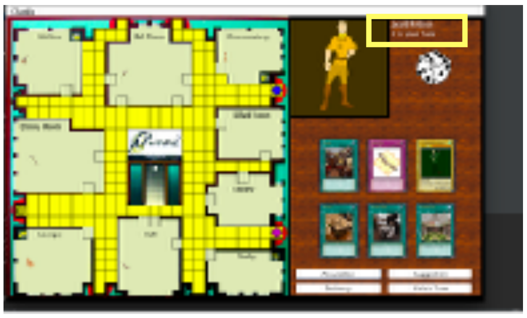
Introduction:

Humans need entertainment. Board games have provided a solution to this problem. In modern times, we continually see the decline of physical games.This is partly because we have developed software that replicates these games, and present them in a more accessible medium (eg smartphone). In this project we will analyse the architecture and design of Cameron McLachlan's attempt to produce a JAVA program which replicates the Cluedo Board game. This attempt can be viewed from the repository https://gitlab.ecs.vuw.ac.nz/mclachcame/Cluedo



I initially cloned this repo and it didn't work. I saw the potential of this project to illustrate my knowledge of software architecture, and so I was determined to run it. I debugged to find that the file path to the txt representation of the board was wrong. After debugging this I managed to play the game, and understand its structure.

History

This game was originally developed by Anthony Pratt in 1943. Cluedo is a murder mystery game, where the main aim is to find out who murdered who, with which weapon. Since its release there have been many more adaptations and spinoffs.

The first board game to be computerised was Code Name Sector. It was released back in the late 1970s. The product revolutionised the gaming industry. Game companies hereafter realised the potential of computerising their classic board games.

This Cluedo software was designed and created for the SWEN 222 project. This course from Victoria university aims to provide its students with knowledge about designing object-orientated software. In particular, one of the main concerns with this course is "implementing and gaining knowledge of Design patterns and design by contract". The aim of this particular project was to implement a game using the design pattern model, view, controller.

DOMAIN

CONCERNS SURROUNDING DOMAIN OF GAME DEVELOPMENT:

The challenge for game developers is to represent a game in a virtual medium. Through this virtual medium we must be able to interact with the game as simply as we would with a normal board game. We will achieve this representation through the construction of 3 main features. We will need to: 1) *present a model of a situation* - This should present a virtual model of the Cluedo game. 2) *render a view* - These concerns should facilitate a way to view the previous feature. 3) *A method to interact with the board game* - There should be a separate package which handles these concerns. This package will have to interact and mange both of the previous features, so that we can **observe** our **interactions** affecting the **model.**

CONCERNS SURROUNDING DOMAIN OF EDUCATION:

The main reason for development of this project was education. In the SWEN 222 course one of the main course learning objectives is to "Appreciate the limitations of different solutions when designing Java programs, particularly with respect to Design Patterns and Design-by-Contract. 3(e)" . This is important to understand, as we can criticise the program for what it is trying to achieve. The program is trying to achieve: not just a virtual representation of the game, but a virtual representation of the game implemented with the correct design.

Domain of open source applications:

This program is open-source software. Game designers and players who embrace open source gaming "believe that sharing, transparency, and rapid prototyping can lead to superior entertainment experiences". This important to understand as it highlights why we need our software to be reusable, adaptable, modifiable and readable. If our software displays these qualities other developers in the open source community can more easily interpret and adapt our projects.Then as a community, we can achieve this “superior entertainment experience”.

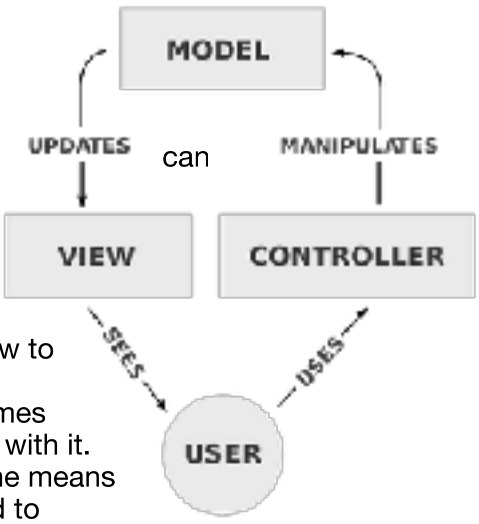
COMPONENT STRUCTURE.

Intro to MVC/why its important:

Model view controller is a design pattern which is used to promote the idea of separated presentation. The idea of separated presentation “is to make a clear division between domain objects that model our perception of the real world, and presentation objects that are the GUI elements we see on the screen”. (Martin Fowler). Domain objects should be completely self contained and have no reference to how we present the data. By separating these concerns we simplify the processes of maintaining and improving software. As well as making it easier to test and debug. This simplification arises from creating a modularity where we can improve alter bits of code independently, without worrying about the details of other components. And more specifically without having to change the other components.

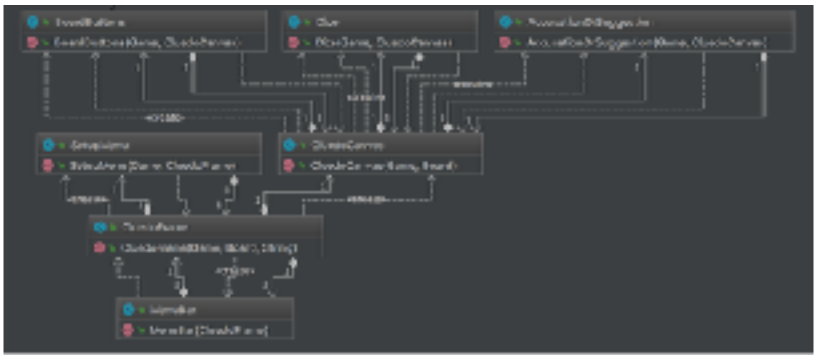
Details of MVC:

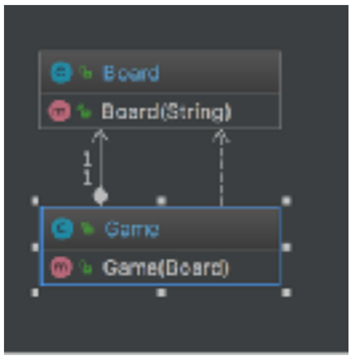
The main components of MVC are the model, the view and the controller. The model represents the actual data, logic and rules of the application. In this case the Model would be the board the players, tiles etc.The model is responsible for managing/ manipulating the data based on user input, which it receives from the controller. The view is just a visual representation of this data. There can be multiple views for a program that correctly implements MVC. The final component is the controller, which manages the view and the model, and also communicates commands between them. E.g if the user queries the controller, the controller then translates this query into a context of the model, which then updates itself. This update will be noted by the controller, and send a command to the view to update itself respectively. Model, view, controller, or some form of it, is usually applied to gaming software. This is due to the dependency games have on user input. We perceive the game, and need a way to interact with it. These interactions involve the utilisation of a user interface, which is the means by which a user interacts with a computer system. MVC is widely used to mediate this interaction.



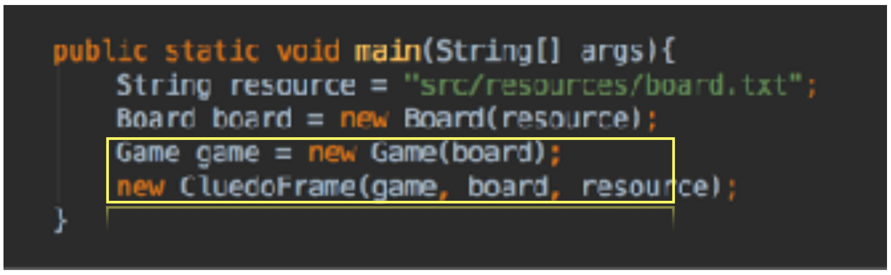
Initial Repo Structure:

the current architecture is very messy and hard to follow. However, my understanding of the project is that Game is the class which holds the main game loop, and could be seen to represent the controller. While the board is the main data structure, which in combination with the other supporting classes (e.g. Player) could be seen to represent the Model component. The CluedoCanvas should represent the view.





Note: Check back page for full class Diagram.
This project does not implement MVC correctly, and my proposed architectural refactor is to ensure that it is. Depicted below is the simplest representation of incorrect implementation of MVC at the highest level.

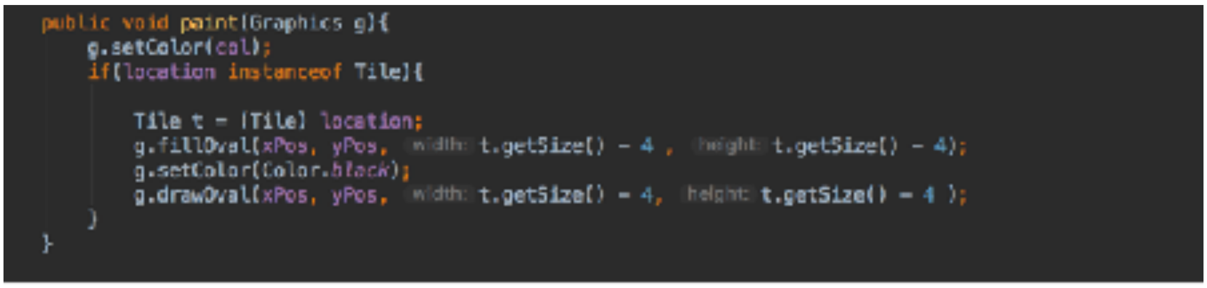


```
public static void main(String[] args){
    String resource = "src/resources/board.txt";
    Board board = new Board(resource);
    Game game = new Game(board);
    new CluedoFrame(game, board, resource);
}
```

As seen to the left the Game is given a board. And the view is given a Game and a Board. `This is inherently wrong. The view should have no field storing the Board, nor theGame. The view should instead, query the controller for the state of the board and render based on the result of this query. Thus, only one initialisation should be made in a Main(), and that is the initialisation of a controller which will initiate and have references to, the view and the model.

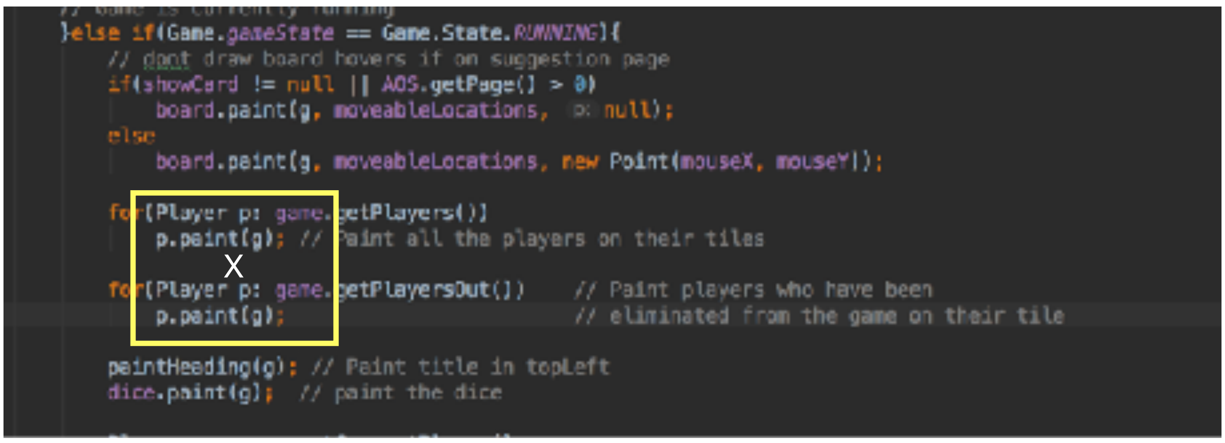
Discussion of individual components and how they are implemented incorrectly:

There are too many examples of poor separation of components. So well concentrate on small examples which illustrate these bad practices. For example, in the package UI we have the CluedoCanvas class which is where the painting of objects should be implemented. Instead,



```
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 );
    }
}
```



```
}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
```

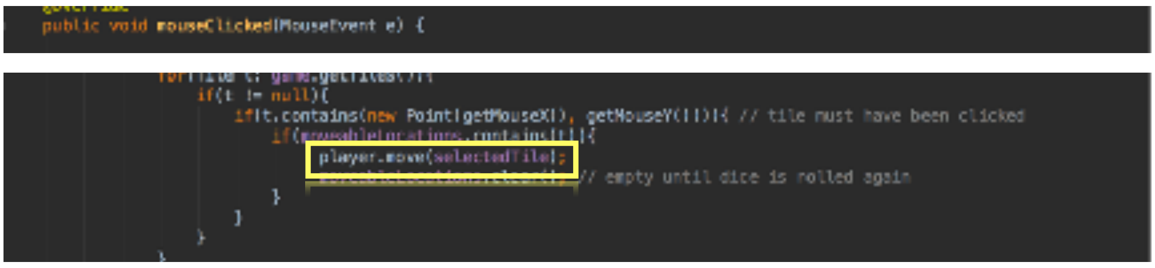
however, the painting is delegated back to the game object - player;

The code snippet above illustrates this. The second piece of code depicted is the CluedoCanvas class which is under the UI package. The first is the method Player.paint(Graphics g), located in the GameObject package. This is obviously not implementing model view controller, as the Player, which is a model object is directly drawing to the output. Instead we should aim to do all the drawing in the UI/view. A more correct way to achieve this would be by be passing the player to a method such as CluedoCanvas.paintPlayer(P) in view which will draw to the graphics based of that players location, etc. Our method in the view would look something like this:

```
private void renderPlayers(int x, int y, Graphics g, Player player){
                                g.setColor(col);
                        if(p.getLocation instanceof Tile){
                                Tile t = (Tile) location;
                                g.fillOval(xPos, yPos, t.getSize() - 4 , t.getSize() - 4);
```

```
                g.setColor(Color.black);
                g.drawOval(xPos, yPos, t.getSize() - 4, t.getSize() - 4 );
    }
```
This will be Called from a GameController class, which will have an UpdateView method gets the current instances of all objects (e.g a Player instance) iterates through them and passes them to the view for painting. One of our main features was to implement a view that represents the data structure, this means rendering responsibilities should be independent and not shared among other packages.

On the contrary we have game logic implemented in the view:



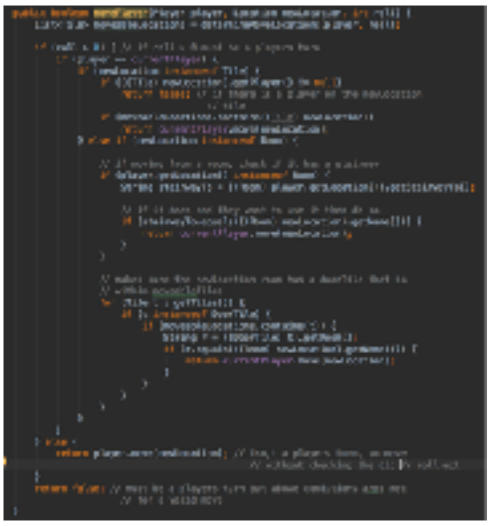
```
    public void mouseClicked(MouseEvent e) {

            for(Tile t: game.getTiles()){
                if(t != null){
                    if(t.contains(new Point(getMouseX(), getMouseY()))){ // tile must have been clicked
                        if(moveableLocations.contains(t)){
                            player.move(selectedTile);
                            moveableLocations.clear(); // empty until dice is rolled again
                        }
                    }
                }
            }
```

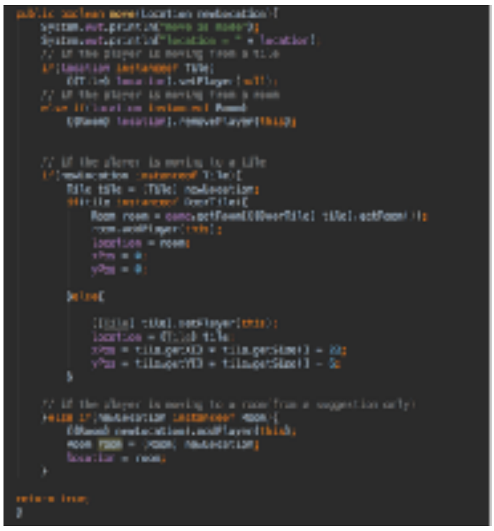
Rather than querying the controller, this method directly modify's its instance of the the model. It should send a query to the controller which should, then call the movePlayer method. Not directly move the Player itself. This is an example of View and Model being implemented poorly, now we will look at the model component having muddled responsibilities.

Code smells & poor coupling/Independence:

The left method comes from the game class(which vaguely represents a controller). Where as the method on the right is from the Player class, thus a game object which is part of the model.



```
public boolean movePlayer(Player player, Location newLocation, int roll) {
    [illegible]
```



```
public boolean move(Location newLocation){
    System.out.println("move in model");
    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 = [illegible].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() - 5;
        }

    // 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;
}
```

Here we can see that the responsibility of moving a player is shared between two components - the model, and the Game class (controller). All game logic i.e deciding if a move is valid or not, should be handled in the model. And as we can see this is what is happening. However the Game method movePlayer() in Player alters the Model too. This does not violate MVC, however it is bad practice as all move responsibilities should be handled by one of the two. A possible solution is having an updateTiles() method in the game which would be called when a player is moved by the movePlayer method. This would update the board, and the relevant locations (rooms/tiles). This way we could compact the Player.Move() method down to only updating its own fields, such as xpos/ypos and location, whilst also achieving more appropriate independence.

Data Structure:

A data structure is a collection of data values, the relationships among these values, and the functions or operations that can be applied to the data. In Cluedo we have Objects storing our data. Object orientated software aims to represent the parts of the real world in a way which the computer can understand. For example The Player Object exists to “organize some (player)data into a single entity that can be passed around and managed as a whole”.

BOARD DATASTRUCTURE:

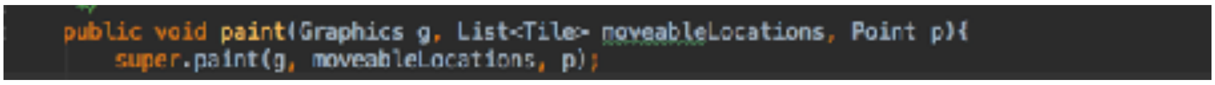
In this program one of our main data structures is the Board. The Board class represents a 2 dimensional array of Tile objects, which represent location on the board. Tile is a parent class which implements the interface Location. Child classes DoorTile, and StartingTile, both extend Tile. Tiles can have players on them, or simply be empty.

It is good to have inheritance evident in a object orientated program as it aims to reduce duplicate code and make it easy to add functionality. However here, this is not the case. In the class Tile, we have a paint method, and then in DoorTile we have an identical paint method. This practice ultimately contradicts the aim of inheritance. Instead the DoorTile should just inherit the behaviour from its parent class eg. super.paint(…) (even though no painting shouldn’t be implemented in the model).

[illegible]

[illegible]

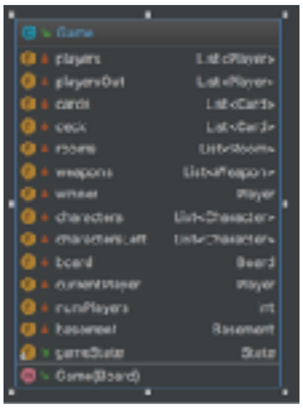
Depicted below is evidence of what child classes of tile should do unless they are painted differently. This is not utilising the full advantage of using objects to store our data. We can organise this “objects” to store our data in more efficient ways.

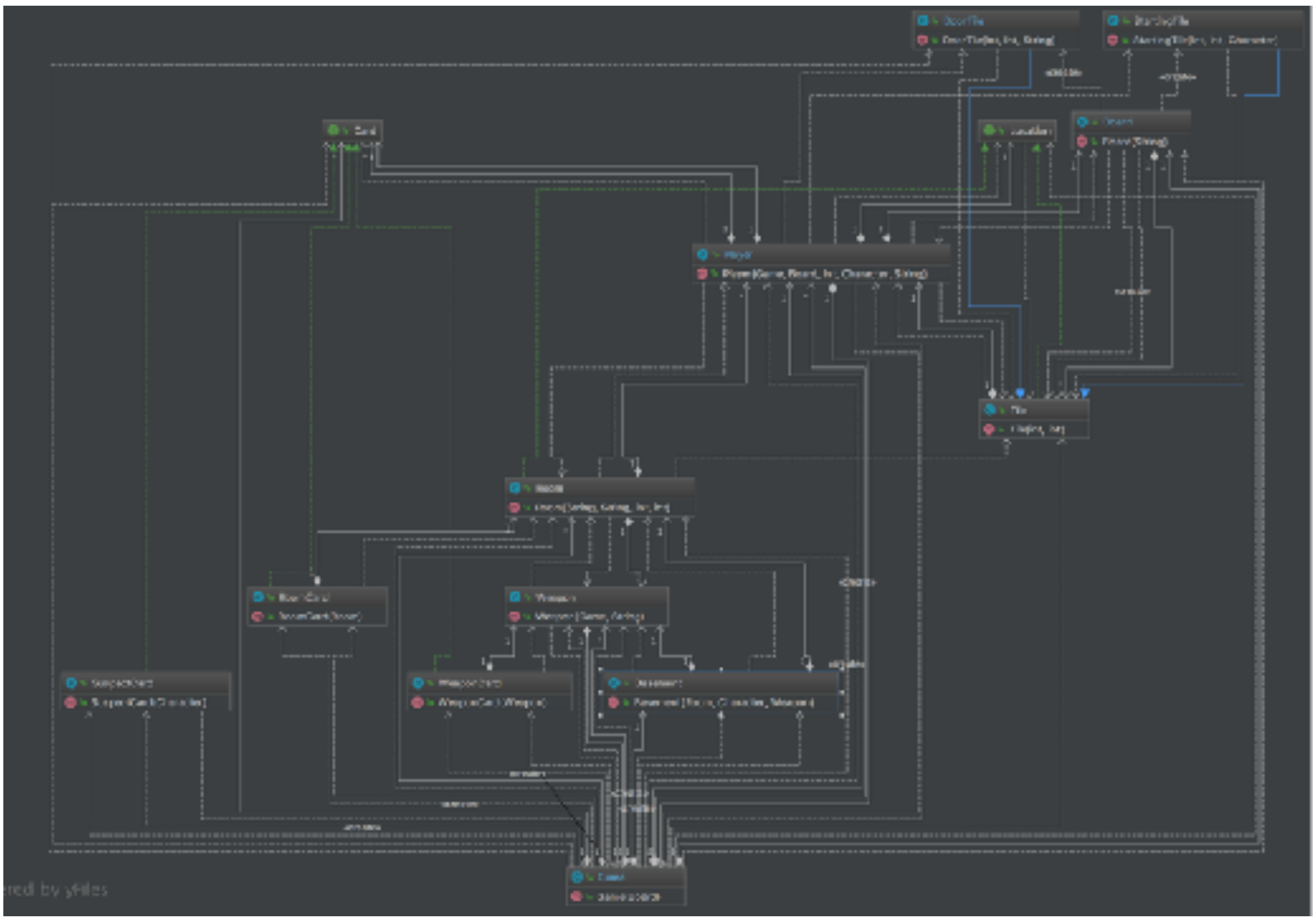


```
public void paint(Graphics g, List<Tile> moveableLocations, Point p){
    super.paint(g, moveableLocations, p);
```

GAME DATA STRUCTURE:

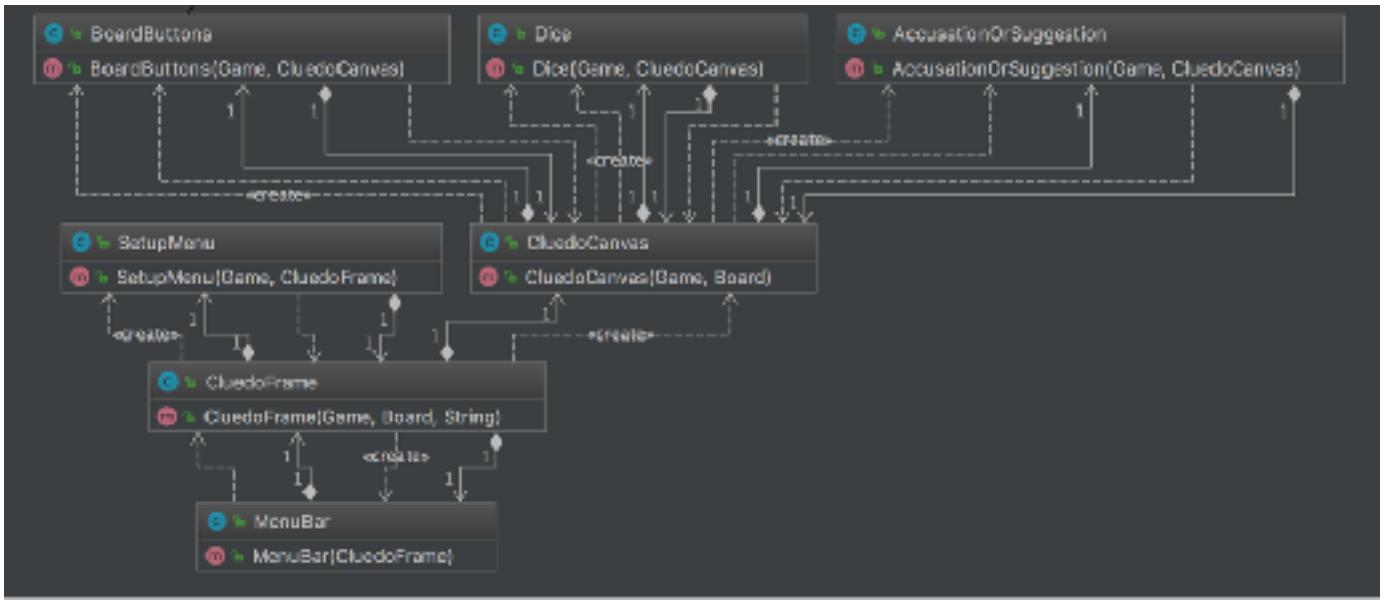
Game is another overlying data structure. It is where the collections of our GameObjects are stored. These collections are abstracted lists. These lists are defined by the type of Objects which are stored in them. Eg we have a players list which has type Player. This Player object will have its own data structures which define what a player is. This definition will be made up of mostly primitive types such as ints, doubles and strings. There are various Game objects stored in lists in the class Game. All of these “objects” interact with each other and in combination represent the Cluedo game as a whole. It is important to have these objects interacting with each other in the correct way. All object classes should have appropriate references to the other Objects they interact with. Additionally all objects with shared functionality should extend a parent class and delegate this shared responsibility to it. In Object orientated programs, objects with there own individual responsibility should be separated. If objects which represent our data are implemented with these practices; we achieve one of the main goals of software design - to minimise maintenance and modification costs of our software. This digram shows how





messy the dependencies are throughout the model.

References:
https://martinfowler.com/eaaDev/uiArchs.html
https://hackernoon.com/objects-vs-data-structures-e380b962c1d2
https://opensource.com/resources/what-open-gaming


BoardButtons
BoardButtons(Game, CluedoCanvas)
Dice
Dice(Game, CluedoCanvas)
AccusationOrSuggestion
AccusationOrSuggestion(Game, CluedoCanvas)
«create»
«create»
«create»
SetupMenu
SetupMenu(Game, CluedoFrame)
CluedoCanvas
CluedoCanvas(Game, Board)
«create»
«create»
CluedoFrame
CluedoFrame(Game, Board, String)
«create»
MenuBar
MenuBar(CluedoFrame)