Partial Design Document

Group 36

Friday 1st November 2019

Candidate Classes and Responsibilities

The chosen candidate classes are: Player, Smart Targeting, Key, KeyDoor and Teleporter.

Player	
Responsibilities Provide and maintain data on the Inventory of the player. Provide and maintain data on the number of points the player has in game. Provide and maintain data on the token count of the player. Support player movement Support adding/removal of items from inventory Player can pick up items	**Collaborators** Level Item
Rough Idea: The entity controllable by the user, that needs to get to the goal in the level. Author: Samuel Roach, Chuks Ajeh Superclass: Entity Subclass: N/A	

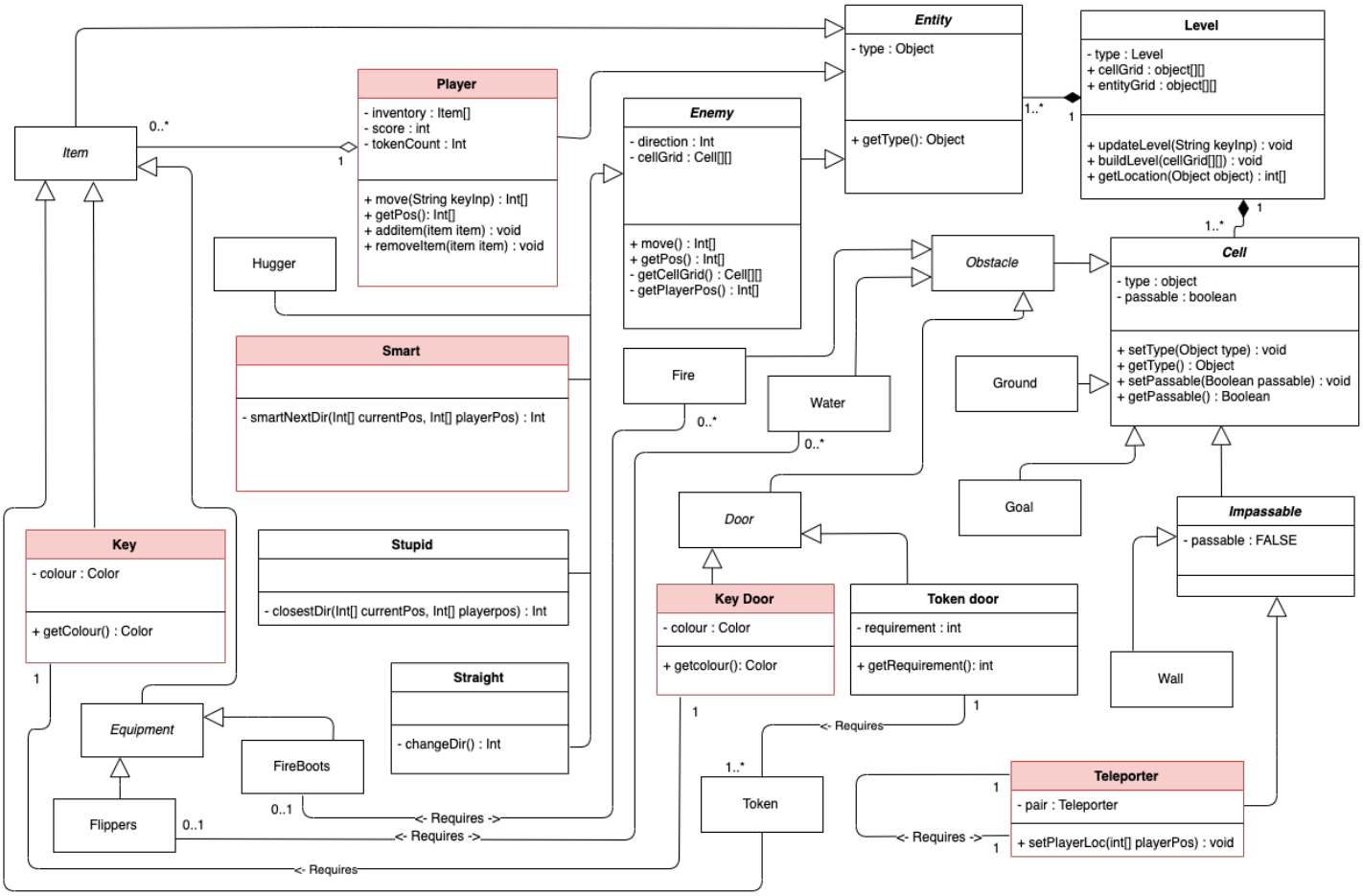
Smart Targeting	
Responsibilities get data on player position use player position data to calculate shortest path to player This will use the Dijkstra or A* Algorithm	**Collaborators** Player
Rough Idea: Variation of enemy that uses path finding algorithms to get to the player. Author: Blake Davis, George Carpenter Superclass: Enemy, Entity Subclass: N/A	

Teleporter	
Responsibilities Moves the player beyond the wall cell by changing the player position Inherits the behaviours of the Impassable class	**Collaborators** Teleporter Player
Rough idea: The cell that allows the player to jump over the wall by teleporting to the same type of cell Author: Ioan Mazurca Superclass: Impassable, Cell Subclasses: N/A	

Key	
Responsibilities Allows the Player to unlock a door that is the same colour as the key Can be picked up by the Player Is consumed and removed from the Player's inventory upon use	**Collaborators**
Rough idea: Represents a collectable and consumable Item required by the Player to complete the level Author: Angelo Balistoy Superclass: Item, Entity Subclasses: N/A	

KeyDoor	
Responsibilities Allows player movement if the specific key colour exists in the inventory Inherits the behaviours of the Door class	**Collaborators** Player
Rough idea: The type of door that allows player movement when they get the correct key in their inventory inventory Author: Chuks Ajeh Superclass: Door, Obstacle, Cell Subclasses: N/A	

Class Diagrams



Class Descriptions

Player

The player class is a sub-class of the Entity class. The Player class itself has no sub-classes. The Entity class has a composition relation with Level, as every entity present in the level will be stored in the entityGrid[][]. We decided to model the relation between Entity and Level as a composition as all instances of Entity should and can only exist within the Level. The Player class also collaborates with Item, as the Item, when not in the entityGrid, the Item must be in the Player's inventory, which is a list of Items.

Teleporter

This class is a sub-class of the Impassable class, which is part of the Cell hierarchy. This class has collaborations with itself, and Player. The Teleporter class needs to interact with itself, as when the player needs to be teleported, the Teleporter must find it's pair and move the Player to that position. As the Player must come into the Teleporter in a direction, the Teleporter must find the Players position.

SmartEnemy

This class represents an enemy who can walk around obstacles while pursuing the player. It inherits from the Enemy class and collaborates with the Player class. Through inheritance from the Enemy class, the SmartEnemy class collaborates with the Level class. This collaboration will allow the SmartEnemy to use an algorithm to find the smartest way to get to the Player's location.

KeyDoor

A subclass of Door, KeyDoor is an Obstacle within the Cell Hierarchy. KeyDoor just collaborates with Player. This collaboration exists to check whether the Player has the relevant Key colour, and if so, then the KeyDoor can be accessed.

Key

Key is a subclass of Item, which is in turn a subclass of Entity. This means that the Key can either exist in the Level or exist within the Player, which thus exists in the Level. The Key has no collaborations, however, it is collaborated with by the Player class, when being removed, picked up, or identified.

Complex Behaviours

Player - move(String keyInp): Int []

The move method within the Player class is one of the most complex behaviours implemented throughout our entire system. It's aim is to return the next position that the player will inhibit in the entityGrid, given a keyboard input. When called with a keyboard input, the method will check it's current position by retrieving it from the getPos method, and then change either the x or y coordinate to return it's new position. This will then be relayed to the entityGrid within the Level, to check whether any interactions will be made (With doors, picking up items etc.).

Player - removeItem(Item item): void

removeItem, as a method, will check through an array of the Player to see if the item requested is within the Player's inventory, and if so, remove it. We plan on implementing the inventory as an ArrayList, which would provide easy manipulation of the Inventory, however because it will be dynamic, this could cause some errors with the Inventory containing more items than can be displayed.

Teleporter - setPlayerLoc(Int[] playerPos): void

In this method, the Players location is set. This method is called whenever the Player enters into a teleporter. Within the method, the Teleporter calculates which side the Player has entered the Teleporter from, and returns which side of it's pair the Player needs to exit. It does this by getting the Pair's location from the Level class, and seeing where the Player is relative to the Teleporter itself.

SmartEnemy - smartNextDir(Int[] currentPos, Int[] playerPos): Int

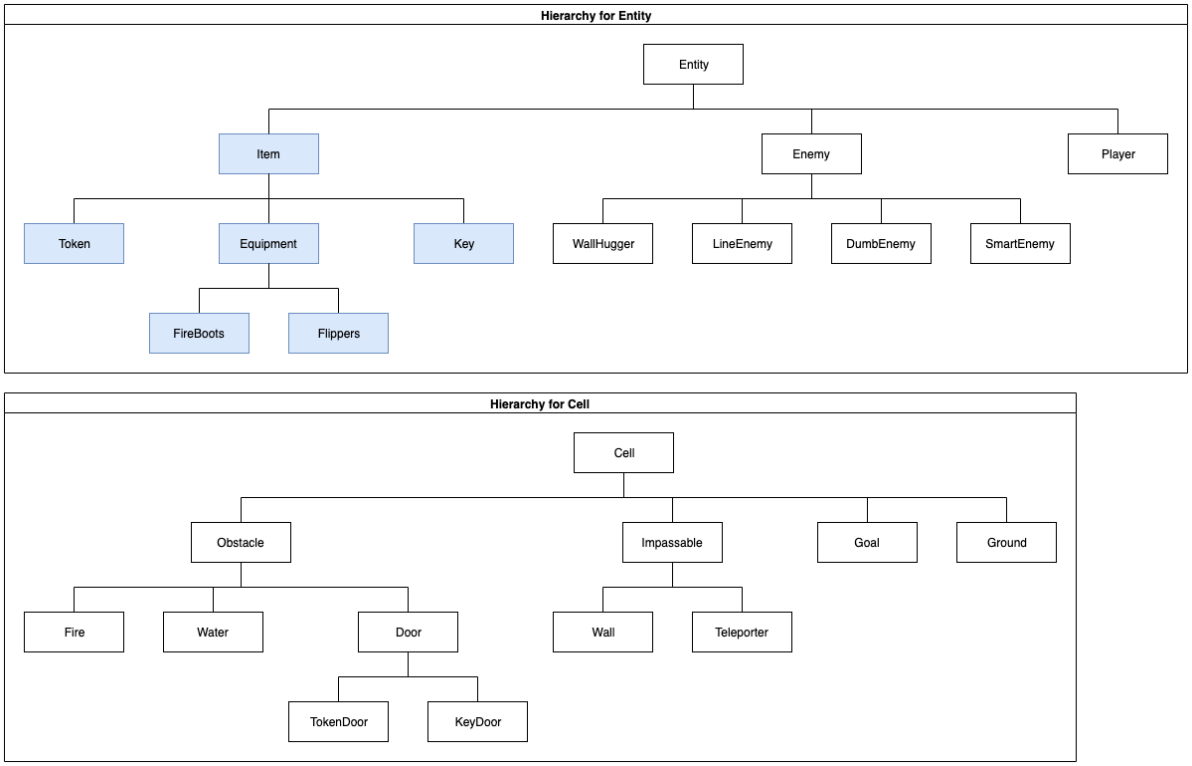
smartNextDir is the method that we will call upon every update of the level, to calculate the next position of the SmartEnemy. This method will also call upon the cellGrid, which is inherited from the Enemy class, but as this is static it won't be needed to be passed in. Ideally we'd implement Dijkstra's algorithm in order to calculate the best route from the SmartEnemy to the Player, although due to constraints in time and efficiency, we may resort to using A* for finding the route which SmartEnemy shall take.

KeyDoor - getPassable(): Boolean

getPassable as a method, checks for the relative Cell, whether or not it can be passed over. This predominantly is used to check whether an Item can be found in the Players inventory, thus making a Cell useable. It checks this by iterating through the Player's inventory until an item is found, of the inventory is fully cycled. This then returns a Boolean saying whether or not the Player can access this Cell.

Hierarchy Description

To begin, please concentrate on the areas highlighted in blue for ease of understanding my explanation.



The class selected is Item which is designed to be an abstract class. This is the superclass to; Key, Token and Equipment (another abstract class which has subclasses FireBoots and Flippers) all these classes can generally be classified as some sort of item. Our use for abstract classes allows us to encapsulate certain ideas and make writing much simpler. This will make it clear that an Item object or an Equipment object should never be instantiated. Instead, we will be forced to instantiate more specific item types such as a Key or FireBoots.

Furthermore, the clear benefit to putting all these items in this hierarchy is the type attribute and getType() method, (inherited by Item from Entity) is inherited down through all highlighted subclasses and so can be classified for further use in validation. This allows our classes to more easily collaborate for instance tokens can be classified as Token type to be then used in TokenDoor class. We can ensure that it's not the Key class being used by mistake.

The other benefit to this is we can provide further specialisation. Such is the case in Equipment. The justification for this is that whilst these are items that can be picked up, these are items (FireBoots and Flippers) that fundamentally behave differently to the items such as Key or token, where token and Key may provide entry to the next level, FireBoots and Flippers change elements of the level and aid the Player to getting the other items, and as such need further specialisation. As well as this, these items are not consumable and will stay with the Player until the level resets.

Overall, we believe our design of in-game items will make their implementation easy to write and easy to maintain and debug. Each item has been separated into subgroups when needed in order to reduce code duplication and allow us to abstract certain behaviours and attributes away in order to focus what differentiate one specific item from another.

Level File Format

The file format will be a text-based file using ASCII. These files will be stored in a simple .txt file as these are most easily accessed and used within Java, and can be easily manipulated for saving and loading files. The file format is as follows:

- Level Size – The first section will be 2 integers, separated by commas. The cellGrid[][] will be expanded to fit this size, where 0,0 is the top left corner of the grid
- Layout – The second section of the file format will be a visual representation of the level. This will contain information on the placement of walls within the level
  - Wall's will be represented by "#"
  - Ground will be represented by " "
- Details – The third section will be the detailing of the level. This detailing will lay out every entity within the level, and all the "special" Cells, comma separated. This will follow an exact order, declaring what the item is, and all its distinguishing properties. The order will be as follows:
  - Player,x,y,
    - Where x is the starting location of the Player on the x axis and y is the starting location of the Player on the y axis
    - This can only be in the level file once
  - EnemyType, x, y, direction,
    - Where EnemyType is the type of enemy which should spawn, x and y are the coordinates where the enemy spawns within the level, and direction is an integer between 0-3, which represents the way the enemy is heading when it first spawns
    - This can be duplicated as many times as needed for enemies within the level
  - Goal, x, y,
    - Where x and y are the coordinates for where the goal will be placed within the level
    - This can only be in the level file once
  - KeyDoor, x, y, colour,
    - Where x and y are the coordinates of the KeyDoor within cellGrid[][] and colour is the colour of the Key which can unlock the door
    - This can be in a level multiple times
  - TokenDoor, x, y, requirement,
    - Where x and y are the coordinates of the TokenDoor within cellGrid[][] and requirements is the amount of Token's which are required to open the door
    - This can be in a level multiple times
  - Key, x, y, colour,
    - Where x and y are the coordinates of the Key in entityGrid[][] and the colour is the colour of the Key
    - This can be in the level multiple times
  - Token, x, y,
    - Where x and y are the coordinates of the Token's location within entityGrid[][]
    - You can have multiple tokens within a level
  - Fire, x, y,
    - Where x and y are the coordinates of the Fire within the cellGrid[][]
    - There can be multiple Fires within a level
  - Water, x, y,
    - Where x and y are the coordinates of the Water within the cellGrid[][]
    - There can be multiple Waters within a level

- FireBoots, x, y,
  - Where x and y are the location of the FireBoots within the entityGrid[][]
  - There can only be one FireBoots within the level
- Flippers, x, y,
  - Where x and y are the location of the Flippers within the entityGrid[][]
  - There can only be one Flippers within the level
- Teleporter, xFrom, yFrom, xTo, yTo,
  - Where xFrom and yFrom are the coordinates of the first teleporter within the cellGrid[][] and xTo, yTo are the coordinates of the second teleporter
  - This will create 2 teleporters, at xFrom, yFrom and xTo, yTo
  - There can be more than one Teleporter line within the level file

We decided to go for this file format as we believe that it'll be the most efficient for the Level design we have chosen. In each of the lines written, the parameters which are passed in from the level file give detail to the level class about where to instantiate an instance of the class within the cellGrid or entityGrid. This allows the level to immediately create the object and place it at it's position for the beginning of the level. This means that rather than having to computer every position at the start of runtime, we have already computed original positions at load-time, thus reducing the work load needed to load and begin playing the Level.

The parameters passed in were all chosen to represent what was needed within the class in order to be able to run. An example of this can be found within the EnemyType line, where we also request that the direction be inputted. This is because direction is an attribute within the Enemy parent class, and tells the level which way the enemy is moving next. This is significantly useful within things such as the StraightLine enemy and the WallHugger enemy, where they simply follow a set of rules.

The order was chosen so that all items which require each other are nearby. An example of this is the doors (KeyDoor, TokenDoor), and there respective requirements (Key, Token). As these are close together, when we load a level file we can check whether or not the Doors have the matching requirements, and therefore can be opened.