SWEN 301 Project 2 - Summary

Proposal:

For this project I wanted to refactor Cameron McLachlan's Cluedo game so that it implements the Model View Controller design pattern. Im especially concerned with separating rendering logic from the game logic.

STAGE ONE:
This will be starting a new project from scratch with 3 main packages; Model, View, Controller. Because the initial packages were not representative of there responsibility. After setting these up I will split the classes into there respective packages. All game logic will go in the model. All UI classes will go into the view. And finally the controller and the main will be in the controller.

STAGE TWO:
After the classes are split into there appropriate ate component of MVC. I'll refactor to make sure that each package's classes are not taking on other components responsibility. For example this is the original Location interface.

```
public interface Location {

        void paint(Graphics g, List<Tile> moveableLocations, Point p);

}
```

Obviously this is wrong in the context of MVC. I will reimplement these classes so that they represent only what they need to in terms of game logic.

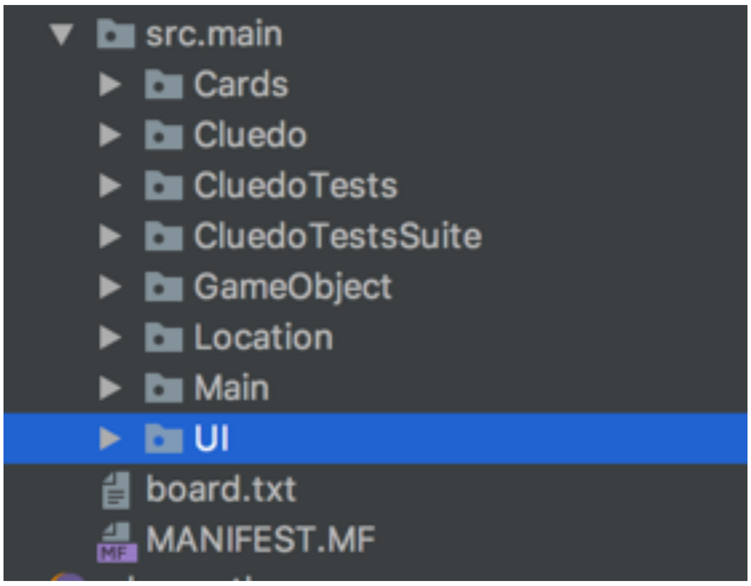
STAGE THREE:
After implementing stage 2, there will be no way of rendering half of the objects present on the board. This is because the painting was done by the objects themselves. We will need methods to render all game objects. These will be located in the CluedoCanvas class which will be in the view package. CluedoCanvas has most of the other drawing so choosing to the methods in this class makes sense. I will create a paintBoard() method which will call paintTiles() and paintRoom() as well as paint roomObjects().

STAGE FOUR:
Will improve independence between the model and the controller. As discussed in the part 1 Essay, the responsibility of moving is shared among the player and the controller. My proposed improvement, is to have the controller handle this logic. This is so the responsibility of updating fields such as player.location, tile.player, room.player is all handled in one class.

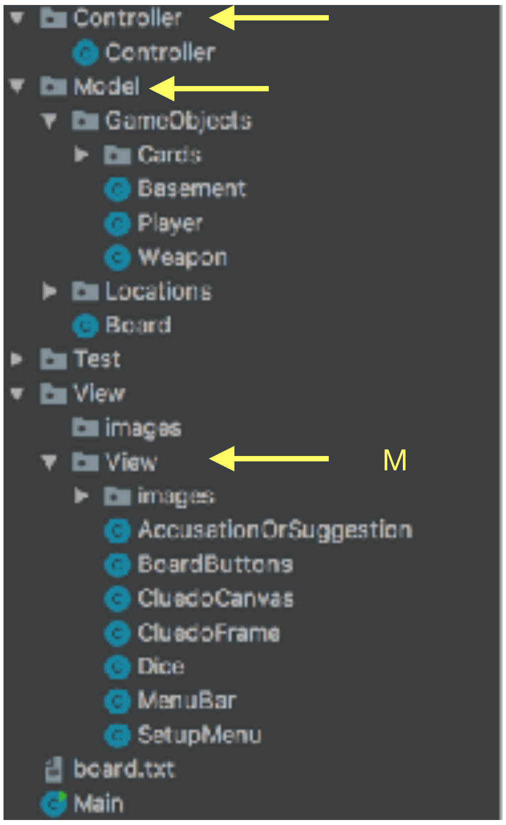
STAGE FIVE:
Refactor the test suite to ensure that all the functionality is still working. The tea suite will have to be adapted to work with the new architecture and design pattern Implemented.

Description/Evidence of refactor:

STAGE ONE:
This initial stage was simple, but very essential. Ensuring the right classes are in there respective component package would ultimately make separating concerns easier to visualise. On the left is how the original packages were laid out. In the right is the new package layout.

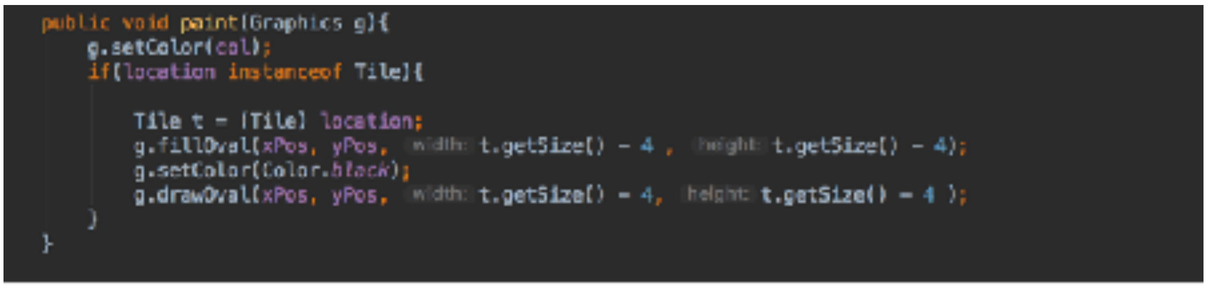


```
▼ src.main
    ▶ Cards
    ▶ Cluedo
    ▶ CluedoTests
    ▶ CluedoTestsSuite
    ▶ GameObject
    ▶ Location
    ▶ Main
    ▶ UI
  board.txt
  MANIFEST.MF
```

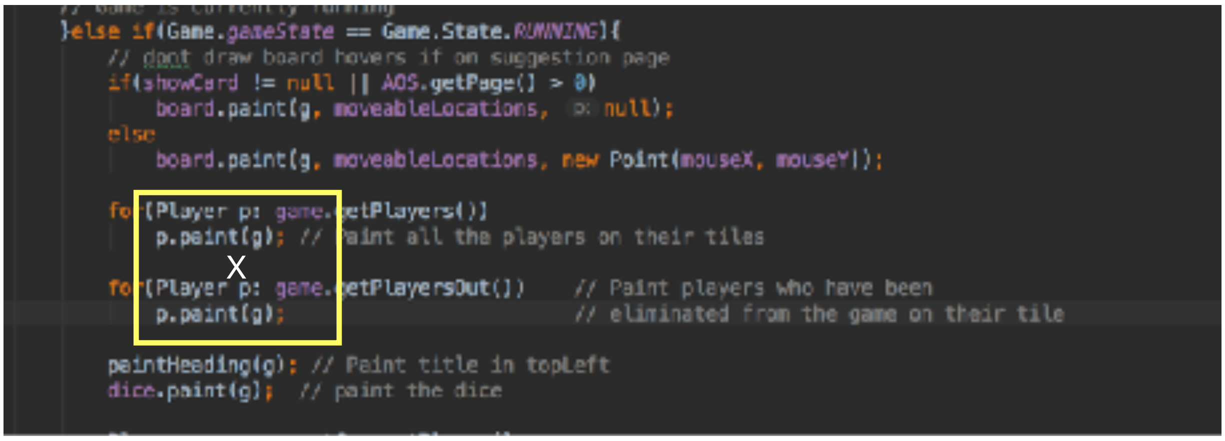


```
▼ Controller
    Controller
▼ Model
  ▼ GameObjects
    ▶ Cards
      Basement
      Player
      Weapon
  ▶ Locations
    Board
▶ Test
▼ View
    images
  ▼ View
    ▶ images
      AccusationOrSuggestion
      BoardButtons
      CluedoCanvas
      CluedoFrame
      Dice
      MenuBar
      SetupMenu
  board.txt
  Main
```

STAGE TWO AND THREE:

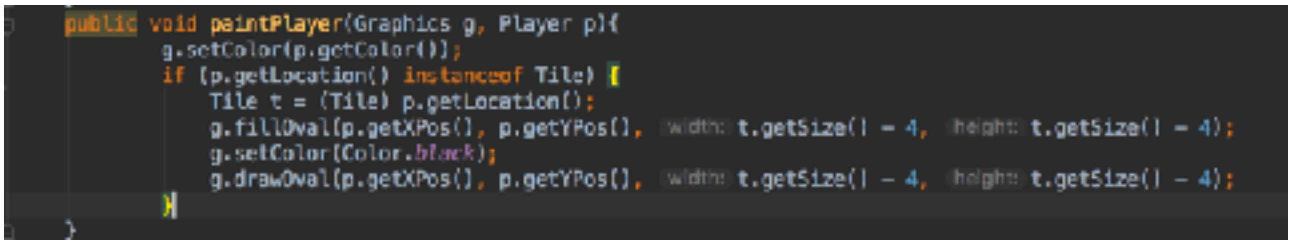


```java
public void paint(Graphics g){
    g.setColor(col);
    if(location instanceof Tile){

        Tile t = (Tile) location;
        g.fillOval(xPos, yPos, width: t.getSize() - 4 , height: t.getSize() - 4);
        g.setColor(Color.black);
        g.drawOval(xPos, yPos, width: t.getSize() - 4, height: t.getSize() - 4 );
    }
}
```



```java
}else if(Game.gameState == Game.State.RUNNING){
    // dont draw board hovers if on suggestion page
    if(showCard != null || AOS.getPage() > 0)
        board.paint(g, moveableLocations, p: null);
    else
        board.paint(g, moveableLocations, new Point(mouseX, mouseY));

    for(Player p: game.getPlayers())
        p.paint(g); // Paint all the players on their tiles

    for(Player p: game.getPlayersOut())     // Paint players who have been
        p.paint(g);                         // eliminated from the game on their tile

    paintHeading(g); // Paint title in topLeft
    dice.paint(g);  // paint the dice
```

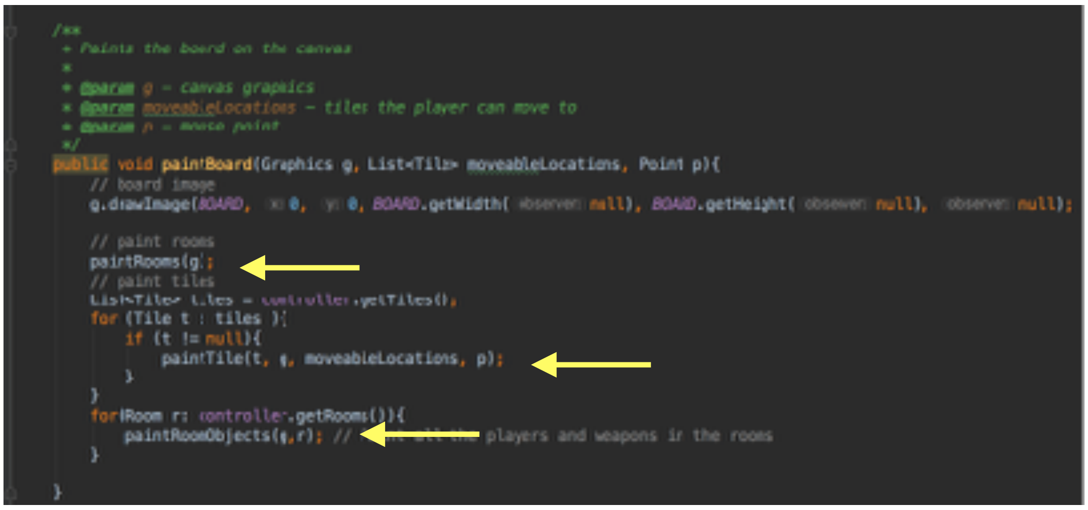
Depicted above is a code snippet from the CluedoCanvas. This illustrates how the instead of painting the player in the View, the rendering responsibility was being delegated back to the player. The first thing to do was remove the method in player. Then I created paintPlayer() method in the CluedoCanvas. This effectively separates View's responsibilities from the model's responsibility.



```
public void paintPlayer(Graphics g, Player p){
        g.setColor(p.getColor());
        if (p.getLocation() instanceof Tile) {
            Tile t = (Tile) p.getLocation();
            g.fillOval(p.getXPos(), p.getYPos(),  width: t.getSize() - 4,  height: t.getSize() - 4);
            g.setColor(Color.black);
            g.drawOval(p.getXPos(), p.getYPos(),  width: t.getSize() - 4,  height: t.getSize() - 4);
        }
}
```

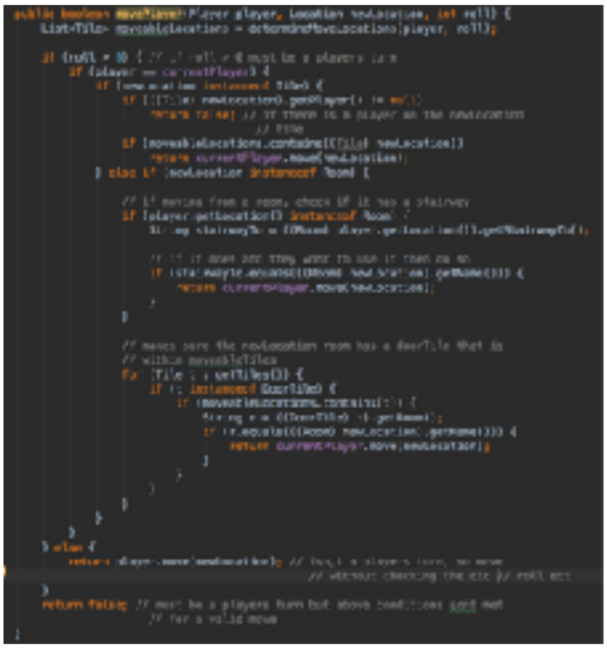
This method is called from a main paint() method in the CluedoCanvas which queries the controller for lists of game objects. e Then iterates through them all and passes them to the paintTile(Tile t), paintBoard(Board b) and paintPlayer(Player) respectively.

Below is more evidence of this refactor. This refactor ensured that rendering responsibilities were handled in the view. Because of this change, our program now reflects software which has been designed with MVC pattern.

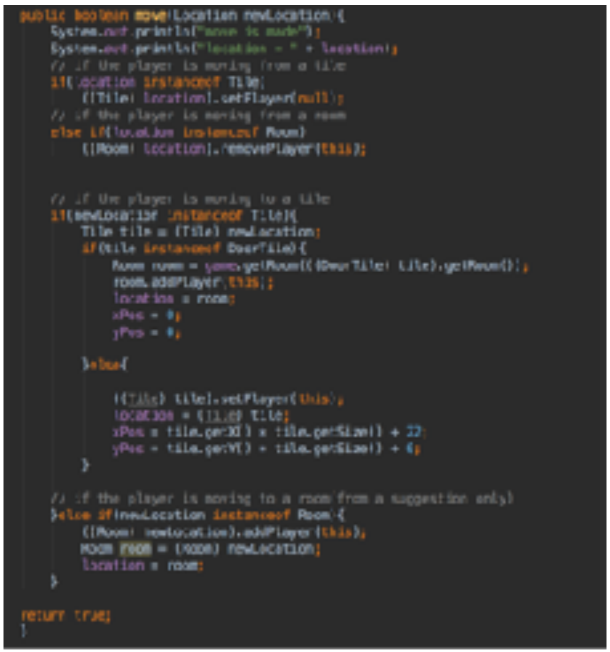


```
/**
 * Paints the board on the canvas
 *
 * @param g - canvas graphics
 * @param moveableLocations - tiles the player can move to
 * @param p - mouse point
 */
public void paintBoard(Graphics g, List<Tile> moveableLocations, Point p){
    // board image
    g.drawImage(BOARD,  x: 0,  y: 0, BOARD.getWidth( observer: null), BOARD.getHeight( observer: null),  observer: null);

    // paint rooms
    paintRooms(g);
    // paint tiles
    List<Tile> tiles = controller.getTiles();
    for (Tile t : tiles ){
        if (t != null){
            paintTile(t, g, moveableLocations, p);
        }
    }
    for(Room r: controller.getRooms()){
        paintRoomObjects(g,r); // paint all the players and weapons in the rooms
    }

}
```

Stagefour:
This is how moving the players was handled before:

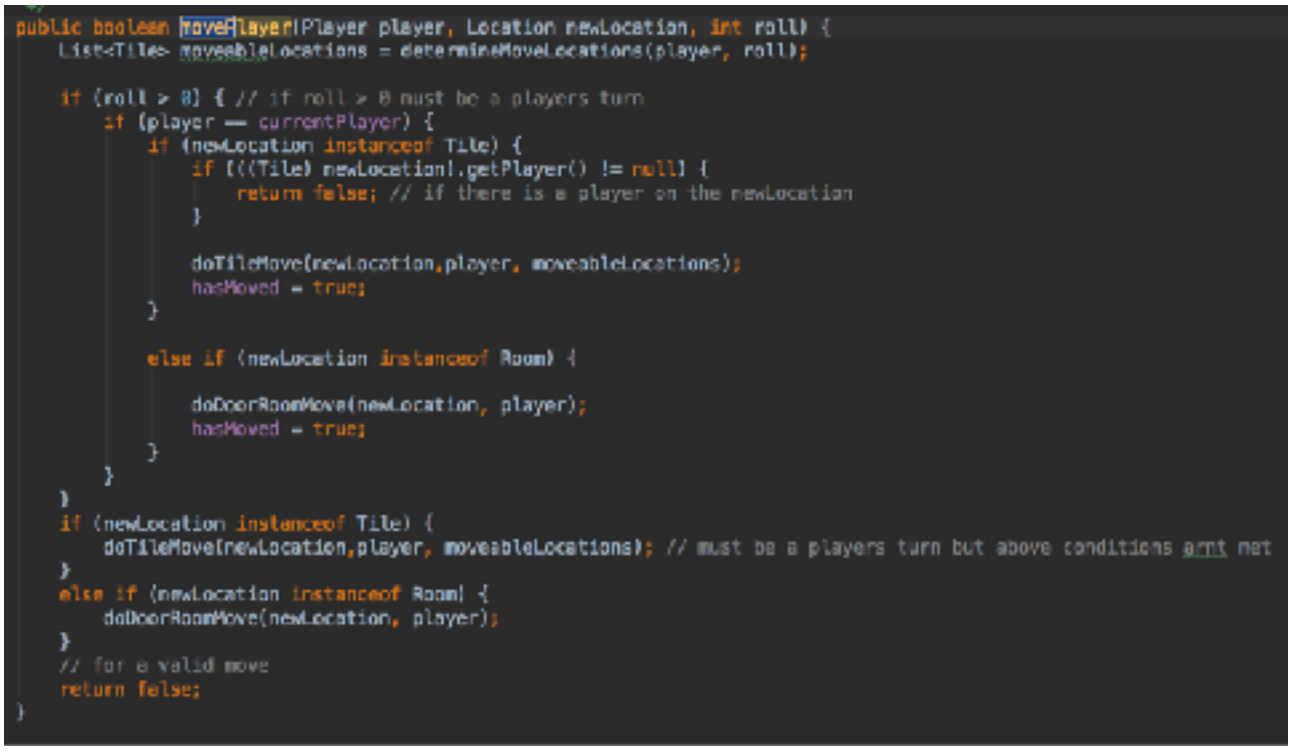


```
public boolean movePlayer(Player player, Location newLocation, int roll) {
    List<Tile> moveableLocations = determineMoveLocations(player, roll);

    if (roll > 0) { // if roll > 0 must be a players turn
        if (player == currentPlayer) {
            if (newLocation instanceof Tile) {
                if (((Tile) newLocation).getPlayer() != null))
                    return false; // if there is a player on the newLocation
                                  // tile
                if (moveableLocations.contains((Tile) newLocation))
                    return currentPlayer.move(newLocation);
            } else if (newLocation instanceof Room) {

                // if moving from a room, check if it has a stairway
                if (player.getLocation() instanceof Room) {
                    String stairwayTo = ((Room) player.getLocation()).getStairwayTo();

                    // if it does and they want to use it then do so
                    if (stairwayTo.equals(((Room) newLocation).getName())) {
                        return currentPlayer.move(newLocation);
                    }
                }

                // makes sure the newLocation room has a doorTile that is
                // within moveableTiles
                for (Tile t : getTiles()) {
                    if (t instanceof DoorTile) {
                        if (moveableLocations.contains(t)) {
                            String r = ((DoorTile) t).getRoom();
                            if (r.equals(((Room) newLocation).getName())) {
                                return currentPlayer.move(newLocation);
                            }
                        }
                    }
                }
            }
        }
    } else {
        return player.move(newLocation); // isn't a players turn, so move
                                          // without checking the die // roll etc
    }
    return false; // must be a players turn but above conditions isn't met
                  // for a valid move
}
```



```
public boolean move(Location newLocation) {
    System.out.println("move is made");
    System.out.println("location = " + location);
    // if the player is moving from a tile
    if(location instanceof Tile)
        ((Tile) location).setPlayer(null);
    // if the player is moving from a room
    else if(location instanceof Room)
        ((Room) location).removePlayer(this);


    // if the player is moving to a tile
    if(newLocation instanceof Tile){
        Tile tile = (Tile) newLocation;
        if(tile instanceof DoorTile){
            Room room = game.getRoom(((DoorTile) tile).getRoom());
            room.addPlayer(this);
            location = room;
            xPos = 0;
            yPos = 0;

        }else{

            ((Tile) tile).setPlayer(this);
            location = (Tile) tile;
            xPos = tile.getX() * tile.getSize() + 22;
            yPos = tile.getY() * tile.getSize() + 6;
        }

    // if the player is moving to a room from a suggestion only)
    }else if(newLocation instanceof Room) {
        ((Room) newLocation).addPlayer(this);
        Room room = (Room) newLocation;
        location = room;
    }

return true;
}
```

Now, all of this ‘move’ functionality has been refactored to be handled in one place. Because all the player.move() method was updating the players fields, we could just call these updates from the controller.movePlayer(Player p) method. Thus in the new refactor the Player.Move() method does not exist.



```java
public boolean movePlayer(Player player, Location newLocation, int roll) {
    List<Tile> moveableLocations = determineMoveLocations(player, roll);

    if (roll > 0) { // if roll > 0 must be a players turn
        if (player == currentPlayer) {
            if (newLocation instanceof Tile) {
                if (((Tile) newLocation).getPlayer() != null) {
                    return false; // if there is a player on the newLocation
                }

                doTileMove(newLocation,player, moveableLocations);
                hasMoved = true;
            }

            else if (newLocation instanceof Room) {

                doDoorRoomMove(newLocation, player);
                hasMoved = true;
            }
        }
    }
    if (newLocation instanceof Tile) {
        doTileMove(newLocation,player, moveableLocations); // must be a players turn but above conditions arnt met
    }
    else if (newLocation instanceof Room) {
        doDoorRoomMove(newLocation, player);
    }
    // for a valid move
    return false;
}
```

The method uses the helper methods doDoorRoomMove(newLocation, player); and doTileMove(newLocation,player, moveableLocations);. These methods update the fields of necessary objects. These fields include Player.location, Tile.player, Room.player etc. This part of the refactor produced better independence whilst still maintaining the original functionality.

STAGE FIVE:

Once the firs 4 staged of refactoring were complete I refactored the test suite. This was pretty easy but there were 3 Tests which I couldn’t get to pass even though the actual game was working as it did originally. The main thing to change in the test suites was just construction of the game itself in each test, after this it was just a matter of changing variable types to fit the new program.

Summary/Reflection:

The refactor went well overall. Although 3 of the tests involving movement don’t pass, I still believe the architecture, and separation of concerns was improved to fit the MVC design pattern. There are further improvements I would make to ensure the view and controller operate more independently, but considering how messy the initial program was I believe the refactor has been significant.

Now that the refactor follows MVC a bit closer, the program can be more easily tested, improved and understood. By modularising our design to seperate the concerns of the model view and controller we significantly increase our ability to change things more easily and quickly. These changes can be made without having trickle down effects that will cause failure or undesired functionality in other components.

As a result any refactoring that would happen from now will be significantly easier than my initial task. This idea of improvement encapsulates the reason why we have open source projects. For example if I updated the GitHub repository another developer could see the potential for a new UI, and because of my refactor easily change it.