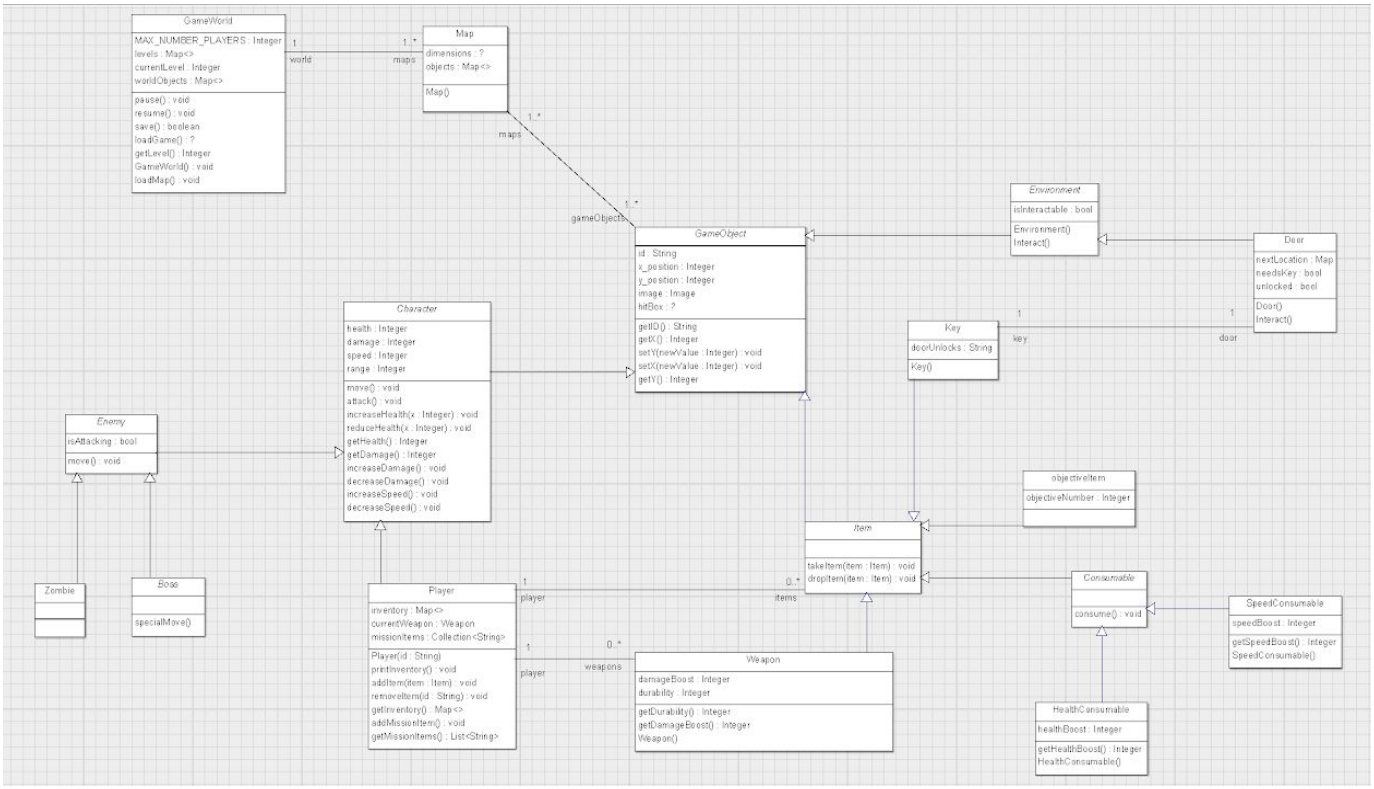
Architecture

Proposed Architecture:

UML Class Diagram:
We have created an abstract model representing the structure of our team's software using a class diagram in UML 2.x. Our model visualises the game's main abstract classes as well as the more general concrete classes and methods. When making our class diagram we wanted our model to abstract away from implementation platform-specific details as this gives us more flexibility in the implementation phase. The diagram was produced using ArgoUML which provided us with a clear UI and was conveniently installed on the university's computers, making it accessible to all team members and other groups.



Non-Standard Notations Used:
Although we designed our model to be platform independent, we also constructed it to support the Java programming language as it was in the specification that we use Java for our project. We made our diagram Java-specific by using Java interfaces such as Collection, we also specify input types, return types and visibility of all methods in the format 'methodName(input : type) : returnType ', as this information is required to construct a Java method. One thing ArgoUML didn't support that limited our Java-specific approach was interface type parameters. This meant our model was more abstract, but we are missing important implementation details. ArgoUML also doesn't define visibility visually and so doesn't allow symbols such as '+' and '-'. Therefore, our model uses non-standard UML notation in this sense, although the visibility of every element is defined within ArgoUML in the diagram file.

Class diagram Justifications:
(FR = functional requirement)

Class	Justification
Environment	- This is an abstract subclass of ‘gameObject’ which defines any immovable object within the game. - Some of the objects in the environment will be interactive such as doors so the environment has a boolean attribute defining whether a player can interact with it.
Door	- This represents any environment object that will be used to transport a player between levels/rooms/map instances. - For each door, there is one type of key that opens it.
Item	- Any object in the game which can be picked up by the player will extend the abstract ‘Item’ class.
Objective item	- Within the player's inventory, there will be a separate one storing the items required for the player to finish the game these are objective items.
Consumable	- Items that have a one time use such as speed boost damage boost.
Speed and health consumable	- These are separate as they have to access different methods to change player attributes (FR 2.0).
Weapon	- This can be equipped to the player causing a change in player damage attributes (FR 2.0). - It can also be stored in the inventory. - Unlike the other item, it has a durability which when depleted will destroy the weapon.
GameWorld	- This class is in charge of game control allowing players to pause/resume, save/load (FR 7.0, 7.1). - It also contains the methods used to change between maps allowing different rooms and environments (FR 4).
Map	- Dimensions attribute allows us to implement rooms and outdoor environments of varying size and shape allowing for many locations (FR 4). - Objects map the objects in that instance of the room with the coordinates of their location.

GameObject	- All game objects must have certain attributes and methods to be able to identify and interact with them, these attributes and methods are contained within this abstract class.
Character	- Characters within the game all share common attributes and methods, for example, the same method ‘move’ will be called for both AI and player characters, so these methods and attributes are defined in the abstract class ‘Character’.
Player	- Other classes that extend character do not have inventories so the player class defines the attributes and methods needed to implement an inventory system.
Enemy	- An abstract class defining methods and attributes shared by the zombie and boss classes.
Zombie	- Concretely implements the methods and attributes defined in the ‘enemy’ class.
Boss	- Abstract class which defines an abstract method ‘specialMove’ which will be concretely defined for each separate boss class.