# Design Patterns

**Singleton**

An example of how I used a singleton was by using the Level Blueprints that are built into unreal engine. I used this to setup the environment in the scene, load the HUD and setup the player controller object, for other scenes It was used to setup the UI and the menus

Here is a series of screenshots of the Level Blueprint. It shows the setup of the HUD, player and initial variable values 

The reason I used the level Blueprint to manage all of this is because I knew it was guaranteed that it was only going to be run once, and there was no chance of an accidental duplication, this is because the level blueprint only has a single instance per level / scene that is run once when the level / scene loads.

Advantages

One of the advantages of the level blueprint is the fact that all the functions in it will always be called before the rest of the game. This meant I could use it to ensure that the required objects had the correct values and were always setup properly.

Disadvantages

One of the disadvantages of the level blueprint I think is that there isn’t really a way to view the active values of certain variables without opening the blueprint up fully. And I had an issue early on where my character wouldn’t sync with the player controller because I had to search for the player index and it wouldn’t let me find the existing player in the scene.

**State Pattern**

I used state patterns for my player and enemy controllers. For my enemy the state was configured using an enum so it would change depending on whether certain requirements were met. The Player only had a couple so it could be controlled by a set of Booleans

A screenshot of a computer

Description automatically generated

In the above code snippet a function State Machine is run on an event timer

Each iteration of State machine

* Checks the value of the enum EnemyStates
  + State Chase
    - Calls the function track player in the enemy controller blueprint which finds the location of the payer character and moves the enemy character towards it.
    - The moving animation will be played based on the movement speed of the enemy being greater than 0
  + State Idle
    - Stops all movement of the enemy character
    - The Idle animation will start based on the movement speed being equal to 0
  + State Attack
    - The Attack Function is called,
    - Set the attacking Boolean to true so the animation can be played
    - The attack animation will override the idle or moving animations and set their playing state to false
  + State Dead
    - The Stop all movement function is called
    - The Death animation is played
    - The Death Function is called at the moment the enemy runs out of health and sets the death Boolean to true

**Advantages**

One of the advantages of using the states was when I started implementing the animation. In the animation event graph I could setup a set of checks so the animation controller knew what state the enemy or player was in and play the set animation accordingly. An example of this is where I used the arming and disarming method to play the same animation in reverse and attach the staff to different parts of the player skeleton mesh depending on which Boolean was active.

Disadvantages

A disadvantage to the state pattern was probably in trying to implement an animation where a character could do different motions at once. The limit to using the enum was that it was only allowed to be one at a time and it meant my enemy wouldn’t have been able to play the attack animation while it was also running towards the player.

# Programming Principles

Dry Code

I kept my code dry by separating into functions and events and calling them instead of using the same code / BP repeatedly

Here is an example of a function I use in several locations

A screenshot of a game

Description automatically generated

A screen shot of a game

Description automatically generated

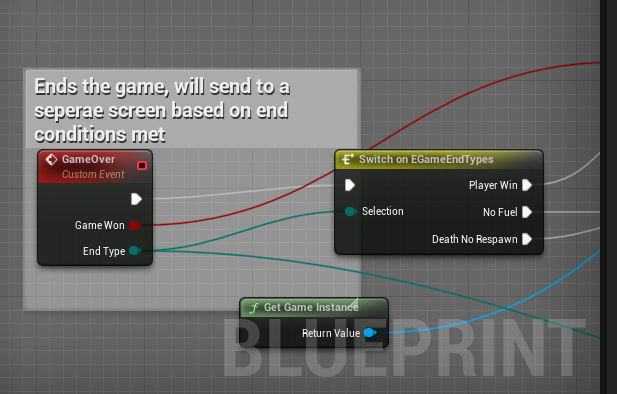
A screenshot of a game

Description automatically generated

Each of these call the same function “Game Over” but slightly different functionality depending on the circumstances of how the game ended. The three ways that the game can end is if

* The Player Wins (Lights all braziers)
* The Player Dies (No respawn points)
* The Player runs out of fuel

The game over function takes the end type and the game won Boolean and changes what it does depending on these values



The function checks the value of the enum and from the 3 possible options it will

* Change which end screen is shown

And it uses the game won Boolean to determine the

* Updated values of games played and (Games Won or Games Lost)

Dry code helps contribute to code maintainability and readability because It allows the re use of functionality rather than repeating the same code or blueprint multiple times. In addition if I need to change the behaviour of a specific function I don’t need to update it in several locations

Encapsulation

I kept data that didn’t need to be easily changed from the inspector hidden and private and only allowed access to the parts that needed it.

Here is an example of the access modifiers I had on specific variables and the interface that I could control for certain variables. For the player class I made the input actions all public so I could access them through the details panel and manually select each one

How it looks in the details panel

A screenshot of a computer

Description automatically generated

A screen shot of a computer program

Description automatically generated

I also made sure to make all the movement functions **private** because I wouldn’t need to access that variable through the interface or by other classes.

A screen shot of a computer program

Description automatically generated

Encapsulation helps maintain code and readability because it allows me to control how variables can be modified and accessed by outside classes.