Ashford Graye Rennie

Software Design

Working Title: Project RPG

Overview

In the project brief, the company has requested a text based adventure game written in Python. It will be delivered as a turn based adventure game, in which information is presented to the player in the style of a tabletop RPG game master - describing environments and events to the player as they unfold.

This document will begin with an overview of the game's story, environment and characters before diving further into gameplay elements and how the game will work. As necessary, sections will be split into 'In Game' and 'In Code' sections.

Characters and Stats

In Game

The game will be comprised of the player character and some NPCs that can be interacted with. There will be several enemies of varying difficulties, all of which will have their own strengths, weaknesses and abilities. Most of the enemies will be encountered in the power station's battle arena.

Player

Our player character is named by the player during the game's introduction. A former member of the military during a war for the empirical force. The player chooses the character's name and former role during the game's introduction to add some immersion and variety into the player role.

Armish Cornwall(definitely need a better name)

An NPC who the player meets on their way to Piston - another former military member who at first seems quite friendly. Introduces the player to the world and provides the player's first usable items.

Station Porter

This character is met after dispatching the game's first enemy in Piston - provides some backstory to Piston and directs the player to the power station.

Power Station Guard

Met by the player when arriving at the power station - this character will inform the player of what they can do in the area.

Old Medic

An old Alliance medic from the war who is no longer at the peak of their abilities. This character will be capable of healing the character, though not to full strength.

Bazaar Vendors - Physical, Armatek and Items

Store vendors that will provide increasingly useful items to the player between rounds of combat in the arena. They will provide some backstory and further explanation of the lore behind different weapon types.

Enemies

Enemies in the game must be varied to provide variety for the player. Therefore, considering five rounds of combat in the arena, with one scripted battle upon arriving in Piston, and fighting once again before leaving - this gives seven necessary enemies.

Considering the scripted nature of when these battles take place, the enemies have been split into four difficulty levels, with their base stats set by difficulty and then adjusted up or down as described in the next pages:

Low - base 75% stats

```
    Vagrant
          Strength:                 none
          Weakness:                 none
          Abilities:
                      Swipes:       Physical, hits between 1–2 times.
                                    Chance to use: 6/10
                      Lunge:              Single Physical
                                    Chance to use: 4/10
```

Medium - base 100% stats, strength and weakness adjustment 20%

```
    Bouncer
          Strength:                 Physical Strength
          Weakness:                 Armatek Defence
          Abilities:
                      Punches:      Physical, hits between 1–3 times
                                    Chance to use: 7/10
                      BodySlam:     Single Physical
                                    Chance to use: 3/10
    Docker
          Strength:                 Armatek Strength
          Weakness:                 Armatek Defence
          Abilities:
                      LoaderFist:   Single Physical
                                    Chance to use: 7/10
                      AnchorStrike:       Single Armatek
                                    Chance to use: 3/10
```

High - Base 150% stats, strength and weakness adjust 30%

```
    Officer
          Strength:                 Magetek Strength
          Weakness:                 Magetek Defence
          Abilities:
                      Pistol Shot:        Single Physical
                                    Chance to use: 7/10
                      ArmaScope:    Single Armatek
                                    Chance to use: 3/10
    Assassin
          Strength:                 Physical Strength
          Weakness:                 Physical Defence
          Abilities:
                      Slice:              Single Physical
                                    Chance to use: 9/10
                      Blades:       Physical, hits between 1–3 times
                                    Chance to use: 1/10
```

Boss - Base 200% stats, strength and weakness adjust 50%

```
Arma Engine
      Strength:               Armatek Strength and Physical Defence
      Weakness:               Armatek Defence
      Abilities:
                  Stomp:      Single Physical
                              Chance to use: 6/10
                  Steam Blast:      Single Armatek
                              Chance to use: 3/10
                  ArmaGatling:      Armatek, hits between 1–3 times
                              Chance to use: 1/10
Final Boss
      Strength:               Physical and Armatek Defence
      Weakness:               none
      Abilities:
                  ArmaStrike: Single Armatek
                              Chance to use: 6/10
                  Frenzy:           Physical, hits between 1–2 times
                              Chance to use: 3/10
                  ArmaFusion: Heals boss by 15% of players remaining HP
                              Chance to use: 1/10
```

In Code

In order to provide better code functionality and design the game in a more modular fashion, it has been decided that the game will use object oriented programming by introducing classes into the code. This will allow the addition of variables and methods over time as the game increases in size and attach them to objects being represented in the game. By calling said variables and methods from the objects they are attached to, we can also provide consistent naming schemes for our code.

Using enemy characters as our example, we know already that there will be more than one kind of enemy. We also know that they will all have a name and stats, as described earlier. To avoid unnecessary amounts of code by providing individual sets of information for each one, we can create a class with a base set of values in already. A pseudocode example is below:

```
class enemy:
     name    <— enemy name
     stat1   <— 10
     stat2   <— 10
     method1()
          code
     method2()
          code
```

From there, we can create a set of subclasses that will adjust the values as needed for that enemy.

```
subclass enemy1 (enemy):
      name  <— enemy1
      Stat1       <— Stat1 - 20%
      Stat2 <— Stat2 + 20%
subclass enemy2 (enemy)
      name  <— enemy2
      Stat1       <— Stat1 + 20%
      Stat2 <— Stat2 - 20%
```

Even though we did not add any code for methods to the subclass, they have already been inherited from the original class and so can still be used. This will be especially useful when coding battle sequences as there needn't be new methods for each enemy - they can be simply called for each class like so:

```
subclass enemy1.method1()
subclass enemy2.method1()
```

Assuming that method1() is some kind of attack based off Stat1, enemy1 and enemy2 will run the same code and come out with a different value. This is what allows us to have a variation in enemy stats without needing to code in full sets of information for each enemy. This improves code clarity for robustness going forwards as new subclasses can be added if the game is expanded later, and improves the games performance by limiting unnecessary code processing.

Interface, Navigation & Interaction

In Game

As the game will be text based, using different colours and styles of text will be used to differentiate information and assist the players understanding of what is on screen. A typewriter effect will also be used to animate the text, providing a visual method of revealing information to the player gradually, instead of simply printing segments of text. This will also assist the player's understanding of what is being presented, as they will not need to scroll up and down the provided text, figuring out what was just printed. This rule will have one exception - when menus are presented to the player. This is simply to avoid wait times for repeated blocks of text to appear, in particular during combat sequences.

With the exception of entering the player's character name at the game's beginning, the entire game will be playable using only the number keys and the enter key. This improves the game's accessibility and reduces the risk of player entry being spurious. The game's 'interface' is a pseudo-UI based around formatting text to resemble a logical structure that is easily followed by the player, comprised of narrated text and menus that the player can choose from. These 'screens' will be implemented based upon what the player is doing at that time - for example, when visiting a location, a location 'screen' will appear. When speaking to an NPC, an 'NPC chat' screen will appear, and so on. By pre-formatting the information displayed in advance we can ensure that the delivery of information to the player is consistent and relevant. For example, the player has no need to see battle functions on screen when visiting an area for the first, or to travel to an area while speaking with an NPC. At the present time, there are five main screen types that will be presented to the player:

Locations

Location screens will relay information about an area to the player and give them actions they can perform, based upon where they are

NPC Interactions

NPC screens will allow the player to converse with or otherwise interact with NPC characters

World Interactions

Interaction screens will show the player information about items of interest they have investigated and provide a means of interaction

Player Information

Player info screens will simply display relevant information about the character. Most screens will have the ability to check the players stats, equipment and items - doing so will load a player info screen before returning the player to what they were doing

Combat Gameplay

Combat screens will show information about player and enemy statistics and provide actions for the player to take.

Within these screens, the text shown will be formatted based upon the information being shown. This provides a clear method of easily differentiating information as will be demonstrated later. The main elements of information being displayed to the player within a game screen are as follows:

Game Master (direct)

Used when the game master is addressing the player directly, as opposed to narrating events. This provides clarity between the game's environment and deliberate instruction, by imitating the duality of a game master's role in a tabletop roleplaying game. Working format - plain white italic text

Game Master (narration)

Used when the game master is narrating events to the player. Working format - gold italic text

NPC Speech

NPC speech is formatted separately from standard narration, to provide a visual cue for when the player is being spoken to by someone other than the game master. Working format - green italic text.

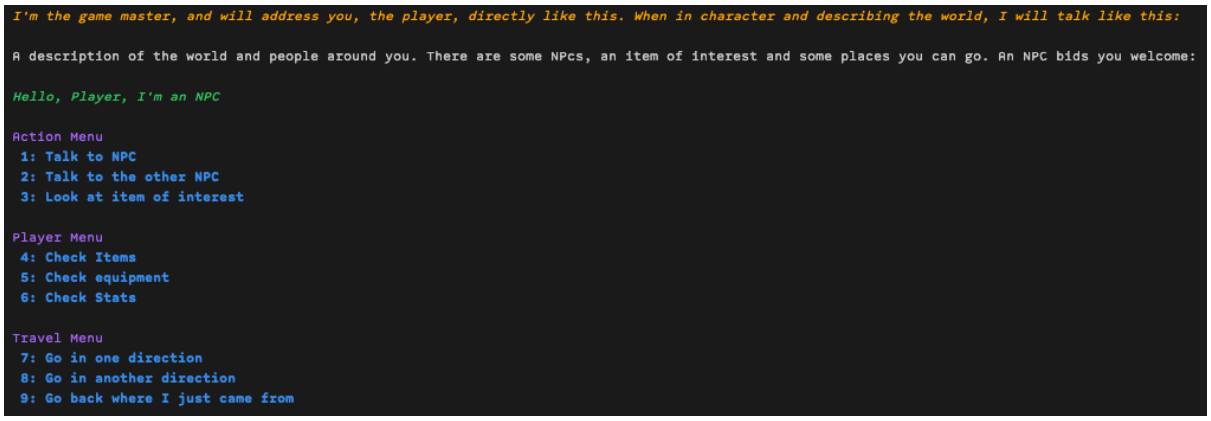
Player Input Requests

options that the player can choose from - options will be provided in a menu format, with numbers that represent the options available. This provides a consistent method of providing player actions. Working format - blue text.

Menu Titles

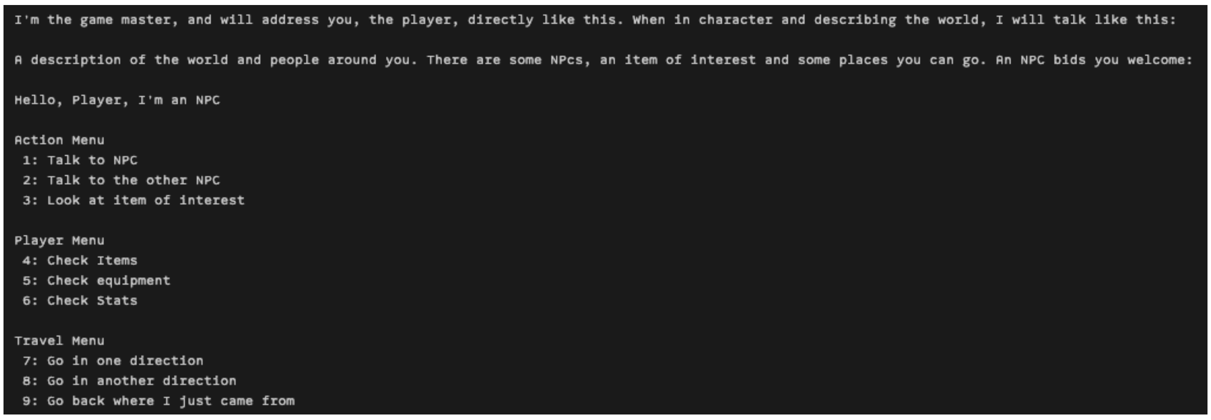
Formatted above the menus of player actions and combat stats such as player and enemy health. This provides an organisational structure to the information being presented to aid the players understanding of what's being presented. Working format - purple text.

Some examples of how this interface will work are given here. (Please be aware that the layout and formatting of game screens may change over time, however the functionality of them will remain exactly the same)



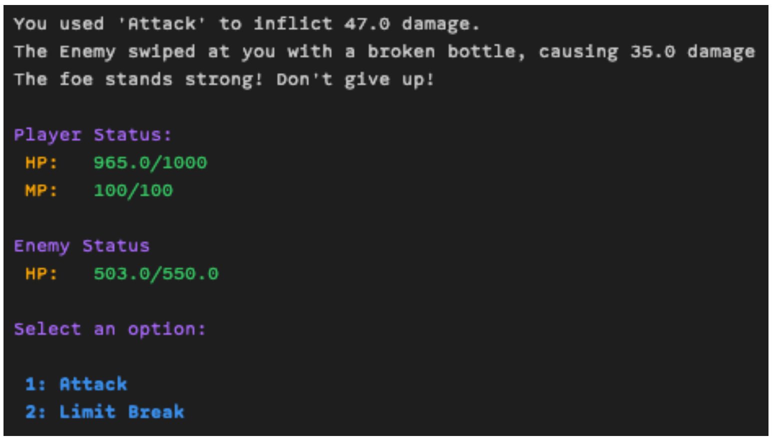
```
I'm the game master, and will address you, the player, directly like this. When in character and describing the world, I will talk like this:

A description of the world and people around you. There are some NPCs, an item of interest and some places you can go. An NPC bids you welcome:

Hello, Player, I'm an NPC

Action Menu
 1: Talk to NPC
 2: Talk to the other NPC
 3: Look at item of interest

Player Menu
 4: Check Items
 5: Check equipment
 6: Check Stats

Travel Menu
 7: Go in one direction
 8: Go in another direction
 9: Go back where I just came from
```

In this first example, the game master introduces themselves to the player directly, before demonstrating an example narration and introducing an NPC who then addresses the player. Following this, the available options to the player are presented. By formatting the differing information like this, the text becomes much easier to understand. Conversely, without such formatting the text would look like so:



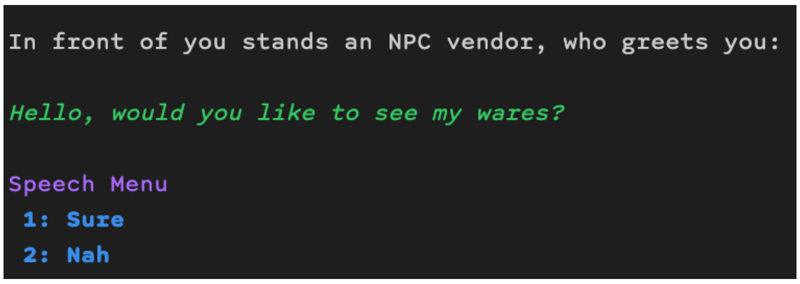
```
I'm the game master, and will address you, the player, directly like this. When in character and describing the world, I will talk like this:

A description of the world and people around you. There are some NPCs, an item of interest and some places you can go. An NPC bids you welcome:

Hello, Player, I'm an NPC

Action Menu
 1: Talk to NPC
 2: Talk to the other NPC
 3: Look at item of interest

Player Menu
 4: Check Items
 5: Check equipment
 6: Check Stats

Travel Menu
 7: Go in one direction
 8: Go in another direction
 9: Go back where I just came from
```

As can be seen, even though the text displayed is exactly the same, without the visual cues it simply appears as a block of text. Although the menu system appears largely the same, the text delivered for direct or narrated speech by the game master and the NPCs spoken lines are indistinguishable from each other.



```
You used 'Attack' to inflict 47.0 damage.
The Enemy swiped at you with a broken bottle, causing 35.0 damage
The foe stands strong! Don't give up!

Player Status:
 HP:    965.0/1000
 MP:    100/100

Enemy Status
 HP:    503.0/550.0

Select an option:

 1: Attack
 2: Limit Break
```

In our second example, an example of the player's combat screen is shown. It incorporates essential information necessary during combat in a manner consistent with how player actions are presented in the main game world. In this example, after the game master has narrated the enemy attack against the player, the same menu titles style shown in the previous example break down player and enemy stats and then provide players with actions to take in the same way that the player is accustomed to in the main game world. This assists the game's approachability by aiming to limit the learning curve required to understand game elements. Notice that the actual stats themselves are in orange text, as this is information being delivered directly to the player instead of narrated. A design decision has also been made to highlight the stats in colour - green when the stat is above 25% and red when it dips below. This is a useful mechanic that allows the player to quickly realise when their character is in danger. It will also indicate to the player that 'Limit Break' abilities can be used. Player character and enemy abilities will be explored in the Characters section.



```
In front of you stands an NPC vendor, who greets you:

Hello, would you like to see my wares?

Speech Menu
 1: Sure
 2: Nah
```

In our third example, we see an example of NPC interaction. Once again, the menu system has been implemented to ensure that the 'UI' remains consistent throughout the game.

As shown earlier, navigation is done by selecting available locations from the menu. In the background, this is accomplished by loading in the area selected, as opposed to changing location values for the player. This avoids repetition for the player where they end up typing in the same commands to move around over and over, as well as providing a consistent method of conveying exactly where the player is by letting them 'see' the surrounding area.

In Code

Screens in code work as classes containing the relevant information and formatting for that screen, and then providing subclasses for instances of those screens. For example, there will be a base location screen class, containing the formatting and layout for what is relayed to the player in every location. It will also hold the functionality required to allow the player to return to their previous location (this will be explained further in the navigation section) Using locations as the example here, the base variables would be organised like so:

```
class NavScreen
    First Visit <- true
    Name <- ""
    Description 1 <- ""
    Description 2 <- ""
    Option 1 <- "-"
    Option 2 <- "-"
    Option 3 <- "-"
    Travel 1 <- "-"
    Travel 2 <- "-"
    Travel 3 <- "-"
```

These are the variables that will then be altered by the subsequent subclasses, which is why they're initialised as blank. The initialising of the option variables as "-" is deliberate - any locations without options for those variables will simply show them on screen as "-" which signifies to the player there is no option. It also acts as a buffer for player attempts to choose it as an action - a simple function with an statement declaring an error statement to be called if the option is "-" will exist. There will then exist a display function which then calls the variables in a pre-formatted way.

```
Class Navscreem
    Display Function ()
         Print <- NavScreen name variable
         If NavScreen First Visit = True
            print <- Description 1
         Else
            print <- Description 2
         Print <- "Travel Menu"
         Print <- "0: Go back to where I was"
         Print <- "1: (travel 1 variable)"
         Print <- "2: (travel 2 variable)"
         Print <- "3: (travel 3 variable)"
         Print <- "Action Menu"
         Print <- "4: (option 1 variable)"
         Print <- "5: (option 2 variable)"
         Print <- "6: (option 6 variable)"
         Print <- Player Menu
         Print <- "7: Check Items"
         Print <- "8: Check Weapons"
         Print <- "9: Check Stats"
         <- Initialise Navigation Selection function
```

For the purpose of simplicity - the above example does not include the actual text formatting - the Print statements will exist as the text subclasses described earlier.

The first option, option 0, and the last three options are always the same, and so do not need to be set variables to be changed.

The display function will be called by a subclass when a location is navigated to, allowing information to be loaded in about the location first and then initialising the location screen itself. The If statement at the beginning is to give the game master an initial description of the area, followed by a different one in subsequent visits. This provides the player with some variety as they move around the world.

In order to facilitate player inputs from the menu, a Selection class for the screen is initialised once the screen itself has finished displaying. There won't be any subclasses which inherit the information, it's more for organisational structure - since a class can have its own variables and functions which can be called specifically, it makes sense to encapsulate relevant code within one instead of declaring it all as top line.

To provide simplicity once again, the pseudocode here will include one example for options and travel, and one example of checking player status. They will all work in the same way. We will visit option 0, returning to the previous area, afterward as it relates to location subclasses and their init functions.

The Selection class works like so:

```
Class navselect:

     Travel1 <- ""
     Option1 <- ""

     Init function:
          Selection <- input from print "what would you like to do?"
          If selection = 1:
               If navselect travel1 variable = ""
                   <- Invalid choice function
               Else:
                   clear screen function
                   set navscreen firstvisit variable<- False
                   call navselect travel1 variable
          Else if selection = 2:
               If navselect option1 variable = ""
                   <- Invalid choice function
               Else:
                   <- clearscreen function
                   <- call navselect option1 variable
          Else if selection = 7, 8 or 9
               <- Call player information screen
```

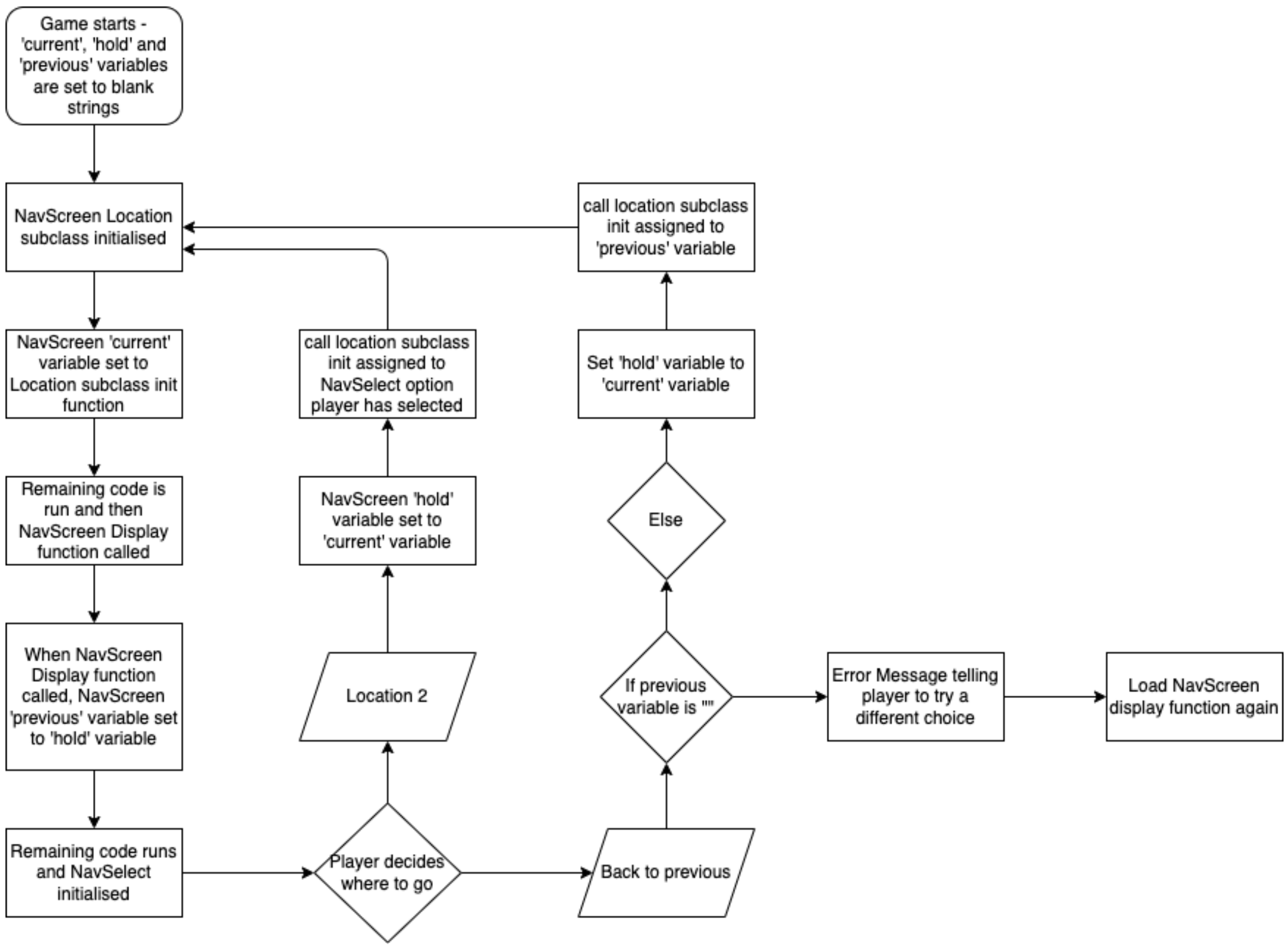
Notice that both options end with "calling" a variable. This is because the location screen subclasses will assign init functions of other subclasses to facilitate navigation and communication.

The navscreen subclasses will bind the navscreen and navselect classes together by declaring variables for both of them before they are called, like so:

```
Class Location (inherits navscreen class)
      Firstvisit <- true
      Navscreen name <- location name
      If location firstvisit = true:
          NavScreen firstvisit <- true
          Location firstvisit <- false
      NavScreen Description 1 <- story element function assigned
      NavScreen Description 2 <- alt story element function assigned
      NavScreen Option1 <- “string description of option”
      NavScreen Travel <- “String description of location to travel to”
      NavSelect Option1 <- story or npc function assigned
      NavSelect Travel1 <- specified location init assigned
```

By assigning functions to the description variables instead of manually writing in the descriptions within the location subclass, all location descriptions by the game master can be contained elsewhere within a story class where functions load in as required. This provides much cleaner code where functionality is being implemented as it isn’t cluttered by huge strings of text between what are otherwise code blocks right next to each other. The same principle applies to assigning functions to the option and travel variables. Functions being called can be stored elsewhere in the code and added as required. This also allows a basic template function to be held in their place within a screen until a more relevant one has been written. By using these template functions we can create the world locations and interaction functionality for testing so that a framework exists to be edited later.

In order to facilitate going backwards to the previous area, three variables are initialised within the navscreen class and then edited by the subclass. - these are the ‘current’, ‘hold’ and ‘previous’ variables. A breakdown of how this functionality works across the navscreen, navselect and navscreen subclasses is on the next page:


Game starts - 'current', 'hold' and 'previous' variables are set to blank strings
NavScreen Location subclass initialised
NavScreen 'current' variable set to Location subclass init function
Remaining code is run and then NavScreen Display function called
When NavScreen Display function called, NavScreen 'previous' variable set to 'hold' variable
Remaining code runs and NavSelect initialised
Player decides where to go
Location 2
NavScreen 'hold' variable set to 'current' variable
call location subclass init assigned to NavSelect option player has selected
Back to previous
If previous variable is ""
Else
Set 'hold' variable to 'current' variable
call location subclass init assigned to 'previous' variable
Error Message telling player to try a different choice
Load NavScreen display function again

As shown above, the process starts by assigning the 'current' variable a location screens init function, then assigns the 'previous' value to the 'hold' value when a display screen is loaded. When the player selects to go somewhere else, the 'hold' variable is set to whatever init the 'current' variable has been assigned. If the player decides to go back to where they were previously, the game checks to see if the 'previous' variable has been assigned "" - this is a defensive strategy to take into consideration that the first navigation screen will have nowhere to return to. Since all three variables are all initialised as "" so they can be changed later, this condition will be met and prompt the player to select another option. The condition can only be met once. The rest of the time, the 'previous' variable will have been assigned the init function of whatever was in the 'hold' variable when the location screen was initialised. At this point, the 'hold' variable is given the 'current' variables value (the init function assigned earlier) and then the init function of the selected locations subclass is called, starting the cycle again.

Text formatting in code works as a class containing all the information required to edit the format of text, then subclasses which put the formatting into practice. This allows us to specify a type of text to be displayed with the string instead of manually formatting each string. The methodology is relatively simple - a text class contains functions to write individual characters at a given speed, and format individual characters with a text colour. It is then given subclasses which call individual elements of the text class, applies them to whatever text has been entered, then resets back to normal so that the formatting does not carry forward.

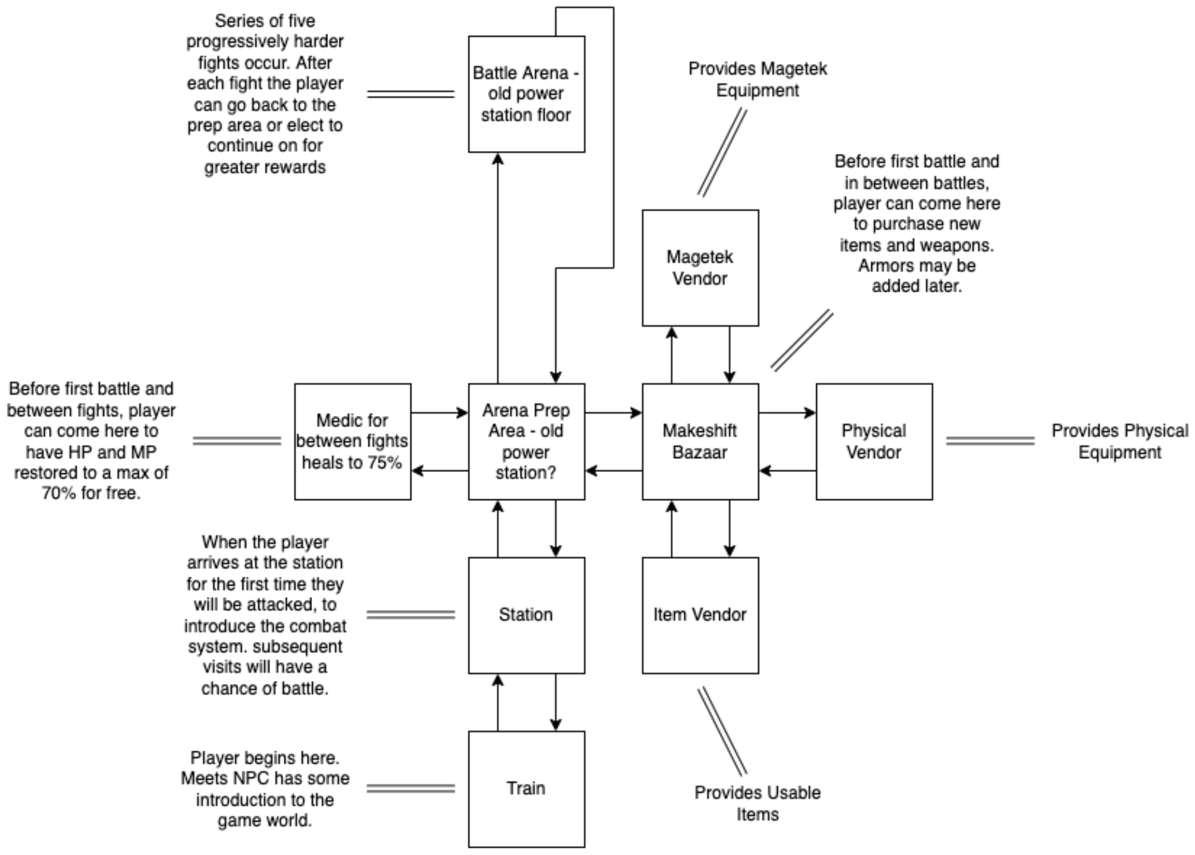
It operates like so:

```
Class Type:
     Write function(text and speed variables where speed = delay in secs)
          For <- each character <- in <- text variable
               Output <- character
               Output <- flush
               Sleep <- speed variable
     Colour function(initialise r, g, b variables)
          Return <- escape character <- foreground colour (r,g,b vars)

     Colour1 = colour function (r,g,b values for colour function)
     Reset = ANSI code for standard text

Subclass (inherit type class)
     Write function ()
          Type class write function <- format text <- string <- reset
```

Further screen subclasses work with the exact same principle. For example npc screens use the same logic as locations to have characters change the way they greet characters when meeting for a second time. Option 0 is still present in all areas, although the logic is altered slightly. NPC interaction screens will simply load the location screen again as opposed to actually going backwards, as the player is only going back a screen, not a location.

Pathing

The gameplay locations and the direction of travel is shown below:


Series of five progressively harder fights occur. After each fight the player can go back to the prep area or elect to continue on for greater rewards
Battle Arena - old power station floor
Provides Magetek Equipment
Before first battle and in between battles, player can come here to purchase new items and weapons. Armors may be added later.
Magetek Vendor
Before first battle and between fights, player can come here to have HP and MP restored to a max of 70% for free.
Medic for between fights heals to 75%
Arena Prep Area - old power station?
Makeshift Bazaar
Physical Vendor
Provides Physical Equipment
When the player arrives at the station for the first time they will be attacked, to introduce the combat system. subsequent visits will have a chance of battle.
Station
Item Vendor
Player begins here. Meets NPC has some introduction to the game world.
Train
Provides Usable Items

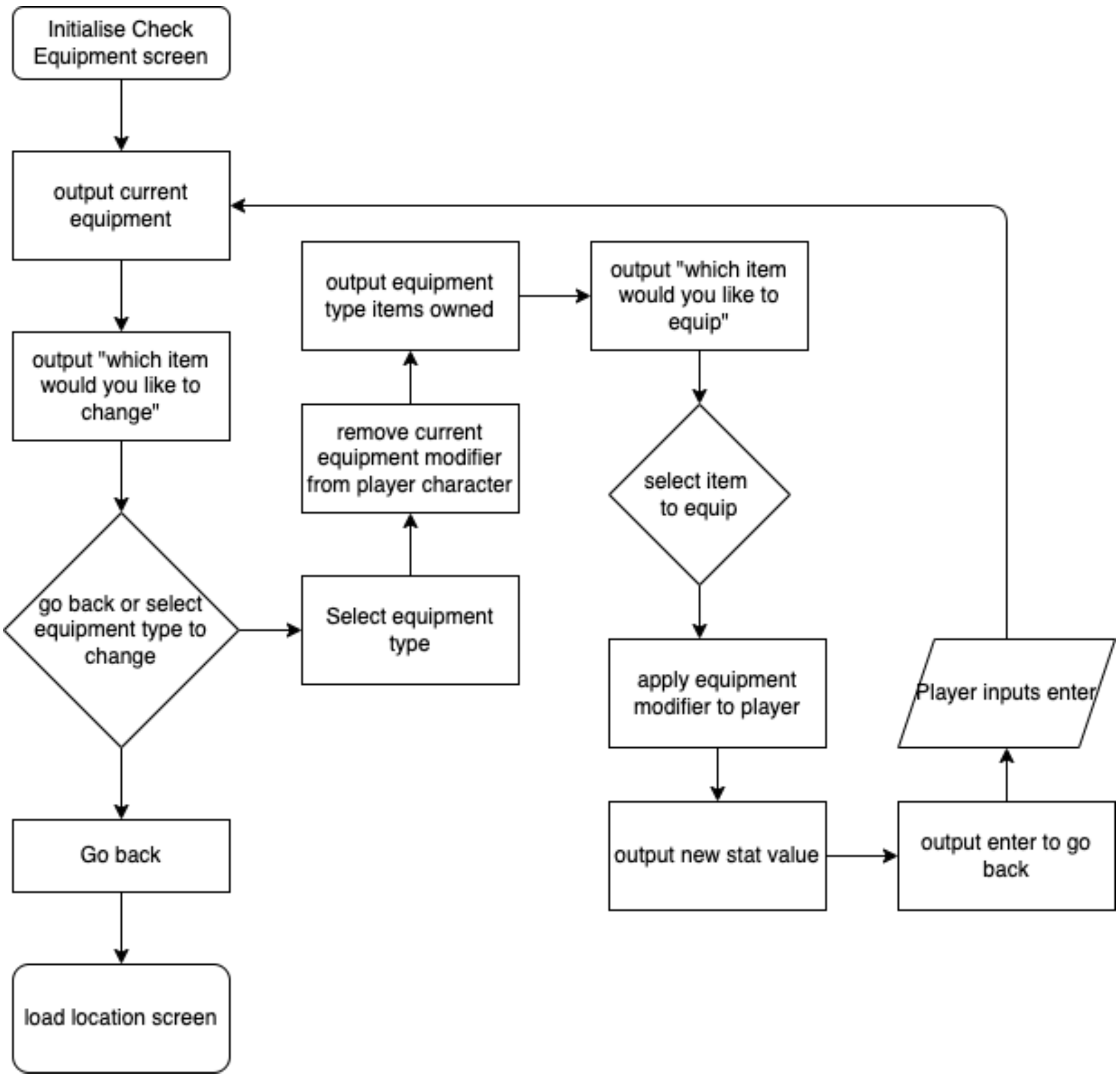
Items and Equipment

In Game

The assignment brief requires the use of or interaction with items and objects in the game world. Since a great deal of time will be spent in battle, this functionality will be incorporated in the form of equipment the player can attach to their character, as well as some offensive and defensive consumable items that can be used during combat.

Attachable equipment will alter the player character's statistics, and items will either boost the players stats (such as healing items) or hinder/hurt the enemy by reducing their stats.

The method of attaching equipment to the player and modifying stats with it is illustrated below:


Initialise Check Equipment screen
output current equipment
output "which item would you like to change"
go back or select equipment type to change
Go back
load location screen
Select equipment type
remove current equipment modifier from player character
output equipment type items owned
output "which item would you like to equip"
select item to equip
apply equipment modifier to player
output new stat value
output enter to go back
Player inputs enter

In Code

To do this we will be using dictionary variables in the game to store not only the names of items and weapons but also the values they hold:

```
Items <— {item1, value; item2, value}
Weapons <— {weapon1, value; weapon2, value}
```

In the example above, if the player chooses to use item1 in battle, the value associated with it will be triggered, be that to help the player or hinder the enemy. If the player chooses to equip weapon2, the value associated with it will alter the player's stat as determined the weapon - if the weapon increases physical strength by 10%, it would appear as:

```
Weapons <- {weapon1, (physical strength variable + (physical strength
variable / 100 * 10)}
```

Items will be executed in code by calling their associated value. For example, if an item will hurt the enemy by reducing it's health by 30 points, we code like so within the combats screen selection class:

```
If player action = (item)
     Enemy health variable - 30
     Player items <- remove item from dictionary
```

Attachable equipment will be accessed from location screens using the 'check equipment' option described in an earlier section. In order to facilitate non-destructively modifying player stats with equipment we will use an equipment modifier variable that is initialised whenever equipment is put on or removed. When attaching equipment, the equipments value is assigned to the players equipment modifier variable, then the variable is applied to the player. When equipment is removed, the modifier variable is removed from the player.

The method for doing this is below: