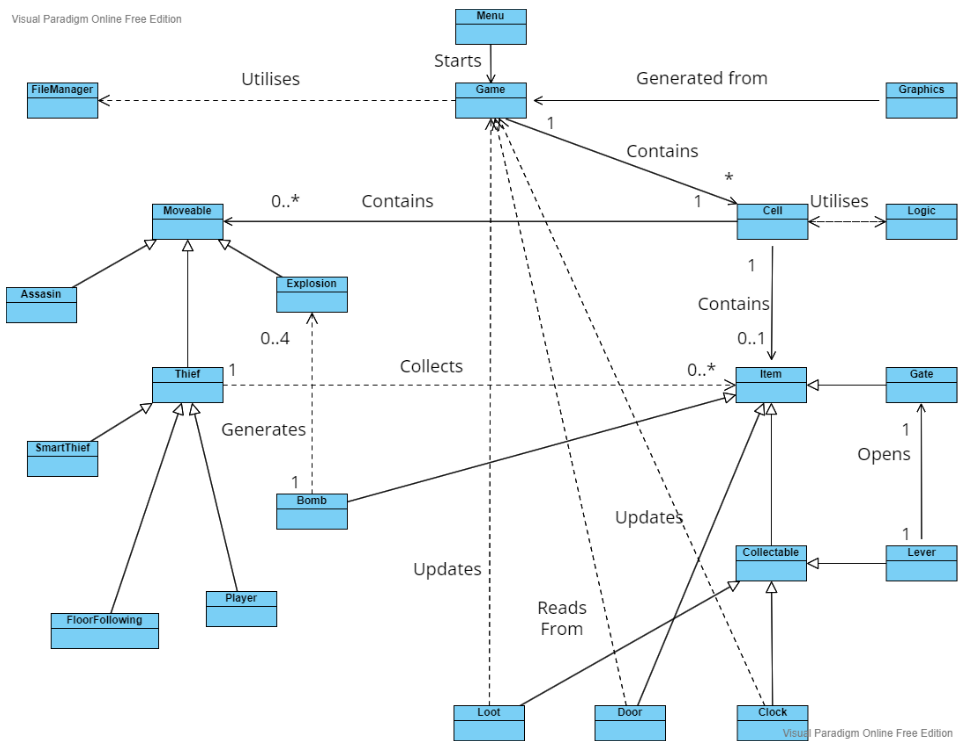
A1 Group 25 Partial Design Doc

Class Diagram:



The "Cell" class can exist without the "item" and "movable' class, just by changing the format of what it holds. Therefore, an association type of a relationship. This also applies to the other classes that use this type of arrow.

Candidate classes:

Rough Idea Behind the Cell:

For a game size of 6x6 there will be 36 cell class objects held in a 2d array within the Game class. It holds information such as the cell colour array of 4, items and moveables.

Cell (created by James)	Collaborations of Cell

+Cell(tileColours : string[], moveable : Moveable, item : Item) : void +getTileColours() : Game$tileColour [] +hasBomb() : boolean +isFlagged(): boolean +removeItem(item : Item) : void +removeMoveable(moveable : Moveable) : void +setItem(item : Item) : void +setFlagged(bool: Boolean): void +setMoveable(moveable : Moveable) : void +setTileColours(tileColours : GametileColour[]) : void -refreshLogic() : void -refreshItem() : void	See below

Responsibilities:

isFlagged:

- Returns true if a bomb is adjacent to the cell

Cell:

- Initialises the cell class setting the initial moveables location and items on the cell
- Takes a string of colours and parses it into an array of enumerator colours

getTileColours:

- Return the tile colours, which will be used by the logic class for checking in the move of a player is valid.

hasBomb:

- Returns a Boolean value if there is a bomb on the cell. Used by logic class to check for collisions.

- Looks at the items in the movable attribute and is if it equals a bomb

removeMoveable:

- Instead of a function moveMoveable, we decided that removing and setting would be better as it takes logic out of the cell class and moves it to logic.

refreshItem:

- When an item is placed or removes, it will call the logic class to update the cell if needed

refreshLogic:

- Does the same as refreshItem but for moveables.

setTileColours:

- When the cell is constructed, it will fill the array of 4 colours in

getTileColours:

- Fetch the array of colours for the graphics class to use which the Game class will access. Will be used for logic if a moveable is allowed to go onto that tile and for graphics.

setMoveable:

- Used to set the moveable onto this class, or remove it. It will call on the logic class for bomb detection and collisions of other moveables.

setItem:

- Used to add an item during the construction.

removeItem:

- Remove item if an event Is triggered from the logic static class. Called when a moveable enters onto the cell.

<u>Collaborations:</u>

Logic Class:

- When a moveable is removed and set again, the cell class must call on the logic class to check for collisions with bombs or other moveables.
- Will check if the move is valid

Game/Board Class:

- The game class holds an array of all the cells, the game class initialises the cell's and this gives them the initial moveable and item locations.
- Game knows how many items are left
- Game holds the clock class

Item:

- Cell contains an instance of an item, which may be a gate, lever, gem…

Moveable:

- Cell contains an instance of a movable, which can move across different cells

<u>Pseudocode:</u>

```
Public setItem(item: Item)
    this.Item = item;
    this.refreshLogic();

Private refreshLogic(item: Item)
    Logic.resolve(item, this);

Public Cell(tileColours: string[], moveable: Moveable, item: Item)
        // n This will take the array of strings and parse them into the correct enumerator of
           colours.
    parsedTileColours = tileColour[3];
    for (colour in tileColours)
    int count = 0;
            for (coloursAvailable in tileColour)
                if (coloursAvailable.toString() == tileColour)
                    parsedTileColours[i] = coloursAvailable
    count++

        // n Set the item into the cell, and run a function to check if it is a bomb and set a
           Boolean true false for other classes to access
    this.setItem(item);

        // n Set the item into the cell and run the logic

    this.setMoveable(moveable);
```

Author: James Wheeler

Rough idea behind Logic:

Logic **(**Created by Joseph)	Collaborates with cell Utiles all of cell's collaborations
+resolve(moveA: Moveable, cell: Cell): void +resolve(moveA: Moveable, moveB Moveable, cell: Cell): void +resolve(item: Item, moveA: Moveable, cell: Cell): void +resolve(item: Item, moveA: Moveable, moveB: Moveable, cell: Cell): void -bombTrigger(cell: Cell): int()	

Logic is a utility class used by the cell class to determine the state of a cell after resolving the all interactions within the cell

Resolve:

Depending on the variables passed, one of the three variants of resolve will be run. The result of resolve will determine what the state of the cell will be after all actions in the cell have been carried out. The main variable that influences the outcome of a cell is the dangerLevel of moveables contained within the cell relative to the lethality of other entities within the cell, with the moveable with the highest danger level winning out. Danger levels come in 3 tiers: 0 for thieves, 1 for assassins and 2 for explosions. Items do not have a danger level. Tier 0 can only kill (pick up) items, tier 1 kills thieves but not items and finally tier 2 kills living moveables and all but some* items within the cell.

*gates,doors and bombs are not destroyed, bombs specifically are detonated instead.

BombCheck:

Whenever a thief interacts with a cell, check if the cell is adjacent to a bomb, if so trigger the bomb

<u>Pseudocode for Logic</u>

case4 -incorporates all other cases but case1 (in which nothing happens)

```
Public (item: Item, moveA: Moveable, moveB: Moveable, cell: Cell)
    Bool bThreatened = False
    Switch (moveA.DANGER_LEVEL)
        Case 0:
            If null == cell.getItem()
                    Pass
            else
                    Cell.getItem().interact(moveA.checkIsPlayer)
                    Cell.deleteItem()
```

```
                Game.decrementItems()
        bombCheck(cell)
        Return
    Case 1:
        bThreatened = True
        Return
    Case 2:
        If 2 == moveB.DANGER_LEVEL
            Pass
        Else
            Cell.deleteMoveable(1)
            If item is bomb
                    Detonate bomb
            else
                    Cell.deleteItem()
        Return
    Base case:
        Throw error "variable DANGER_LEVEL out of scope. Must be 0,1 or 2"
        Return
If null != moveB
    Switch (moveB.DANGER_LEVEL)
        Case 0:
            If null == cell.getItem()
                Pass
            else
                Cell.getItem().interact(moveA.checkIsPlayer)
                Cell.deleteItem()
                Game.decrementItems()
                bombCheck(cell)
            Return
        Case 1:
            aThreatened = True
            Return
        Case 2:
            If 2 == moveA.DANGER_LEVEL
                Pass
            Else
                bThreatened = False
                Cell.deleteMoveable(1)
                If item is bomb
                    Detonate bomb
                Else if item type == collectable
                    Cell.deleteItem()
        Base case:
            Throw error "variable DANGER_LEVEL out of scope. Must be 0,1 or 2"
```

```
                Return
     If aThreatened AND bThreatened
         Pass
    Else if aThreatened
         cell.deleteMoveable(0)
    Else if bThreatened
         cell.deleteMoveable(1)
    return
```

-in theory the base case should never be run as there is no method to change the danger level.

```
BombCheck(cell: Cell):
        If null != cell.isFlagged
                Trigger bomb at flagged square
```

Rough idea behind bomb:

Bomb **(**Created by Kamille)	Collaborates with cell Collaborates with explosion Collaborates with board
-numberOfFlags: int -fuseTime: int -triggered: boolean +static final tickRate: int -selfCoords: int()	
-triggerCountdown(): void -generateFlags(): void +decrementCooldown(): -explode(): void	

<u>Responsibilities:</u>

Countdown

- To initiate the count down, before the explosion.
- Each tick is one second
- The tick method will be called before the bomb detonates.
- Start one method, another method to detonate. When the tick method reaches 0 seconds left, another method named “explode” will be called.Explode
- All bombs may explode
- The method “explode” will be called and will communicate with the class Explosion, in order to generate 4 explosion instances.

Collaborators:

Explosion Class

- The subclass of Bomb, Explosion, will tell the class Movable that the bomb was detonated and will impact other items.

Cell Class

- The class Bomb will inform the cell class that it must be deleted, upon destination.
- This is because the bomb class is stored in cell

Game/Board Class

- This class also stores the location of any other bombs, that may need to be told to cell to delete.
- Game Stores the location of all loot, clocks, levers in the bombs path. Therefore, the location of any items that were impacted by the detonation.

Logic Class

- The logic class will check if a flagged square, near the board, has been trespassed.
- If a square is flagged, the logic class will tell the bomb class to start the countdown.
- It will check with the Game/board class for the location of the detonated bomb.

Pseudocode for Bomb

```
triggerCooldown()

    Triggered = True

generateFlags()

    selfX = selfCoords[0]

    selfY = selfCoords[1]

    board.getCell((selfX+1), selfY).setFlagged(True)

    board.getCell((selfX-1), selfY).setFlagged(True)

    board.getCell((selfX), selfY+1).setFlagged(True)

    board.getCell((selfX), selfY+1).setFlagged(True)

decrementCooldown()

    If fuseTime != 1
```

```
        fuseTime = fuseTime -1
    Else
        explode()
```

Inheritance hierarchy

The Flying Assassin, Floor Following Thief and Player are all subclasses of Moveable and form an inheritance hierarchy. Each subclass has their own similar and differing attributes and methods, which makes them perfect for an inheritance hierarchy.

Every moveable shares attributes such as speed, direction, and danger level, so all of them adopt these from the superclass Moveable. This reduces repetition and ensures clarity across all moving objects in the game. It is also very useful in cases such as the logic class, as this relies on using the danger levels to determine the outcome of a conflict on a cell between moveables and items. Having a range of values to compare massively simplifies this class in comparison to comparing each item and moveable individually to one another. The same applies to the methods required by each moveable, such as the ability to move to the next cell and to track their forward direction. While the moveables have differing ways of moving, whether that be controlled by the player, using AI or following a set path, they all require the base ability to move, and having this under the superclass allows for this, and allows for movement to work the same way across all moveables.

However, each subclass also requires their own attributes and methods to permit them to perform their specific tasks. Moveable is an abstract class, and therefore all objects which use it must be from an object of one of its subclasses. The Flying Assassin contains the method “turnAround” which simply permits it to rotate back on itself when it hits the level border; the Floor Following Thief contains the method “nextDirection”, which calculates the next valid spot it may take while keeping to the outside path; finally the Player, which takes in inputs from the user using the “takeInput” method to determine direction.

Both the Floor Following Thief and the Player are subclasses of a class called Thief, however the Flying Assassin is not. Thief is a subclass of Moveable which is a superclass to only 3 different subclasses under Moveable. As these 3 subclasses share similar thief-related methods and attributes that the other moveables don’t, having this subclass means a lack of repetition even under the Moveable class.

Putting both these and our other Moveable subclasses under a superclass provides consistency and a lack of repetition between similar classes, as well as cutting down the amount of code significantly.

Author: Eveleen McGovern

The Level File Format

The File System can be split into two main components: the profile storage and the level storage.

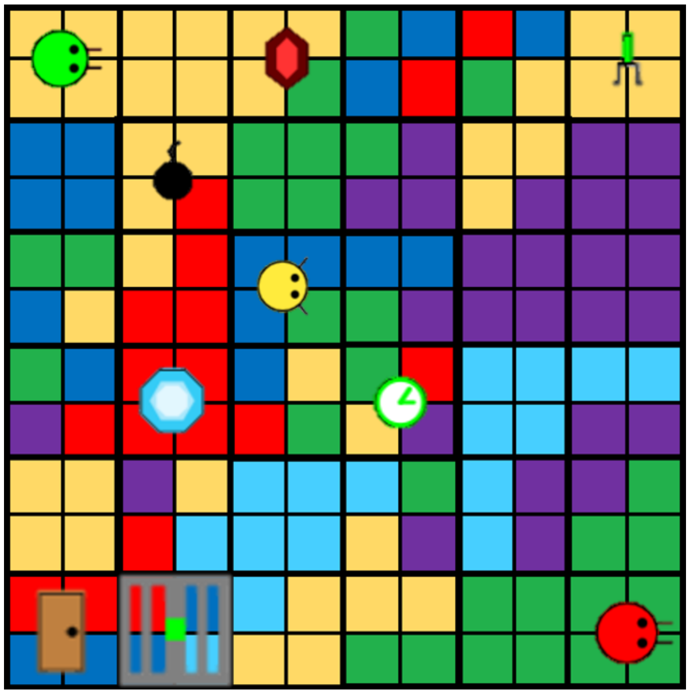
Profiles will be stored on a database that is kept in the same directory as the main exe. This database can be accessed via SQL and is an effective way of storing the player profiles. Player profile stores their chosen name, their maximum level unlocked and whether or not they have an unfinished game. The Player's name will have to be unique as this is the primary key for the Player Profile table. In addition to the player profiles the high scores tables will be kept on a database. Keeping them both in the same database allow for referential integrity i.e. A high score can't be added without a valid player profile.

The levels should be stored in a txt file. The levels will be stored alongside the launcher in a folder named levels, within the folder there will be individual folders for each level with the name corresponding to the level. (insert level txt design) When the user saves an unfinished game, a new level file is created and stored in a directory with the player's profile name as its directory name. This would mean that the next time the player logs on they can pick up where they left off.

Figure 1

```
1> 6 6

2> YYYY YYYY YYYG GBBR RBGY YYYY

3> BBBB YYYR GGGG GPPP YYYP PPPP

4> GGBY YRRR BBBG BBGP PPPP PPPP

5> GBPR RRRR BYRG GRYP CCCC CCPP

6> YYYY PYRC CCCC CGYP CPCP PGGG

7> RRBB RBBC CYYY YYGG GGGG GGGG

8> Loot Ruby 2 0

9> Loot Diamond 2 3

10> Lever Green 5 0

11> Gate Green 1 5

12> Door 0 5

13> Clock 3 3

14> Bomb 1 1
```

Figure 2



15> Player Position 0 0

16> Smart Thief 5 5

17> Thief 2 2

18>

The board size is defined in row one in a standard numerical format.
The colours of each tile is determined by the four letter block being read left to right i.e. top left -> top right -> bottom left -> bottom right.

The item’s position is defined by the coordinates given; this refers to the tile’s position within the board array.

Author: Mathew Wiliams