# FIT2099 - Assignment 1

**// DESIGN RATIONALE //**

## Overview:

This document supplements the UML outline of the rogue-like CLI game. There are three major systems considered in the design: the movement system, the item system, and the combat system. Each system works in tandem to provide game functionality, and is comprised of a set of similar internal interactions within the overall program. Each system is described in greater detail below.

Note that the terms 'player' and 'user' are not used interchangeably in this document. 'Player' refers to the object instance within the game system, and 'user' refers to the physical person playing the game from outside the digital system.

One entity which is not described in detail is the environment / game engine / substrate. This entity manages the program as a whole and extends the provided game engine. Among its duties are:

- ➔ Taking user input
- ➔ Managing game progression, allowing relevant entities to perform actions per unit time ('tick')
- ➔ Managing an overall map of game locations and all entities stored within it
- ➔ Generally acting as both an indexing system to track game objects and initiating key actions based on user input

The precise implementation of the substrate is not of contention at this stage of the design, and will be alluded to but otherwise kept intentionally abstract.

### Movement system:

Where a position is prescribed for a class instance, it is in the form of a 'vector2d' - this will likely be implemented as a simple 2 element array which holds the coordinates of that instance. These coordinates refer to a position on the map of the room which exists within the substrate - the substrate will be tasked with initialising all object instances which require a position, so it can ensure these values are valid and provide an interface for instances to check that it remains valid.

Within a given room, there may be:

- ➔ **Environment Objects**
  All squares of the map must have an environment object on them. Other object instances below exist on a given environment object.
  There are three type of environment objects, all of which will inherit a common class:
  - Ground
    Ground tiles never block the player or NPCs. They may have an item on them (described below)
  - Wall
    A wall never allows the player not NPCs to walk on or through its position. Some walls are designated as map edges, which are automatically generated at the edges of the map by the substrate on initialisation. Beyond these walls, there is no map to exist on - good exception handling should minimise the chance of any entity erroneously existing outside map edges.
  - Door
    Doors may or may not block NPCs and players, depending on whether or not they are locked or not. Doors may be unlocked by the player using a key in their inventory.

- ➔ **The player**
  A single instance which must exist on the map, controlled by user input. The player instance holds its position and has two methods to get and set this position. These are described below.

- ➔ **NPCs**
  This includes hostile NPCs and neutral NPCs. NPCs may be on a space which is ground or an unlocked door. They may walk over items. Hostile NPCs will not exist in the same space as neutral NPCs, they will not spawn in range of the neutral NPC.

- ➔ **Items**
  All ground type environment objects have an inventory of size one, which may contain an item instance or a null value. Items will be described further in the inventory section. When the player is on a ground tile in the map which contains an item, they will automatically pickup that item and add it to the player inventory - the user will be given a popup on that turn to indicate this.

  Any NPC which walks over a ground tile with an item on it will not affect the item (though this may be a feature added in future). Miniboss NPCs drop a rocket engine on death.

The interface for moving any NPC or the player instance is polymorphic, and uses the getPosition() and changePosition() public methods for that instance to get and set its position respectively.

If the new position specified is invalid, either because it is not adjacent to the instance on the map or because it is blocked by an environment object, the instance's position will not be changed. In implementation this method may return a boolean representing whether or not the position was changed successfully, but for now the UML will indicate void.

## Inventory system:

Specific objects will be given the ability to hold and manage items.

**Player:**
They player is able to hold any number of item as it manages items via its own internal arraylist - where it can remove and add items as needed. The list can hold any item, and any item subclass type, therefore allowing the player to simultaneously hold keys, plans and rocket parts.

A player is able to pick up items off the ground if they are adjacent to the position it is held. The game engine will validate this action by confirming their positions and if that action is possible,

The only times items are removed, is when they are used in some way. This is can occur when they either trade rocket plans for rocket parts with the NPC called Q or place the part on a rocket pad for later use in building the rocket.

**Ground**:
Each ground object on the map is able to hold at most one item, quite simply, this is just to denote that an item is laying on the ground and is able to be picked up by the player. If there is no item, it will be indicated by a null type.

**Rocket Pad:**
Like the player, it has its own inventory in the form of an array list. However, its inventory will be exclusively for the rocket part types as those are the only items that will be used in building the rocket.

In keeping with polymorphic practice, the methods hasItem(), takeItem(), and addItem() are used when interacting with the inventory of the rocket pad - the validity of the items passed will be checked internally. In the event an item is unable to be placed, it will be added back to the player's inventory (as the player instance is globally defined, it is accessible and visible to the rocket pad instance).

There may exist only one rocket pad on the map.

## Combat system:

The types of interaction between the player and grunts depends on the type of grunt (The subclasses).

**Goons:**
They will continuously move towards the player, because of this, they must know the players position, and their own position at all times.To do this, the goon needs to work continuously with the game engine to keep track of everyone's positions and for it to tell them the positions it requires when called. This is also important for how the goon makes attack calls on the player, as they need to be in an adjacent position for the attack to be made and need to know when they are so they can begin attacking. The players way of attacking is exactly the same to this as well - except that the user has autonomy over where the player moves and if they choose to attack when adjacent (Goons will always attack if adjacent to the player).

As for how the actual attack works, the goon has a integer value of how much damage it deals. When it attacks the target player, that amount of health is deducted from the player's total health.

In addition, every turn they will attempt an insult, regardless of their position from the player. It will be called every time the goon makes a turn, having a certain amount of chance to succeed or fail.

**Ninja:**
As for the ninja, it relies on the game engine to tell it when the player is within 5 units of its position. Once the player has been found, the ninja will actively move one step away each turn and make an stun call on the player regardless of distance.

If the stun is successful, it will add to an attribute within the player object called called stun counter which is a integer value of how many turns they have left stunned - While this counter is above zero, the player can't make an action and will tell the game engine that their turn is complete without doing anything. Each turn this counter is active, it decrements and also prevents extra stuns from adding more to the counter.

**Miniboss system:**
The miniboss is similar to the grunts in that it has its own health attribute, is able to make an attack and has a set amount of damage. It differs because unlike the grunts, it will not be aware of its position because it doesn't need to know due to its stationary nature. Only the game engine itself will know its position in order to allow the player to make an attack on it based on adjacency.

It will also have a different of attacking methods, which may be implemented by giving it a range of different attacks of which it can choose to inflict upon the player at random each turn. (Eg. One turn it may decide to stun the player and then the next turn make a ranged attack) This design will be developed upon at a later time.

Once the miniboss' health has reached zero, it will be marked as defeated and it will call a method within itself that drops a rocket part onto the ground - of which the player can decide to pick up.