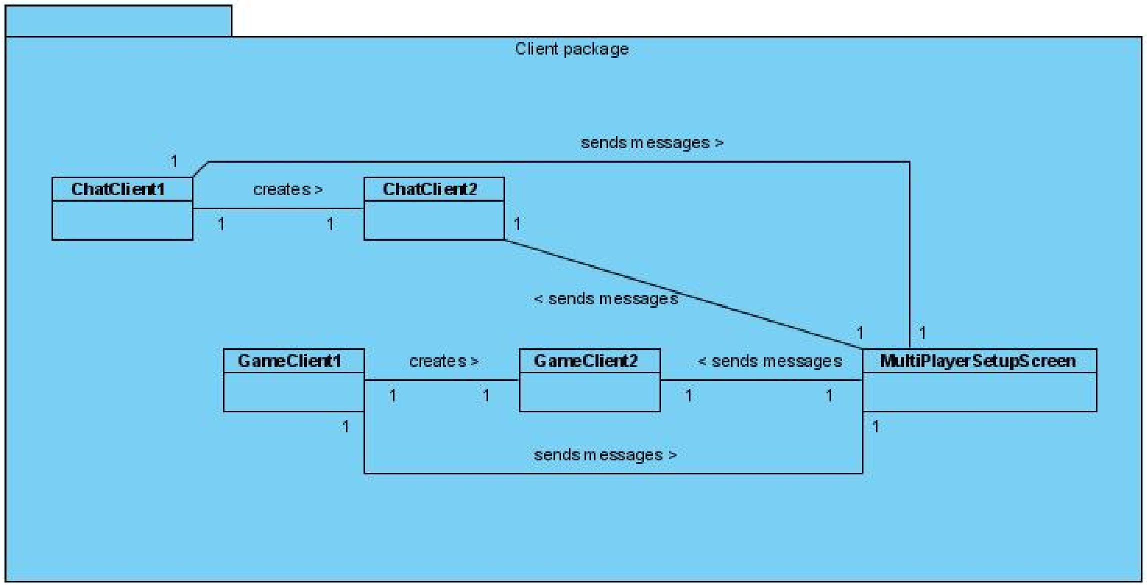


Diagram 2.4.5: Client Package

The Client package is basically the other side of the Server package. The client package has both the chat client classes and the game client classes. In addition it has the MultiPlayerSetupScreen due to its close relationship with the client classes. The Client package basically handles the client side of the network message passing.

5. Board Package

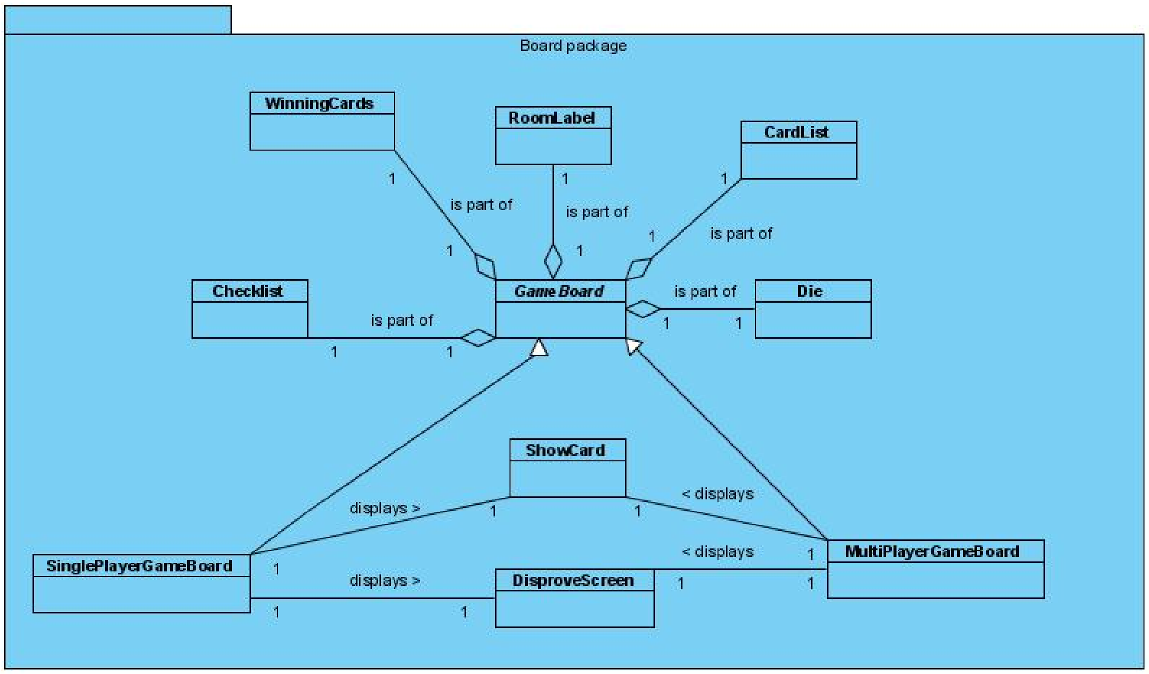


Diagram 2.4.6: Board Package

The Board package consists of all the classes that make up the game board. This includes all the GUI classes as well as the SinglePlayerGameBoard class and the MultiPlayerGameBoard class. All the classes in the board package have some type of graphic component associated with them, since they all display information related to the game.

3: Network Architecture

3.1: Introduction

The network design for the game uses the star topology. The host of the game creates the server, which all other players connect two. All players including the hosting player are clients. All messages sent from the clients will go to the server for processing. In many cases the server will echo the message back by going through a loop of connected IP addresses. The clients then will process the message. Even the host player must go through the message protocol. For example, if the player hosting the game clicks on the roll button it will not do anything until the message is sent and received by that player. Even though the hosting player does not need to use the protocol because the game client and server are on the same computer it does anyway. This allows for some error detection for the connection because the single message will be sent to all connected players. If an error occurs it will be recognized and handled in most cases.

3.2: Threads

In addition to the basic star topology the server and clients are broken down into several threads. The two main threads are the game thread and the chat thread. The game thread is for game messages and the chat thread is used for chat messages. When a client connects to the host a game thread and chat thread are created on both the client and server side. These two threads are broken up into two more threads each on the client side; one for input and another for output. All relevant data on the server side is stored in a class called ServerData. This class keeps the information on who is connected to the server. By using the ServerData class, each thread on the servers can send messages to all other clients by using the IP addresses saves in the ServerData class. The following diagram shows what the network architecture looks like. The dark blue lines show all inbound streams to the chat server and the light blue lines show all outbound streams from the chat server. Likewise the dark green lines show all the inbound streams to the game server and the light green lines show all outbound streams from the game server.

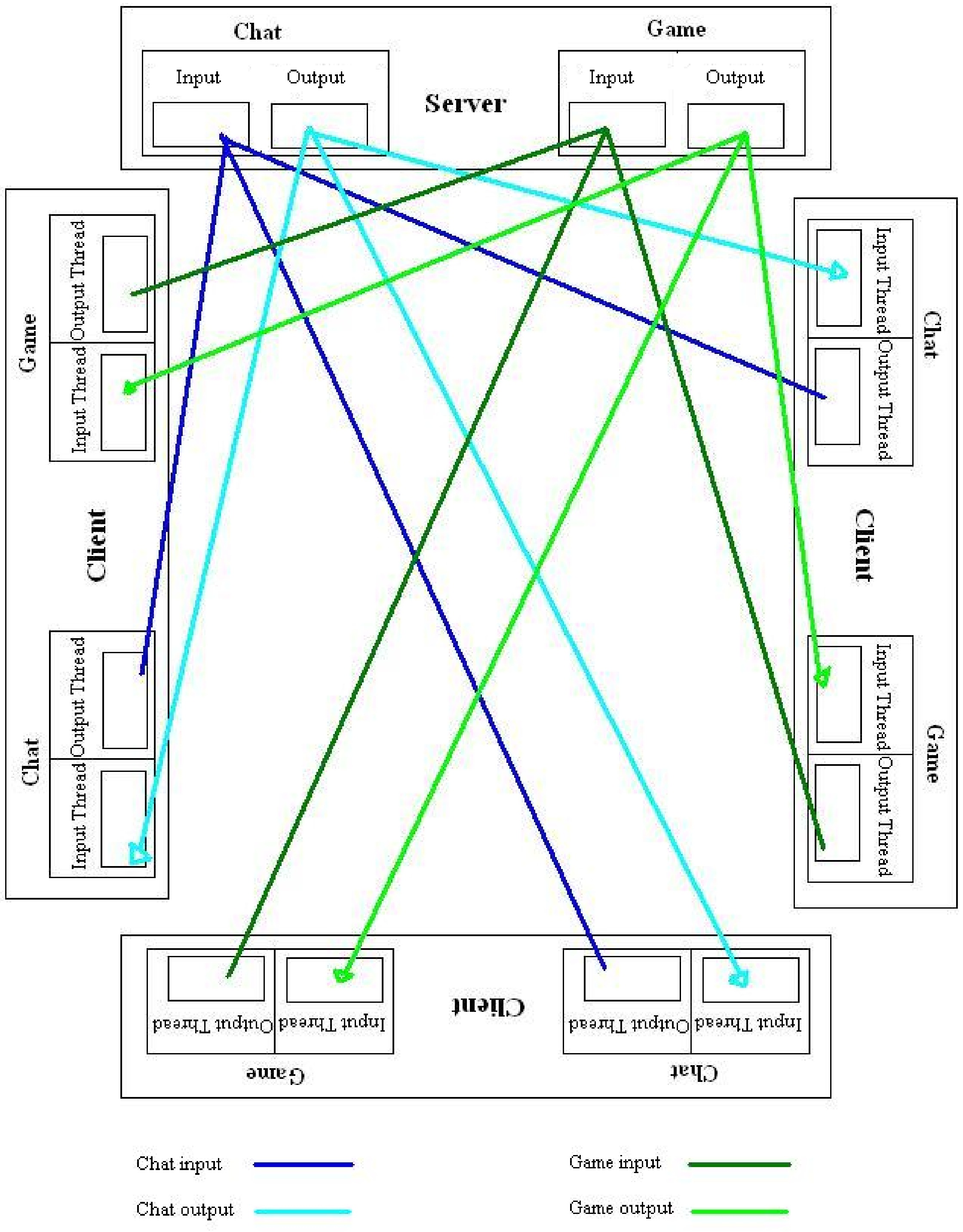


Figure 3.2.1: Network Architecture

The chat was designed to be very simple. It is so simple that if a player leaves the chat threads are unaware of it. This is because the player numbers are assigned through the game thread and the chat thread and game thread may not share the same position in their respective IP address arrays on the server side. No lookup methods were created. Instead the solution to this was to catch any errors in the chat output stream. An error here would mean that it could not deliver a message to that client. When the error occurs, the array spot of the IP address is known and can be removed from the IP address array for chat. Basically all the chat thread on the server does is receive messages and echo them back to all the other players. The chat threads on the client side are simple as well. The thread with the outbound stream just takes a message passed as a parameter and sends it to the server. The threads with the inbound stream take the message from the server and pass it to the method to update chat messages. This method is either in the setup screen or the game screen depending on the state of the game.

The game threads on the client side do exactly the same thing as the chat threads with the game messages. The game thread on the server side is also similar to the chat thread on the server side. The difference is that it only accepts several predefined messages. It does some type of processing with these messages if it needs to and it also sends messages back to one or all clients based on the processing done.

3.3: Message Protocol

Both the game client and server have predefined messages serving as the communication protocol. A list of all the messages is shown below with their parameters and a brief description of what each one will do. The chat thread also has one predefined message with the purpose of closing the chat connection.

Table 3.3.1: Server Game Thread Accepted Messages

Message	Parameters	Description
$$CloseGame$$	None	Closes the connection
AssignPlayerNumber	None	Finds the next available player number; Sends the number to the client
Leave	Player number	Updates server data information; Sends a message that the player has left; Closes the connection
Boot	Player number	Updates server data information; Sends a message that the player has been booted
Ready	Player number	Sends a message that the player is ready; Starts the game countdown if all players are ready
characterbox	Player number; Character box number	Updates the server data; Echoes the message back to all players
playerbox	Player number; Player box number	Toggles the player box data; Echoes the message back to all players
playername	Player number; Player name	Sets the player name; Echoes the name back to all players
playerleft	Player number	Updates server data information; Sends a message that the player has left; Closes the connection
hostIP	None	Sends the host's IP addresses to the player
sendstate	None	Sends the state of the game to the player
StartGame	None	Sets the game started state to true; Send the message that the game has started
<Unknown message>	None	Echoes the message back to all players

Table 3.3.2: Multiplayer Setup Screen Accepted Messages

Message	Parameters	Description
playerbox	Player number; Player box number	Makes GUI visible for the box
characterbox	Player number; Character box number	Updates the character box
playername	Player name; Player number	Sets the respective player name
PlayerNumber	Player number	Enables respective GUI
xPlayerNumber	Player number	Sets the player number
nospots		Send message to chat server to close; Send message to game server to close; Notify the player there are no spots; Close setup screen; Bring up main screen
Ready	Player number	Sets the player ready status
internalIP	IP address	Sets the IP screen's address field
externalIP	IP address	Sets the IP screen's address field
playerleft	Player number	Updates GUI; Closes the chat for the leaving player; Closes the game for the leaving player; Closes the setup screen for the player; Brings up the main screen for the player
playerboot	Player number	Updates GUI; Closes the chat for the booted player; Closes the game for the booted player; Closes the setup screen for the player; Brings up the main screen for the player
mperror_SameCharacter	None	Notifies the player about the error
mperror_PlayersNumber	None	Notifies the player about the error
Startcountdown	Number of players; Starting turn	Starts the countdown; Sets the total players; Sets the starting turn
Game starting in 5 seconds	None	Notify players
StartGame	None	Create game board; Dispose setup screen
<Unknown message>	None	Nothing

Table 3.3.3: Multiplayer Game Board Accepted Messages

Message	Parameters	Description
characterbox	Character number	Sets the character for the player
playername	Player name	Sets the player name for the player
playerbox	Player type	Sets the player name based on the type
donesendingstate	None	Sends a message to fix the characters; Sends a message to fix the player names; Sends a message to fix the cards
fixcharacterNames	Character number; Character name	Sets the character for the spot number
fixPlayerNames	Player number; Player name	Sets the player name for the player
gamelookupspot	Player number; Lookup spot	Sets the array lookup spot for the player
fixcards	Card number; Owner	Sets the correct owner to the card
donefixingcards	None	Creates the winning cards class; Creates the mycards class; Creates the cardlist class
playerTurn	Turn spot	Sets the current turn; Updates players GUI (if necessary); Starts computer AI (if necessary)

Suggest	Person suggestion;
	Weapon suggestion;
	Room suggestion;
	Type
		Sets the person suggestion;
		Sets the weapon suggestion;
		Sets the room suggestion;
		Calls suggest function for more processing

warped	Player number; Warped value	Sets the warped value for the player
rolla	Die number	Rolls the die to the number (computer player)
roll	Die number	Rolls the die to the number (human player)
endturn	None	Increments the turn
secretpassage	None	Moves the player through the secret passage
moveplayer	Row; Column	Moves the player to the spot
showcard	Card number; Card name	Shows the card to the correct person; Continues processing
timestate	Time state	Sets the time state
disprove	Disprove player	Creates the disprove screen
playerleft	Player number	Notify players
valid	Turn number; Valid or invalid	Sets the turn to valid or invalid
restart	None	Restarts the game; Creates a new game board; Uses existing data from the setup screen; Resets all other data

makeguess	None	AI makes the guess (for computer); Sends the player’s suggestion (for human)
<Unknown message>	None	Nothing

By looking at all the message it is noticeable that the clients do most of the managing. There are much more messages that the clients handle than the server does. The server mostly just keeps connection information and the initial game state before the game starts. Everything else is managed by the clients. More specifically the hosting player’s client manages a lot of the game. All computer players are managed by the host’s client and the fixes to the data are also controlled by the first player (player 0). Player 0 is always the host of the game. Every other player manages his or her own turn and related data and decisions.

The following sequence diagram shows the message timing for making a suggestion. In this case there are three players in the game and player 2 is making the suggestion. Player 1 is disproving the suggestion. Only players 1 and 2 are shown as actors because player 3 does not supply input in this example. All three players have their respective game boards which interact with the server.

Player 2 clicks the suggest button on the suggestion screen to make the suggestion. This message is passed from the suggestion screen to the server. The server echoes back the message to all three players’ game boards. All three players’ game boards will cycle through the disprove turns while updating the game messages for all the players who cannot disprove the suggestion. When the cycle gets to player 1 who can disprove the suggestion player 1’s game board created a disprove screen. Player 1 then chooses a card to disprove with. The card is sent as a message to the server which then echoes back the card to all the other players. All the other players receive the message, but do not execute any game message updates yet. Each player must synchronize to ensure that the message is not executed before the player cycling has finished. The show card screen is created for player 2 to show player 2 the card that player 1 chose to disprove with. After player 2 acknowledges the card, the show card screen enables the correct buttons for player 2.

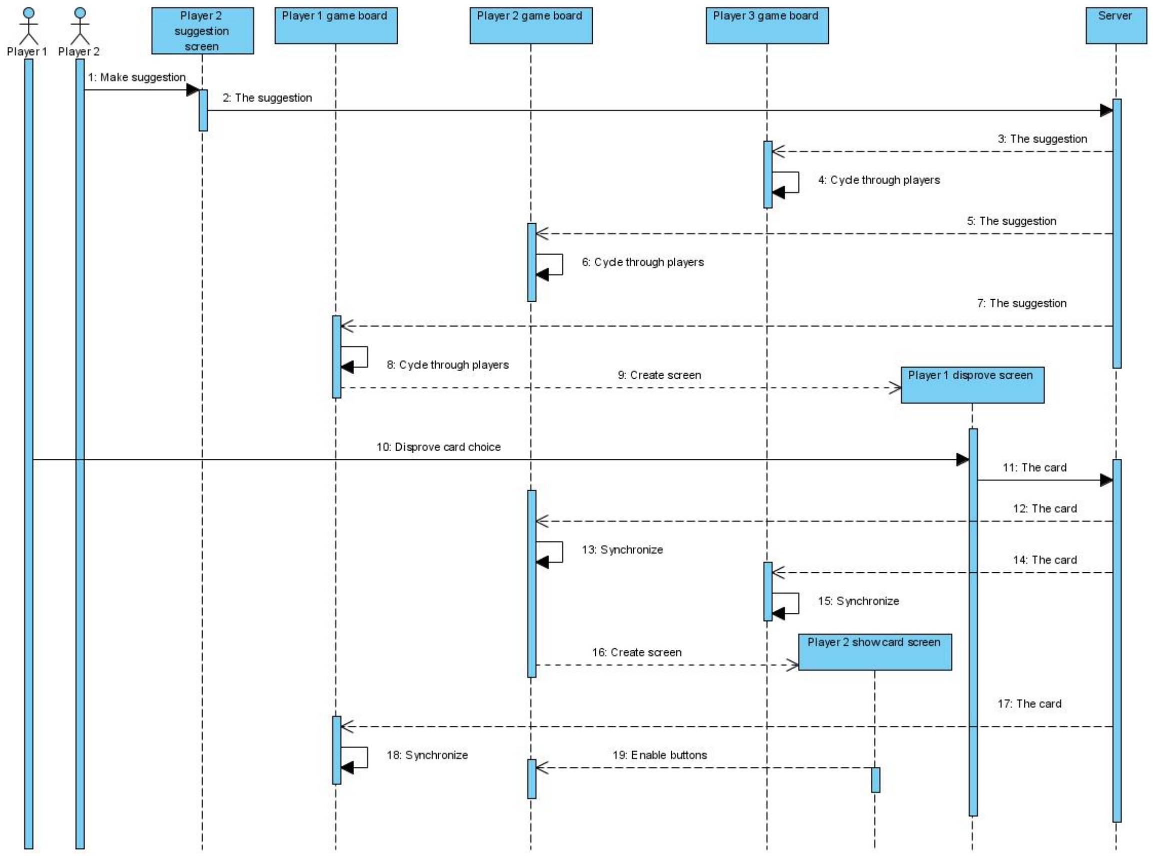


Figure 3.3.4: Sequence Diagram

4: Use Cases

4.1: Introduction

The Use Cases listed below contain all the options a player has to choose from during the game. All lines without numbers are assumed to be a 1-1 relationship. Any other relationships with different numbers will be shown. The main screens in the game are the main screen, setup screens, and game board screens. All other screens are not as important and may have their own use cases or be combined into one of the main use cases. The use of the include statement in the use cases indicates a less important screen being used by one of the important screens. These use cases that are included in a more important screen have their own separate use case to display their functionality to the user. The use cases are listed in order from the user's point of view from when they can access each screen and associated options.

4.2: 1. Main Screen System

When the user starts the program the main screen will be the first screen to show up. At the main screen the user will have the options to choose to start a single player game or a multiplayer game. The user will also have options to view the rules, turn the soundtrack on or off, and exit the game.

A. Single Player Game

If the user chooses a single player game the main screen will disappear and the single player setup screen will appear.

B. Multiplayer Game

If the user chooses a multiplayer game a dialog box will first appear on top of the main screen. The dialog box has some preliminary multiplayer setup options that the user must choose before starting a multiplayer game.

C. View Rules

If the user chooses to view the rules a dialog box with the game rules loaded from a file will display. The user can also close the rules after reading them or leave them open. This option will be available from every screen.

D. Close Rules

The close rules option will be available from the rules screen that is displayed. It is optional to close the rules as they can be left open throughout the game if desired.

E. Toggle Sound

Selecting or deselecting the sound will turn the game sounds on or off.
This option will be available for every screen.

F. Exit

The exit button will exit the program. There will be an exit menu option from every main screen.

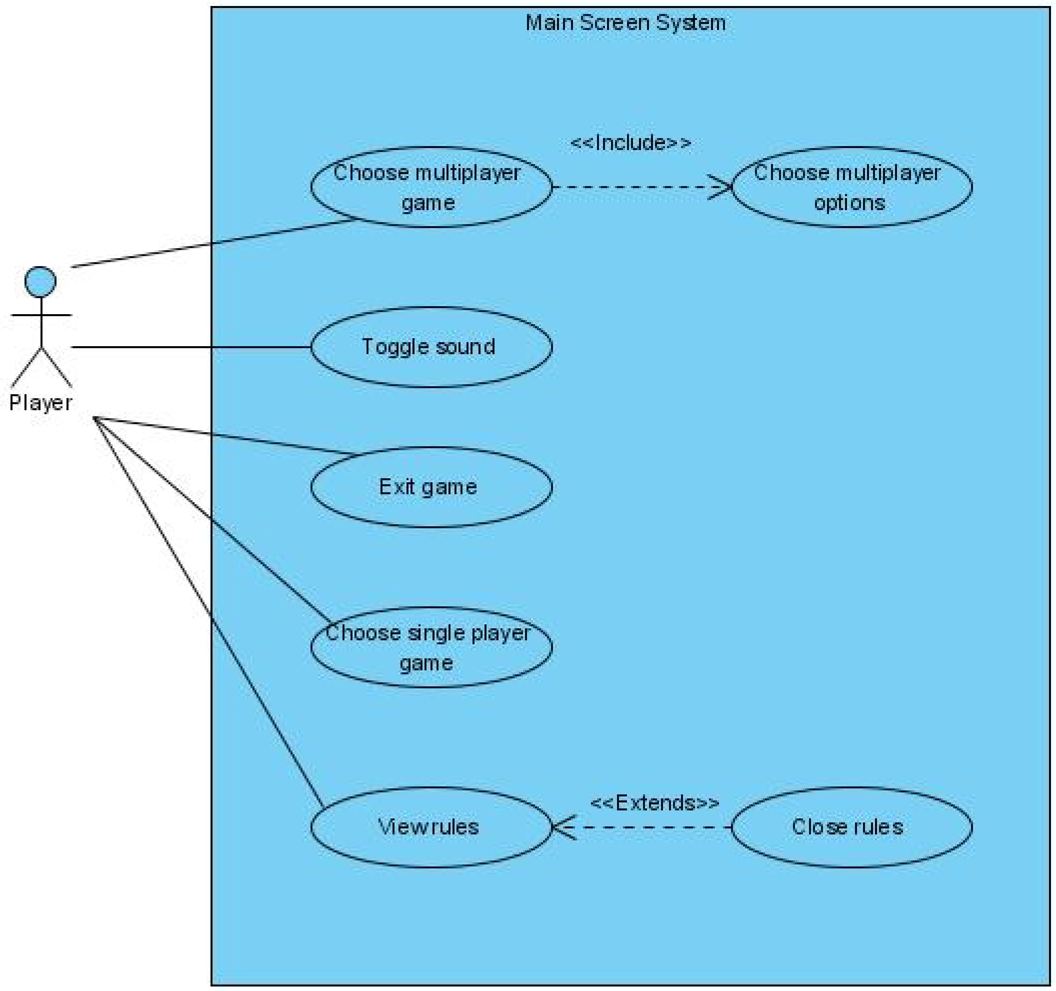


Diagram 4.2: Main Screen Use Case

4.3: 2. Multiplayer initial setup screen

The multiplayer initial setup screen is a small dialog box that appears after a user selects a multiplayer game from the main screen. Before a multiplayer game can be started the user must decide to either host or join the game. Once the user chooses to either host or join a game the multiplayer setup screen will appear upon making a successful connection. The main screen and the multiplayer initial setup screen will disappear.

A. Host Game

If the user chooses to host a game a server will be created. The user will then connect to his or her own server as a client. This will bring up the multiplayer setup screen.

B. Join Game

If the user chooses to join a game the IP address entered is what the game will use to find the host of the game. If the host is found and a connection is successfully established the user will be brought to the multiplayer setup screen. If a connection is unsuccessful an error message will appear and the user will remain on the current screen.

C. Enter IP Address

The user can enter the IP address of the host computer if the user wishes to join a game. When the user clicks the join game button the IP address entered will be used to find the host. If the user chooses to host a game, the IP address entered will be ignored.

D. Cancel

The cancel button will close the multiplayer initial setup screen. This will make the main screen active again. The main screen is always visible while the multiplayer initial setup screen is up, but it is not clickable until the multiplayer initial setup screen is closed.

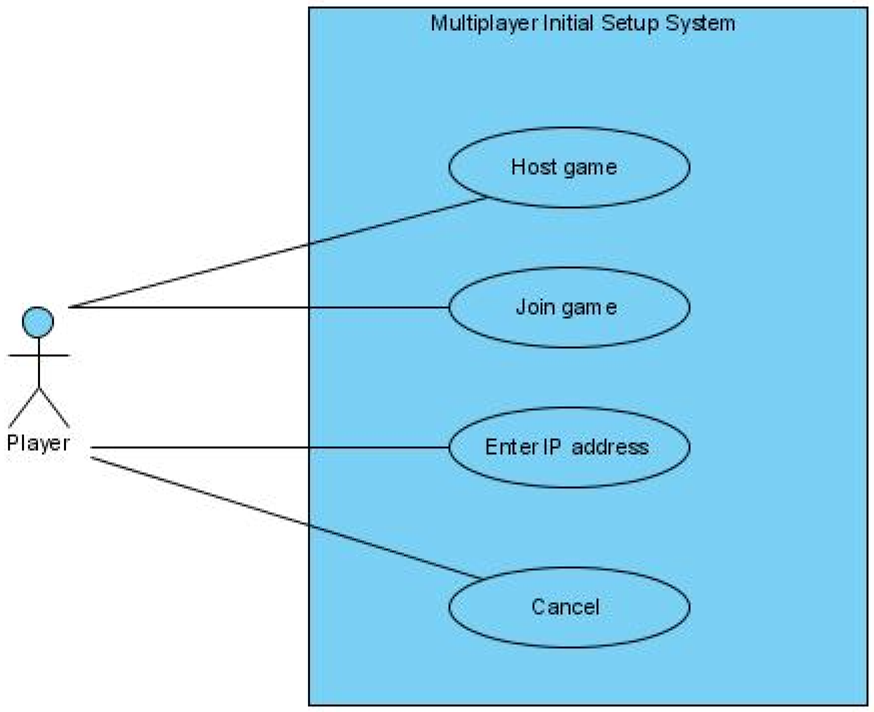


Diagram 4.3: Multiplayer Initial Setup System Use Case

4.4: 3. Single Player Setup System

After selecting single player mode from the main screen the user will be brought to the single player setup screen. At this screen the user will be able to choose how many computer players to add and which game piece they will control. The user will also have the options to view the rules, turn the sound on or off, start the game, and exit to the main screen or the entire game. The user can choose to start a game or exit to the main screen. Choosing to start a game will transition to the next important screen, the game board. Choosing to exit to the main screen will bring the user back to the main screen.

A. View Player

The user will be able to view all the players and characters for the game at this screen. This includes managing each player. The manage player use case will explain further.

B. View Message

The user will be able to view all the messages from the setup screen. These messages include important game notification like error messages from invalid selections. In addition the user will be able to copy messages.

C. Copy Message

The user can copy any amount of the messages by selecting the text and clicking copy. The user cannot cut the messages from the text area or paste messages to the text area. This option is more useful in the multiplayer setup screen, but is available for the single player setup screen too.

D. Start Game

The user may choose to start the game at any time. Before the game actually starts, the game will check to see if valid starting conditions are met. This includes at least two other computer players and that all players have selected different characters. If these conditions are met than the single player setup screen will close and the game board screen will be brought up, where the game will start. If one or more valid starting conditions are not met, an error message will be displayed indicating what the error was and the user will remain at the single player setup screen to fix the game options.

E. View Rules

This is the same as the main screen option.

F. Close Rules

This is the same as the main screen option.

G. Toggle Sound

This is the same as the main screen option.

H. Exit Game

Exiting the game will close the screen and exit the game. Everything visible or not will be closed and the game program will end.

I. Exit to Main Screen

Exiting the main screen will close the single player setup screen and bring up the main screen.

J. Exit

There are two types of exiting from the single player setup screen. Exit game and exit to the main screen both are a type of exit. Either button will close the single player setup screen. Based on which one is chosen determines whether or not the main screen is brought back yup or the game is closed.

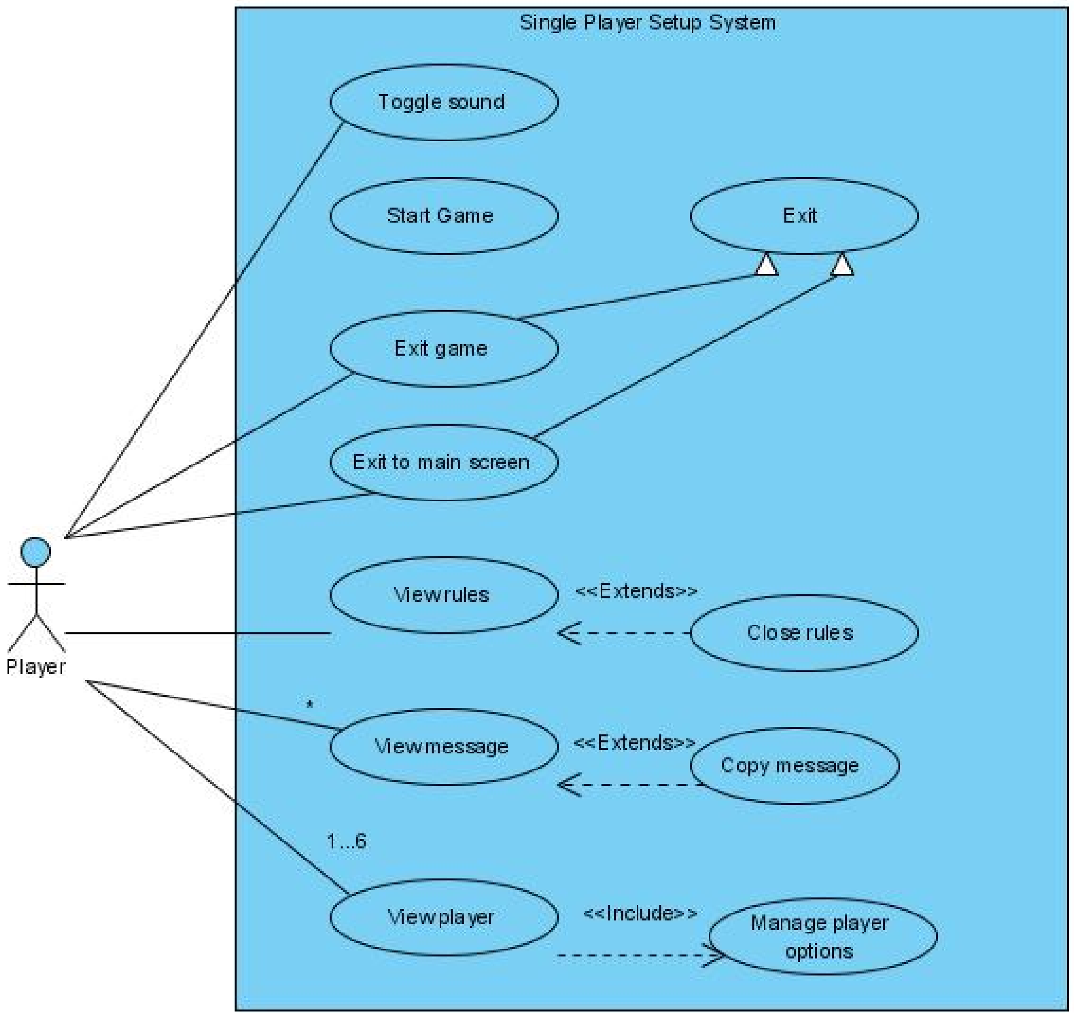


Diagram 4.4: Single Player Setup Screen Use Case

4.5: 4. Player Management System

The player management system is used by both the single player setup system and the multiplayer setup system. It is included in both use cases. At a setup screen players have options to choose their name and character. The single player game player is represented as a host extending from a player. The host of the game has all the options at the setup screen. The host of a multiplayer game extends from the host of a single player game and has the additional option to boot other players. The client extends from the player and has the option to choose only his or her name and character.

A. Enter Name

All users can enter their own name from the single player or multiplayer setup screen. In a multiplayer game a player must hit enter to confirm his or her name before clicking the ready button.

B. Choose Character

A host of a game can choose up to six characters. The host can choose all the characters in a single player game and all the AI player's characters in a multiplayer game. The host cannot choose another player's character in a multiplayer game. A client can choose only his or her character in a multiplayer game.

C. Choose Player

A host of a game can choose up to six other players to be in the game. The first player is automatically chosen since the host must include himself/herself. A client of a game can only chose whether or not he/she is in the game.

D. Boot Player

The host of a multiplayer has the option to boot all other human players that may be connected to the game. The host cannot boot himself/herself, but can exit the game.

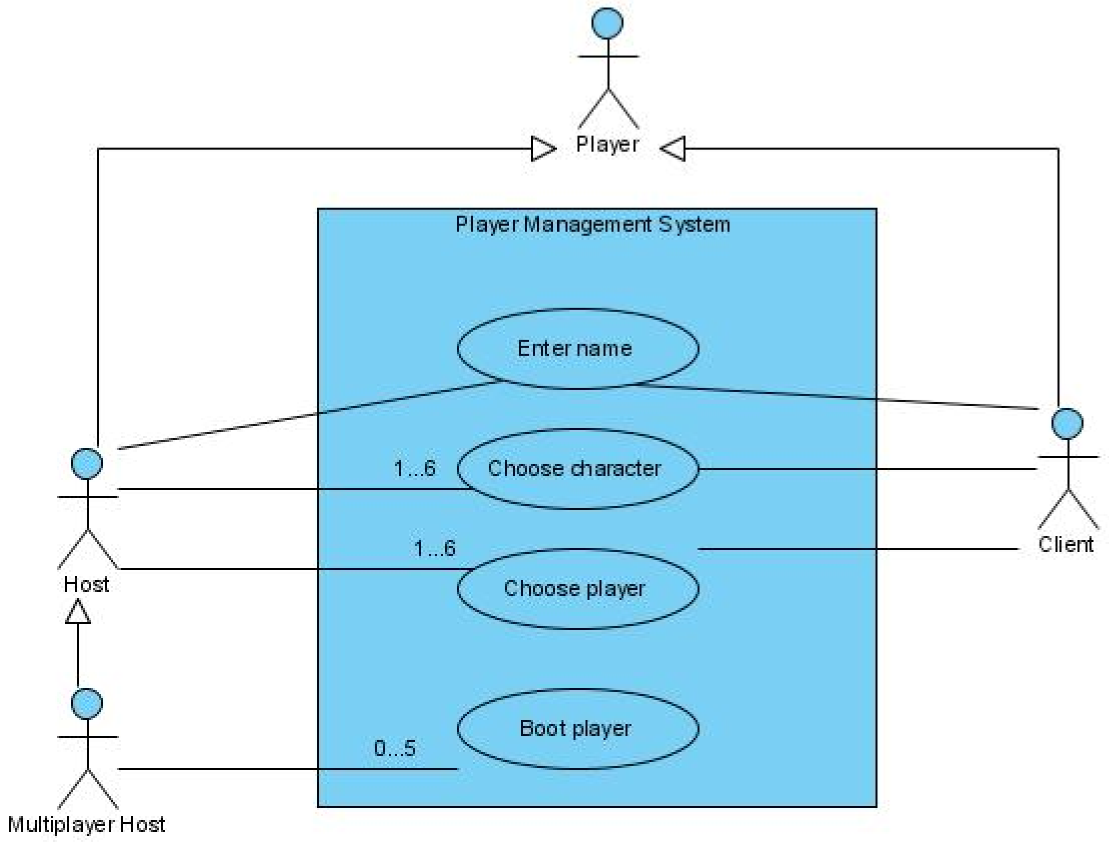


Diagram 4.5: Player Management System Use Case

4.6: 5. Multiplayer Setup System

After selecting to either host or join a game and successfully connecting to the server, the multiplayer setup screen will be displayed. This is similar to the single player setup system, except there are a few more options. Most of the new functionality is in the message system, since the players can talk to each other. There is also a slightly different way to start a game.

A. View Player

This is shown in the single player setup system

B. View Message

Viewing messages has more options than it did in the single player setup system and includes a separate use case to manage the messages.

C. View Rules

This is the same as the main screen option.

D. Close Rules

This is the same as the main screen option.

E. Toggle Sound

This is the same as the main screen option.

F. Exit Game

This is the same as the single player setup system except that exiting may end connections ungracefully depending on the method used to exit. Clicking the "x" in the upper right corner of the screen will basically kill the game. This is what the user typically wants when the "x" is clicked so the connections are not ended gracefully, but are handled in an acceptable way.

G. Exit to Main Screen

Exiting the main screen will close the multiplayer setup screen and end the connections to the game. The main screen will be brought back up.

H. Exit

This is the same as the single player setup system.

I. Toggle Chat

All chat messages can be toggled on or off. Toggling the chat on or off will not show any incoming chat messages. Previous messages that have been received will not disappear if the chat was previously on. Likewise messages that have been received while chat was disabled will not reappear when the chat is enabled.

J. Toggle Ready

The ready button is what determines when the game starts. All players will have the option to select or deselect their ready button. If the player has not chosen a character or confirmed their name they will receive an error message and the ready button will be deselected. When all players have selected their ready button the game countdown will start. During this time any player can deselect their ready button to stop the countdown. Once the countdown has completed the multiplayer setup screen will disappear and the multiplayer game board screen will be brought up.

K. View Host IP

All players including the host of the game can view the host's internal and external IP address.

L. Copy IP Address

A player can copy the IP addresses to paste in another text window.

M. Close IP Screen

A player can close the IP screen. While the IP screen is active a player cannot access the multiplayer setup screen. The multiplayer setup screen will remain visible but not active. Once the IP screen is closed the multiplayer setup screen will become active again.

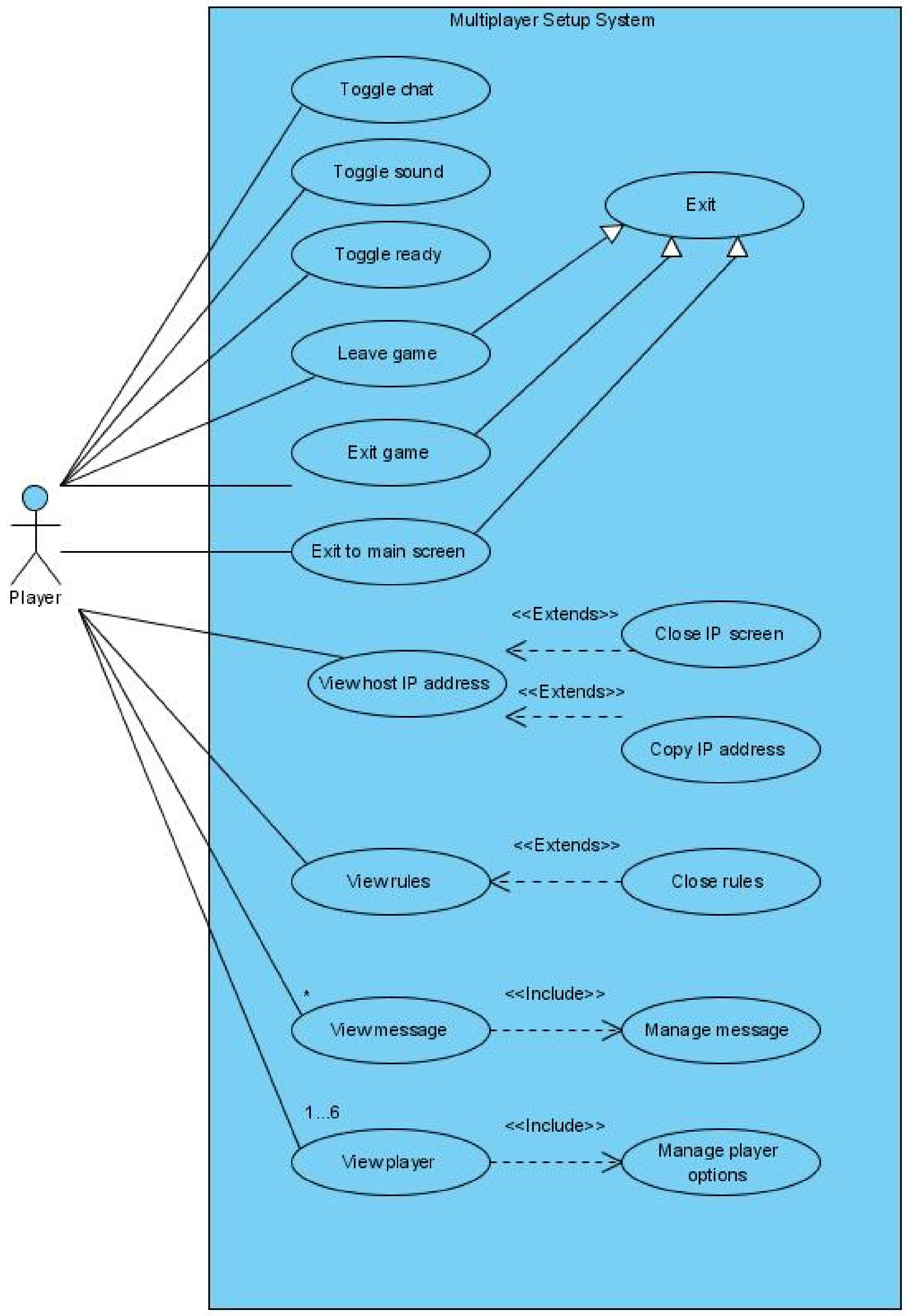


Diagram 4.6: Multiplayer Setup System Use Case

4.7: 6. Message Management System

The message management system allows players to manage game messages from both the multiplayer setup screen and the multiplayer game board screen. This use case is included in both the multiplayer setup system and the multiplayer game board system. The options shown are what a user can do with the text.

A. Cut Message

A player can cut messages from the text field that he/she has entered text in. A player can cut any number of messages. A player cannot cut messages from the game messages area.

B. Copy Message

A player can copy any number of messages from either his/her text field of the game messages field.

C. Paste Message

A player can paste any number of messages into his/her text field but not the game messages field.

D. Send Message

A player can send any number of messages to the other players connected to the game. After a player has hit enter with the text field active a simple error check process must be successful in order to send the message. The message typed must not be blank and must not exceed a certain length. The message must contain ASCII characters. An error message will appear for a player who tries to send an invalid message.

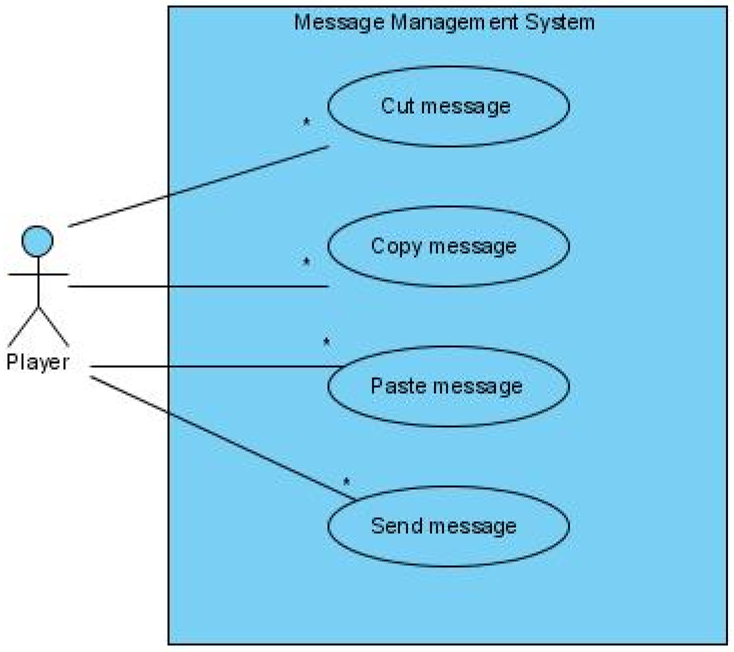


Diagram 4.7: Message Management System Use Case

4.8: 7. Single Player Game Management System

The single player game management system is the system that managed a single player game. The associated screen is the single player game board. From this screen the player has all the game options available. Some options are enabled or disabled depending on the state of the game.

A. View Message
The user will be able to view all game related messages on the left side of the screen.

B. Copy Message
This is the same as the single player setup system.

C. View Rules
This is the same as the main screen option.

D. Close Rules
This is the same as the main screen option.

E. Toggle Sound
This is the same as the main screen option.

F. Exit Game
This is the same as the single player setup system.

G. Exit to Main Screen
This is the same as the single player setup system.

H. Exit
This is the same as the single player setup system.

I. Make Suggestion/Accusation
A user has the option to make an accusation at any time during his/her turn. A user also has the option to make a suggestion at appropriate times during his/her turn. Both use the same screen in an almost identical process.

J. Choose Person
When making a suggestion or accusation a player must choose the person as one of the cards.

K. Choose Weapon
When making a suggestion or accusation a player must choose the weapon as one of the cards.

L. Choose Room
When making a suggestion or accusation a player must choose the room as one of the cards. For suggestions a player has no choice but to choose the room he/she is currently guessing from. For accusations a player can choose any room.

M. Roll
At the start of a players turn, the player can choose to roll the die.

N. Choose Spot
After rolling the die the player will have the option to move to a valid position on the board. If the spot is a door, the player will be moved into the appropriate room.

O. End Turn
At any time during a player’s turn, the player can end his/her turn. This will disable any buttons related to the player’s current turn that may still be enabled.

P. View Checklist

At any time a player can view his/her checklist. The checklist runs in a separate screen and the use case is included in the single player management system and the multiplayer management system.

Q. Close Checklist

The player has the option to close the checklist at any time. The checklist data is saved and can be reopened at any time. The screen is merely invisible while it remains closed, but it still exists.

R. View Card List

The player can view his/her own cards at any time during the game. The cards are shown as images and text on the screen.

S. Close Card List

The card list can be kept open and closed at any time similar to the checklist screen. Likewise the data still exists but the screen is invisible.

T. Disprove Suggestion

A player can disprove any number of suggestions as long as he/she has one or more cards to disprove with. When it is the player’s turn to disprove a suggestion, the player will be presented with a popup screen with the images of all the cards the player can disprove the suggestion with. The screen cannot be closed until one of the cards is chosen. While this screen is open the game board screen remains inactive.

U. View Winning Cards

After an accusation the winning cards screen will popup.

V. Close Winning Cards

The player can close the winning cards screen after viewing the winning cards. This will make the game board screen active again.

W. View Card

After another player has chosen a card to disprove with the card will be shown to the player. The image of the card will be shown as a popup window and must be closed in order to access the main screen again.

X. Restart Game

At any time during the game the player can restart the game.

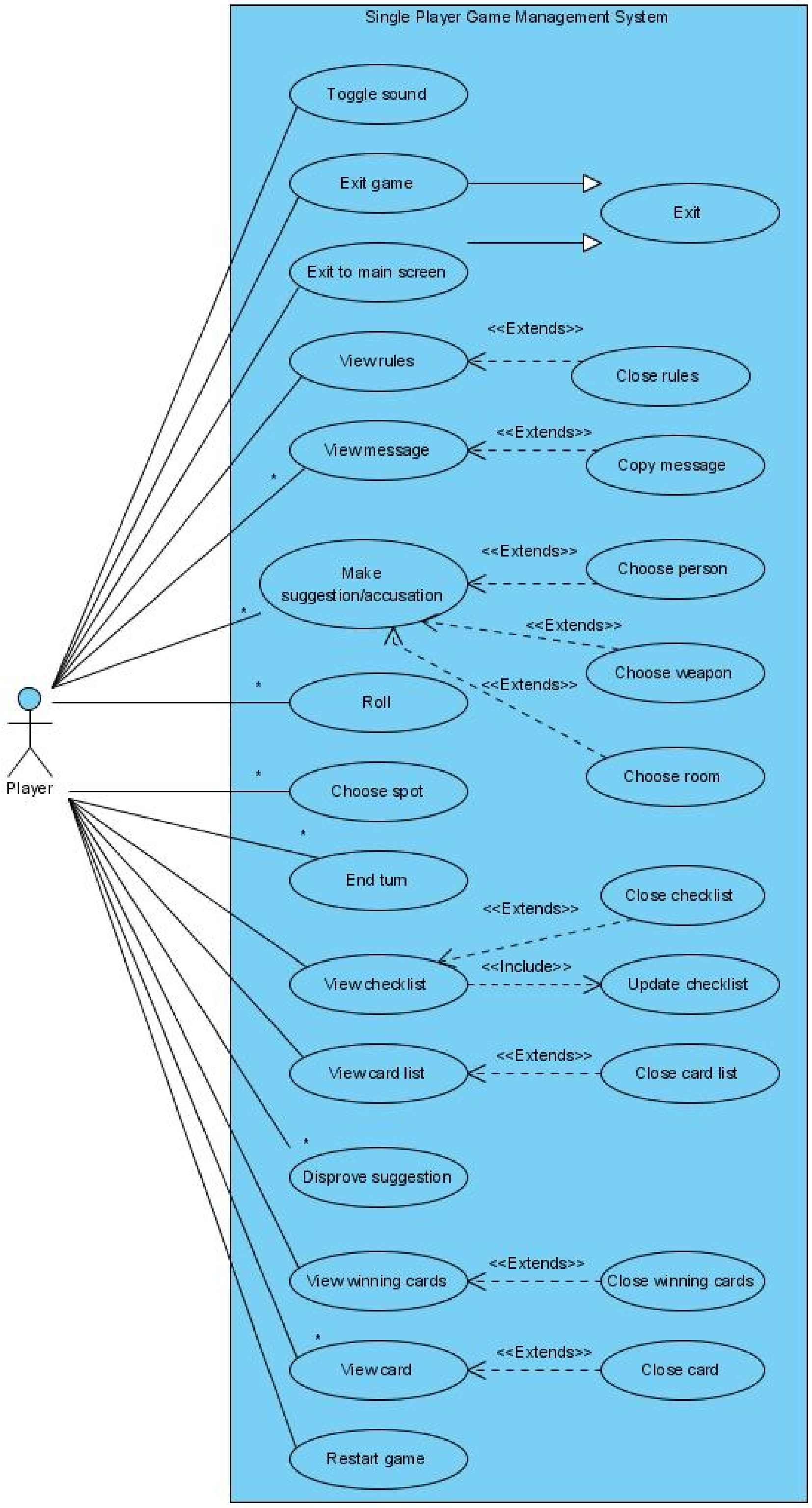


Diagram 4.8: Single Player Game Management System Use Case

4.9: 8. Checklist System

When a player chooses to view the checklist from either the single player game management system or the multiplayer game management system the checklist system is used. The player uses the checklist system and the checklist screen to update the checklist that holds the information the player has gained throughout the game. The checklist screen is a separate screen that can be kept open for the duration of the game, while the game board screen is open.

A. Change color
The user can change the color of the symbol to add to the checklist.

B. Select color
The user can select the color to use. The colors available are black, red, orange, gray, green, blue, and purple. The colors all represent the color of the players in the game, with the exception of black.

C. Add Symbol
The user can add a symbol to any spot on the checklist by left-clicking on the spot. If a symbol already exists then that symbol is replaced by the new symbol.

D. Select Symbol
The user can select the "X", "O", or "*" symbol to add to the checklist.

E. Remove Symbol
The user can remove any number of symbols on the checklist by right-clicking on them. Right-clicking on an empty spot will simply do nothing.

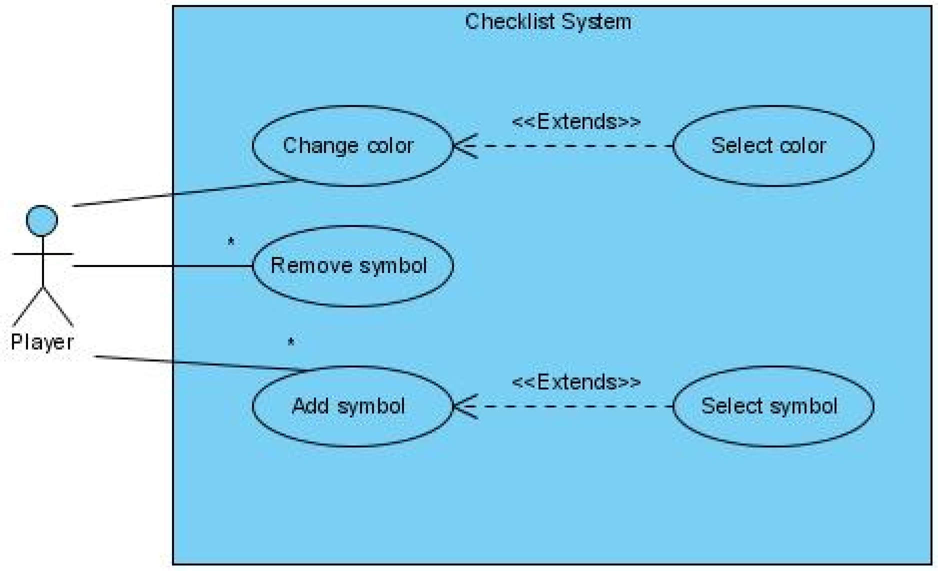


Diagram 4.9: Checklist System Use Case

4.10: 9. Multiplayer Game Management System

The multiplayer game management system is similar to the single player game management system. All options available in the single player game are available in a multiplayer game. The only difference is that only the host has the option to restart a game. In addition the multiplayer game management system has some additional features that are also in the multiplayer setup system.

A. View Message

This is the same as the multiplayer setup system.

B. Copy Message

This is the same as the single player setup system.

C. View Rules

This is the same as the main screen option.

D. Close Rules

This is the same as the main screen option.

E. Toggle Sound

This is the same as the main screen option.

F. Toggle Chat

This is the same as the single player management system.

G. Exit Game

This is the same as the single player setup system.

H. Exit to Main Screen

This is the same as the single player setup system.

I. Leave Game

All players have the option to leave the game. This will terminate all connections and bring the player back to the main screen.

J. Exit

This is the same as the single player setup system.

K. Make Suggestion/Accusation

This is the same as the single player game management system.

L. Choose Person

This is the same as the single player game management system.

M. Choose Weapon

This is the same as the single player game management system.

N. Choose Room

This is the same as the single player game management system.

O. Roll

This is the same as the single player game management system.

P. Choose Spot

This is the same as the single player game management system.

Q. End Turn

This is the same as the single player game management system.

R. View Checklist

This is the same as the single player game management system.

S. Close Checklist

This is the same as the single player game management system.

T. View Card List

This is the same as the single player game management system.

U. Close Card List

This is the same as the single player game management system.

V. Disprove Suggestion

This is the same as the single player game management system.

W. View Winning Cards

After an accusation the winning cards screen will popup.

X. Close Winning Cards

This is the same as the single player game management system.

W. View Card

This is the same as the single player game management system.

Z. Restart Game

At any time during the game the host of the game can restart the game.

AA. View Host IP

This is the same as the multiplayer setup system.

AB. Close IP Screen

This is the same as the multiplayer setup system.

AC. Copy IP Address

This is the same as the multiplayer setup system.

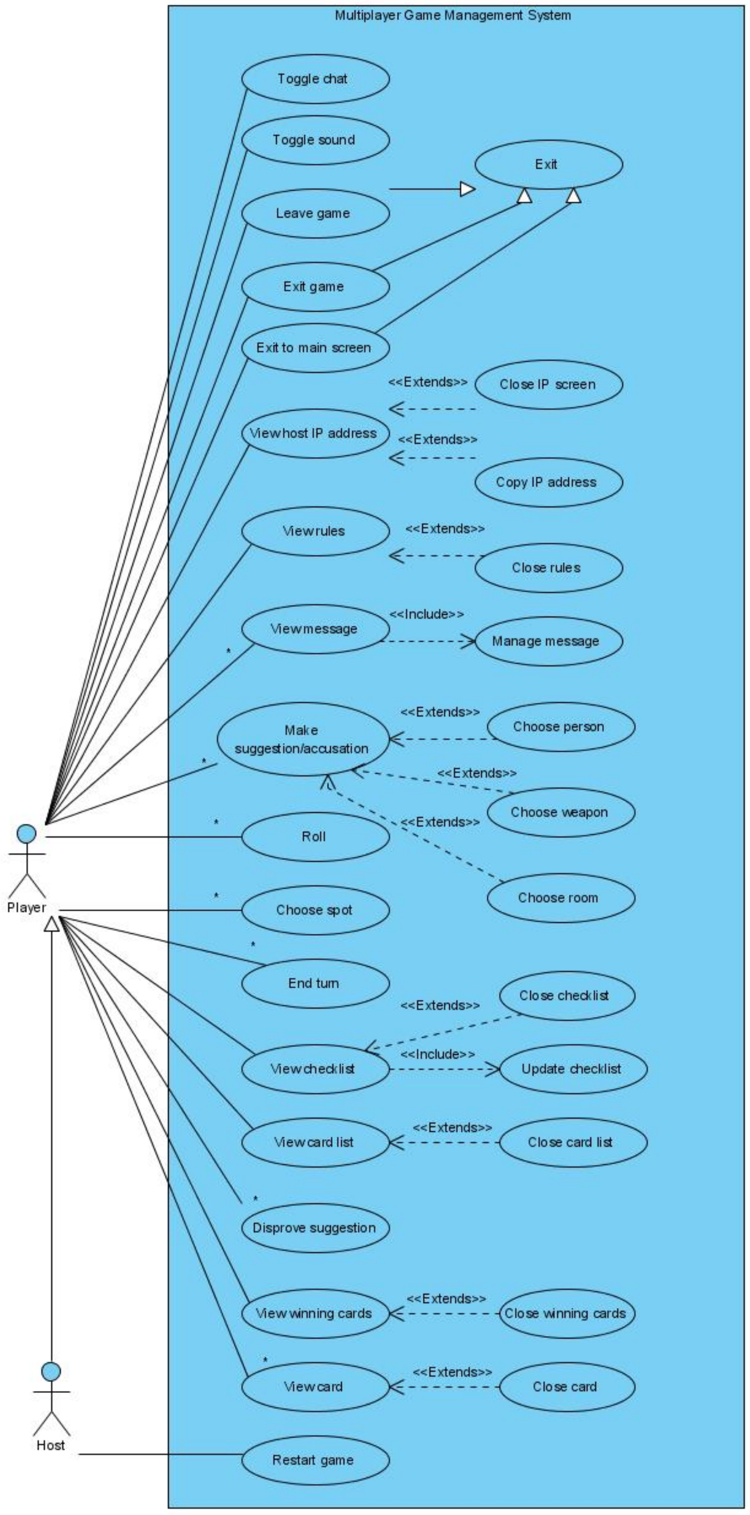


Diagram 4.10: Multiplayer Game Management System Use Case

5: User Interface Design

5.1: 1. Main Screen

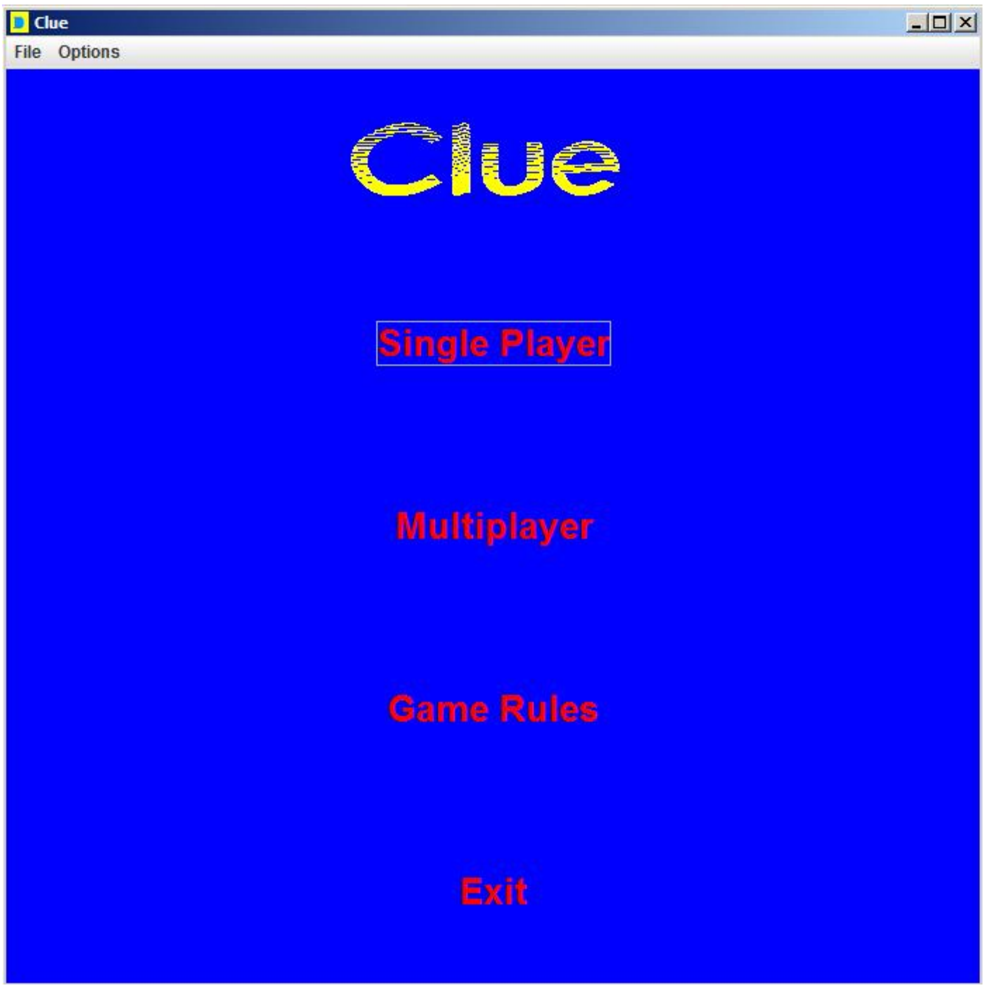


Figure 5.1: Main Screen

The main screen, as shown above, will have the logo at the top with the following clickable buttons in red below. At the very top is a menu bar with some basic options such as toggling sound or exiting the game. From this screen the Single player setup screen, rules screen, and multiplayer preliminary setup screen can be accessed.

5.2: 2. Single player setup screen

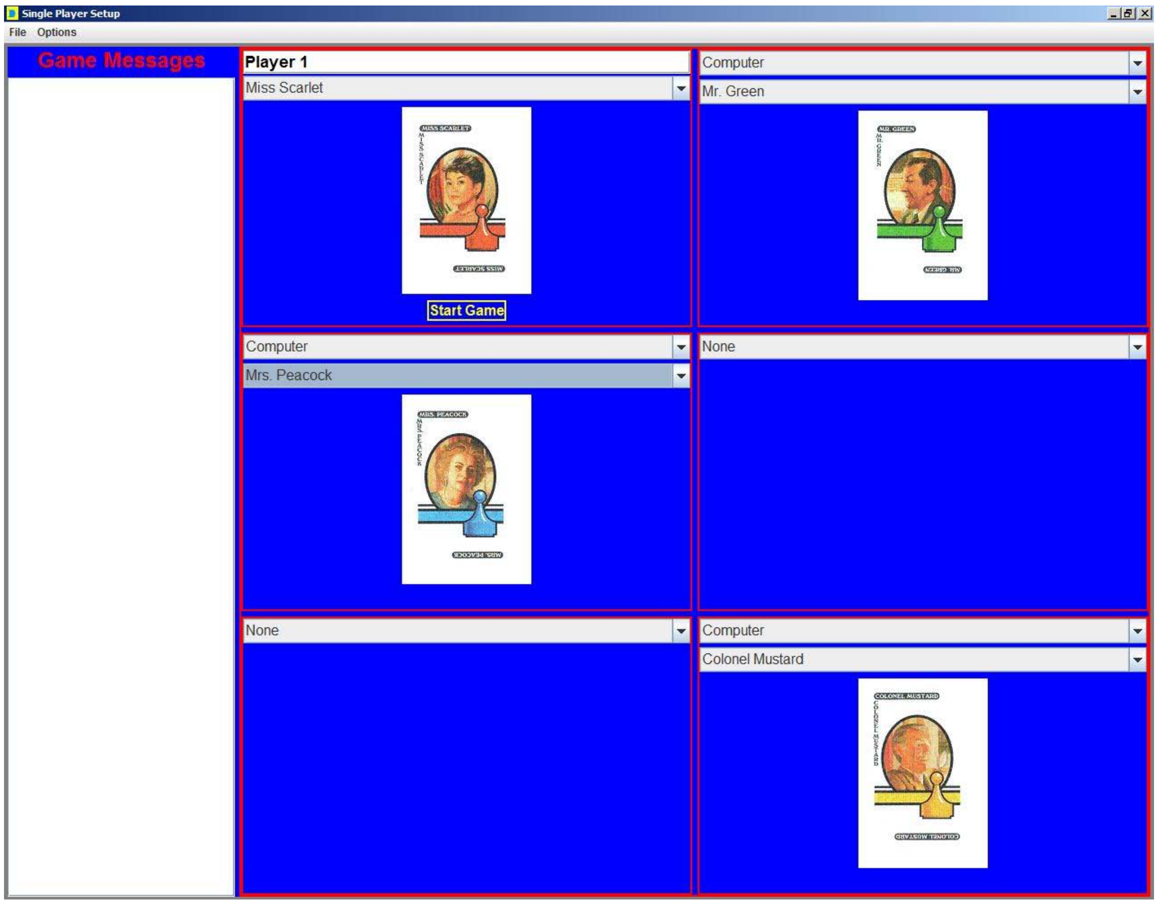
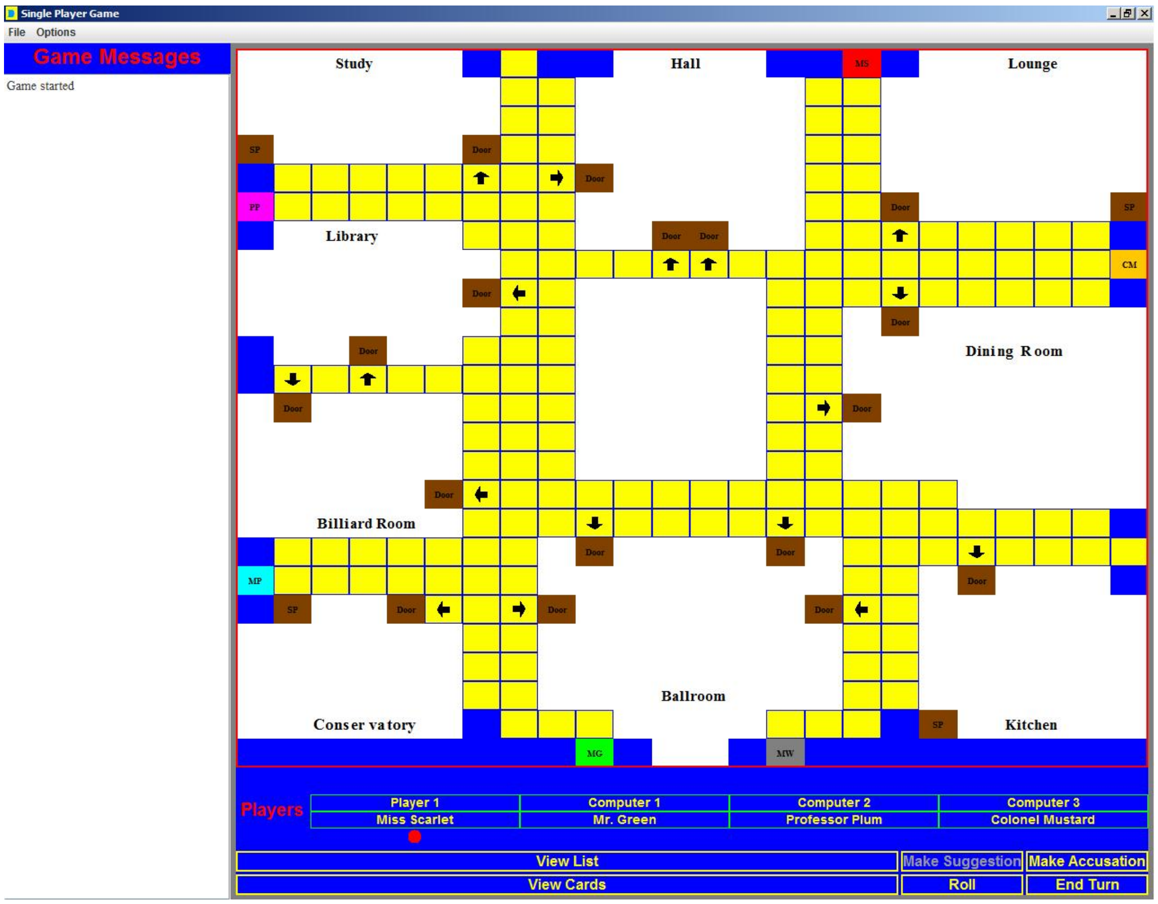


Figure 5.2: Single Player Setup Screen

The single player setup screen has game messages to the left. These messages will inform the user if any selection that has been made is invalid when the user clicks the start game button. To the right are six boxes representing possible players. The user has a name field, where a name can be entered. The computer players do not have this field and are supplied a default name. Each main box has three fields. The top field is where the user can select whether a computer player should be active in that spot or not. If the computer player is selected another box will appear to choose the character for the player. The character image is shown below the character selection box. In addition to these fields the user will have a start game button in his corresponding box. The multiplayer setup screen will be similar to the single player setup screen. Some differences will be that all players will be able to choose their name, but not other people's names. Instead of a start button each player will have a ready button in a multiplayer game. When all players have toggled their ready button the game will start. The host of the multiplayer game also has a boot button to boot other players from the game. Also below the game messages there will be a small text box for the users to type and send messages to each other. The screens will look nearly identical.

5.3: 3. Game Board



Game started

Figure 5.3: Game Board

The game board screen is the most complicated of all the screens. To the left once again are the game messages. Any critical game information will be shown and added to the text area. When the text area fills up a scroll bar will appear allowing the user to scroll up and down to see messages that have appears at any time during the game. This is a key feature added to the game, since the current computer version of Clue does not have it. With this feature users will be able to keep track of the game much better, which also results in better game play.

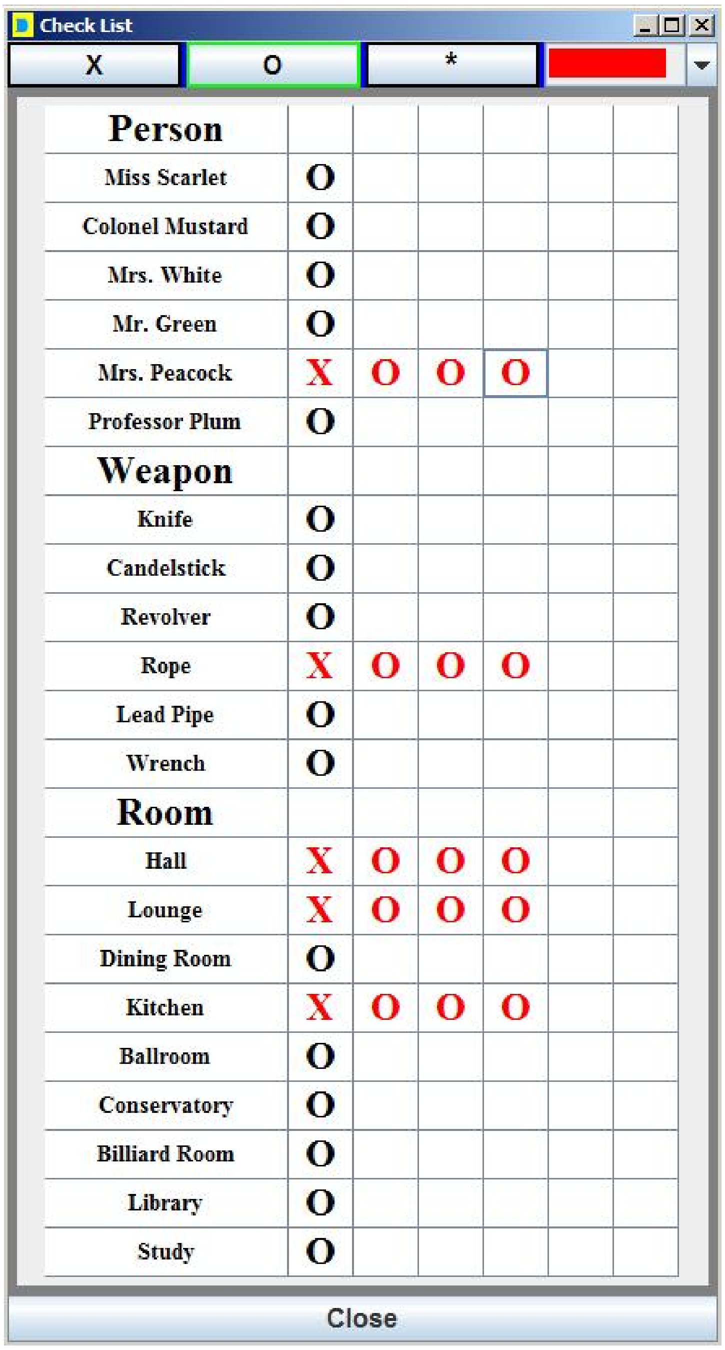
To the right of the game messages is the game board. The game board may not look exactly like the actual physical game board, but it is the same board and everything is in the correct spot. The board is a table, and each cell is a square. The white squares represent a room and have no border. The yellow cells represent spots on the board where a player can move to. The physical game board represented doors as something similar to a line; however this was not doable programmatically. Therefore the doors are represented as brown cells labeled “door” as part of the room. In addition to the doors, arrows are supplied on the yellow cells to show how the player must enter the room. If the cell does not have an arrow pointing to a door, then a player cannot move from that cell to the door cell, even though the two cells are adjacent. The players are represented as their respective colors with their initials.

Another key feature is improved movement. The old game forced a player to click on all the spots in order in order to move. This game allows the user to just click on the spot to move to and the game will do the rest. This occurs first by having the user roll the die. Once the die is rolled, the game logic uses a complex recursive algorithm to determine all the spots on the board that the user can move to. All these spots are then outlined with a green border. All a user has to do is click on one of the valid spots. The game will use another complex recursive algorithm to determine the path the user will take to reach the spot. Then a timer is used to show the animation as the user’s piece moves from one spot to the next until it has reaches its final spot. The added intelligent logic makes the game much easier to play than the current version.

Below the game board are boxes to display all the players, their characters, and whose turn it is. On top is the player’s name, and underneath the name is the player’s character. The turn order is also represented as the order of the boxes. Underneath the player’s character is a red circle to show whose turn it is. The red circle moves whenever the player’s turn changes. It will always be underneath the correct player.

At the bottom of the screen are six buttons. The view list and view cards buttons allow the user to bring up those respective screens. The other four buttons may be enabled or disabled depending on the state of the player. For example, if the player has already rolled the die then the roll die button will be disabled since the player cannot roll twice. The end turn button will end the users turn. The make accusation button and make suggestion button will bring up another screen. The roll die button will also bring up a small screen to show the animation of the die rolling.

5.4: 4. Checklist

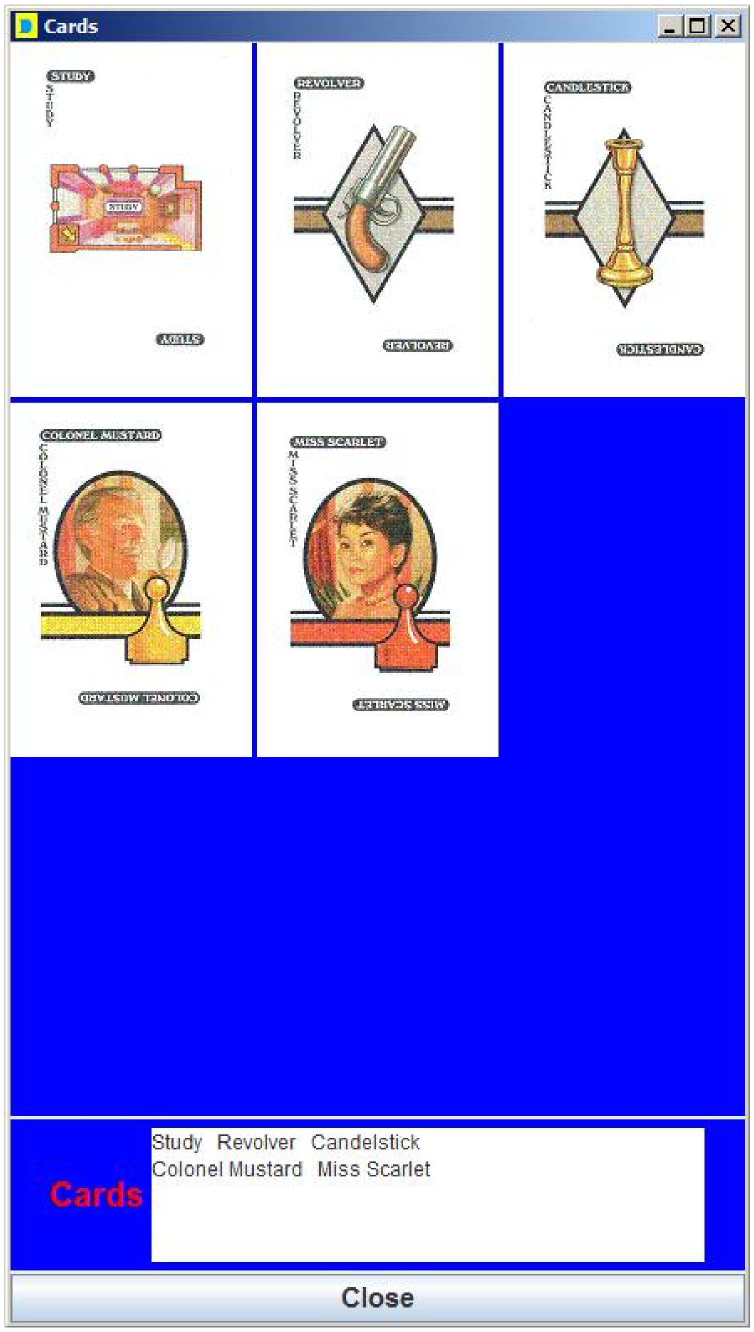


Person						
Miss Scarlet	O					
Colonel Mustard	O					
Mrs. White	O					
Mr. Green	O					
Mrs. Peacock	X	O	O	O		
Professor Plum	O					
Weapon						
Knife	O					
Candelstick	O					
Revolver	O					
Rope	X	O	O	O		
Lead Pipe	O					
Wrench	O					
Room						
Hall	X	O	O	O		
Lounge	X	O	O	O		
Dining Room	O					
Kitchen	X	O	O	O		
Ballroom	O					
Conservatory	O					
Billiard Room	O					
Library	O					
Study	O					

Figure 5.4: Checklist

The check list screen allows the user to record data they have gained throughout the game. The main part of the screen is a table broken down into three sections; a person section, a weapon section, and a room section. There are six columns for each thing, where a user can mark information. The user has three choices in symbols shown as buttons at the top of the screen. The 'X', 'O', and '*' symbol can be used. The selected symbol will be outlined in green to show the user what the currently selected symbol is. All a user has to do is click on a cell and that symbol will appear there. To clear the symbol from the cell a user can right click the cell. The box to the right of the symbols is the colors. There are six colors the user can choose from. The close button on the bottom will hide the screen.

5.5: 5. Card list



Study Revolver Candelstick
Colonel Mustard Miss Scarlet

Figure 5.5: Card list

The card list shows the user what cards they have. The top is a 3x3 grid, which in most cases will not always be fully populated by cards. The cards are shown in the grid in a sorted order. The weapons, people, and rooms will always be grouped together. Below the images is a text list of the cards also being displayed in a 3x3 grid and grouped in order. Below that is a close button to allow the user to hide the screen. The text list was implemented since the images are not good quality and the visually impaired user might have trouble reading the name from the fuzzy image.

5.6: 6. Suggest / Accuse screen

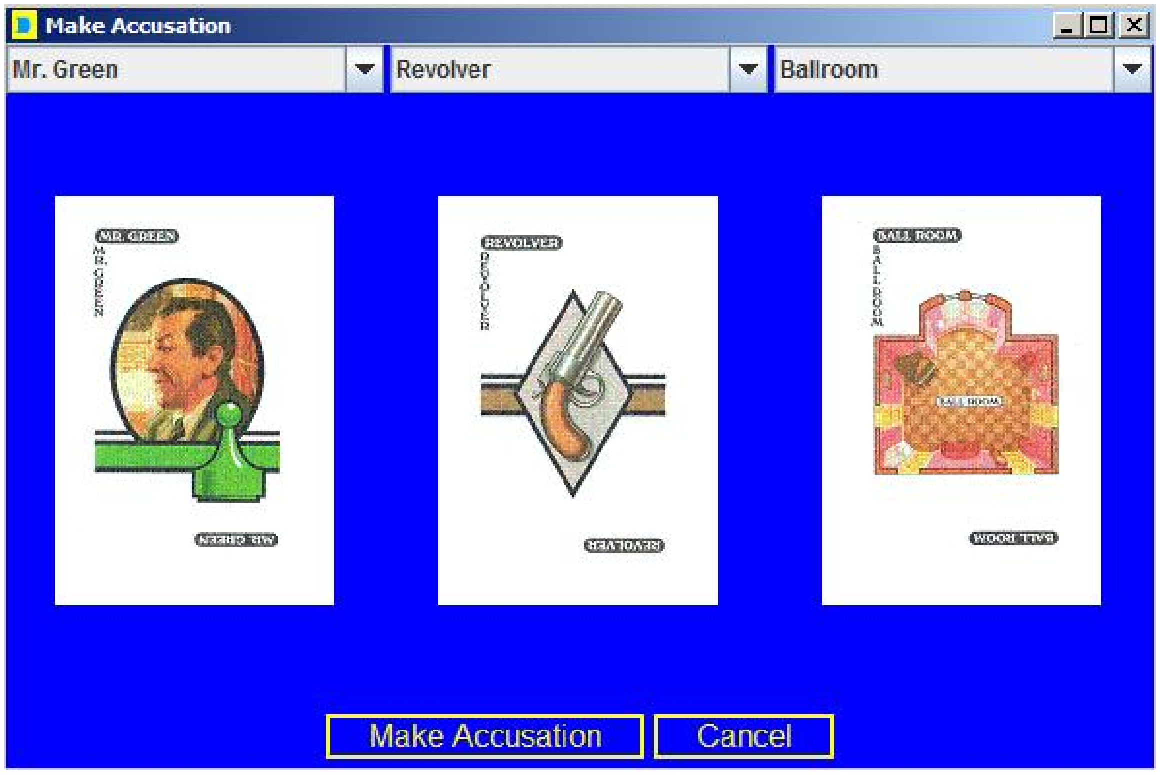


Figure 5.6: Suggest/Accuse Screen

The Suggest and Accuse button both bring up the same screen. However, based on which button was clicked the screen will have different properties. At the top are three choice boxes one for the person, one for the weapon, and one for the room. The room box will be disabled and reselected if the make suggestion button was clicked. Otherwise it will be selectable as shown above. When the user has selected an item the corresponding image will appear below it. When the user has chosen all three items the make accusation / make suggestion button will become enabled, allowing the user to continue. The user may also exit this screen with the cancel button.

5.7: 7. Winning Cards screen

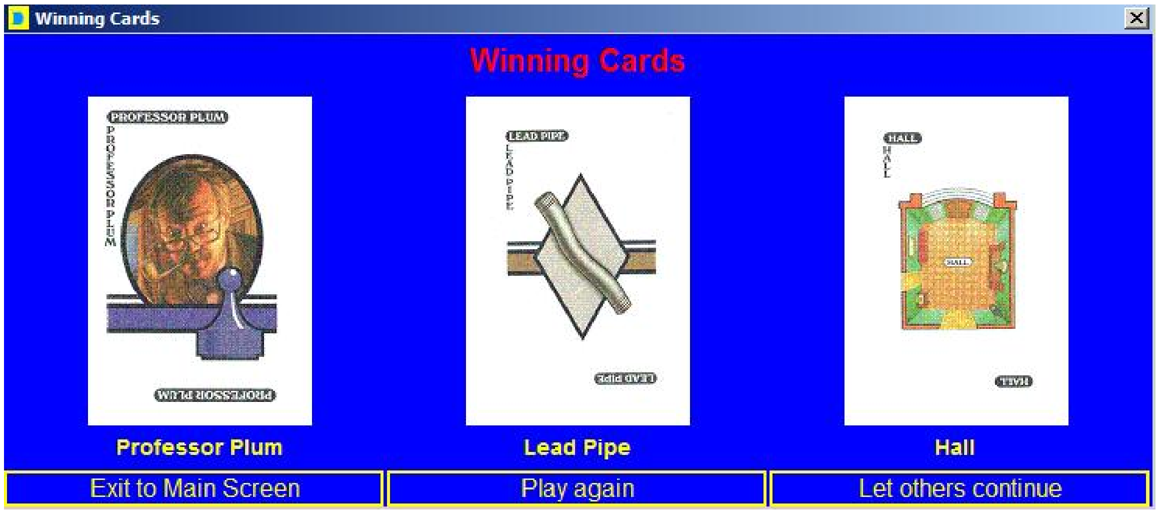


Figure 5.7: Winning Cards Screen

The winning cards screen shows the three winning cards. The person card is on the left, the weapon card is in the middle, and the room card is on the right. In a single player game the options to exit to the main screen, play again, or let others continue will be optional below the images.

5.8: 8. Disprove Cards screen

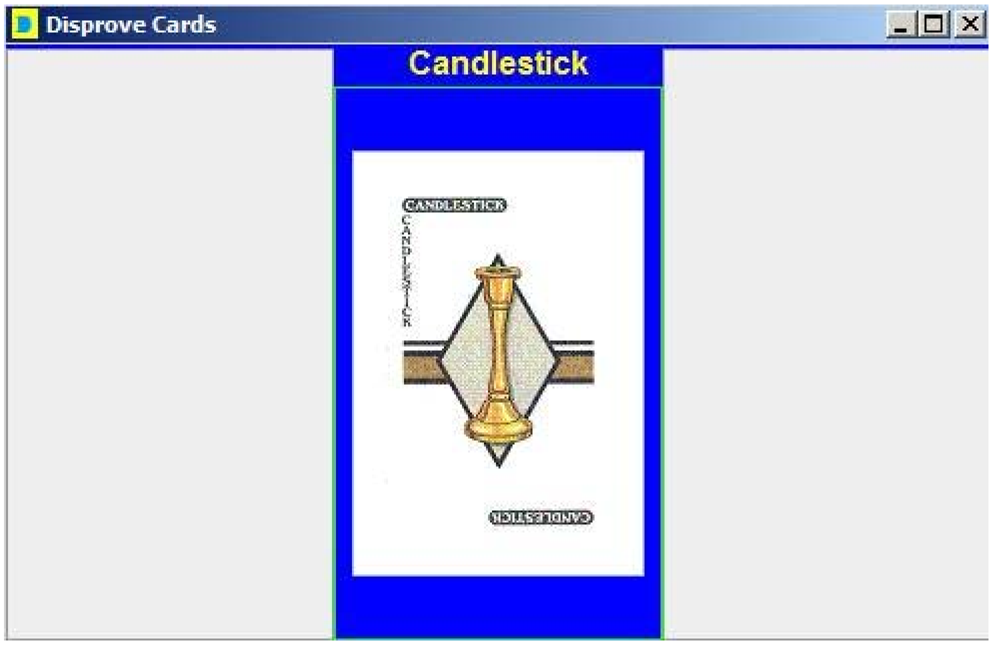
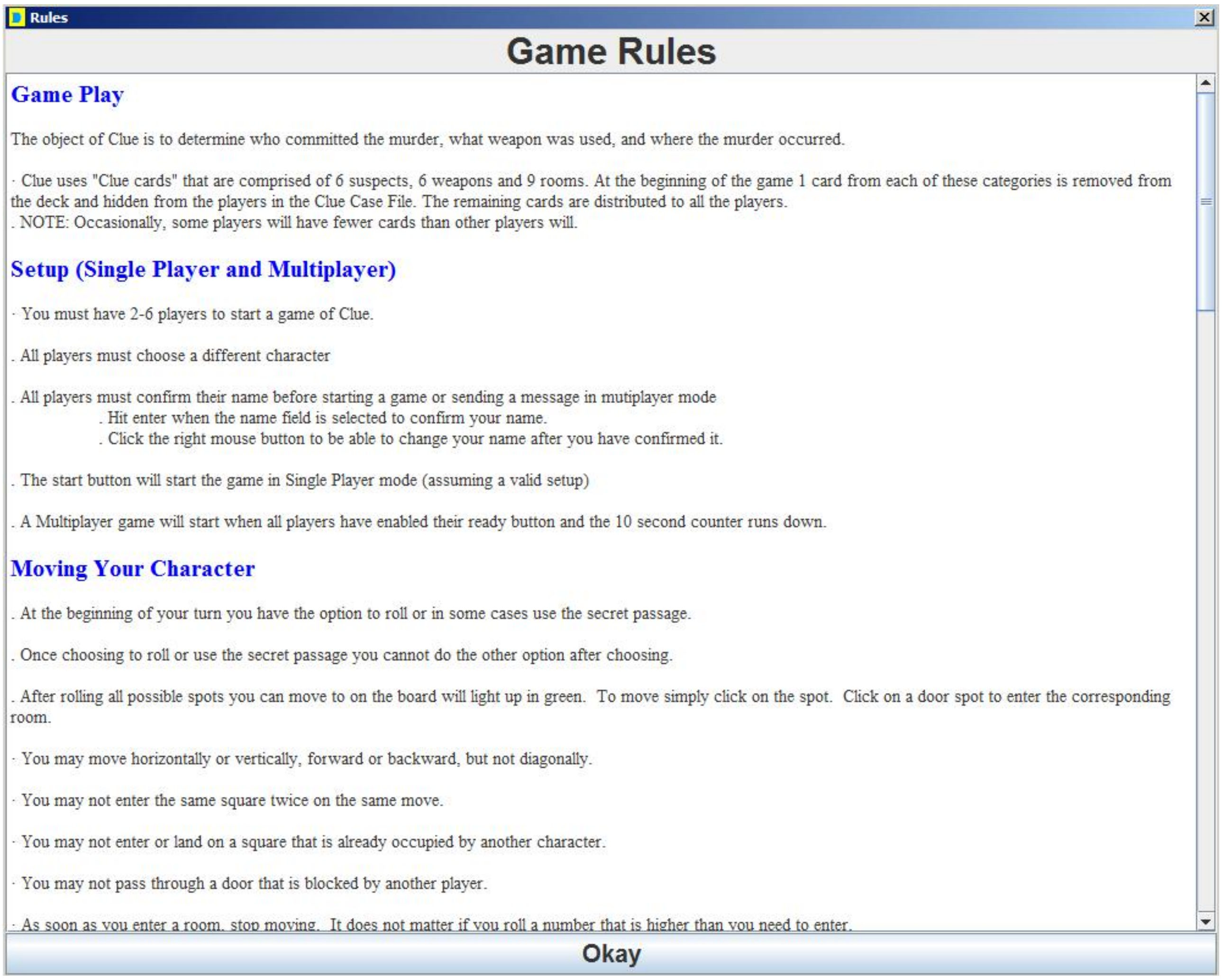


Figure 5.8: Disprove Cards Screen

The disprove cards screen pops up when a player has at least one card to disprove with. The person card is on the left, the weapon card in the middle, and the room card on the right. Images of all cards that the person actually has will appear. Cards that the person does not have will not appear and the space will be left blank. Each image is clickable to choose as the disprove card. The “x” button in the upper right corner is specialized to force a user to choose a card before closing the screen. If the “x” button is clicked the screen will not close. Instead a message will pop up asking the user to choose a card to disprove the suggestion with.

5.9: 9. Game Rules screen



Game Rules

Game Play

The object of Clue is to determine who committed the murder, what weapon was used, and where the murder occurred.

· Clue uses "Clue cards" that are comprised of 6 suspects, 6 weapons and 9 rooms. At the beginning of the game 1 card from each of these categories is removed from the deck and hidden from the players in the Clue Case File. The remaining cards are distributed to all the players.
. NOTE: Occasionally, some players will have fewer cards than other players will.

Setup (Single Player and Multiplayer)

· You must have 2-6 players to start a game of Clue.

. All players must choose a different character

. All players must confirm their name before starting a game or sending a message in mutiplayer mode
. Hit enter when the name field is selected to confirm your name.
. Click the right mouse button to be able to change your name after you have confirmed it.

. The start button will start the game in Single Player mode (assuming a valid setup)

. A Multiplayer game will start when all players have enabled their ready button and the 10 second counter runs down.

Moving Your Character

. At the beginning of your turn you have the option to roll or in some cases use the secret passage.

. Once choosing to roll or use the secret passage you cannot do the other option after choosing.

. After rolling all possible spots you can move to on the board will light up in green. To move simply click on the spot. Click on a door spot to enter the corresponding room.

· You may move horizontally or vertically, forward or backward, but not diagonally.

· You may not enter the same square twice on the same move.

· You may not enter or land on a square that is already occupied by another character.

· You may not pass through a door that is blocked by another player.

· As soon as you enter a room, stop moving. It does not matter if you roll a number that is higher than you need to enter.

Figure 5.9: Game Rules Screen

The rules screen has the title on the top, the text in the middle, and an “okay” button on the bottom. The text is customized so that each section title is in blue and the other text is normal. The text is both vertically and horizontally scrollable. The horizontal scroll bar may or may not appear depending on the size of the screen and the length of each piece of information. The “okay” button closes the screen just like the “x” button.

5.10: 10. Windows Navigation Diagram

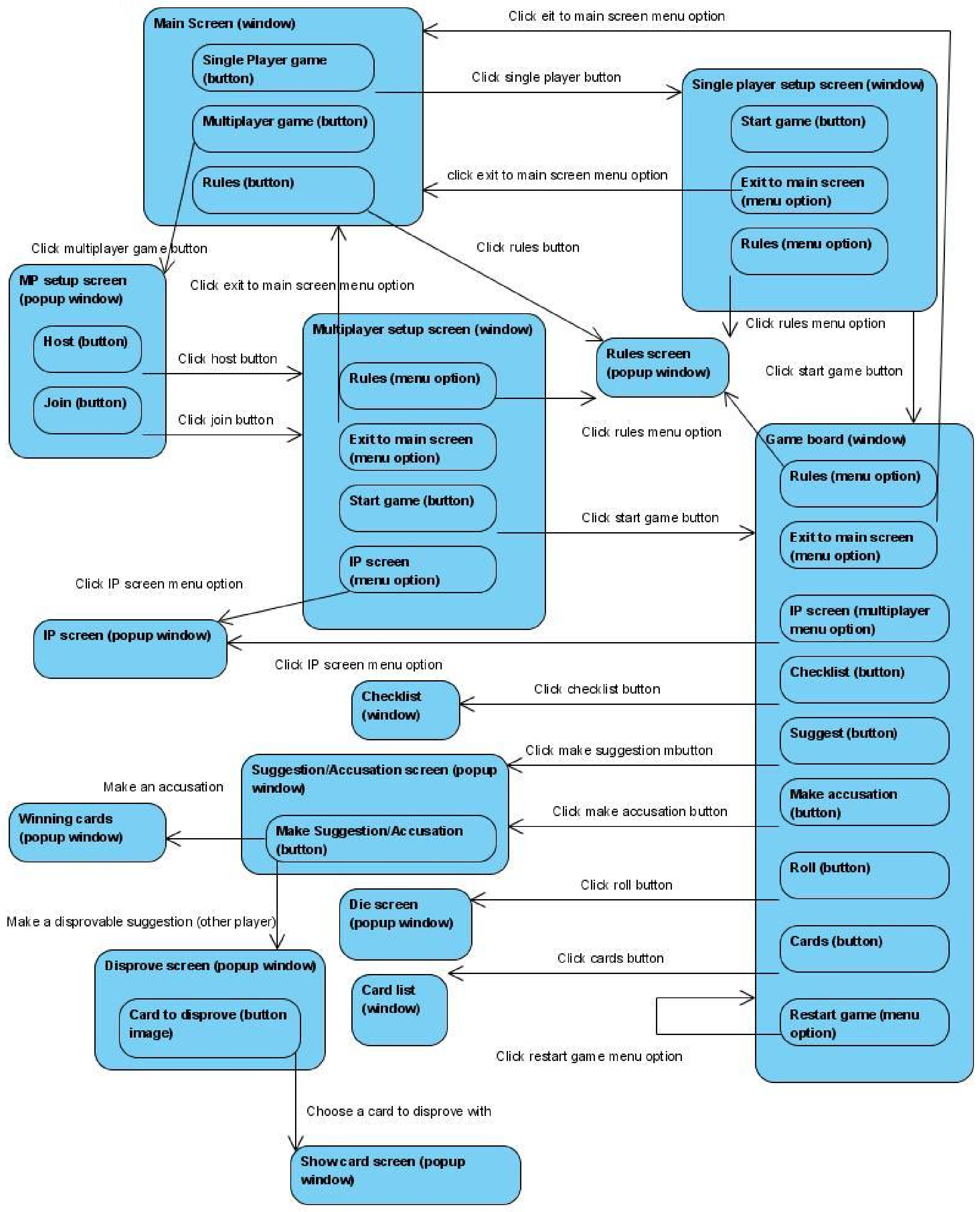

Main Screen (window)
Single Player game (button)
Multiplayer game (button)
Rules (button)
Click eit to main screen menu option
Click single player button
Single player setup screen (window)
Start game (button)
Exit to main screen (menu option)
Rules (menu option)
click exit to main screen menu option
Click multiplayer game button
Click rules button
Click exit to main screen menu option
MP setup screen (popup window)
Host (button)
Join (button)
Multiplayer setup screen (window)
Rules (menu option)
Exit to main screen (menu option)
Start game (button)
IP screen (menu option)
Click rules menu option
Rules screen (popup window)
Click start game button
Click host button
Click join button
Click rules menu option
Game board (window)
Rules (menu option)
Exit to main screen (menu option)
Click start game button
IP screen (multiplayer menu option)
Click IP screen menu option
IP screen (popup window)
Click IP screen menu option
Checklist (button)
Checklist (window)
Click checklist button
Suggest (button)
Click make suggestion mbutton
Suggestion/Accusation screen (popup window)
Make an accusation
Winning cards (popup window)
Make Suggestion/Accusation (button)
Click make accusation button
Make accusation (button)
Roll (button)
Click roll button
Die screen (popup window)
Make a disprovable suggestion (other player)
Disprove screen (popup window)
Card to disprove (button image)
Cards (button)
Click cards button
Card list (window)
Restart game (menu option)
Click restart game menu option
Choose a card to disprove with
Show card screen (popup window)

Figure 5.10: Windows Navigation Diagram

The windows navigation diagram shows how all the screens can be accessed. Multiple windows can be open at the same time depending on what the window is and what button was clicked. For example while the game board window is up, the card list and checklist windows can be open at the same time. However once the exit to main screen button is clicked the game board window will close and the main screen window will become visible again. When this occurs, all screens related to the game board will close as well including the card list and checklist windows. Another point to make is that when an accusation is made the player who made the accusation will see the winning cards screen. The other players in the game will only see the winning cards screen if the accusation is correct. Also when a suggestion is made the disprove screen will pop up for the first player who can disprove the suggestion and not the player who made the suggestion. Likewise the show card screen will not appear for the player who has disproves the suggestion. It will only appear for the player who made the suggestion. Another note is that the IP screen menu option is only available in multiplayer games from the game board window.

6: Game Mechanics

6.1: Introduction

The game is run by something similar to a state machine. The main difference is that a regular state machine would not give the desired effect. The game needed the timing correct so it does not play too fast. Most of the processing is done instantly, which would cause several problems. This means that the animation for moving a player, messages for disproving a suggestion, AI decisions, and sounds will all not function correctly. Everything would happen too fast for the player to see. Also sounds would be overlapping and be inaudible. For these reasons the simple state machine was done with timers and time states instead. The timers cause the game to play at a correct speed and the time states are basically the original state machine. When a timer event occurs the event is processed based on the time state. The time state is set prior to the timer starting.

6.2: Timers and Time States

Instead of having just a single timer the combination of three timers was used. Although only one timer runs at a time, three timers were used instead based on preset delay times. If only one timer was used the delay time would have to be changed every time the time state was changed for the correct delay time. With three timers, all the timers can have different preset delay times that never have to be changed. This allows the correct timer to start based on the event that occurs without having to change the delay times. The three timers and their times are listed below as well as all the possible time states the game has. All the time states occur in both a single player game and a multiplayer game; however the implementation of the functionality may differ.

Table 6.2.1: Timers

Timer	Delay Time (ms)
t	2000
t1	500
t2	4000

Table 6.2.2: Time States

Time State	Description
-1	Displays possible movement spots after a human player rolls
0	AI player makes the roll decision
1	AI player chooses a spot to move to
2	AI player ends their turn
3	The human player is moved from one spot to the next spot
4	A player has accused correctly and sound is played
5	A player has accused incorrectly and sound is played
6	The disprove turn cycles
7	The AI player is moved from one spot to the next
8	The AI player makes a guess

6.3: State Machine

Another way to think of the game is without the timers. This will show just the state machine for a player. At the start of a turn a player is either in a room or not in a room. If the player is not in a room their only option is to roll the die. If the player is in a room they can also roll the die. In addition they could have been called into the room or the room could have a secret passage. If they were called into the room they can make a suggestion from that room. If the room has a secret passage they can use the secret passage. If they were called into a room with a secret passage the player has the option to make a suggestion or use the secret passage. After a player a player has rolled, the player is either in a room or not in a room. This state is different from the initial state because it is after the player has rolled. If the player is not in a room then the only option the player has is to end his or her turn. If the player is in a room the player has the option to make a suggestion. Once a suggestion is made the player can end his/her turn. At all states the player has the option to end his/her turn or make an accusation. The following state machine diagram does not show all the lines for ending a turn or making an accusation because it would make the diagram unreadable. Instead only the lines to make an accusation and end a turn that happen at probable state transitions are shown.

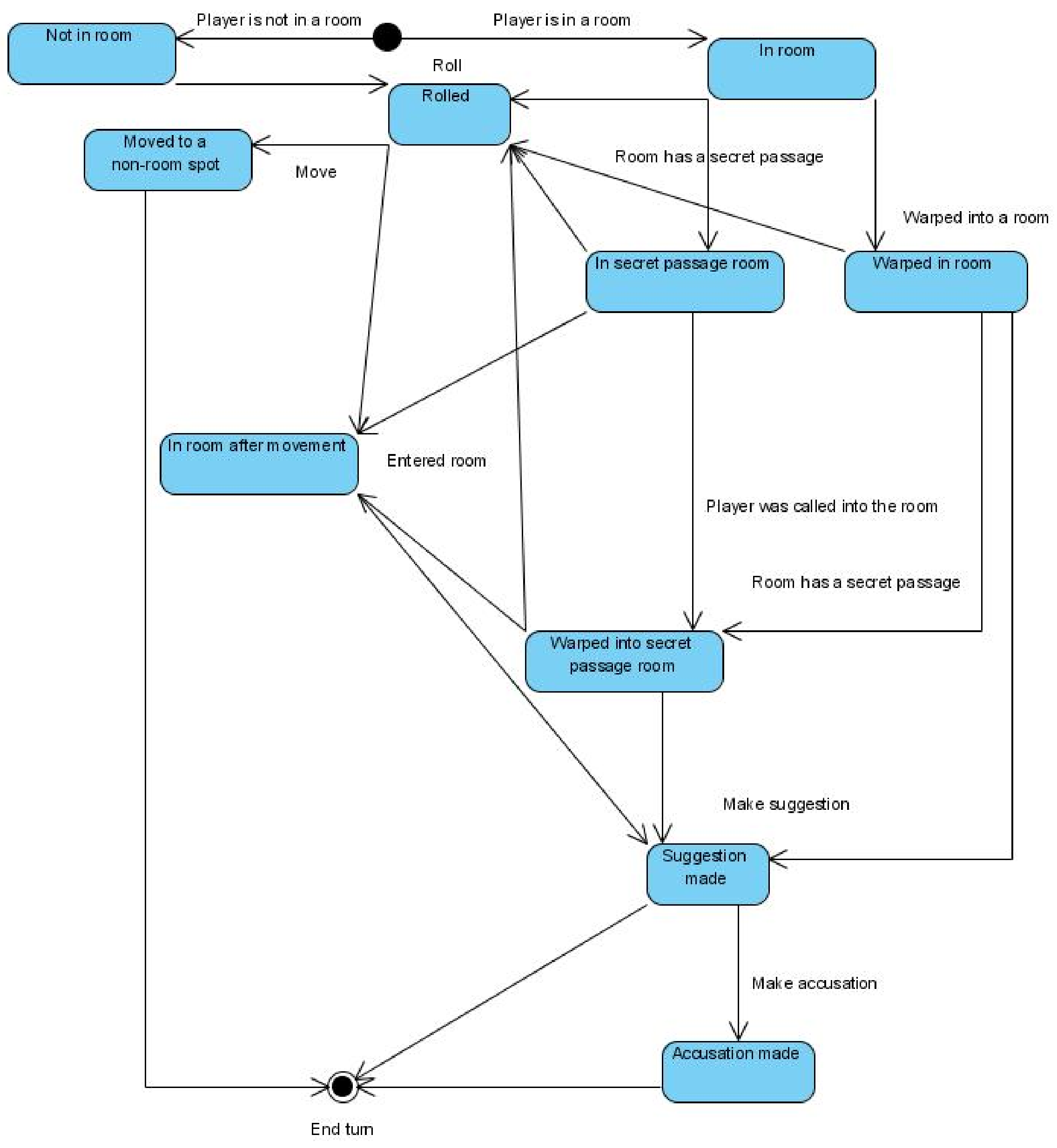


Figure 6.3: Turn State Machine

7: Artificial Intelligence (AI)

7.1: Introduction

The AI for the game was broken up into several separate parts. The parts are the roll decision, movement decision, guess decision, solution deduction, and choice of a card to disprove with. Having the algorithms broken up into different parts made the AI easier to create, since all the parts did not have to explicitly work together and could be designed separately. Some AI algorithms work together implicitly such as the movement decisions and guess decisions. The movement algorithms use the information acquired by the deduction algorithms to choose where to go. The guess making algorithms assume that a good room was chosen and it makes the best guess possible for the chosen room. All three algorithms (deduction, movement, and guessing) are entirely separate but all of them assume that each of the others is doing its job correctly. If one algorithm fails then all of the algorithms could possibly fail as a result.

7.2: Guess Array

The most important data structure used by the AI is an array of all the information that the AI has gained about the cards. This array is called guessArray[] in the actual code. The dimensions of the array are the number of players by the number of cards (guessArray[numberOfPlayers][21]). For each player the array holds information about what the respective AI player knows about each other player's cards. The table below shows all the possible values the array can hold and what each one means.

Table 7.2.1: Guess Array Values

Value	Meaning
-2	The card is one of the winning cards.
-1	The card is unknown for that player. The player may or may not have the card.
0-5	The respective player has the card.
20-25	The respective player does not have the card.

The next table shows which cards map to the card numbers (array elements) of the guess array.

Table 7.2.2: Card Values

Array Element	Card
0	Miss Scarlet
1	Colonel Mustard
2	Mrs. White
3	Mr. Green
4	Mrs. Peacock
5	Professor Plum
6	Knife
7	Candelstick
8	Revolver
9	Rope
10	Lead Pipe

11	Wrench
12	Hall
13	Lounge
14	Diniing Room
15	Kitchen
16	Ballroom
17	Conservatory
18	Biliard Room
19	Library
20	Study

An example array is shown below. The example is taken from the start of a two player game for player 0.

Table 7.2.3: Example Guess Array

Player 0 Cards			**Player 1 Cards**	
Card Number	**Value**		**Card Number**	**Value**
0	0		0	21
1	0		1	21
2	0		2	21
3	20		3	-1
4	20		4	-1
5	20		5	-1
6	20		6	-1
7	0		7	21
8	0		8	21
9	20		9	-1
10	20		10	-1
11	0		11	21
12	0		12	21
13	0		13	21
14	20		14	-1
15	20		15	-1

16	20		16	-1
17	20		17	-1
18	20		18	-1
19	0		19	21
20	20		20	-1

In the above example all the cards the player 0 has are represented as the value of 0 (the player number). All the cards player 0 does not have are represented as the value of 20 (the player number + 20). The cards for player 1 have two values at the start of the game. Player 1 either does not have the card (because player 0 has it) or the card is unknown. The value of 21 represents all the cards player 1 does not have and the value of -1 represents all the cards that are currently unknown about the player. An important note is that all the information shown above is just the guess array for player 0. This means that all the information known about all the other players' cards is from the perspective of player 0. Each player has their own guess array containing information from their perspective on all the other players in the game. Something else to note is that the guess array should be kept in a correct state. For example if player 0 finds out that player 1 does not have card 20, then the value for player 1 card 20 is set to -2. However this also means that the value for player 0 card 20 should be set to -2. Similar situations occur quite often, usually after any new piece of information is gained especially with more players. This leads into the AI deduction algorithm, which also keeps the correct state of the guess array.

7.3: AI solution deduction algorithm

7.3.1: Introduction

The deduction algorithm is broken up into several different operations and is well designed. The main function is called think(), which calls all the other functions. The other functions in order are fillGuessArray(), eliminationGuessArray(), singleMemory(), foundAll(), and foundAllButOne().

7.3.2: The fillGuessArray() function

As mentioned the correct state of the array is kept so the first function called is fillGuessArray(). What this function does is look through each player's cards for the values of -2 and the values 0-5. If it finds the value of -2 for any player and any card it will set that card's value for all the other players to -2. If it finds the values 0-5 it sets all the other player's values for that card to 20 + their player number. This keeps the guess array data in the correct state. Since the code for this algorithm is so intuitive; it is shown below.

```
public void fillGuessArray()
{
        for(int i=0; i<numberOfPlayers; i++)
        {
                for(int j=0; j<21; j++)
                {
                        if(guessArray[i][j] == -2)
                        {
                                for(int k=0; k<numberOfPlayers; k++)
                                {
                                        guessArray[k][j] = -2;
                                }
                        }
                        else if(guessArray[i][j] >= 0 && guessArray[i][j] <= 5)
                        {
                                for(int k=0; k<i; k++)
                                {
                                        guessArray[k][j] = k+20;
                                }
                                for(int k=i+1; k<numberOfPlayers; k++)
                                {
                                        guessArray[k][j] = k+20;
                                }
                        }
                }
        }
}
```

7.3.3: The eliminationGuessArray() function

Once fillGuessArray() has completed the array data is in a correct state and some basic thinking can be done. The next method called is named eliminationGuessArray(), because it works by the process of elimination. It determines if none of the players have a certain card and then designates that card as one of the three winning cards. The algorithm is broken up into three sections, one for the person, one for the weapon, and one for the room. For all the cards of each player a count is kept of each card for the value of 20-25. If the count is equal to the number of players then it means that all the players do not have that card and that card is one of the winning cards. Once again the code is easily understandable and is shown below.

```
public void eliminationGuessArray()
{
        int cardCount[] = new int[21];

        for(int i=0; i<21; i++)
        {
                cardCount[i] = 0;
        }

        for(int i=0; i<numberOfPlayers; i++)
        {
                for(int j=0; j<21; j++)
                {
                        if(guessArray[i][j] >= 20 && guessArray[i][j] <= 25)
                        {
                                cardCount[j]++;
                        }
                }
        }

        for(int i=0; i<21; i++)
        {
                if(cardCount[i] == numberOfPlayers)
                {
                        for(int j=0; j<numberOfPlayers; j++)
                        {
                                guessArray[j][i] = -2;
                        }
                }
        }
}
```

7.3.4: The singleMemory() function

The next function is called singleMemory and as the name suggests involves the memory of all the previous guesses made by all the players. The word single is in front because originally future plans were to have another type of memory that ended up not being necessary. The singleMemory operation works off of a vector that has all the guess information stored in it. The information stored is who made the guess and who disproved the suggestion. It basically works by eliminating possible disprove cards as the game goes on. For example if player 1 guesses Professor Plum with the Knife in the Study and player 3 disproves the suggestion then is it known that player 2 does not have Professor Plum, the Knife, or the Study. It is also known the player 3 has at least one of those cards. Later in the game if player 2 suggests Professor Plum with the Revolver in the Study and if player three cannot disprove the suggestion then it can be deduced that player 3 must have the Knife. The concept is similar to a system of equations such that if player 3 has A, B, or C and player 3 does not have A or C, then player 3 must have B. A nice bonus that made this method easy to code was that time was not a factor. For this application, time not being a factor meant that all possible equations did not have to be solved, since the current state held all the past information. All that had to be done is iterate through all the past guesses and compare them to the current state of the guess array. This method gave the exact same effect as solving all the systems of equations would have, but in a more ingenious way.

7.3.5: The foundAll() function

The foundAll() method determines if all of one players' cards have been found. In this case the remaining unknown cards can all be known that the player does not have them. The method works by counting all the cards the player is known to have and comparing that value to the total number of cards the player has. If the values are equal then the player cannot have any more cards. This thinking is so obvious that it is often overlooked and forgotten by most people.

7.3.6: The foundAllButOne() function

When all the cards except one are known of a group (person, weapon, or room) then the remaining unknown card must be one of the winning cards, since it cannot be any of the other cards. For each section a count of the known cards is determined as well as which cards are known. If the count is equal to the number of cards in the section -1 then the remaining unknown card is one of the winning cards.

All the simple methods combined create a powerful and perhaps perfect deduction algorithm. After all the processing is done fillGuessArray() is called again to put the guess array back into a correct sate since changes may have been done by previous steps of the algorithm. The only possible weakness of the algorithm is actually its greatest strength. The algorithm is perfect based on the facts of the game. However the algorithm only works off of facts. Probabilities based on intuition are not considered. For example if a player keeps guessing the same two cards that are known to be owned by that player, then it is almost certain that the player does not have the third card guessed. The only reason a player would guess three of his/her own cards would be to try and throw someone else off as it offers no real benefits. It is basically a wasted guess in a game where each guess is important. Unfortunately the AI does not consider this information when deducing a solution since it is not factual. If the AI did consider this information, a player could throw off the AI quite severely with a silly guess. That is why the values for not having a card in the guess array are 20-25. Originally it was planned to have values 10-15 for cards a player probably has and values of 30-35 for cards a player probably does not have, but considering such probabilities made the AI very susceptible to silly guesses. The AI should never make an incorrect accusation and therefore can only use facts in deducing a solution. The algorithm for making a guess does not have consequences and does use probability to try and obtain information much faster than random guessing.

7.4: AI Guess Decision Algorithm

7.4.1: Introduction

The guess decision initial structure is basically just a bunch of “if” statements based on important conditions. Once the correct path is chosen the guess is then formulated based on the path chosen. The two main conditions tested are what is known about the current room and what information is needed. The current room value for the guess array is a key determining factor of what type of guess should be made. If the AI player has the room card of the same room the AI player is in then the guess would have to get information on only weapons and people. The information that is needed is also a big determining factor. If the winning cards for the person and weapon are known, then the room information is the only information that is needed. A guess would be formulated in such a way to force the needed information to be given. Other factors include if the AI player is likely to be going back to the room and how many rooms are known. Most cases fall under four categories and are broken up into functions based on the categories. The functions are isolateRoom(), personKnown(), weaponKnown(), and personWeaponNotKnown().

In addition to those functions the basic conditions, that decide which of those functions to call, also use functions to simplify the code. The functions that determine whether a condition is true or false include needPerson(), needWeapon(), and willGoBackToRoom(). The needPerson() and needWeapon() conditions are simple. They look through the guess array for a value of -2 for a person or a weapon. If either function finds a value of -2 for their respective group it returns false, meaning that because it knows the winning card then it does not need to look for it for the respective card group. The willGoBackToRoom() function uses the current room value of the guess array as well as what information is ultimately needed to determine if the player might go back to the room again in the future. For example if the room is unknown and the player needs to find the winning room card then the player might go back to that room to try and get the information needed. Another example would be if someone else has the room card for that room. Then it would be a waste to go back to that room because it is already known that someone else has it. In this case the player would not likely go back to the room unless it is in the path to another more desired room.

The personKnown(), weaponKnown(), and personWeaponNotKnown() functions are all similar and only the personKnown() function will be discussed. The personKnown() function is called when the winning person card is known, likewise the weaponKnown() function is called when the winning weapon is known. The personWeaponNotKnown() function is called when neither the winning person card nor the winning weapon card is known.

7.4.2: The personKnown() function

When the winning person card is known a guess has to be made to figure out the weapon. Note that the room suggestion always has to be the room the player is in so the is no decision for the room. The algorithm goes through some preliminary setup to get the information needed then has two steps in choosing the cards (one step for each card). The information gathered is which weapons are known and how many people do not have each weapon. Based on this information a priority group is assigned to each weapon. A priority group is only assigned to weapons which are not known. A priority group of 50 is assigned to weapons where all but one player does not have. This is the highest because if the last player does not have that weapon card then that card is one of the winning cards. A priority group of 30 is assigned to all other weapons where the number of people who do not have the people is greater than one. This is because it is more likely that those weapons are the winning cards since there are less people that could have them compared to the priority group of 20. Priority group 20 is that last group where the number of people who do not have the weapon is equal to one. This means that the current AI player only knows that it does not have the weapon and knows nothing else about any other player with regards to that weapon. This is the lowest priority group because the maximum number of people would need to not have the card in order for it to be the winning card.

Once the priority group values are set for all the weapons a random number is chosen within the group range. If all three ranges were in use then the random number range would be 1-100. If there were no weapons with the priority group of 30 then the number range would be 1-20 and 50-100. The random number decides which card is chosen to be guessed. The variables involved in the calculation are the random number, card to choose, increment, and base. The increment is calculated by taking the range and dividing it by the number of cards in that range. If 3 cards had a priority group of 50 then the increment would be 50/3 = 16 (truncated). This gives all three cards an equal chance to be chosen from the group. The base is where the numbers in the range for the priority group start. The first group of 20 is from range 1-20. The base is this case is 1(actually it is 0 and then adjusted to 1). Finally the card to choose can be calculated in the following formula.

```
choiceSpot = (priorityGroup - (base+1)) / increment;
```

The line of code is the actual formula used. The choiceSpot is the spot within the priority group of the card. The priorityGroup variable is actual the random number. What happens is the random number needs to be converted to an array spot of one of the cards. The base is used to lower the random number to the range of values starting at zero. The increment is used to divide into that number to give all cards (array elements) an equal chance. After all the converting and calculations the weapon choice is made. The weapon array is iterated and the currentSpot variable is incremented each time a weapon is in the chosen priority group. When the currentSpot is equal to the choiceSpot the weapon guess is set to the array element + 6.

An example is shown below of how the algorithm works.

Table 7.4.2.1: Initial Information

Initial information			
Weapon Status		*Number of players who do not have the card*	
Array element (card-6)	**Known**	**Array element (card-6)**	**Number**
0	TRUE	0	N/A
1	FALSE	1	3
2	TRUE	2	N/A
3	TRUE	3	N/A
4	FALSE	4	2
5	FALSE	5	5

The initial information requires no calculation and is gathered from the guess array for this example. The weapon cards normally occur after the people cards so the actual array element refers to the weapon card -6(the number of people cads). The values for the number of people who do not have the card are N/A for all known cards, because known cards will not be guessed.

Table 7.4.2.2: Guess Priority

Guess Priority	
Array element (weapon)	**Priority Group**
0	N/A
1	30
2	N/A
3	N/A
4	30
5	50

The priority for each weapon is calculated above. For all the cards with the number of players who do not have the card equal to N/A also have a priority of N/A. Elements 1 and 4 have a priority of 30 because the number of players who do not have the card is greater than 1 and less than the total number of players -1. Element 5 has a priority group of 50 because all the players except one do not have the card (six players assumed).

fiftyCount = 1

thirtyCount = 2
twentyCount = 0

priorityGroup = 27

base = 20
increment = 30/thirtyCount = 15

choiceSpot = (priorityGroup - (base+1)) / increment;
choiceSpot = (27 - (20+1)) / 15;
choiceSpot = (27-21) / 15
choiceSpot = 6/15
choiceSpot = 0

Array element (weapon) chosen = 1
Card chosen = Array element (weapon) chosen + 6
Card chosen = 7 = The Candelstick

The above calculations are straight forward from the rules described earlier. The priorityGroup variable was randomly chosen. The weapon chosen is 1 because the choiceSpot is 0; therefore the first weapon of the selected priority group is chosen. Since the priorityGroup (random number) is 27 the group to choose from is 30 because the priorityGroup variable is in the range of 21-50. This weapon choice translates to card number 7, which is the Candelstick.

Now that the weapon is chosen, the person can be chosen in the second step of the algorithm. Since the person winning card is already known, the person guess will have to be a card that no other player can disprove with. That will force the other players to give the weapon information or the room information. An initial count of all the person cards is done to see how many person cards the AI player owns. If the AI player does not have any person cards then the person to guess is going to be the person winning card. This is because the person is winning card is the only card the AI player can guess with to force other information. Note that this assumption works because if the person winning card was not known then this function would not have been called. If the AI player has one or more person cards then one of those cards is chosen at random. After this is done all three cards making up the guess are chosen and the guess decision is complete.

The functions weaponKnown() and personWeaponNotKnown() work similarly to the personKnown() function as described above. The weaponKnown() function works exactly the same except the person and weapon cards are reversed. The first part of the algorithm will involve the person cards and the second part will involve the weapon cards. In the case for personWeaponNotKnown(), the first part is done with both cards. These three functions cover most scenarios involving a guess making decision. The other scenarios involve isolating the room information and getting any information possible from a room that someone else already has.

7.4.3: The isolateRoom() function

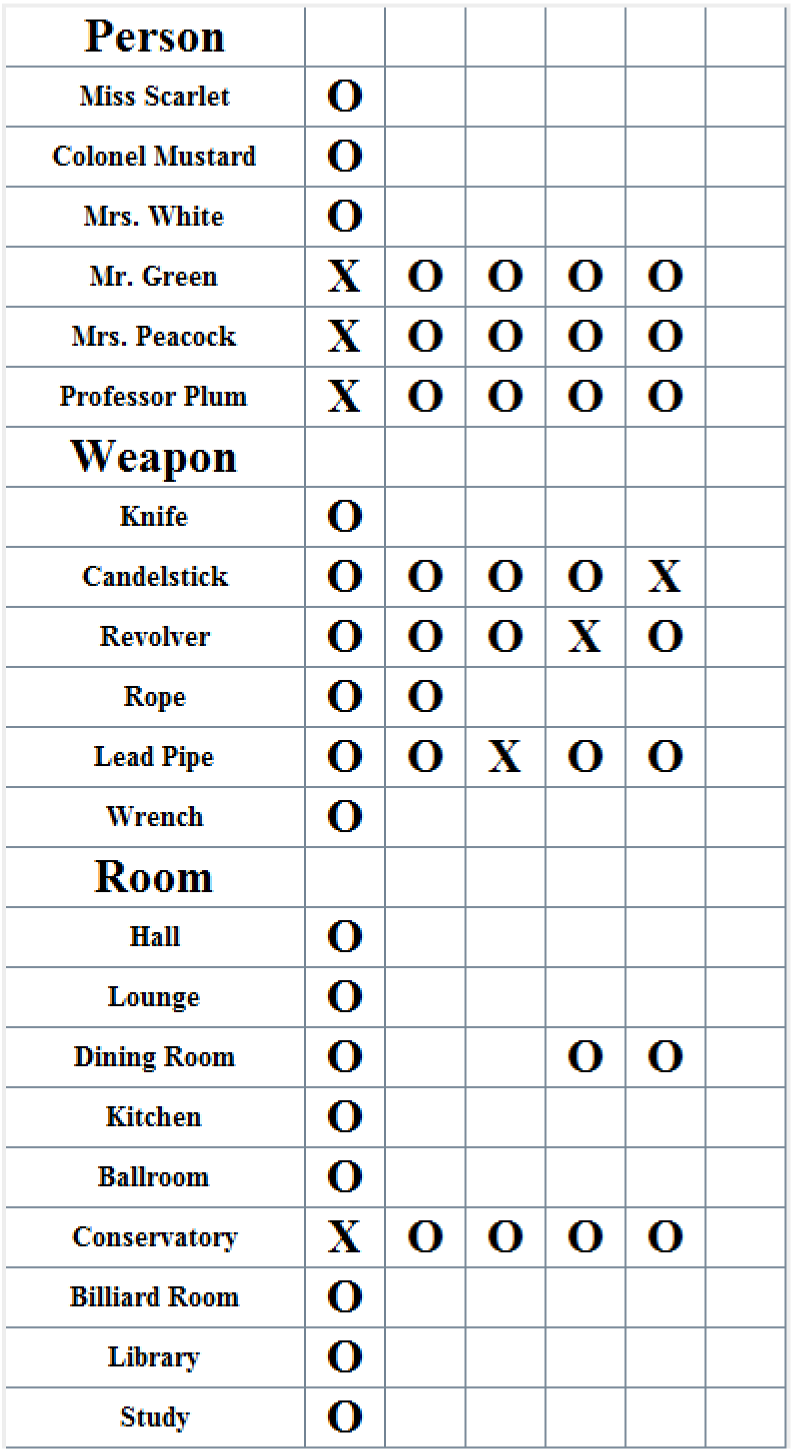
The room card is the most important card to figure out because it can only be guess if the player is actually in the room. There are also three more room cards than there are person or weapon cards (9 rooms, 6 person/weapon cards). There are two situations where the room information is forced. The first and most obvious is when the AI player only needs the room information. The second is based on probability. Even if the person and weapon cards are not known there is still a chance that the room information will be forced. As the number of rooms known increases the chance that the room information will be forced also increases. The player will want to force the room information and move on to the next room with few rooms remaining. The chance the room information will be forced is called isolatePercentage in the actual code, because it is the percentage that the room card will be isolated for information. A table is shown below of the isolatePercentage numbers based on the number of rooms known. The isolatePercentage is compared to a random number between 1 and 100. If the random number is less than the isolatePercentage then the room will be chosen to be isolated and the isolateRoom() function is called.

Table 7.4.3.1: Isolate Percentage Values

Rooms known / isolatePercentage	
Number of rooms known	isolatePercenatge
<= 2	30
3	50
4	75
>4	100

What basically happens in the isolateRoom() function is that a room and weapon card that the player owns are chosen. This will force the other players to disprove with the room card if they have it. Here is where the situation arises of what if the room cannot be isolated. In other words the current AI player does not own any person or any weapon cards to force the room information. This situation also occurs when a player is in a room where another player is known to have that room card. In that case a guess must be made to try and get any information before it becomes the person with the room card's turn to disprove.

The algorithm starts by searching through the players to see what the farthest player away from the current player has their room information unknown. This is what the person and weapon guess will have to pass in order to isolate the room. The same method is done with each person and weapon card. One of the person or weapon cards will be chosen out of the ones that can be chosen in order that the room information will be obtained. A walkthrough of the algorithm is shown in the following example with five players with player 1 making a guess from the Dining Room. The CheckList screen is used to shown the information from the perspective of player 1.



Person						
Miss Scarlet	O					
Colonel Mustard	O					
Mrs. White	O					
Mr. Green	X	O	O	O	O	
Mrs. Peacock	X	O	O	O	O	
Professor Plum	X	O	O	O	O	
Weapon						
Knife	O					
Candelstick	O	O	O	O	X	
Revolver	O	O	O	X	O	
Rope	O	O				
Lead Pipe	O	O	X	O	O	
Wrench	O					
Room						
Hall	O					
Lounge	O					
Dining Room	O			O	O	
Kitchen	O					
Ballroom	O					
Conservatory	X	O	O	O	O	
Billiard Room	O					
Library	O					
Study	O					

Figure 7.4.3.2: Player 1 Checklist

In this example the cards player 1 knows that someone has are marked with an X under the column of the player who has the card. The cards that player 1 knows someone does not have are marked with an O under the column of the player who does not have the card. All blank spots are unknown to player 1, whether that player has the card or not.

The first thing the algorithm does is look to fine the farthest player that player 1 needs room information from. As shown in the CheckList in order to get all the information on the Dining Room the suggestion must get past player 3. Player 3 has the last unknown spot from player 1 on the Dining Room. This number (3) is stored in the variable farthestUknown in the actual code.

The person choice is easy since player 1 has three people to choose from. Since the three person cards are owned by player 1, they can all be chosen. The other three person cards have no information and cannot be chosen to isolate the room information, since any player could have any of them. A random number between 0-3(exclusive) is chosen. To give a concrete example, let's say the number 2 meaning that Professor Plum was the card selected.

The weapon choice is not as easy. Player 1 does not own any weapon and must choose a weapon card to force the room information. Each weapon card is looked at to see how far each weapon can go in the disprove cycle before it could be used. The knife and the wrench are unknown for player 2 and therefore can only go as far as player 2. Since the farthestUknown variable is 3 neither the knife nor the wrench are valid choices. The rope is first unknown for player 3, but this is also not a valid choice because the number must be greater than the farthestUknown variable. Likewise player 3 has the lead pipe, making it an invalid choice. The revolver and the candelstick are owned by players 4 and 5 respectively. Both players 4 and 5 are greater than the farthestUknown variable, which is 3. This means that either the revolver or the candlestick could be used to get all the information needed on the Dinging Room. A random number between 0-2(exclusive) is chosen. To give a concrete example, let's say the number 0 meaning that candelstick was the card selected. This results in the final guess being Professor Plum with the Candlestick in the Dining Room.

There are times where it is not possible to isolate the room. In this case a count is done for all cards in the group with only the unknown status. This at least gives a chance to isolate the room information. Then a random card among those choices is chosen. These situations are less likely because the movement decisions almost always put the AI player into a room where it can make a good guess from.

7.5: AI Roll Decision

The roll decision can be either to roll the die, make a suggestion, or use a secret passage. It is influenced by a few main factors. These factors are if the player is in a room or not, if the player was called into the room, and if the room has a secret passage. If the player is not in a room then the player cannot make a suggestion or use a secret passage so the only choice is to roll. Likewise if the player is in a room, but was not called in and the room does not have a secret passage, then the player also must choose to roll the die. If the room does not have a secret passage but the player was called into the room, the AI player has the option to make a suggestion or roll. The AI player will then consider the information about the current room that it knows and the information that it needs to obtain. If it makes sense to guess then the AI player will make a suggestion, otherwise it will choose to roll. The AI player also makes a similar decision for being in a room with a secret passage, but not called in. If the room that the secret passage leads to is a good choice to make a suggestion from then the AI will choose to use the secret passage, otherwise it will choose to roll. The last case is when the AI player has been called into a room with a secret passage. This is when the AI has all three options available. It can roll, make a suggestion, or use the secret passage. The AI will try to avoid rolling since it takes the most time to move. The AI will again consider the information of the current room, the secret passage room, and the close rooms. It will make the best decision on what to do considering the information the AI player needs to acquire. This is also where some functions of the BoardMap class are used to determine the closest path to a desired destination. That information is used to decide whether it is best to roll or use the secret passage to reach the destination. The BordMap class functions are explained in the BoardMap game algorithms section of the paper.

7.6: AI Movement Decision

7.6.1: Introduction

If the AI player has chosen to roll the die, the player must then decide where to go after the die has been rolled. This is the most complicated decision in the game and required a lot of additional help. The BoardMap class was designed specifically to help get the information needed for this decision. The main function in the code is called chooseSpot() and takes the number rolled as an input parameter. The additional function chooseSpot2() does most of the work where chooseSpot() makes the generic decisions. A generic explanation is shown here and the methods that these methods use from the BoadrMap class to get board information are explained in the other algorithms section.

7.6.2: The chooseSpot() function

The first thing done is to gather all the possible spots the AI could choose from. This is done by using the BoardMap getPosibleSpots() function. All the possible spots are stored as well as all the possible door spots. If any of the rooms are good choices to go to then the spot choice will be the corresponding door spot. If there are no rooms to choose from or all the rooms are bad choices then the chooseSpot2() function is called.

7.6.3: The chooseSpot2() function

The chooseSpot2() function starts by finding the four closest rooms and the distances to each one by die roll by using the BoardMap class methods. If the rooms are too far away then sentinel values are used such as a distance of 1000 so that these values will not be used later in the decision. For each room a search is done to find the closest room to that room that the AI player will want to go to. The result of the search returns the closest room in the path to the closest room for each room. If there are any duplicate rooms then the distances of the duplicates are set to zero. This is done because the distance to the closest room and the distance to the closest room to that room are added together. The room with the closest distance is stored. If any of the original distances are less than the set of combined distances that value is then stored as the closest distance. Now the AI player has made the decision on where to go. However it still must decide how to get there. All the possible spots and their distances to the chosen room are calculated and the closest spot is chosen. After that a simple test is done to see if the AI player would rather go back to the same room if it is in a room. If it does then a spot outside the door is chosen instead.

7.7: AI Choosing a Card to Disprove With

The simplest decision the AI makes is what card to disprove a guess with. The first thing the AI player checks for is to see how many cards it can disprove with. If there is only one option then that card must be chosen. If there are two or three options then the AI proceeds to the next steps of the algorithm. If the AI player has already shown that player one of the choices it will show that card again. This prevents the suggesting player from gaining any new information unnecessarily. If that is not possible the algorithm moves on to the next step. It then looks to see if any of the cards being guessed have already been shown to over half the players in the game. If this is true for any of the cards then that card is chosen to disprove the suggestion with. The reason for this is to prevent new information from being gained by the players. Since the card is known to over half the players then the AI player might as well choose it to show to another player. This is the correct choice is most cases, but can backfire. The AI does not know what each player's cards are from the other player's perspective. This means that the suggesting player may only need the card that everyone else already knows. In this case the algorithm does not show the optimal card. The algorithm continues to the last step if none of the previous conditions are met. The last step randomly generates a percentage. The percentage is roughly 80% for person and weapon cards combined and 20% for room cards. This means that there is roughly a 20% chance the AI player will choose the room card to disprove with since in most cases the best choice is not to show the room card since it is the hardest to get. Idealistically the room card should never be shown if it does not have to be in most cases. The problem with that is if the person and weapon percentage was 100% then other players could get additional information from the AI. If the AI chose to disprove with the room then the other players would know it does not have the weapon or the person if their percentages were kept at 100%. Therefore the percentage is kept at 80%, which is high but not high enough to assume other information. Out of the combined 80% for the person and weapon cards roughly 40% is assigned to each the person and weapon card. All percentages are not exact due to the randomness they were generated with; thus making the disprove choice more random, but still confined to certain values.

7.8: AI Conclusion

The five separate AI algorithms make the AI as a whole pretty good. In almost all cases the AI player will make the correct choice. In some rare situations the AI player will make a valid choice but perhaps not the best one. The strength of the AI is that it is extremely smart when choosing what to do using a lot of information. Most players would not consider some of the information the AI does. This also means that human players know that the computer players will not make a silly decision and might be able to deduce some additional information from the decisions it makes. Another weakness is that the AI does not have all the information. It has a lot of information but is missing some that other human players easily consider. This includes what cards everyone has from another player's perspective. This is an important piece of information because the AI can make some assumptions on what other players have based on what they are doing from what they know. The reason this was not implemented is because it would mean the AI would be working off of assumptions. The AI would be assuming a player is trying to get a certain piece of information or move to a certain room. For example if player 1 knows that player 2 only needs the room card, then whenever player 2 makes a guess from a room unknown to that player, no other player will have the person and weapon cards in the guess. This means that player 2 either owns those cards or those are the winning cards. This is a highly probable assumption, but if the AI did this then human players could start making bad guesses to throw off the AI. The difference between human players and AI players is intuition to figure out when and how to use the assumed additional information. Coding AI intuition would not have been practical because it does not add much to the strength of the AI algorithms and would have taken a long time to code well functioning intuition. This weakness gives all human players a slight advantage against the computer players. This is not that bad because people like winning more often than losing.

8: Other Important Algorithms

8.1: Introduction

The GameManager class and the BoardMap class provide several algorithms related to the game. The GameManager class algorithms handle much of the game board data manipulation such as movement. The BoardMap class methods also handle movement but not the actual movement. The BoardMap class uses a copy of the game board array to work off of so no actual changes are made. This is because the BoardMap class is used only by AI players when thinking about where to move. They need to work the game board but do not want to change it. So a copy of the game board is used instead. A list of the most important algorithms is shown below.

- GameManager class
  - calculateMoves()
  - findPath()
- BoardMap class
  - findClosest()
  - findClosestRooms()

8.2: The calculateMoves() method

The calculate moves method is used by the GameManager class to get all the possible spots a player can move to in a given turn. It takes the number of the die rolled, the x position, the y position, and whether not it is being called recursively as parameters. The function definition is "public void calculateMoves(int dieRoll, int px, int py, boolean firstSpot)". It works by taking the current spot and looking down, right, up, and then left in that order to see if those spots are valid movement spots. If they are the calculateMoves() function is called recursively from those spots with the die roll decremented. The same method continues until the die roll is zero or there are no more valid spots to move to. The function then returns. All the recursive calls can update the stack that has all the possible moves. When the recursive part is done, every spot on the stack is removed and those spots are set to a green border indicating that they are valid choices for the player to move to. The first call to calculateMoves() then returns back to the GameBoard class that called it allowing the player to continue to choose one of the spots with a green border to move to. A walkthrough of an example is shown below. The number rolled is two.

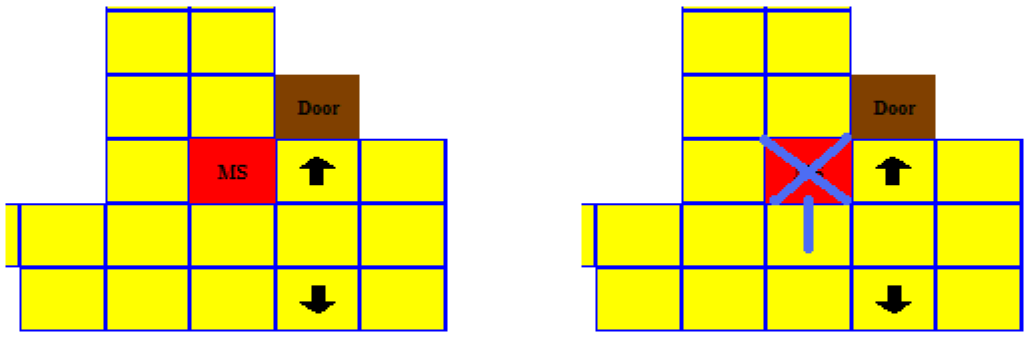


Figure 8.2.1: Calculate Moves 1

The algorithm starts at the spot of the player who is Miss Scarlet in this example. The image on the left is what the game board actually looks like before the algorithm starts. The image on the left shows what the algorithm is doing. The blue X indicates the current spot the algorithm is working from and the line indicates the spot where the algorithm is looking to. From the current spot on the board the algorithm looks down to see if that spot is valid. In this case it is, so the calculateMoves() function is called again with die roll = 1 and that spot is added to the stack.

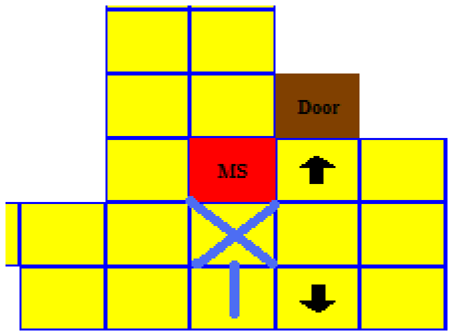


Figure 8.2.2: Calculate Moves 2

Now the algorithm's does the exact same thing. It looks down from the current spot to see if that spot is valid. That spot is also a valid spot, so the algorithm again makes a recursive call to itself with the die roll = 0. Since the die roll is zero the next call will just return so the image for that step is not shown. The next step is to look to the right to see if that spot is valid.

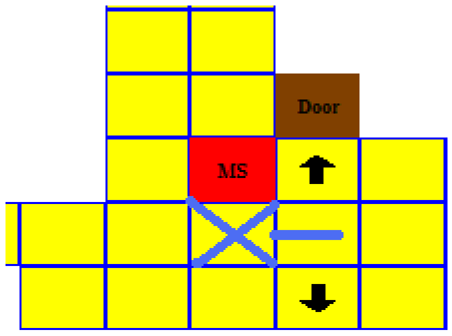
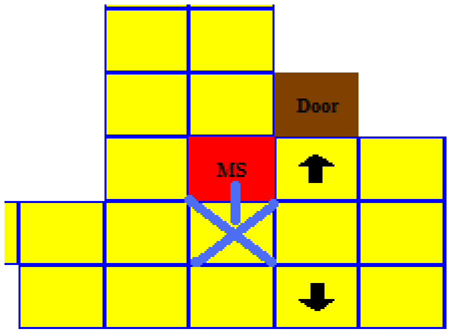
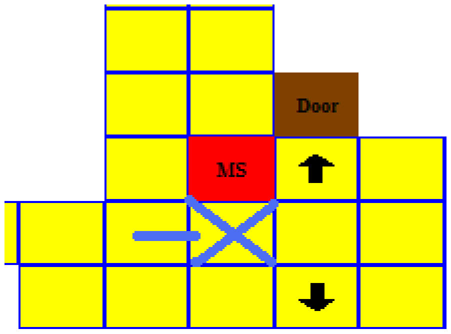


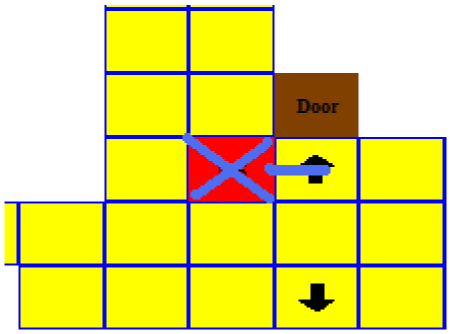
Figure 8.2.3: Calculate Moves 3

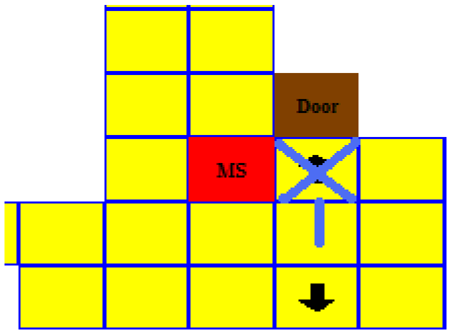
The spot to the right is valid so that spot is added to the stack and the algorithm calls itself again with the die roll = 0. Once again since the die roll is zero it will just return.

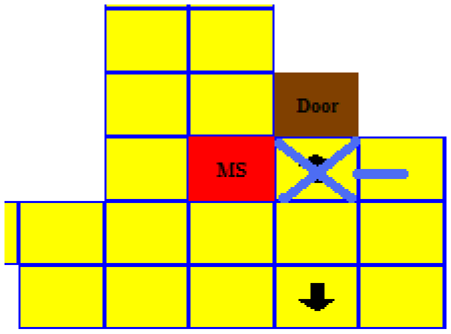
The algorithm continues using this method until all recursive calls return and all directions are checked for each call. The next images show in order how it works continuing from where it left off.

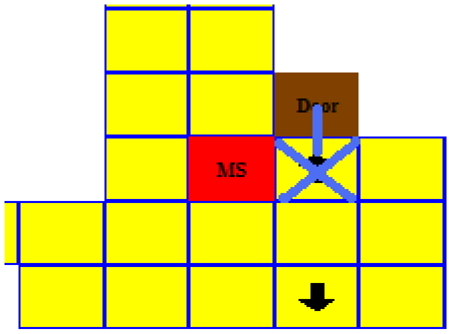


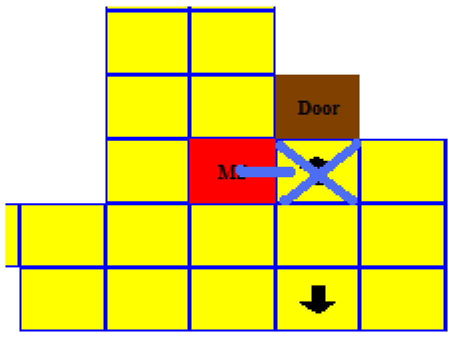


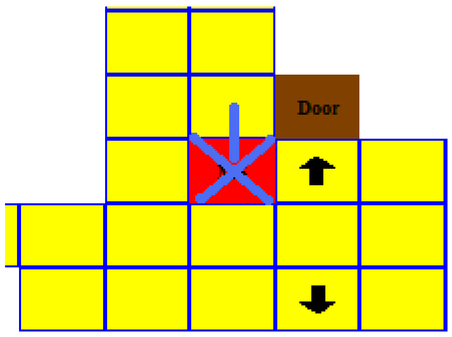


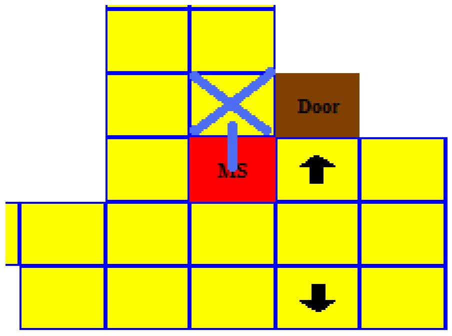


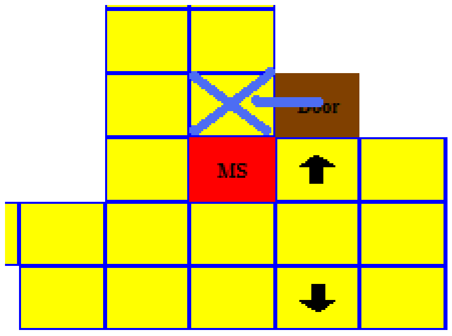


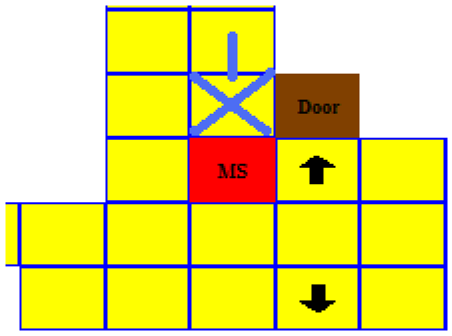


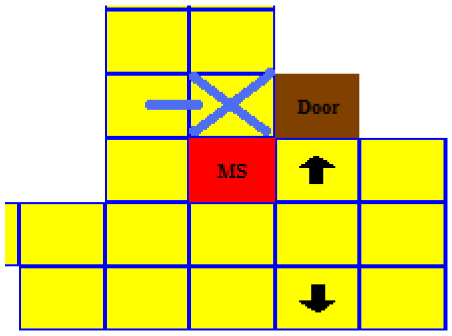

Door
MS

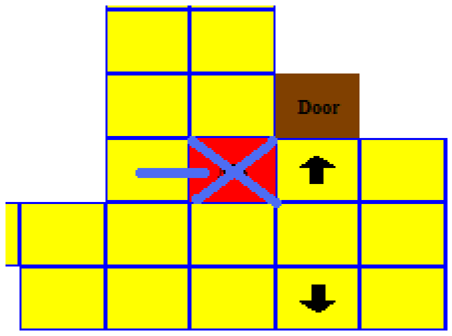

Door
MS

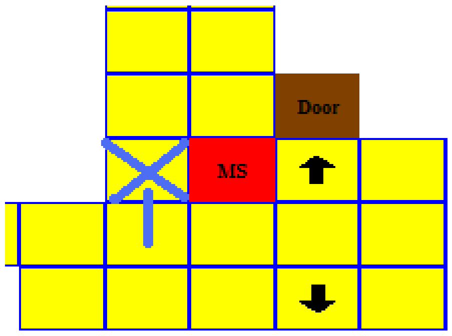

Door
MS


Door
MS


Door
MS


Door
MS


Door
MS


Door
MS

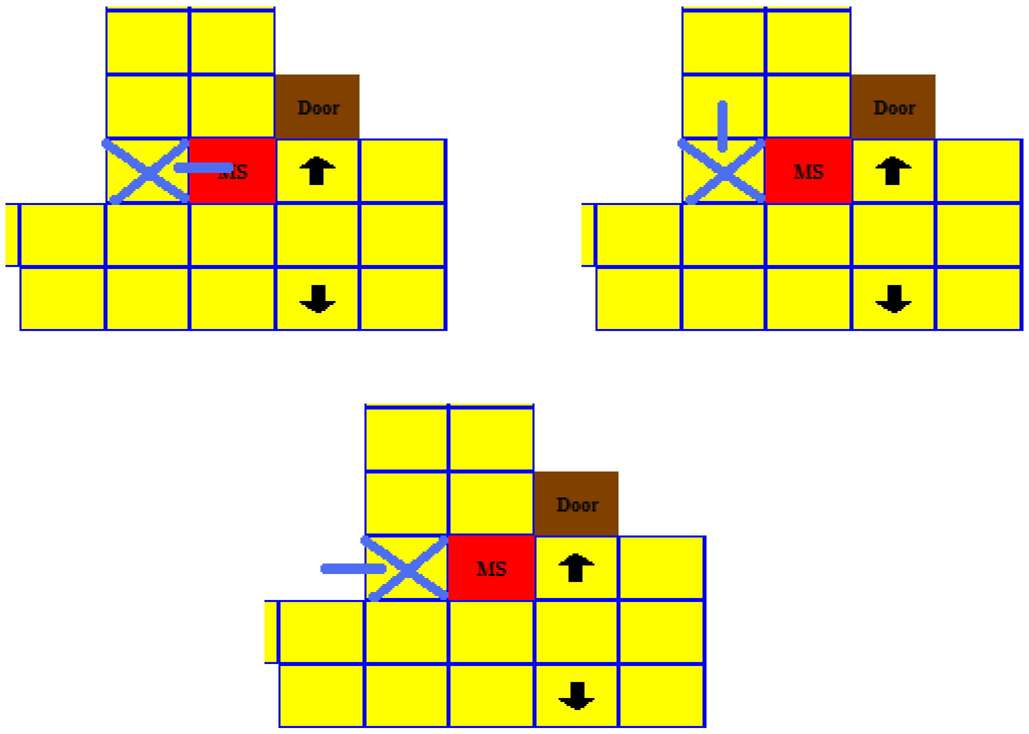


Figure 8.2.4: Calculate Moves 4

As you can see a lot of work is done to determine all the valid moves; hence why an example with the die roll only equal to two was chosen. At this point all the valid spots are added to the stack. The image below shows the order in which they were added.

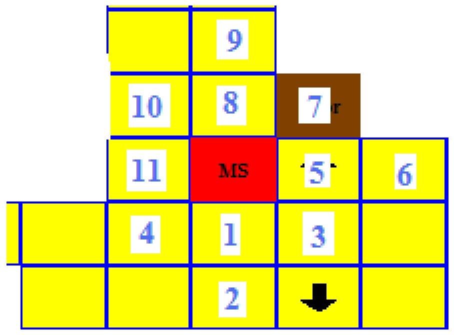
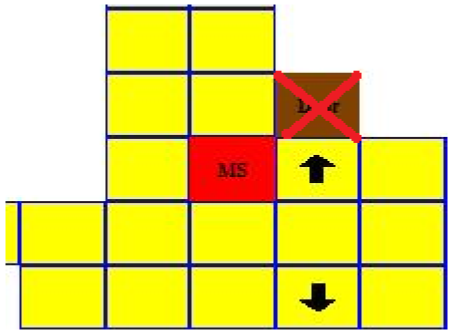
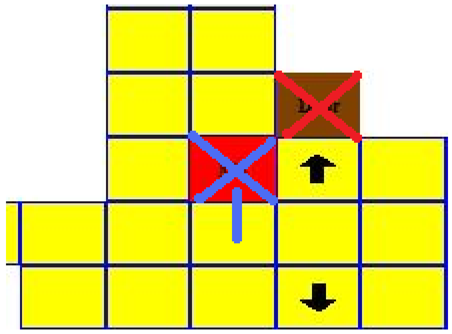


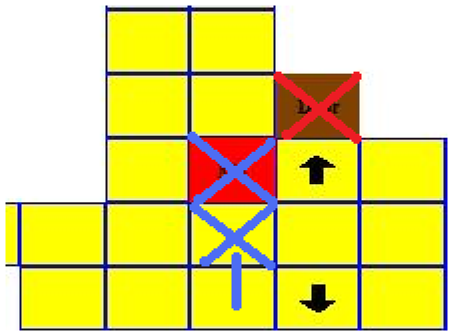
Figure 8.2.5: Calculate Moves 5

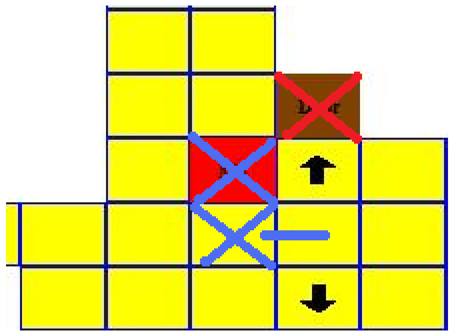
8.3: The findPath() method

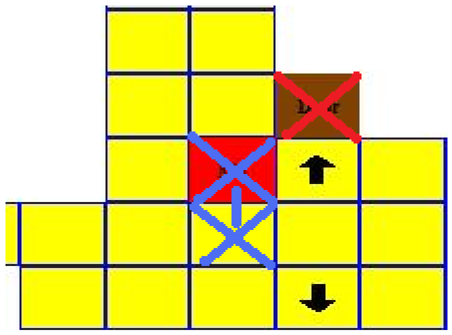
After a valid spot is chosen the findPath() algorithm is used to find a correct path to that spot. The findPath() method takes the current spot, destination spot, and die roll as parameters. The function definition is "public boolean findPath(int x1, int y1, int x2, int y2, int roll)". Ultimately all the spots in the path are added to a stack. When the algorithms is finished the GameBoard class will move the player to each of these spots as they are popped off the stack until the destination is reached and the stack is empty. The findPath() algorithm uses a similar recursion technique as the calculateMoves() method. It returns true if the destination is reached and false if the die roll is zero. The result of the method is used to determine if the current spot on the stack should be there. Of the method returns false then that spot is not part of the path and that spot is removed from the stack. A walkthrough of the previous example is shown for the findPath() method. The desired destination is marked as a red X and the spots in the path are marked as a blue X. The blue line indicates where the current spot is looking to go.

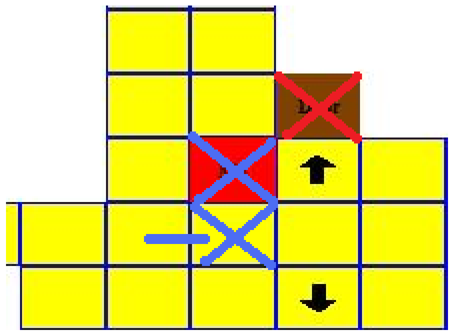












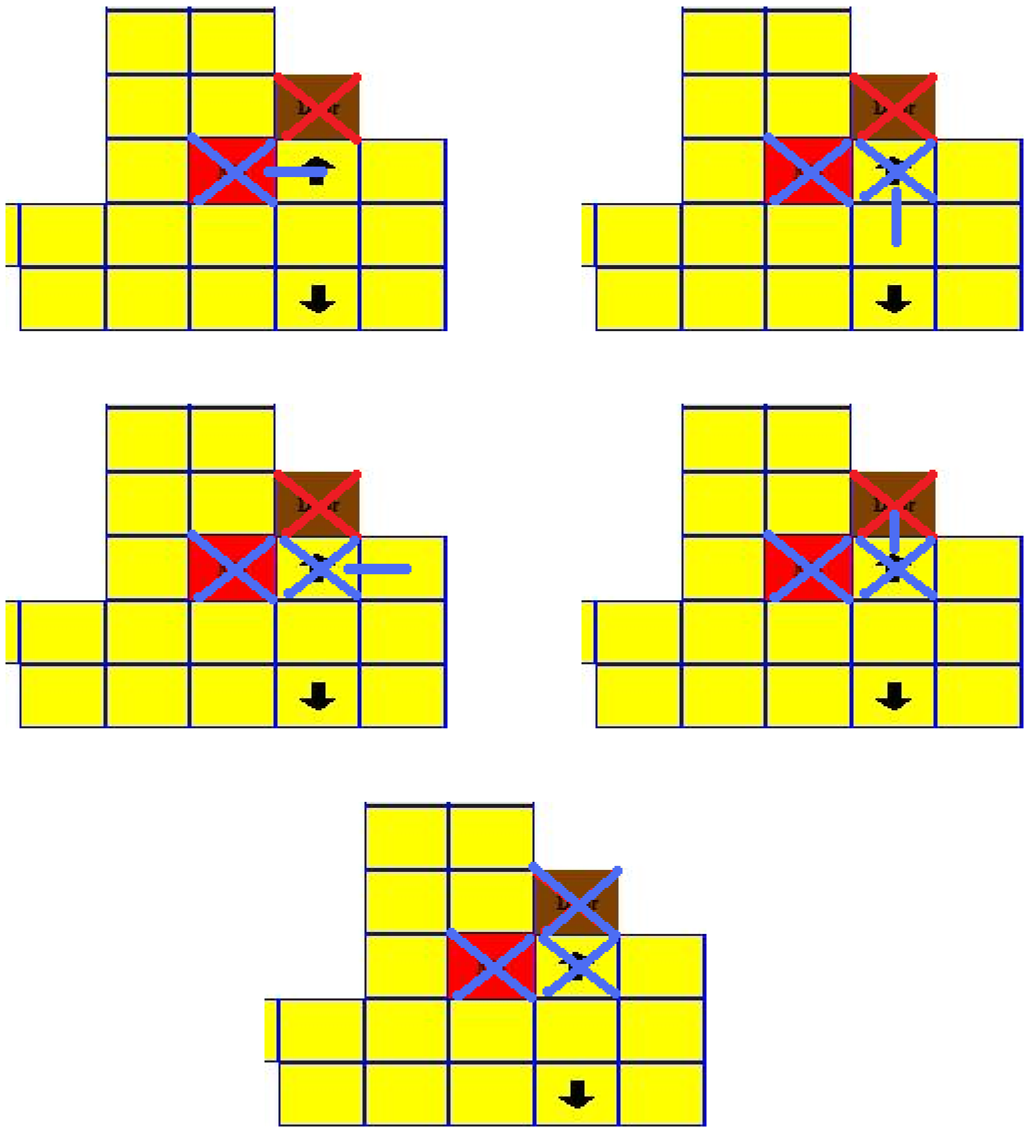


Figure 8.3: Find Path

The algorithm looks for spots the same way the calculateMoves() method does. Essentially it works counterclockwise and recursively from the current position. It is finished when it finds the destination through a correct path. Another nice thing about the algorithm is that is guarantees that all the spots are added to the stack in order. This was done by adding all spots to the stack before checking if they are valid or not. If the return value of the recursive call is true then the spot is kept on the stack, otherwise it is removed.

8.4: The findClosest() method

The findClosest() method in the BoardMap class is for the AI class to use in order to find the closest room in the path to the closest room. It is used as a part of the AI function to calculate moves to get data. It is needed because the computer may need to get to a certain room, but that room is not close. This method returns the closest room in the path to get to the desired room. With the closest room returned, the computer player can attempt to move towards that room, which would be the next step in its path. This way the computer player will not get lost trying to find a room. It will always keep going to the closest room in the path to the desired room until it reaches the desired room. This method takes two parameters. One parameter is an integer that represents the current room to run the algorithm from. The computer player does not need to be in a room to use this algorithm. If the computer player is not in a room the AI functions actually find up to four close rooms in rolling distance and use all four of those with this function. The second parameter is a Boolean variable representing if the computer player is looking for unknown rooms or not. If the computer player already knows the winning room card then the computer player does not want to go to unknown rooms. The desired room must be the winning room or a room owned by the player.

The algorithm starts by searching through all the rooms and adding all the rooms that are desired to a stack. This is dependent on the Boolean parameter passes to the function and determined by what knowledge the computer player has in its guess array. The stack name is called the unknownStack and is a misnomer. The purpose of the function was originally just to find unknown rooms and was later tweaked to find known rooms too. The next step finds the distance in number of turns from all the rooms in the stack to the current room. All distances from one room to another are in a lookup table called roomArray[9][9][2] that is initialized in the constructor of the BoardMap class. All rooms that have the minimum distance are added to another stack called the roomStack. Finally a random number is generated from zero to one less than the number of rooms in the roomStack. The value is used as the destination room index value. The function returns the room value from the roomArray[roomIndex][destinationIndex][1].

8.5: The findClosestRooms() method

Another important method in the BoardMap class is the findClosestRooms() method. The findClosestRooms() method is used by the AI class to find the four closest rooms by roll from the current spot on the board. These results can be used by the AI class to use when calling the findClosest() method if needed. The findClosestRooms() method takes no parameters and returns a two dimensional integer array containing four rooms and the distances to each one. The algorithm is shown with an example below.

First we initialize the array closestRooms to the initial values. For all four rooms we initialize the room number to -2 and the distance to 1000. We also create a new stack, which the call to closestRooms_calculateMoves() will use.

```
closestRooms[0][0] = -2;
closestRooms[0][1] = 1000;

closestRooms[1][0] = -2;
closestRooms[1][1] = 1000;

closestRooms[2][0] = -2;
closestRooms[2][1] = 1000;

closestRooms[3][0] = -2;
closestRooms[3][1] = 1000;
```

s = new Stack.

In this example the player is Miss Scarlet and the player is in the Dining room. An image is shown below to show the section of the board the algorithm will be working with.

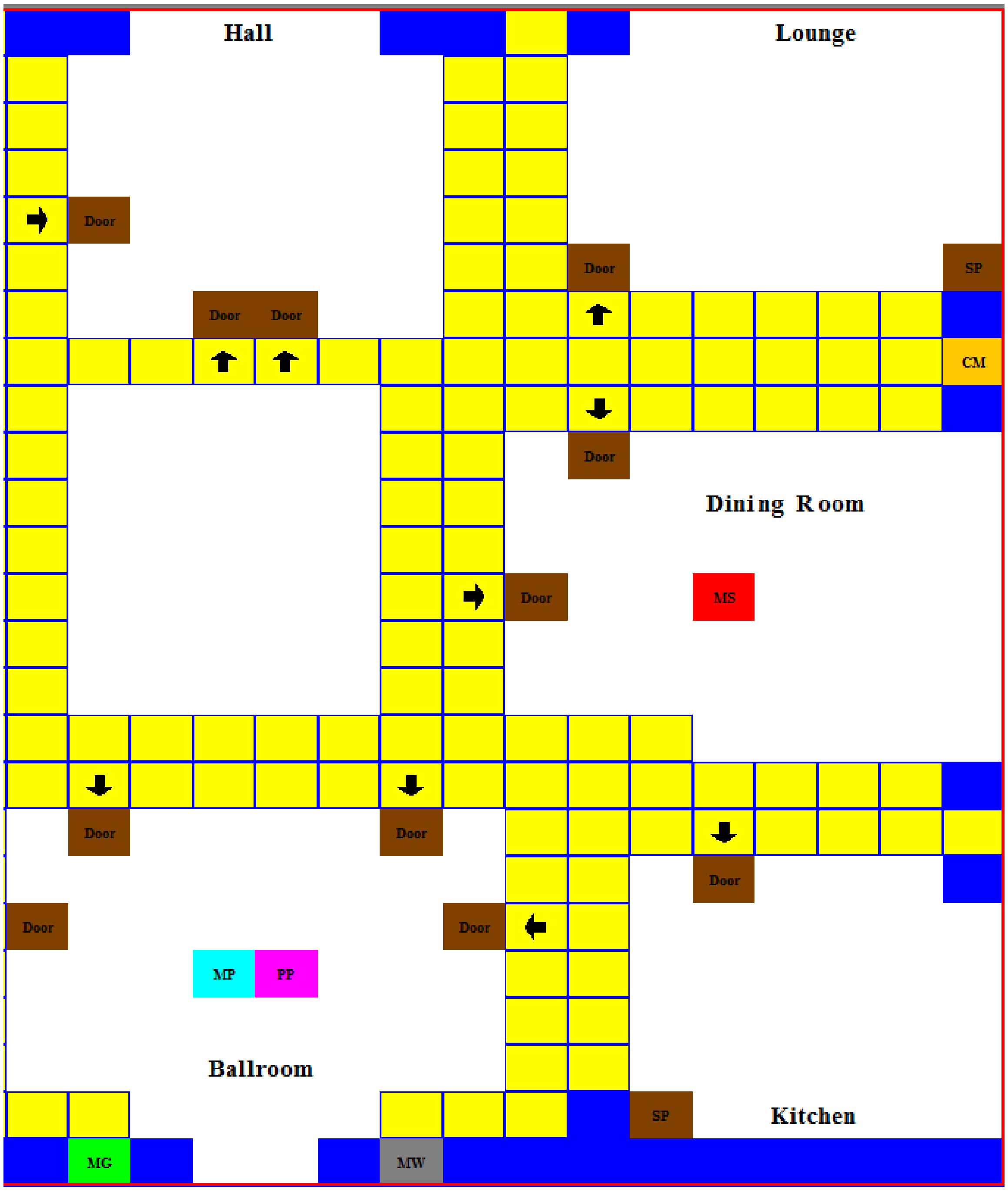


Figure 8.5.1: findClosestRooms() 1

Since Miss Scarlet is in a room a new stack has to be created for all the door spots to work off of. The stack doors will hold all the doors spots as points. The stack is shown below with the doors added to it. The top door is on the top of the stack. The x value of the point is the row and the y value of the point is the column of the door spot.

```
doors
Point(9, 17)
Point(12, 16)
```

For each door the algorithm holds the dieCount and the roomCount to go through the main loop. The main loop executes until either the roomCount exceeds four or the dieCount exceeds 13. This means the loop ends when there are four rooms or the distance to search for a room has gone past thirteen. After each iteration the dieCount is incremented.

```
dieCount = 0
roomCount = 0
```

For each iteration the method closestRooms_calculateMoves() is called. This is similar to the calculateMoves() function of the GameManager class. All the new possible moves are added to the stack s, which was initialized earlier. For each dieCount this method is called to further populate the stack. The stack is then searched to see if any of the values on the stack are doors to rooms. The next image shows the spots on the board that have been added to the stack. They are numbered with the iteration that they were added on. Note that this is only working from the top door right now and has not considered the other door.

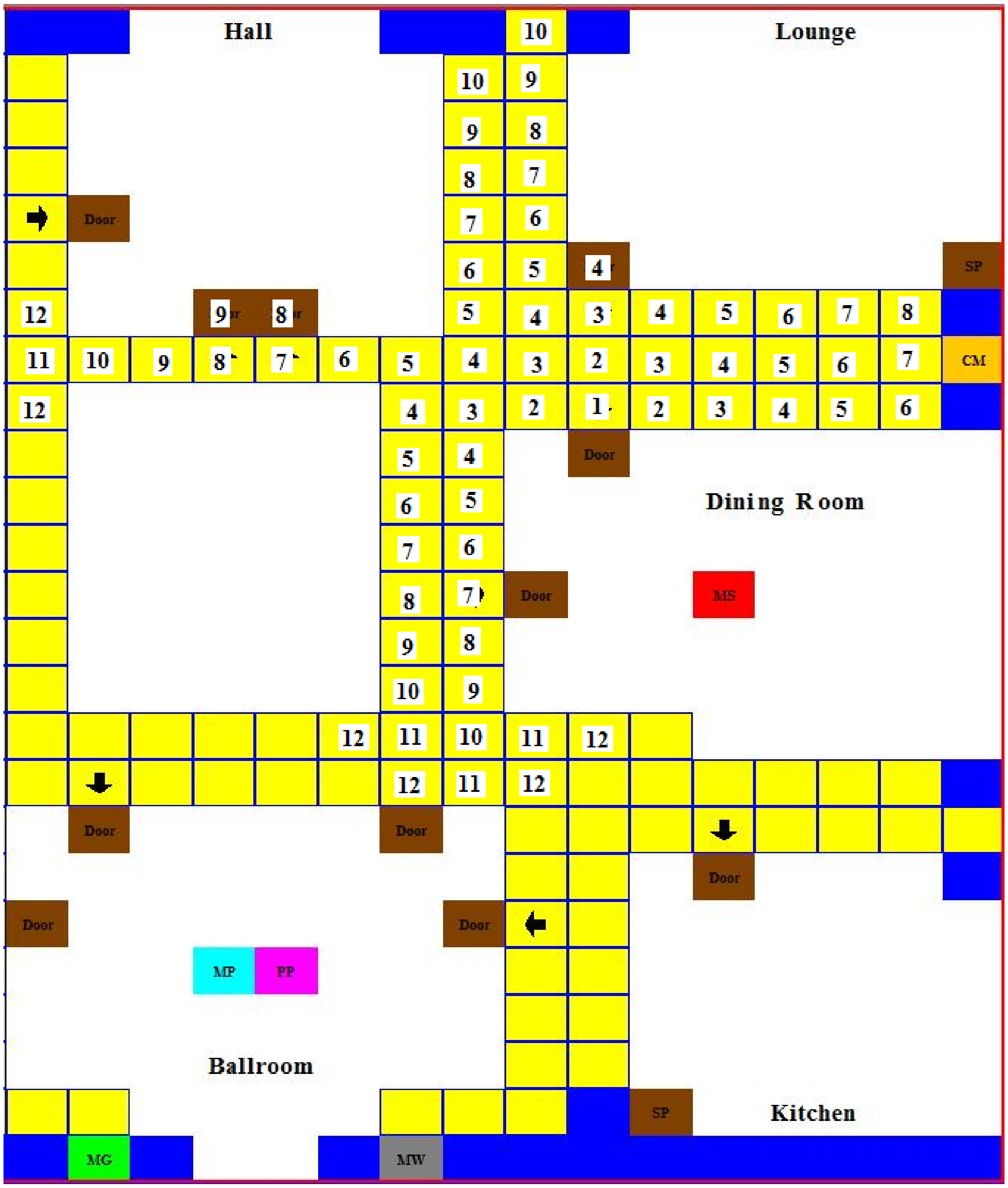


Figure 8.5.2: findClosestRooms() 2

At iterations 4, 8, and 9 door spots are added to the stack. If there are spots in the closestRooms array the rooms are added to the array if they have not been added already. If the rooms have already been added only the shortest distance is kept. For example on iterations 8 and 9 the doors to the hall can be reached. However the only distance kept is the distance of 8. The room array does not hold duplicate rooms and only hold the shortest distance to each room. Also if more than four rooms are found then only the rooms with the four shortest distances are kept. The room array will look like this after this all thirteen iterations. The first two rooms are assigned the room numbers and distances to the room. 13 is the lounge and 12 is the hall.

```
closestRooms[0][0] = 13;
closestRooms[0][1] = 4;

closestRooms[1][0] = 12;
closestRooms[1][1] = 8;

closestRooms[2][0] = -2;
closestRooms[2][1] = 1000;

closestRooms[3][0] = -2;
closestRooms[3][1] = 1000;
```

After the main loop the following variables are updated. The spot stack s is now empty due to the iteration process that searches for room spots on the stack. It pops all the points off the stack when it searches.

```
dieCount = 13
roomCount = 2

doors
Point(12, 16)
```

Now the same thing is done with the other door spot. The dieCount and roomCount variables are reset. This is okay because additional checks are in place to ensure the closestRooms array is correct. The following image shows the spots on the stack after the loop has completed. Note that since four rooms were found before the dieCount reached 13, the loop terminated after 11.

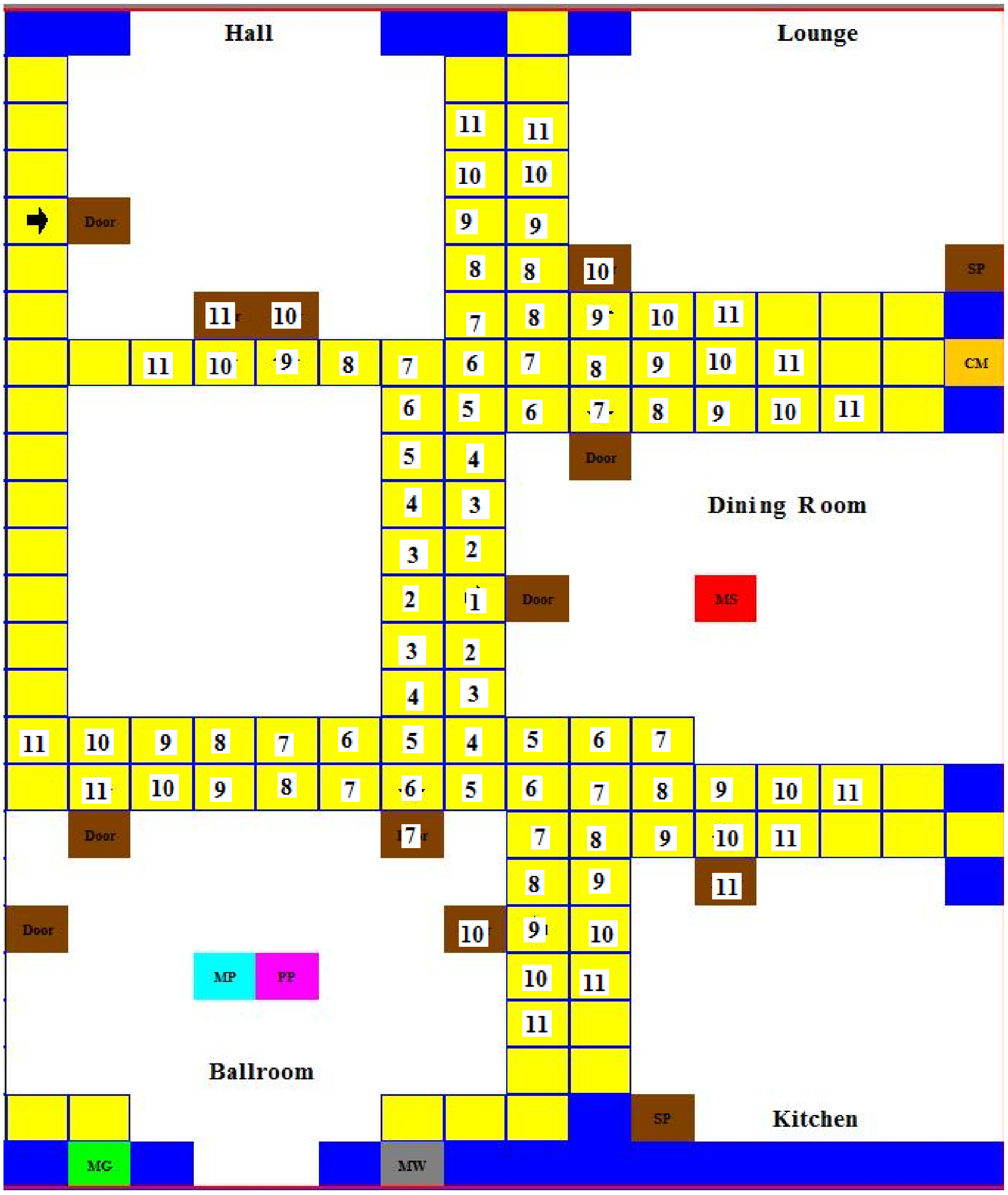


Figure 8.5.3: findClosestRooms() 3

Now all the door spots have been searched. After iterations 7, 10, and 11 room spots were found. The spots for the hall and the lounge are not added to the closestRooms array because they are already there and the previous closest distance is smaller than the one that the second door spot produced. The ballroom is added to the third spot of the array with a distance of 7 and the kitchen is added to the final spot of the array with a distance of 11. The final array looks like this. The ballroom is room 16 and the kitchen is room 15.

```
closestRooms[0][0] = 13;
closestRooms[0][1] = 4;

closestRooms[1][0] = 12;
closestRooms[1][1] = 8;

closestRooms[2][0] = 16;
closestRooms[2][1] = 7;

closestRooms[3][0] = 15;
closestRooms[3][1] = 11;
```

This array is returned to the calling functions of the AI class and the information is used for further processing on deciding where the computer player should move to. These rooms are typically used in calculating other rooms that are desired. The combination of distances from a roll to these rooms and these rooms to a desired room is used for the final room decision. After that the closest spot available is calculated to the room that was chosen and the computer player moves to that spot.

8.6: Conclusion

The algorithms discussed for the GameManager class are essential to playing the game and making the game easier for the user. The user does not need to search for all his/her possible spots nor find the path to those spots. The two GameManager class algorithms do all the work for the player. All the player has to do is click on the desired spot that is already highlighted for them. The BoardMap algorithms themselves do not provide a solution to a whole problem. Instead they provide unique and complex ways to calculate critical information for the AI class to use. The AI class considers all this information that is calculated from the BoardMap class algorithms and uses it for further use. This could be calling more BoardMap class algorithms with the data or comparing it with other data to make the final decision on where to move. Since the decision on where to move is so complex the BoardMap functions are essential to the AI class to help with obtaining the information to deduce a decision from.

9: Tests

9.1: Unit Test Plans

1. Test that cards are dealt correctly from the GameManager class

In the GameManager class the methods to deal cards should deal the cards correctly. It should first set the valid winning cards, which consists of one person, one weapon, and one room. Next it should deal the remaining cards randomly to the players until there are no more cards left. The cards should be dealt to the players in a consistent way so that an uneven number of cards will always be the same uneven number for every player. The special method that prints out all the cards and what player they were assigned to will verify this test.

2. Test that all sounds work correctly in the PlaySound class

The PlaySound class plays all sounds in the game. All the sounds that are used should be tested here. This will catch any typos in code or bad sound files. A simple test which attempts to play each sound one at a time will ensure that all the sounds work.

3. Test that the game rules load correctly

The game rules are stored in a text file that the program loads. The text file contains special commands similar to html tags. These commands tell the program what size and color the text should be. The tags should be tested that they do what they should do and produce the correct results.

4. Test that the AI class correctly disproves suggestions

The AI players should only be disproving suggestions when they actually can disprove the suggestions. Also the AI players should choose their cards intelligently if they have more than one choice. The first part of this test can be easily done with a print out of the cards. When a computer player disproves the suggestion with a card, that card can be checked with the card print out and verified that the computer did disprove with a valid card. The second part of this test is much harder. The intelligence of the AI was coded probabilistically to prevent predictability. Therefore the AI will choose the best card most of the time, but not all the time to prevent being predictable. It also has special cases where it will choose certain cards based on past events, sometimes in a probabilistic way and other times 100% of the time. Careful observation of several instances of all situations should be observed and compared with the cards the computer had to choose from.

5. Test that the GameManager class correctly calculates possible movement spots

When the player rolls the die all possible spots within the range of the number rolled should appear highlighted. If a player rolls a four, only spots that are within four moves should be highlighted. No other spots should be highlighted. With knowledge of the code, trial of most of the situations should be enough for this test. The results can be visually seen, once the visual part of the test has passed.

6. Test that the increment turn function in a multiplayer game works correctly

The increment turn function in the multiplayer game is the equivalent of a main loop in a program. It is the function that keeps the state machine moving, continuing to proceed to the next player. It should increment the players turn and handle the next player’s turn correctly. Players that have accused incorrectly should be skipped and AI players should be fully controlled by the host’s game client.

7. Test that the AI deduction algorithms work correctly

This algorithm actually is made up of several smaller unit tests because the deduction algorithm was broken up into several smaller steps. The first step makes sure that the array with the data is in a correct state. The second step sets cards to be the winning cards if no player has the cards. The third step uses systems of equations and the memory of the AI to deduce what cards a player must have. The fourth step determines if all of one player’s cards have been found and knows that the remaining unknown cards for that player cannot be owned by the player. The fifth step determines a winning card if all the other cards in a group are known. This means the last card must be the winning card. Lastly the data is put into a correct state again. All these methods should be tested to ensure that each one of them works correctly. The print out of the AI player’s cards will show if each one is working correctly based on the information the AI player was given.

8. Test that the AI makes a good guess

The AI should make good guesses that it can obtain useful information from. The way it decides what a good guess is explained in the AI algorithms section. All guesses should be examined with the data that the AI player knows to determine if the guess is a good guess. This test should also make sure that the AI is correctly guessing in the first place and not giving bad guess values.

9. Test that the AI calculates distances from spots to rooms correctly

The AI uses the BoardMap class function to calculate the distance from a spot to a room. This function is called in a loop with the die roll increasing. When the roll reaches the destination room, that roll value should be the minimum distance to the room. The function is similar to the GameManager function except it was modified for AI use. The function should not actually modify the game board data.

10. Test that all errors are checked at the setup screens

No game should be allowed to start with invalid data choices. The setup screens should check to ensure that all the choices are valid before starting a game. If there is an error the player(s) should be notified with the appropriate error message.

9.2: Functional Test Plans

1. Test that there are no dead ends

The user may have one or more option at the current screen they are on. In no case should the user click a certain series of buttons that leads him/her to a screen that he/she cannot get out of. There should be some type of back button for every screen, to go back to a previous screen. This test can be done by trying out several or all possible combinations that may lead to a dead end.

2. Test how well a network game can hold up when one or more people disconnect/reconnect one or more times

In a multiplayer game, users that disconnect should be able to reconnect if allowed to by the host of the game. This test should test if the game state is upheld with one or more people disconnecting and reconnecting several times. This test can be done by creating a multiplayer game with several people, then having one or more disconnect and reconnect at various times.

3. Test that all buttons clicked will bring up the correct screens

Clicking some buttons allow the user to transition from one screen to another. This is where information is passed from one class to another, or even to multiple classes. Tests should be done to ensure this information is passed correctly and not lost or mishandled in any way. The correct screens should display and the old screens should disappear. The correct information should display where applicable. Also certain features of the new screens should be enabled or disabled based on the information passed. Some transitions include, transitioning from the main screen to a setup screen, transitioning from setup screen to a game board screen, and bringing up a suggest/accuse screen. The main transition is from the setup screen to the game board screen. All players and their names should be shown in the correct order on the game board screen. The information should also be correctly stored in the game manager class, which is highly interactive with the game board class.

4. Test that all information loaded into a game is correct

For a single player game the information from the setup screen should be correctly passed to the game board. It should also display correctly on the game board. In a multiplayer game the same thing happens except for all players. Every player creates their own data, which must be fixed. The data from the first player, who is the host, should be sent out to all players. All the other players should receive the information and correct their data. All players should have the same data before the game starts.

5. Test that all the game messages appear correctly

The game messages have different sizes and colors based on what the message is. The messages were done by passing the message with the style to be sued for that message. All messages should have their text appear correctly. Player chat should appear blue and suggestion cards should be black and bold. Other messages should be normal. Also grammar should be correct for sentences generated with card names in them.

6. Test that all images and options display correctly on the setup screens

The player should be able to choose up to five other computer players to be in the game. When a computer player is selected a character box should appear where a player can then select a character. When a character is selected the corresponding image should appear. Tests for this class should make sure all these options are available and that the boxes and images correspond to each other. An exhaustive test for all options can be done and the results can be seen visually. Multiplayer screens should have limited options for players other than the host. The host should have more options such as adding human or computer player spots and booting connected players.

7. Test that the SinglePlayerGameBoard class runs the game correctly

The "main loop" to run the game exists in the SinglePlayerGameBoard class. It should correctly signal the player when it is their turn. Likewise it should correctly call the AI methods when it is their turn. The game should never "freeze" under any circumstances. Testing this can be done both with white-box and black-box testing. Going through the code to make sure nothing will discontinue the "main loop" is a good way to test this. Also observing how the game plays while playing will ensure it does not get stuck anywhere. Indefinite pauses should only be by the player and the game should resume correctly when the player allows it to. These situations are when the player is making any decision. The "main loop" is not an actual while loop, but just a method which will set off timers. These timers will set of other timers based on what choices were made. Eventually some of the timers will call the method again. That is how the game turns and state are handled. Ensuring that the loop will not be broken means checking all end states of the timers. The end states should be the only states to call the original function and they should all be reachable by other states. This is best done by examining the code carefully; however any errors should be detectable when playing the game.

8. Test that a multiplayer game runs correctly

The same methods apply here as they did for testing a single player game. The difference is that a multiplayer game works off messages than direct function calls. The end result should be the same as a single player game however the method is completely different. No bad states should be reached and all players should be in sync.

9. Test that the class containing game board position always matches what is shown on the game board

Two issues can occur here. One is that the information is correct and the screen is not updated. Another is that the information is either lost is mismatched. Comparing print outs to the visual game board can show if the information is being displayed. Likewise reading the print out can confirm if the position is correct.

10. Test that the algorithms for moving will display correctly on the game board

The algorithm for showing a user where the user can move should appear on the game board. The game board appearance should be refreshed after the algorithm has finished so the new outlined cells that the user can choose from will appear outlined. These cells should also be selectable and be shown correctly. The movement animation should also display correctly. The correctness should not be a problem since both classes interacting here use the same array for data storage, hence should always be the same. The screen refresh is the main factor. If the screen is out of date then the information will be correct but will not be seen as correct. Simple visual tests to ensure the screen is in an updated state should be enough.

9.3: Integration Test Plans

1. Test that connections are setup and closed properly

The host of the game creates the server. The server listens for connections and creates a game thread for game messages and a chat thread for chat messages. The threads listen for input from clients and send output to the clients if needed. All data needed for the threads is stored in a common storage class for all connected threads called the ServerData. This class contains the connection information as well as some important game information. Connecting players create threads for input and output for both the game and the chat. All these connections on both the client and the server side should be setup correctly, so that no threads get mixed up. Also the connections should close properly when it is time for them to close.

2. Test that sound is correctly toggled from all screens

Since sound events can be created from several classes it is important that these classes correctly interact with the sound class. From any screen the user should be able to toggle the sound. Therefore when any sound events occur, all events must first pass the check to see if sound is on. If sound is not on then the sound should not play. If sound is on then the sound should play. In addition to making sure the sound toggle works, the sound should also be the right sound. Unit tests may work for the sound class and the class that uses the sound class. However the two classes must match up correctly. If one class chooses sound 14 to play, and the sound class has sound 14 but it is stored as sound 13, then there is a mismatch. The numbers should correspond to the correct sounds in both classes and not just one for each class.

3. Test that all screens only appear when they should

Users should not have access to certain screen at certain times. The die roll screen should not pop up at the main screen as well as any game board screen. The integration of several classes with user interfaces should not clash. When one change is made it should affect the other user interfaces. If the game board is closed, all the user interfaces related to the game board should close too.

4. Test that a game is restarted correctly

When a game is restarted all players should be set back to their starting positions on the board. The cards should be dealt again. All data from the old game should be either reset or destroyed. All related user interfaces should be also reset or destroyed. A restarted game should be the same as a new game with no left over effects from the previous game.

5. Test that players only have access to their own data and not other player's data

For human players a simple visual test will determine if the cards are theirs or not. If the cards match the array of cards for them then it is correct. Likewise a similar method can be used for computer players. Another issue is that computer players should only have access to their own information. This means that although the game has access to all information, it should hide some of it from other parts of the game that should not know about it. This is so that the computer cannot cheat in the game. The only way the computer players should be able to gather information it does not have, is to play the game. The computer players should be watched during some simulations to make sure the data they gain is from the algorithms to play the game, and not magically stealing the data.

6. Test that the IP address for the host is correctly retrieved and received

The internal address is not a problem, but getting the external address is a little tricky. The hosting computer cannot directly get its own external IP address. An outside source must be used to retrieve the host's external IP address. This address must be retrieved correctly and both addresses much be sent to any players who connect to the host. A simple visual text can confirm the correctness of the internal IP addresses. The external IP address can be proven correct if another player can connect to it.

7. Test that the die roll animation correctly interacts with the die class and the game board screen

When a player chooses to roll the die, the button should be disabled after it is clicked. Following that should appear a small popup screen with the die on it. The die class generates random numbers which should match the image shown on the screen. Also the image should update correctly and pause on the final change of the die value. The final value shown should match the final value stored in the die class. Lastly the image should disappear. Everything should be reset for the next player. In a multiplayer game the final value of the die shown must be the same for all players. The intermediate values to simulate the roll do not have to be the same.

8. Test that the game state is preserved the same for all players in a multiplayer game

Many things can happen during a multiplayer game. Players can lose connections at any time. Players can also be sending similar messages to the server at any time. The state of the game should always be the same for all players. If player 1 rolls the die, then every player in the game should also see the die rolling for player 1. Therefore most of the server side functions should be synchronized and in order. That should preserve the game state. This can be tested by having several users interacting with the game at the same time and seeing if the game state will be the same for all of them. Loss of connection should be handled in some acceptable way. Since the internet is a factor, nothing in the game should be based on time like a single player game. It takes different amount of times to send packets to different places. The game has been modified from its original version to accommodate this. Everything is done by message passing. This test should ensure that the messages always leave the game in a correct synchronized state for all players.

9. Test that the AI makes the correct decision to roll

The roll decision involves the integration of the AI class and the BoardMap class. The AI class holds the data about the cards and the BoardMap class works off the data related to the position on the board. The roll decision is when the AI player decides to roll, suggest, or use a secret passage. The simple decision is when the player is not in a room because rolling is the only option. However when the player is in the room and has both the option to use a secret passage and suggest as well as roll things get more complicated. The algorithm is explained in the AI algorithms section. The result of the decision can be seen on the screen. The correctness of the decision should be verified based on the data obtained and used for the algorithm. If the roll decision is working correctly the algorithm should produce correct results. Also the algorithm should produce good solutions for all situations.

10. Test that the AI chooses the correct spot to move to

The most complicated decision in the game is where the AI chooses to move. The movement algorithms are explained in both the AI algorithms section and the other algorithms section. It involves both the AI and BoardMap classes. Nearly all the BoardMap functions are used to gather information on the game board. The AI class interprets the return values from the BoardMap class functions to produce the correct spot to move to. The test should prove that all the algorithms give logical results to the problem of deciding where to move to. In addition to being logically correct the algorithm must be functionally correct too. All the data values produced by all the functions involved should be correct. Also all the values should be interpreted correctly. The test of this decision involves three parts. The functions should be called with the correct values, return the correct values and those values should be interpreted and used correctly by the calling function. The combination of all three results should produce a correct value. This value for the spot to move to should also logically agree with the computer player's situation. The situation involves all the information the computer player has gained about all the cards and room proximity.

10: Test Results

10.1: Unit tests

1. Test that cards are dealt correctly from the GameManager class

Result: Passed

Description: All the cards are dealt correctly in both single player and multiplayer games.

2. Test that all sounds work correctly in the PlaySound class

Result: Passed

Description: All sounds are loaded correctly to the correct name.

3. Test that the game rules load correctly

Result: Passed

Description: The rules screen appears correctly. This verifies that the rules have been loaded correctly. It also verifies that the tags for sizes and colors are interpreted correctly.

4. Test that the AI class correctly disproves suggestions

Result: Passed

Description: The AI disprove function works correctly both logically and functionally.

5. Test that the GameManager class correctly calculates possible movement spots

Result: Passed

Description: The possible move spots are calculated correctly and appear on the game board.

6. Test that the increment turn function in a multiplayer game works correctly

Result: Passed

Description: All turns are incremented correctly with AI and human players.

7. Test that the AI deduction algorithms work correctly

Result: Passed

Description: All parts of the solution deduction algorithm work correctly both functionally and logically.

8. Test that the AI makes a good guess

Result: Passed

Description: All parts of the guessing algorithm are functionally correct. In some rare cases the choice the AI makes is not logically the best choice but is still an acceptable guess choice.

9. Test that the AI calculates distances from spots to rooms correctly

Result: Passed

Description: The distance calculation is correct.

10. Test that all errors are checked at the setup screens

Result: Passed

Description: All errors are checked and error messages are displayed.

10.2: Functional tests

1. Test that there are no dead ends in the program state

Result: Passed

Description: The option to exit to the main screen from the file menu bar prevents all dead ends. All main screens that are closed will open up another screen so that the user is not left with a blank screen. The one exception is during a network connection failure during the setup phase. This in most cases is handled correctly and the user is brought back to the main screen after acknowledging the error message. In some odd cases the screens will fail to transition because the connection has not failed but has not connected either. This idle time may cause the user to think the program has crashed. What happens is that the timeout for the connection attempt has not been reached yet. For some reason the connection is not processed. The test passes because it works in all but the rarest cases.

2. Test how well a network game can hold up when one or more people disconnect/reconnect one or more times

Result: Passed with exception

Description: At the setup screen any number of people can connect or reconnect as long as there are open spots. If there are no spots then the connection is refused. If the host reconnects all the players are notified with an error message and are brought back to the main screen. During the game if a player disconnects all other players are notified upon the next game message sent. If the player disconnects during his or her own turn no game messages are sent. This is because the player who disconnects is currently the only player who can send game messages at the time. This will result in the game appearing to have frozen. The game is not frozen. The game is waiting for the disconnected player to make some type of decision although the player is disconnected. This freezes the state machine so there is nothing the other players can do except to restart the game or exit the game. No notification is sent that the player has left if the player disconnects during his or her own turn. If a player leaves gracefully a message is sent regardless of whose turn it is.

3. Test that all buttons clicked will bring up the correct screens

Result: Passed

Description: All buttons clicked bring up the correct screens

4. Test that all information loaded into a game is correct

Result: Passed

Description: All the information loaded into a game is correct. The multiplayer game data is fixed for all players before the game is started.

5. Test that all the game messages appear correctly

Result: Passed

Description: All game messages appear correctly. Grammar, font, size, and color are all correct.

6. Test that all images and options display correctly on the setup screens

Result: Passed

Description: All appear correct.

7. Test that the SinglePlayerGameBoard class runs the game correctly

Result: Passed

Description: The state of the game runs smoothly with no dead ends or incorrect states.

8. Test that a multiplayer game runs correctly

Result: Passed with exceptions

Description: The state of the game is kept in a correct state. Due to the speed of the messages some messages arrive faster than others. This may make some events happen with more or less delay than intended. Sounds may overlap as a result. The only problems that occur are explained in functional test 2.

9. Test that the class containing game board position always matches what is shown on the game board

Result: Passed

Description: The game board is refreshed every time changes are made so the board is current with the array containing the board information.

10. Test that the algorithms for moving will display correctly on the game board

Result: Passed

Description: The movement spaces are lit up correctly.

10.3: Integration tests

1. Test that connections are setup and closed properly

Result: Failed

Description: The connections are not closed properly but do close when they need to. The close is ungraceful and throws an error which is caught and ignored in most cases so the game can continue. The main server is never closed when the host exits to the main screen. When a new game is hosted the error is caught and ignored that the connection that was still open is now reset. This may result in some odd things happening if a player exits from the setup screen and players still attempt to connect. The connections are setup correctly but do not close properly.

2. Test that sound is correctly toggled from all screens

Result: Passed

Description: Sound works from all screens.

3. Test that all screens only appear when they should

Result: Passed

Description: All screens that should not appear are either disabled for future use or disposed of if they are currently visible.

4. Test that a game is restarted correctly

Result: Passed

Description: Both single player and multiplayer games restart correctly. The restart methods simply call the constructor of the class with the previous data from the existing instance of the class. The constructor sets the current instance of itself to the old instance of itself. The calling class is not known to exist or not at this point. It is not known whether the old data from the previous game still exists in memory. The current games will work fine because the data is correctly passed to the new instance of the class. The interesting thing is the question of the call stack. The class keeps constructing itself when the game is restarted, but it is unknown of what happens to the previous instance of the class. Whether or not the previous instance is overwritten remains unknown. In theory a shallow copy is done so this would reset the old instance of the class to the new instance of the class with correct data reconstructed. However the calling class then disappears. There may or may not be a memory leak. If there is it is not harmful because the memory size is too small. For there so be any memory issues a person would need keep restarting the game thousands of times.

5. Test that players only have access to their own data and not other player's data

Result: Passed

Description: All players' data access is limited to their own. No mismatches occur under any circumstances.

6. Test that the IP address for the host is correctly retrieved and received

Result: Passed with exceptions

Description: Everything works fine except for the external IP address retrieval. For an unknown reason it will produce an error on rare occasions. It is dependent on an external website to make a connection to. This website will then be able to get the computer's external IP address and return it to the computer. If the external website not owned by the program has an error or is taken down then the game can no longer retrieve the host's external IP address. It is a vulnerable point, but seems to work fine in most cases. The error produced is handles correctly and has no impact on the rest of the game. It will just show the external IP address as error instead of the IP address.

7. Test that the die roll animation correctly interacts with the die class and the game board screen

Result: Passed with exception

Description: This works fine in all cases except when the AI takes a long time to decide where to move. The line of code to update the game messages on what die number was rolled is in the correct location. The problem is some type of threading error with java threads unrelated to my own threads. Java has a thread to update user interfaces which does not always execute in order.

The AI takes a little bit longer to decide where to go from certain rolls from the ballroom. This may be a ten second delay. During this time the die animation has already rolled however the game messages have not updated on what number was rolled. The AI is processing and the thread is busy with that processing. The Java thread to update the screen is apparently not executed until the AI is done processing. When the AI has finished with the decision on where to move the game messages are updated with the message of what die number was rolled. This is a Java error which no work around was used in this case. The consequences are minimal in this case.

8. Test that the game state is preserved the same for all players in a multiplayer game

Result: Passed

Description: The game state is correct. The hosting player's client controls the game state messages. It is the only player who can send most messages to all the other players about the state of the game. This means all control is through that player and all other players follow the lead of that player. This means multiple messages will not be creates and confusion will not occur. Since several things happen based on timing certain functions require synchronization before executing. A problem that was fixed was that messages were being received before the time state was finished handling the previous message. The problems resulting from this were hard to track down. This was fixed by forcing any messages to wait for all active timers to become inactive before processing the message. This means that the game synchronization has some leeway. The timing of messages can cause players to not be in perfect synchronization, however important messages will not execute until synchronization is complete. This means that the game state can be different for the players at a given time but nothing will execute if the state is different.

9. Test that the AI class correctly makes die rolling decisions

Result: Passed

Description: The AI roll decision works correctly in all but the rarest situations. In these situations the incorrect decision is still an acceptable choice. This is because rare special situations are not logically coded into the decision. All the involved parts work correct functionally.

10. Test that the AI chooses the correct spot to move to

Result: Passed with exceptions

Description: In most cases the AI chooses the correct spot to move to. Some problems may occur if the AI has to travel a long distance to reach its destination. In this case it may not make the best decision but will still make an acceptable one. Another issue happens with dealing with blocked rooms. The AI player will not consider rooms blocked by a player. Instead of moving close to the room and waiting a turn, which is the best choice in many situations, the AI player will decide to move towards the next closest room that is needed. Since there are so many situations to consider when dealing with movement the algorithm is acceptable. In almost all cases it works correctly. In those rare cases it still makes a decision that is acceptable. The algorithm is slow at times due to the complexity. In uncommon situations it may take ten second to decide where to go.

11: Conclusion/Future work

The Clue computer game turned out very well. The game as a whole worked as well as it was expected to. No program is perfect but this game is close enough. The program incorporated several areas of computer science including UML, java programming, artificial intelligence, multithreading, client-server computing, and additional network architecture. Other concepts such as state machines and complex recursive algorithms were explored as well. The Clue computer game is a well-functioning combination of all these areas.

One key observation made is that when dealing with a multiplayer game the game must be synchronized. This concept is easier said than done. For future projects a more careful multiplayer architecture will be considered before coding it. One piece of the code that required the most debugging was the initial lack of synchronization. Another key observation is that UML diagrams do not look good outside of the environment they were created in. The only way to make them readable in a word document is to deviate from convention. The entire UML diagrams that follow convention are too big to reasonably fit in a word document. Since most of the documentation involved UML diagrams, this was a reoccurring problem.

There is not much left to do for future work for the Clue project. Aside from tweaking some minor bugs, the game turned out great. The minor bugs are nothing to be concerned about and probably not worth fixing. Overall the Clue computer game project was very successful.

12: References

[1] Visual Paradigm 7.0 Community Edition. Retrieved April 10, 2011, from http://www.visual-paradigm.com/product/vpuml/editions/community.jsp

[2] Dennis, Alan, Barbara Wixom, and David Tegarden. *System Analysis Design UML Version 2.0*. 3rd ed. John Wiley & Sons, 2009.

[3] Clue by Parker Brothers, 1992 edition.

[4] Clue – Murder at Boddy Mansion, 1998, Hasbro Interactive Inc.

* **DOUGLAS SELENT** received his B.S. in Computer Science at Merrimack College in 2009 and his M.S. in Computer Science at Rivier College in 2011. He has recently been accepted into the doctoral program for Computer Science at Worcester Polytechnic Institute. Doug’s favorite subject in Computer Science is algorithms. His favorite thing to do is to create his own algorithms to solve unique problems. In his free time he likes to play video games and watch the Patriots win on Sunday.