2.1 Partial Design Document

<u>Group 9</u>

Noah Stebbings	976098
George Cook	984336
Cai Sidaway	982445
Sindasu Sibanda	985526
Hao Wu	690523
Lok Sang Fong	690713

2.1.1 – Candidate classes and responsibilities
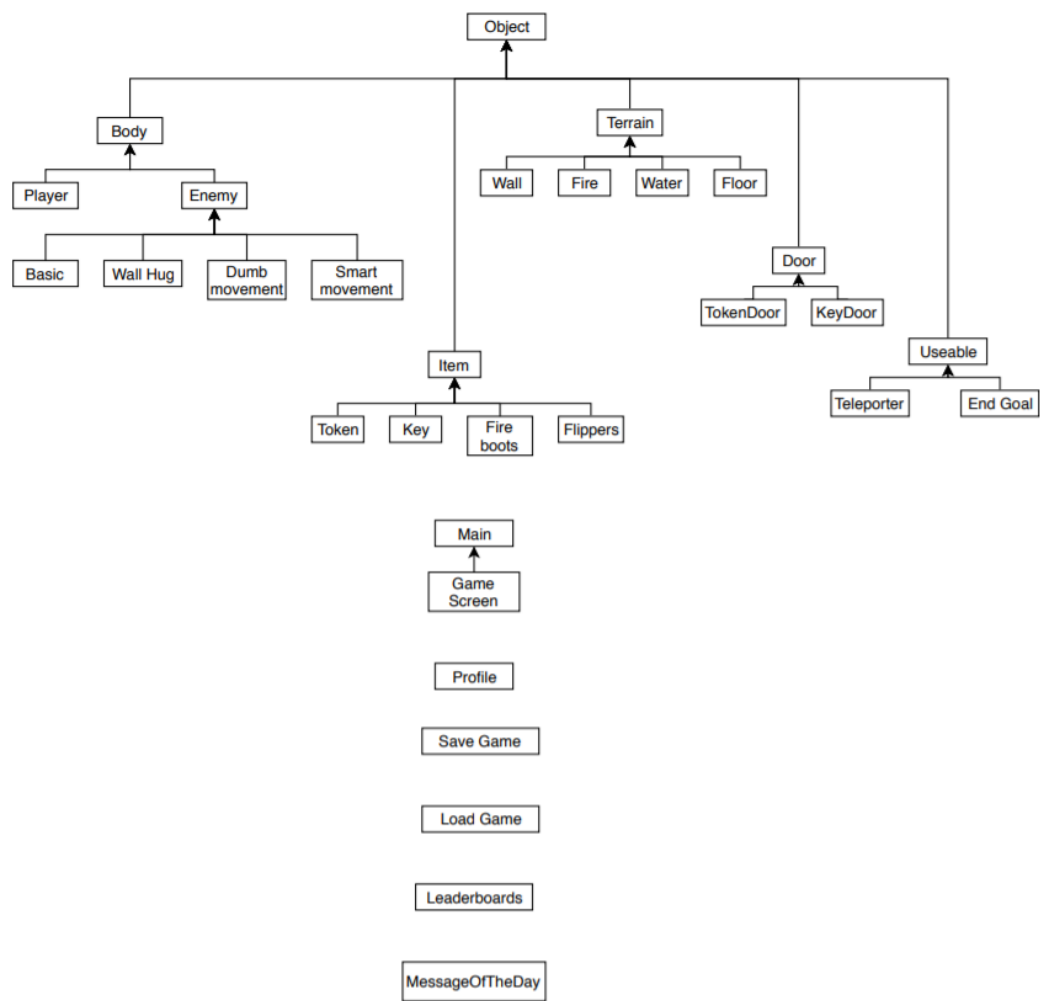
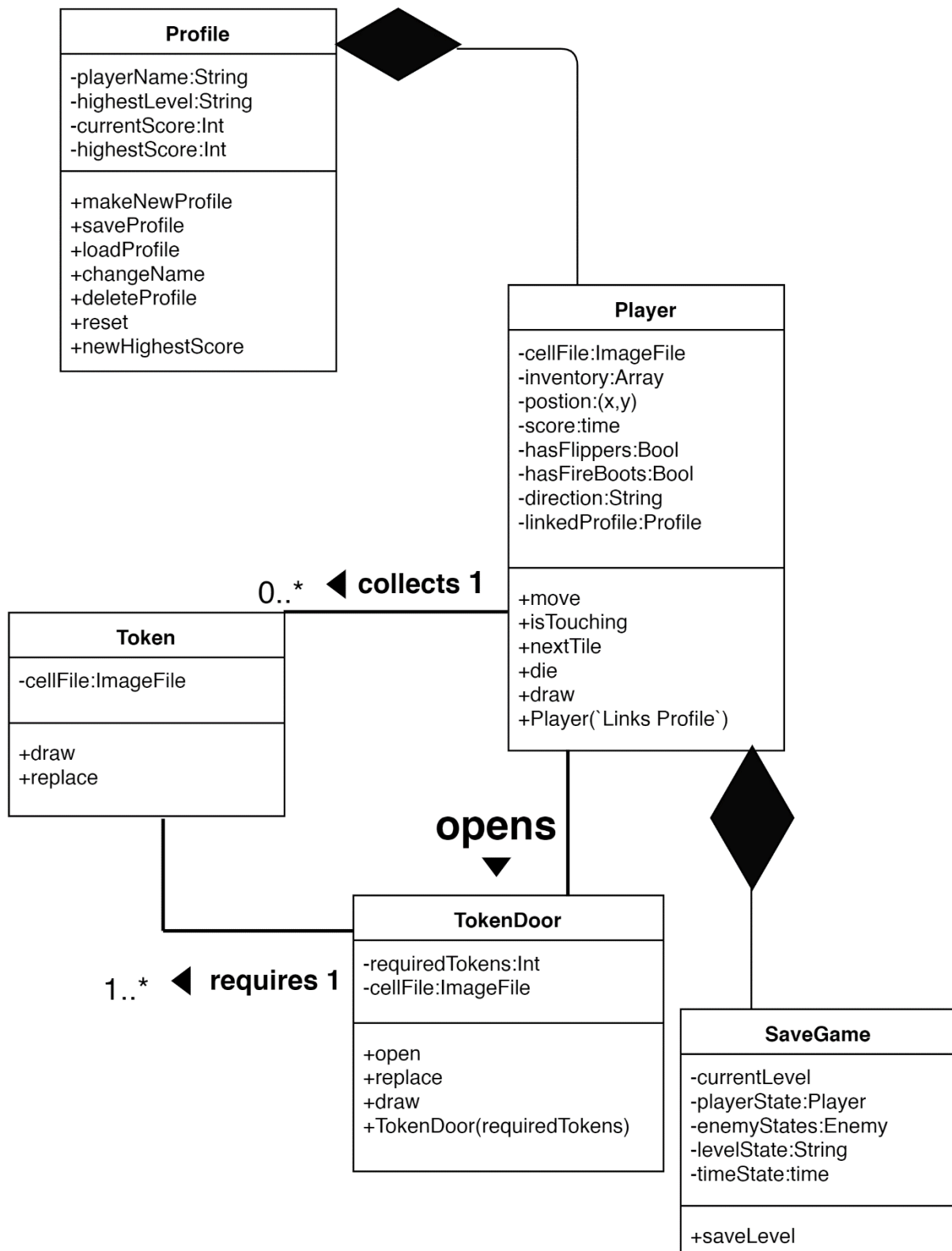| TokenDoor:	
Author: Noah Stebbings	
This class contains more specific information about the door, such as the number of tokens needed to open it.	
Superclass: Door	**Subclasses:**
Responsibilities: NumberOfTokens	**Collaborations:** Player

SaveGame	
This is a class which will allow the user to save the current state of the game. The game will also be saved after every move, to ensure data persistence. **Author:** Mohammed Raihan **Super class:** N/A **Sub class:** N/A	
Responsibilities: Save level state, inventory state, enemy positions, player position, current completion time of the level	**Collaborators:** - Game - Profile

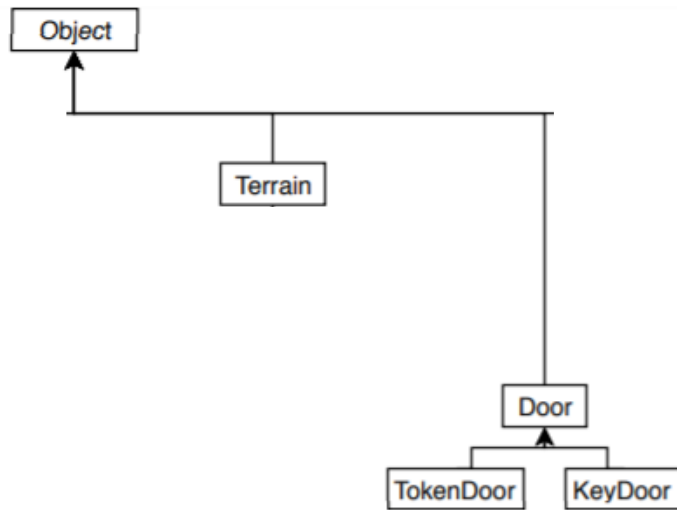| Token – Designed to house details orientating around the collectable tokens within the game	
Author: Cai Sidaway Superclass: Item Subclasses: N/A	
Responsibilities	Collaborators
- Contains a more accurate draw method - Has a 'collect' method that will call the redraw method in the **item** class, and will interact with player's inventory class, adding item to it. - Boolean *collected* will change to true, allowing enemies to walk over item	Player, Door, Item

Player	
In this class, we would designed the code that the way to move of the character controlled by player and store information of items and keys Author: Hao Wu Super class: Body Sub class: N/A:	
Responsibilities	Collaborators
Move Store data	Body Key Item Door

Profile	
This is a class that holds common information about `existingProfile` and new `newProfile` class.	
Author: George Cook	
Super Class: N/A	
Sub Class: newProfile, exisitingProfile	
Responsibilities:	Collaborators:
-playerName	-main
-highestLevel	-gameScreen
-currentScore	-profile
-constructor	
-deleteProfile	
-changeName	
-reset	
-createProfile	

2.1.2 Class Diagrams

Profile

-playerName:String
-highestLevel:String
-currentScore:Int
-highestScore:Int

+makeNewProfile
+saveProfile
+loadProfile
+changeName
+deleteProfile
+reset
+newHighestScore

Player

-cellFile:ImageFile
-inventory:Array
-postion:(x,y)
-score:time
-hasFlippers:Bool
-hasFireBoots:Bool
-direction:String
-linkedProfile:Profile

+move
+isTouching
+nextTile
+die
+draw
+Player(`Links Profile`)

0..* ◀ **collects 1**

Token

-cellFile:ImageFile

+draw
+replace

opens ▼

1..* ◀ **requires 1**

TokenDoor

-requiredTokens:Int
-cellFile:ImageFile

+open
+replace
+draw
+TokenDoor(requiredTokens)

SaveGame

-currentLevel
-playerState:Player
-enemyStates:Enemy
-levelState:String
-timeState:time

+saveLevel

Hierarchy descriptions:

We have chosen to put these classes into an inheritance relationship as it allows us to use abstract methods and superclasses. We made this design choice as it removes repetition of methods and attributes, as they can be put in a more general superclass, rather than being repeated several times in smaller methods. Another advantage is that the code will be more readable, making it easier for group collaboration during the development cycle, thereby increasing the overall quality of the code. A good example of this in our classes is the *door* superclass, and the *TokenDoor* and *KeyDoor* subclasses. Without the door superclass, both of the subclasses would require a method *openDoor()*, which would do the same thing, in opening the door when the requirements of the door are met. Using the superclass *door* saves having to repeat this method in both classes, as it can instead just be written once inside the superclass.

In the section of our class hierarchy shown above, both *object* and *door* are subclasses of the class *object*. This class contains even more general information than *door*, such as information such as the objects coordinates, its texture, and a general draw method. *Terrain* and *Door* are not in subclasses of each other however, as they only need to share information stored in *object*, for example, both objects require *coordinates*, however *terrain* does not require a method *openDoor()*, as it is not a door, and therefore doesn't need to be opened.

Level file format.

```
ExampleLevel - Notepad
File  Edit  Format  View  Help
#######################
#E#######t-----------r#
#R#######-----####----#
#-------3---^-F--------#
##############-########
#-------------------t#
#G############--------#
#---------#############
#P-------g#############
######################.
```

This is an example of how we plan to create the levels. We will use a basic ASCII format in order to hold information about what tile is in what location.

In this example:

- "#" is a wall tile
- "E" is the end goal
- "t" is a token
- "-" is a floor tile that can be walked on
- "r" is a red key
- "R" is a red door
- "3" is a token door requiring 3 keys to open
- "^" is a basic enemy that moves up and down
- "F" is a fire tile that can only be walked over by a player with fire boots
- "G" is a green door
- "P" is the player start location
- "g" is a green key
- "." is a symbol that tells the code that there is no more level information after this point.

A file reader would read through this document when the level is loaded and would be able to translate this into information that can then be used to generate the levels in the game.

The document would then be followed by additional information, such as the direction certain enemies move, etc. This information would just be stored as a series of integers and Strings, that would make no sense on their own, but the file reader would be able to interpret.