Technical Design

Samurai Shampoo

v 1.1

```
public class Player : Entity
{
    bool right;
    bool left;
    bool up;
    bool down;

    [HideInInspector]
    public bool CanMove;

    [Header("UI")]
    [Tooltip("Text which displays above the player")]
    [SerializeField] private TMP_Text interactiveText;
    // numerous features can interact with the UI, li
    [SerializeField] private IngameUILogic ui;

    public static Player Instance;

    Rigidbody2D rigidbody2D;
    Animator animator;
    SpriteRenderer spriteRenderer;

    public void Awake()
    {
        if (Instance == null)
            Instance = this;
```

```
Transform
Position          X 0
Rotation          X 0
Scale             X 0.8
Rigidbody 2D
Body Type         Dynamic
Material          None (Phys
Simulated         ✓
Use Auto Mass
Mass              1
Linear Drag       20
Angular Drag      0.05
Gravity Scale     0
Collision Detection Continuous
Sleeping Mode     Start Awak
Interpolate       None
Constraints
Info
 Position         X 0
 Rotation         0
 Velocity         X 0
```

Coach: Martijn ter Schegget
Minor: HBO-ICT Games Programming

Authors:

Student	**Student Number**
Kylian de Vries	1125209
Nathan Snippe	1131636
Rixte de Wolff	1121381
Luke Tobin	1166180
Joey Einerhand	1167318
Koenraad Drost	1123524

Version control

Version	**Changes**	**Date**	**Comments**

1.0	First draft	16-4-2021	-
1.1	Health feature chapter	22-4-2021	

Distribution

Person/Organization	Date	Version

Index

1. Introduction

The technical design document to inform technical people about the features in Samurai Shampoo. This document describes, in-depth, how technical features of Samurai Shampoo are designed and implemented. The way this document is written describes the technical details of the implementation of certain features, and also explains the features with appropriate diagrams.

2. Tools, frameworks and libraries

The Unity Engine (LTS release 2020.3.0f1) is used as an engine for our project. This has a few advantages:

- Not writing a custom **engine allows more time to be spent actually creating the game** and working on AI/Game feel/etc.
- There are a lot of free **libraries and "assets"** available for the Unity Engine, a lot of which have very permissive **licenses.**
- Compared to engines such as Godot, Unity is a more **mature engine**. Therefore it's probable that Unity has focussed more on **performance, less bugs, and documentation**. It also has a bigger (And because of that, a more active) **community**.
- Unlike engines such as Game Maker Studio 2, the Unity Engine provides relatively **permissive licenses** without having to spend a single cent.
- Unity focuses more on, and is better in, **rapid prototyping** and development for small to medium games compared to the Unreal Engine.

Unity, without any external packages or plugins, only supports the programming language C#. Unity focusses purely on supporting C#, which means any other language would lack the integrated support Unity creates for C#. This is the reason C# is used for this project.

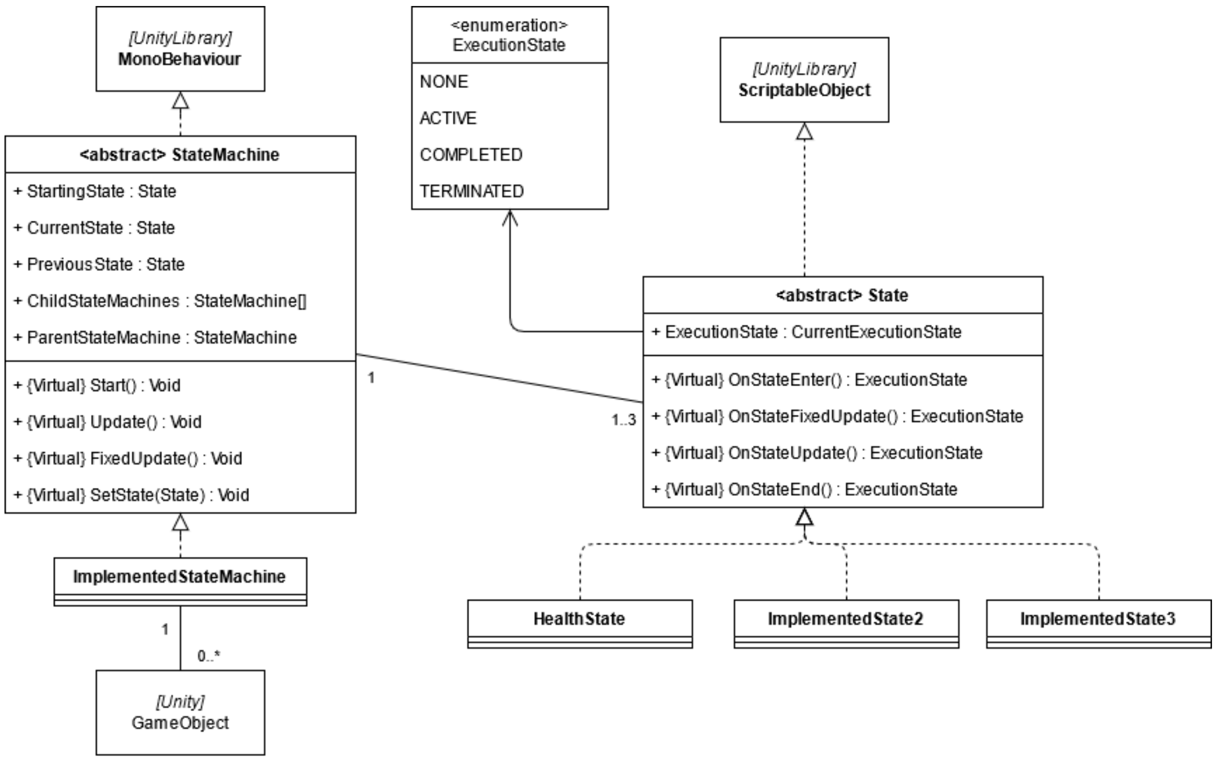
3. Software Architecture

The systems of Samurai Shampoo are written on top of the existing Unity architecture. As an example, the “entity component system” serves as an encapsulation for unity components, in order for classes like Entity to keep track of features like Movement, HP, or AI. In unity, scripts can be added to “Game objects” in order to add functionality to them. There is a way to ‘get’ the components of a certain game object and call them, as if they were fields of a game object, but doing so is expensive in terms of processing power, especially since the code would have to call this ‘getting’ each time a component is required. To work around this issue, each Entity class, which is attached to a Game object, fetches each feature attached to the same game object, and keeps track of said features throughout the game inside its class, without having to ‘get’ the components from the game object each time they are needed. This reduces the required processing power, since it only has to ‘get’ the components once.
This is an example of how the Samurai Shampoo architecture is built upon the existing Unity architecture.

4. Software Design Patterns

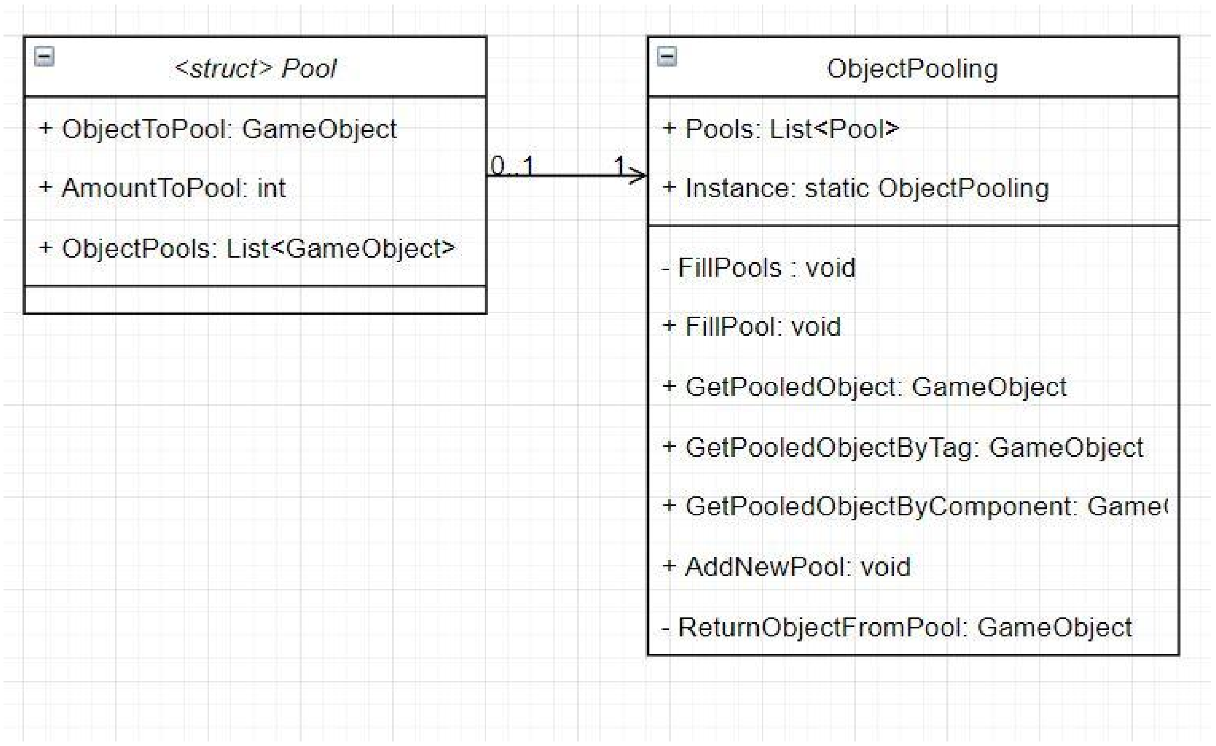
This chapter contains the design patterns that are used in Samurai Shampoo.

4.1. State Machine Pattern


[UnityLibrary]
MonoBehaviour
<enumeration>
ExecutionState
NONE
ACTIVE
COMPLETED
TERMINATED
[UnityLibrary]
ScriptableObject
<abstract> StateMachine
+ StartingState : State
+ CurrentState : State
+ PreviousState : State
+ ChildStateMachines : StateMachine[]
+ ParentStateMachine : StateMachine
+ {Virtual} Start() : Void
+ {Virtual} Update() : Void
+ {Virtual} FixedUpdate() : Void
+ {Virtual} SetState(State) : Void
1
1..3
<abstract> State
+ ExecutionState : CurrentExecutionState
+ {Virtual} OnStateEnter() : ExecutionState
+ {Virtual} OnStateFixedUpdate() : ExecutionState
+ {Virtual} OnStateUpdate() : ExecutionState
+ {Virtual} OnStateEnd() : ExecutionState
ImplementedStateMachine
1
0..*
[Unity]
GameObject
HealthState
ImplementedState2
ImplementedState3

The State Machine Pattern is used in situations where entities need to be able to switch between states. For example if an enemy is trying to chase the player, or is attacking the player. The State Machine removes the need for long switch statements and retains clarity when more states are added later. This project's specific implementation of the State Machine also tracks the previous and starting state. This allows the State Machine to switch back to a working state in cases where switching to the new state fails.
This implementation of the pattern also allows for nested state machines. Whilst nested state machines are not currently used, it might be used in the future for, for example, a boss' AI state machine, which could have an "attack" state, which would use a state machine to determine which attack it uses.

4.2. Object Pooling

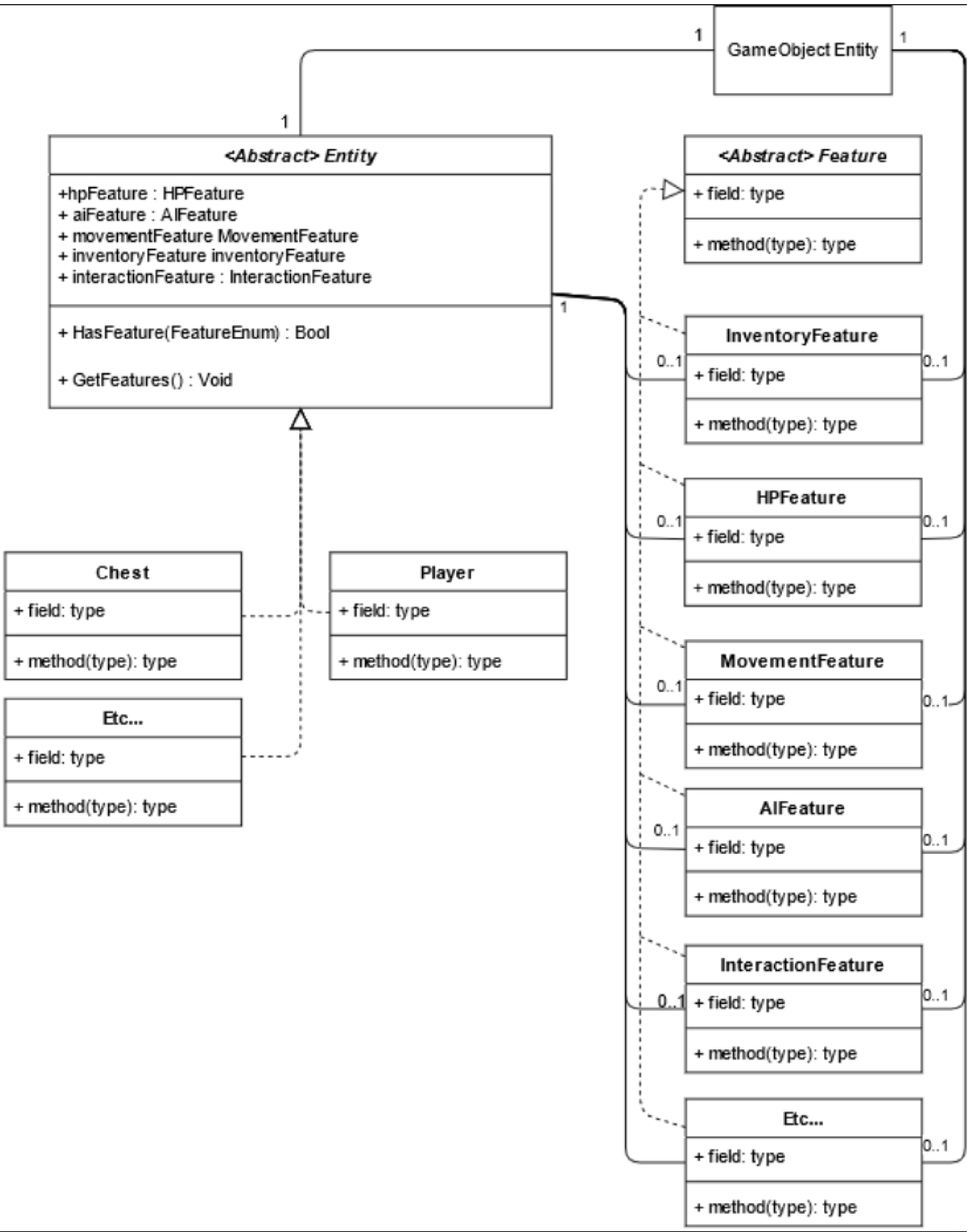

<struct> Pool
+ ObjectToPool: GameObject
+ AmountToPool: int
+ ObjectPools: List<GameObject>
0..1
1
ObjectPooling
+ Pools: List<Pool>
+ Instance: static ObjectPooling
- FillPools : void
+ FillPool: void
+ GetPooledObject: GameObject
+ GetPooledObjectByTag: GameObject
+ GetPooledObjectByComponent: Game
+ AddNewPool: void
- ReturnObjectFromPool: GameObject

The object pooling design pattern initializes a set of predefined (within inspector) GameObjects into the scene within what is called a “pool” and decativates each of the new objects. Through the use of the ObjectPooling class, another class can request an already existing GameObject from a pool. This allows the class to cut down on heavy usage of Instantiating and Destroying, but rather recycle GameObjects. This is especially useful in scenarios where you want to constantly create and destroy objects, such as with projectiles and particles.

5. Entity

This chapter contains all elements relating to the Entity class. All non-static game objects that affect the gameworld in some way will implement this class. This provides a stable definition of functionalities that the core game loop can call whenever an interaction between objects needs to be handled. These functionalities are defined in ‘Feature-class’ implementations called ‘features’. To avoid having to check for interface implementations for each object, all features are implemented through the *Entity* class itself, as seen in the class diagram below.

This chapter will first explain these features. Then the Entities that use these features are discussed.



For performance we use booleans to check whether a specific subclass of *Entity* has enabled a certain feature or not.
Separating out each feature into a self contained interface also drastically reduces the number of subclasses needed and prevents subclasses from containing duplicate code for their individual functionalities.
The Component Decoupling pattern is used for the entity component system[1].

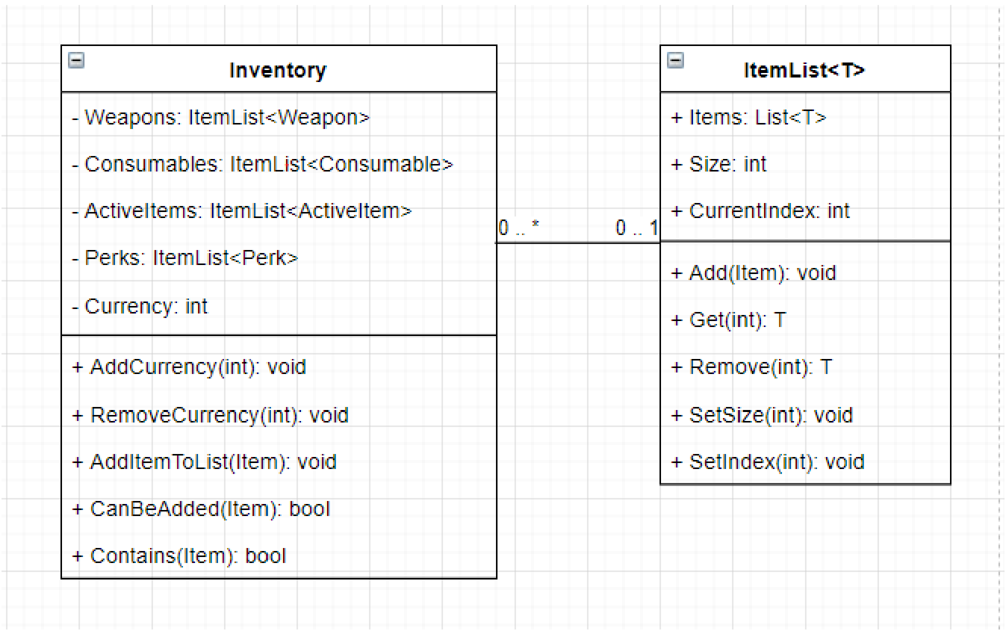
[1] https://gameprogrammingpatterns.com/component.html

5.1. Movement Feature

The movement feature is what allows Entities to move around the game world. This feature relies on the standard Unity force based movement, which works by adding a force vector to a Rigidbody 2D. Using this system makes it easy to use built-in collision detection and physics, like friction and mass.

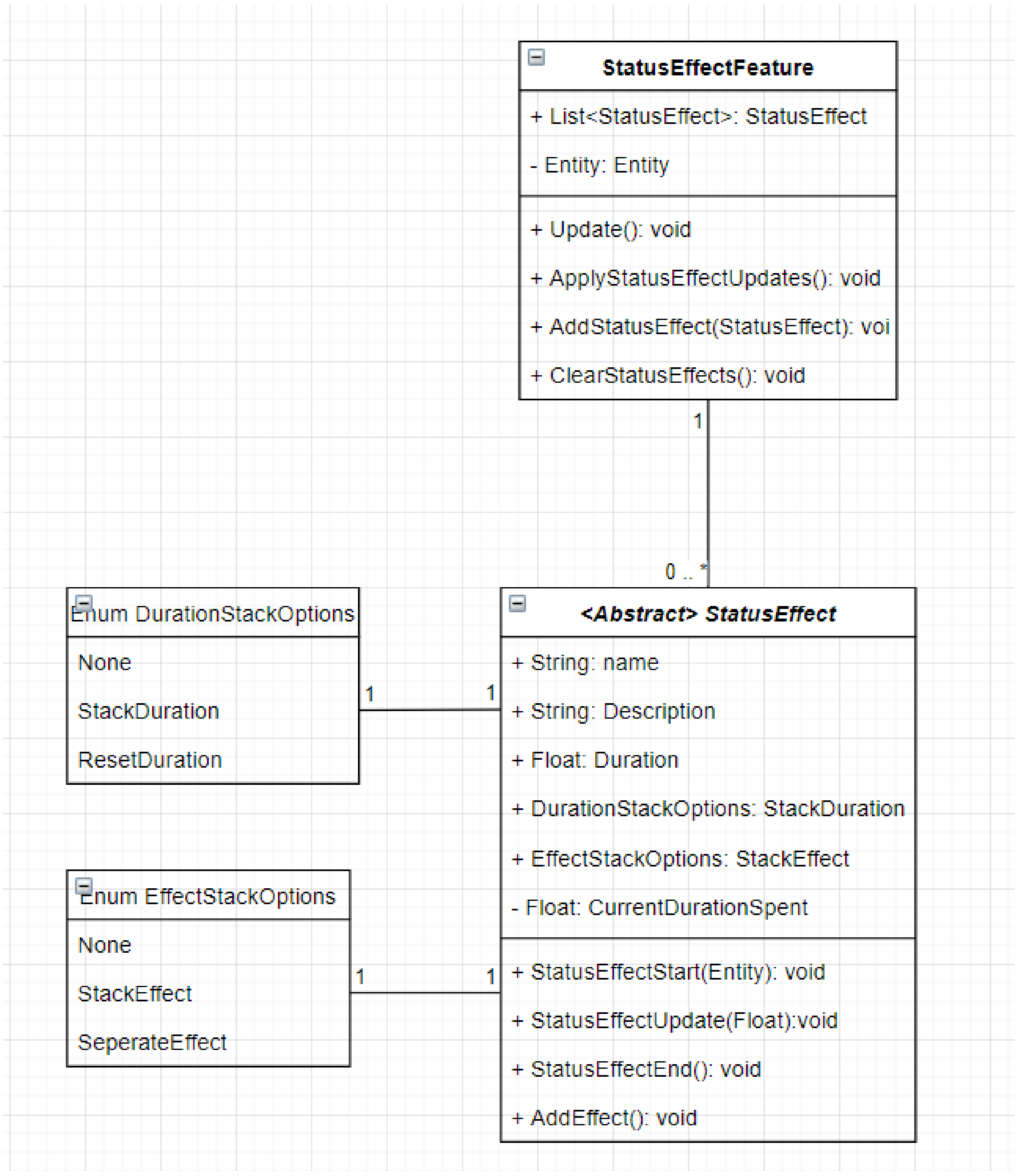
The feature contains two types of movement methods, ‘Move’ and ‘Push’. The ‘Move’ methods are intended to be used in continuous movement. These methods use the member variable *Force* to determine the strength of the force that gets added to the rigidbody. The ‘Push’ functions are meant for effects that occur once, like knockback, and take a force parameter to determine its power.

5.2. Inventory Feature



Each entity can contain an inventory. An inventory has an ItemList for every relevant item type. An ItemList is a generic list with functionality to limit the amount of items stored and to have the ability to switch out items if the list is full. When you add an item to a full list the item at index is dropped. Index and size can be changed during the game to allow for inventory buffs. The inventory contains an amalgamation of functions. The CanBeAdded function checks if the ItemList of the specified type has room for another item. If it doesn’t, the item can be added anyway by using the AddItemToList function, but the player will have to do it manually by pressing a key. InstantConsumables can always be added because they should always activate no matter the status of the inventory. The inventory also has functionality for the currency. Currency can be added or removed but will never go below zero.

5.3. Status Effect Feature


StatusEffectFeature
+ List<StatusEffect>: StatusEffect
- Entity: Entity
+ Update(): void
+ ApplyStatusEffectUpdates(): void
+ AddStatusEffect(StatusEffect): voi
+ ClearStatusEffects(): void
1
0 .. *
Enum DurationStackOptions
None
StackDuration
ResetDuration
1
1
<Abstract> StatusEffect
+ String: name
+ String: Description
+ Float: Duration
+ DurationStackOptions: StackDuration
+ EffectStackOptions: StackEffect
- Float: CurrentDurationSpent
+ StatusEffectStart(Entity): void
+ StatusEffectUpdate(Float):void
+ StatusEffectEnd(): void
+ AddEffect(): void
Enum EffectStackOptions
None
StackEffect
SeperateEffect
1
1

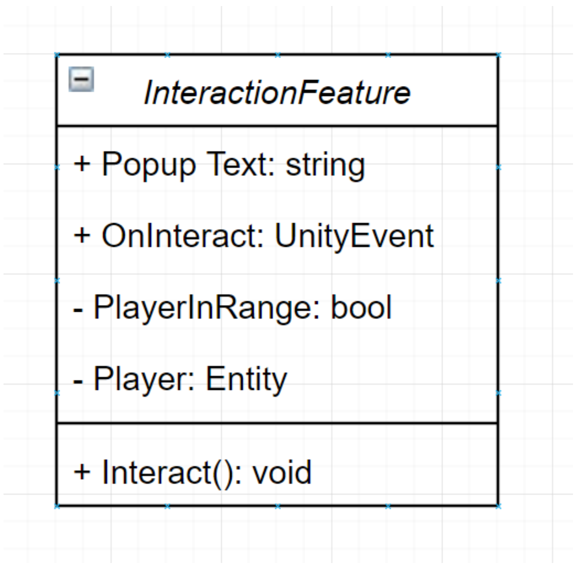
StatusEffectFeature is a class that can be added to any entity that we want to have status effects. to give an entity a status effect call the AddStatusEffect function and give it an instance of a class that derives from StatusEffect. The removal of StatusEffects will be done by the StatusEffectFeature based on the Duration.

StatusEffectStart is a function that gets called when the status effect is first applied. In case of a stat buff for example this function can be used to Add temporary stats to another feature. StatusEffectEnd will need to remove these stats again. this function gets called when the duration of the status effect runs out.
StatusEffectUpdate updates the currentDuration. It can also be used for other effects that need to be triggered every tick. Regen for example.

A status effect also has enums for stack options. These are there in case the entity manages to get the same status effect twice.
If both enums are set to none the second status effect won't do anything.
If DurationStackOptions is set to StackDuration then the duration of the second statuseffect will be added to the duration of the first.
If DurationStackOptions is set to ResetDuration then the currentDurationSpend of the status effect will be set to the Duration.

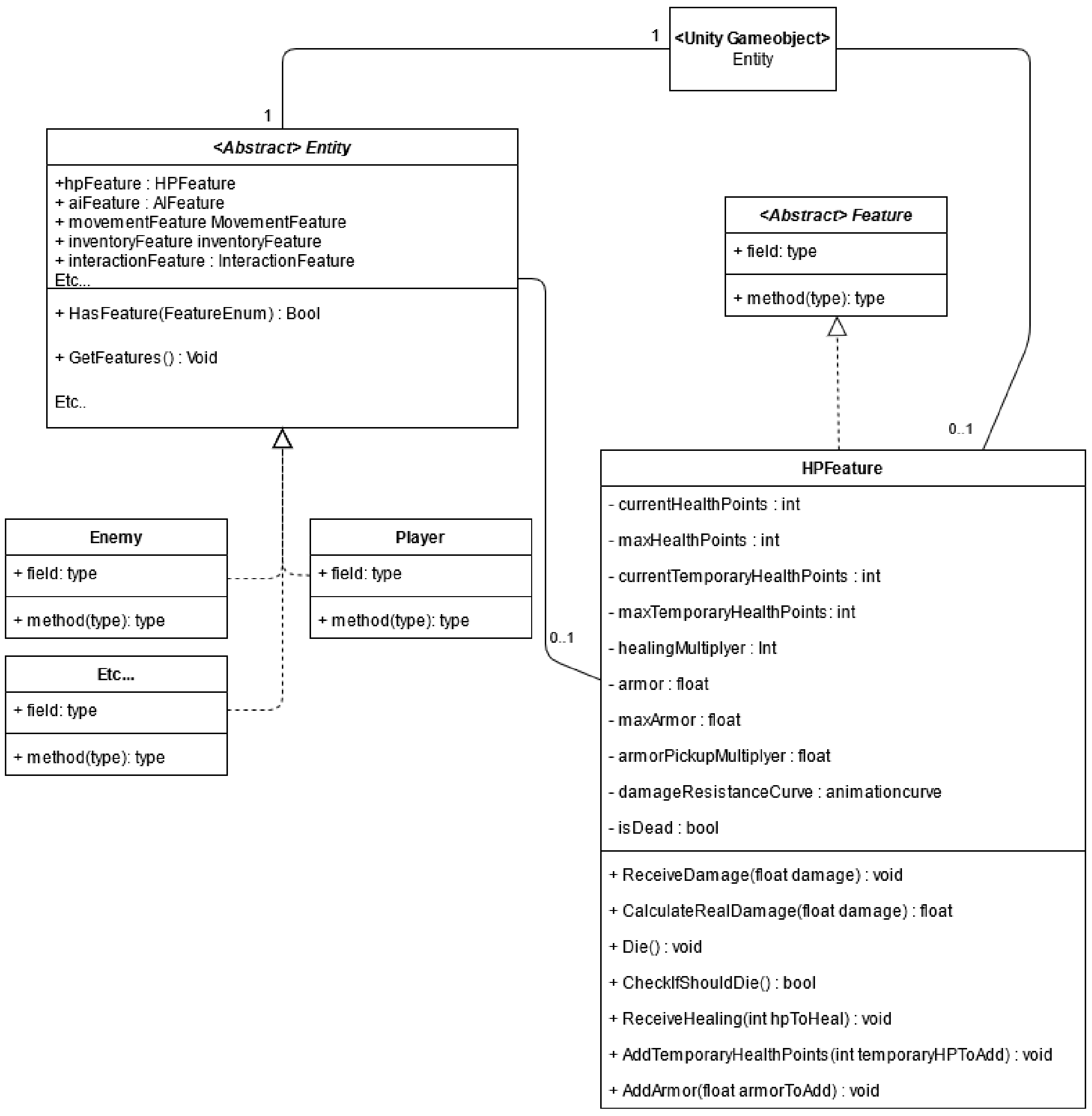
If EffectStackOptions is set to StackEffect then the AddEffect function is called. You will have to implement how the effect stacks yourself. Don't forget StatusEffectEnd might also need to change.
If EffectStackOptions is set to SeperateEffect then the entity will get the same statuseffect twice. This is probably fine from a game design standpoint but it is worse for performance and might clutter the UI.

5.4. Interaction Feature



OnTriggerEnter sets PlayerInRange to true. while it is true Update() will check if the F key is pressed and if it is Interact() will be called. Interact will trigger the OnInteract event which calls any method which has been set within the inspector, or overridden.
OnTriggerExit wil set PlayerInRange to false again.

PopupText is a text that should explain very briefly what the interaction does and what button you need to press. This text will be displayed above the player when it is in range.
A reference to the player gets saved in case the interact function needs it.


1
<Unity Gameobject> Entity
1
<Abstract> Entity
+hpFeature : HPFeature
+ aiFeature : AIFeature
+ movementFeature MovementFeature
+ inventoryFeature inventoryFeature
+ interactionFeature : InteractionFeature
Etc...
+ HasFeature(FeatureEnum) : Bool
+ GetFeatures() : Void
Etc..
<Abstract> Feature
+ field: type
+ method(type): type
0..1
Enemy
+ field: type
+ method(type): type
Player
+ field: type
+ method(type): type
Etc...
+ field: type
+ method(type): type
0..1
HPFeature
- currentHealthPoints : int
- maxHealthPoints : int
- currentTemporaryHealthPoints : int
- maxTemporaryHealthPoints: int
- healingMultiplyer : Int
- armor : float
- maxArmor : float
- armorPickupMultiplyer : float
- damageResistanceCurve : animationcurve
- isDead : bool
+ ReceiveDamage(float damage) : void
+ CalculateRealDamage(float damage) : float
+ Die() : void
+ CheckIfShouldDie() : bool
+ ReceiveHealing(int hpToHeal) : void
+ AddTemporaryHealthPoints(int temporaryHPToAdd) : void
+ AddArmor(float armorToAdd) : void

5.5. Health Feature

For the implementation of the health feature, three main factors have been used:

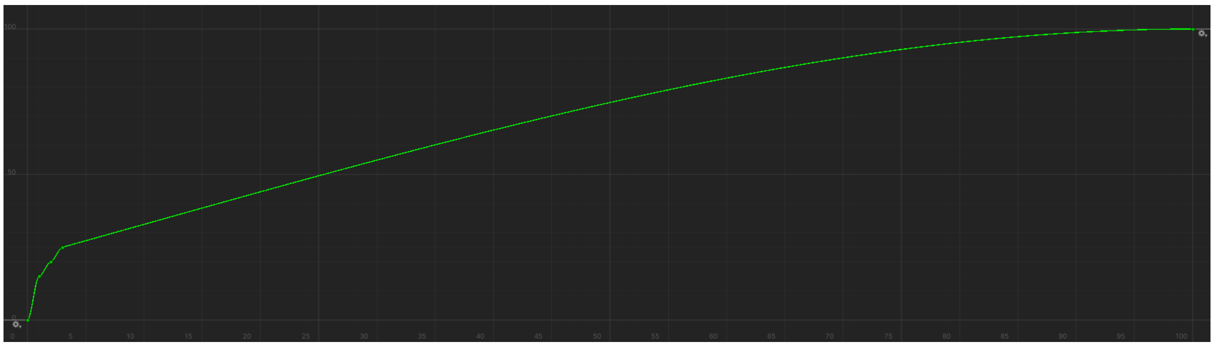
1. Health points, which represent the entity's current amount of health. Once this reaches 0, the entity should die. An entity's health can be healed up to its maximum number of allowed health points- so in a sense, an entity can regain its lost health.

2. Temporary hit points. These are hit points that in some way get added to an entity, but which are not limited by the maximum number of allowed health points.
   When an entity gets attacked, the attack reduces any number of temporary hit points before damaging the entity's health points.

   The combination of raising the player's total health beyond their allowed maximum health points, and the fact that the temporary hit points will always be reduced first, is advantageous to the player, because it allows the player to stack up on temporary health points, take damage, and not worry about losing their actual health points. The downside for players is that these temporary health points cannot be “regained” by healing like health points can- once the player loses these temporary health points, they’re gone. The only way to gain more temporary health points is through objects like powerups, potions, etc.

3. Armor. Armor isn't like (temporary) health points. Instead of taking damage and lowering its armor points, armor instead gives an entity a damage reduction rate per armor. In order to give the player the option to have as much armor as they want, a non-linear graph is used to determine how much armor reduces how many percent of the incoming damage.

   The current damage resistance versus armor amount looks roughly like this:

   Where the X is the amount of armor (from 0 to 100), and the Y is the damage resistance (from 0 to 100).

   Currently, an entity gets 15% damage reduction with 1 armor, 20% with 2 armor, and 25% with 3 armor.
   This implementation rewards players with each armor they get, but the amount of damage resistance the player gets after 3 armor is negligible. This is to give the player the choice of getting as much damage resistance as they want, but not encouraging the hoarding of armor.

Both the receiving health and receiving armor are able to be influenced by multipliers which give an increase in health and armor by multiplying the received numbers by their multipliers.

5.6. AI Feature

5.6.1. Behaviours

Behaviours are implemented by using a state machine. Currently we are using 2 state machines, one for the movements and one for attacking.

The movement state machine currently has 4 states:

- Idle, enemy does nothing until the player comes close enough
- Stand, enemy checks if it is at the desired distance from the player

- CloseIn, enemy moves to the player until it is at desired distance
- BackAway, enemy moves away from the player until it is at desired distance

The AIFeature script has a few variables that are used by these states:

- back away, the distance at which the enemy decides to back away
- close in, the distance at which the enemy decides to close in
- target distance, distance at which enemy stays in place, should be in between close in and back away
- active distance, distance at which enemies agro

The attack state machine currently has 3 states:

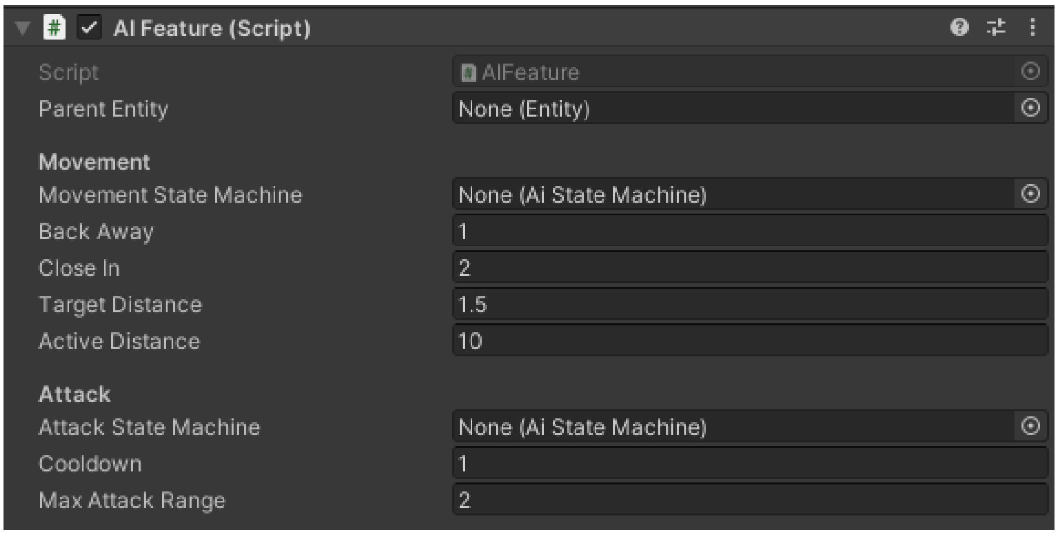
- idle, enemy does nothing until the player comes close enough
- cooldown, enemy waits for x amount of time to attack
- attack, calls attack function then immediately switches back to cooldown

The AIFeature script has a few variables that are used by these states:

- cooldown, amount of time between attacks in seconds
- attack range, range at which enemies try to attack, this value should be higher than target distance.

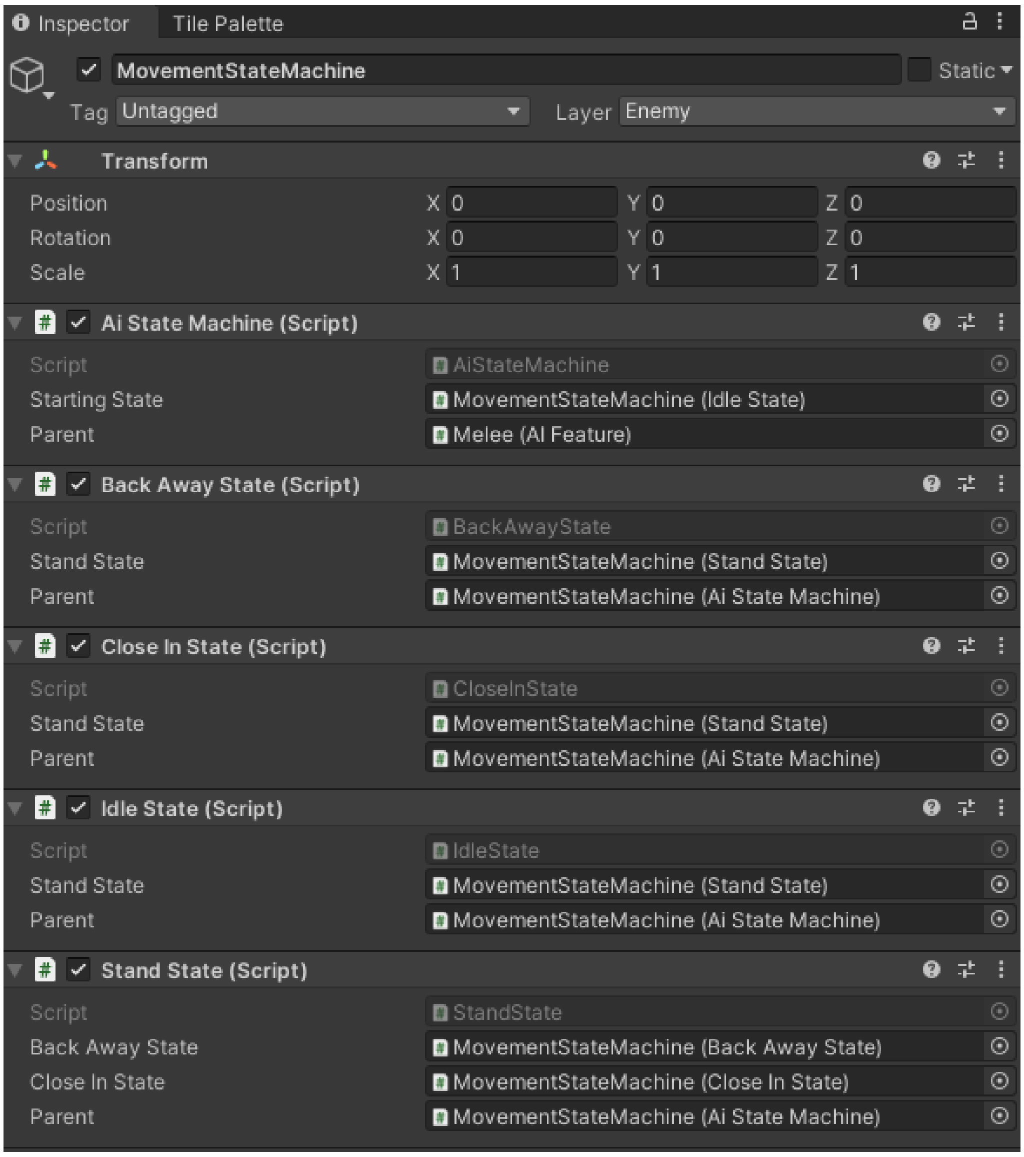
5.6.2. Implementation

To implement the AIFeature, an AIFeature script has to be added to the desired entity. The script is then automatically assigned to the entity script. The current AIFeature script contains two state machines that handle attacking and movement. These state machines should be put in seperate child gameobjects to prevent the entity's inspector from becoming overpopulated with scripts. The state machine scripts are added to the serialized fields of the AIFeature script.


AI Feature (Script)

Script	AIFeature
Parent Entity	None (Entity)
Movement	
Movement State Machine	None (Ai State Machine)
Back Away	1
Close In	2
Target Distance	1.5
Active Distance	10
Attack	
Attack State Machine	None (Ai State Machine)
Cooldown	1
Max Attack Range	2

The general variables that are used to tune the behaviour are also put into the AIFeature script as serialized fields for ease of use in Unity.


Inspector | Tile Palette

✓ MovementStateMachine | Static

Tag Untagged | Layer Enemy

Transform

Position	X 0	Y 0	Z 0
Rotation	X 0	Y 0	Z 0
Scale	X 1	Y 1	Z 1

✓ Ai State Machine (Script)

Script	AiStateMachine
Starting State	MovementStateMachine (Idle State)
Parent	Melee (AI Feature)

✓ Back Away State (Script)

Script	BackAwayState
Stand State	MovementStateMachine (Stand State)
Parent	MovementStateMachine (Ai State Machine)

✓ Close In State (Script)

Script	CloseInState
Stand State	MovementStateMachine (Stand State)
Parent	MovementStateMachine (Ai State Machine)

✓ Idle State (Script)

Script	IdleState
Stand State	MovementStateMachine (Stand State)
Parent	MovementStateMachine (Ai State Machine)

✓ Stand State (Script)

Script	StandState
Back Away State	MovementStateMachine (Back Away State)
Close In State	MovementStateMachine (Close In State)
Parent	MovementStateMachine (Ai State Machine)

The states are also added to the state machine gameobjects. The states contain serialized fields of each other to switch between themselves. Upon switching, the state instance is sent to their state machine script which is stored as the 'Parent' variable.

5.6.3. Extension

The primary way to extend this behaviour is to write more state scripts and add transitions in other states. The problem with extending the behaviour scripts is that we put all the variables the state scripts use in AiFeature. This makes the basic scripts easier to edit, but will probably not work for more complicated scripts. The variables should probably be stored in the states that use them. That way it is possible to use the same state more than once with different variables.

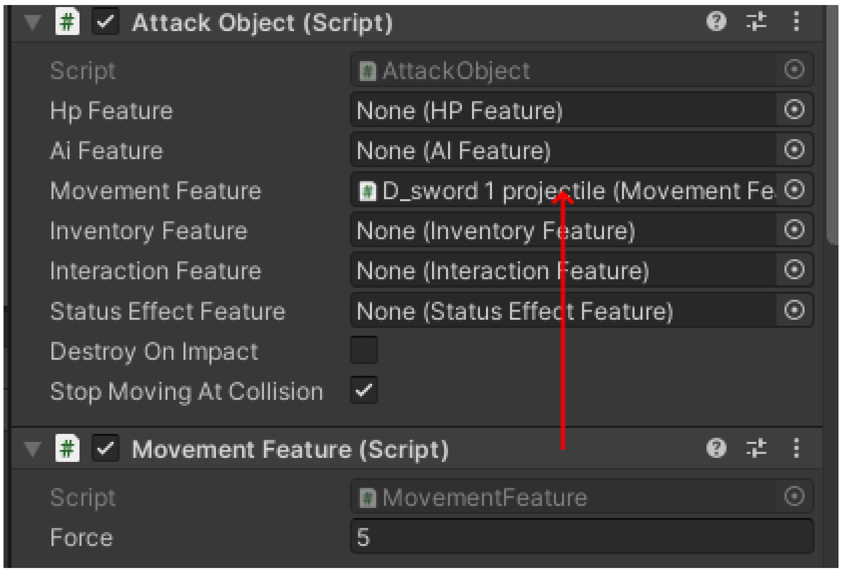
5.6.4. Boss AI

Because bosses are going to have unique behaviour I didn't add the variables for that behaviour to the AI feature. These variables will have to be changed in the state scripts. Also the bosses I made for now all use only one state machine. That's because 2 of them don't move and the third uses its movement as an attack.

5.7. AttackObject

The AttackObject is instantiated by the WeaponItem class when an attack is executed and is responsible for interacting with Entities. When an AttackObject hits an Entity, it calls the Entity's health and status application methods and applies the status and damage. The AttackObject is either destroyed on impact or destroyed when the AttackDuration expires. AttackDirection is the relative position to the weapon GameObject in the scene, which the AttackMovement script uses as the starting position for its movement.

When creating a new AttackObject, the *MovementFeature* script that is attached to the AttackObject prefab needs to be added to the MovementFeature field of the Entity, *AttackObject* (script), as shown below.

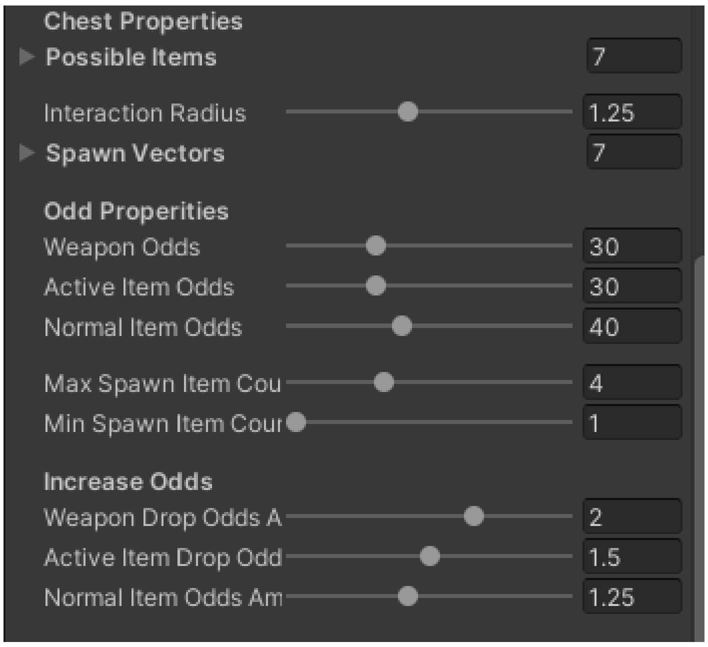


Attack Object (Script)	
Script	AttackObject
Hp Feature	None (HP Feature)
Ai Feature	None (AI Feature)
Movement Feature	D_sword 1 projectile (Movement Fe
Inventory Feature	None (Inventory Feature)
Interaction Feature	None (Interaction Feature)
Status Effect Feature	None (Status Effect Feature)
Destroy On Impact	☐
Stop Moving At Collision	✓

Movement Feature (Script)	
Script	MovementFeature
Force	5

Object Pooling
Attack Objects are generated during loading using the Object Pooling framework. This makes it so the same attacks can be reused, making the system less CPU-intensive. The sprite of an AttackObject can be changed by the weapon, allowing the developer to reuse AttackObjects for similar attacks, rather than creating entirely new objects for tiny changes.

5.8. Chests



Chest Properties

▶ Possible Items	7
Interaction Radius	1.25
▶ Spawn Vectors	7

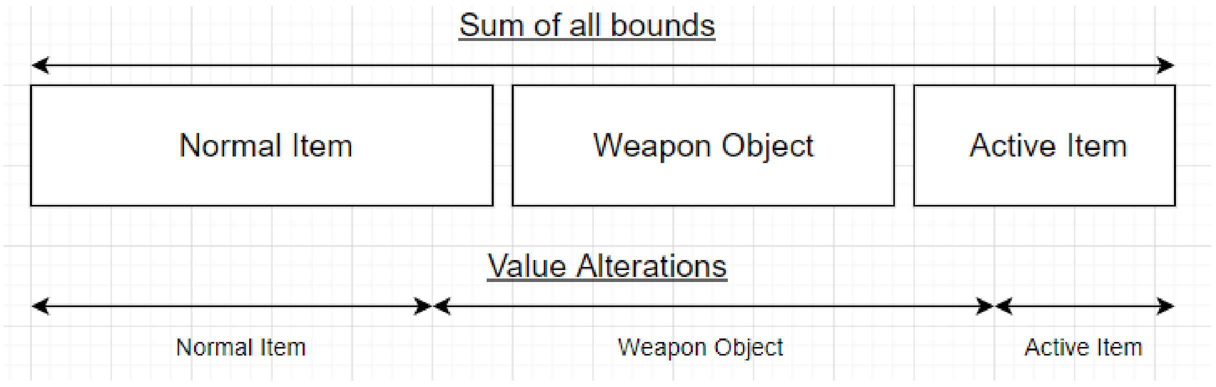
Odd Properities

Weapon Odds	30
Active Item Odds	30
Normal Item Odds	40
Max Spawn Item Cou	4
Min Spawn Item Cour	1

Increase Odds

Weapon Drop Odds A	2
Active Item Drop Odd	1.5
Normal Item Odds Am	1.25

The chest implementation is an example of an entity that utilizes the interaction feature. Chests are set up designer friendly with modularity in mind, through the inspector anyone can fill in a list of possible items which a chest can drop. A chest has a series of variables and options which are customizable to weigh and possibilities.



When the chest is opened an alteration to the odds are applied based on the player state. For example, if the player has a low amount of health, the odds of the chest dropping a healing based item will be increased.

6. Level

Throughout the game, the player traverses several levels, which in turn are made of several rooms. Each level is loaded in a separate scene. Traversal between scenes is handled by the level transition object.

If the developer wants to be able to run a scene without going through other scenes, the scene must include the following objects, which can be found in the prefabs:

- Player
- Camera
- SceneLoader
- LoadingScreen

Each level has a boss room, which opens a path to a new level when cleared. A level consists of multiple rooms that are stitched together at their doors. First, the structure and contents of rooms will be explained, then the generation of the levels.

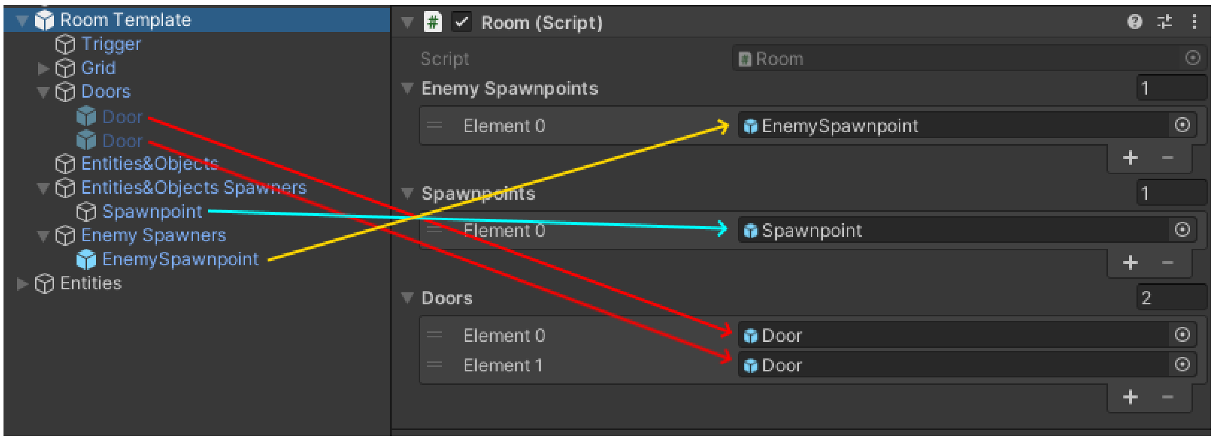
6.1. Rooms

A room is a predesigned part of a level, a prefab. Most rooms will contain enemies, in which case it will close when a player enters and open only when all enemies have been slain to force the player to interact with the content. The structure of the room prefabs is set up in a way to allow new rooms to be created without having to write any additional code. The scripts only need to be changed or expanded if completely new room types with different behaviour are created. Even then, the new room prefab can simply use a new script that inherits the existing *Room* script.

6.1.1. Structure

The 'Room' script works with a couple of child gameobjects. These gameobjects should not be changed or renamed if you are not planning on rewriting the code. In other cases, you should be able to create additional rooms without any coding.

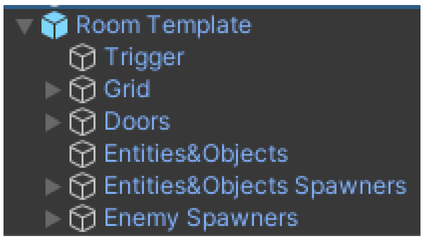
To make a new room function properly, it is very important to also add its spawnpoints and doors to the Room script component. This can be done by adding the objects to the relevant fields, as seen below. The Room script is a component of the main room object, called "Room Template" in the example.


```
Room Template
  Trigger
  Grid
  Doors
    Door
    Door
  Entities&Objects
  Entities&Objects Spawners
    Spawnpoint
  Enemy Spawners
    EnemySpawnpoint
  Entities
```
```
Room (Script)
Script                Room
Enemy Spawnpoints                       1
  Element 0           EnemySpawnpoint
Spawnpoints                             1
  Element 0           Spawnpoint
Doors                                   2
  Element 0           Door
  Element 1           Door
```

In a similar vein, each of a spawnpoint's waves has a list of enemy prefabs that can spawn there. These also have to be added in the spawnpoint's script fields (but not to the object tree). More on spawnpoints can be found in 6.1.5.

Spawnpoints can also be added to the ClearedSpawnpoints list. Spawnpoints that are added here only spawn the entity upon clearing the room (defeating all the enemies of each wave).

Each room prefab is made up of the following child objects:



```
▼ Room Template
    Trigger
  ▶ Grid
  ▶ Doors
    Entities&Objects
  ▶ Entities&Objects Spawners
  ▶ Enemy Spawners
```

The objects "Doors", "Entities&Objects", "Entities&Objects Spawners" and "Enemy Spawners" don't add any functionality themselves, they only serve to make the object tree more readable.

Trigger
An area that triggers when the player enters its collision box. When triggered, it calls the rooms *ActivateRoom()* method. There is more information in 6.1.2.

Grid
Contains the Tilemaps of the room. There is more information in 6.1.4.

Doors
The object that contains the doors of the room. There is more information in 6.1.3.

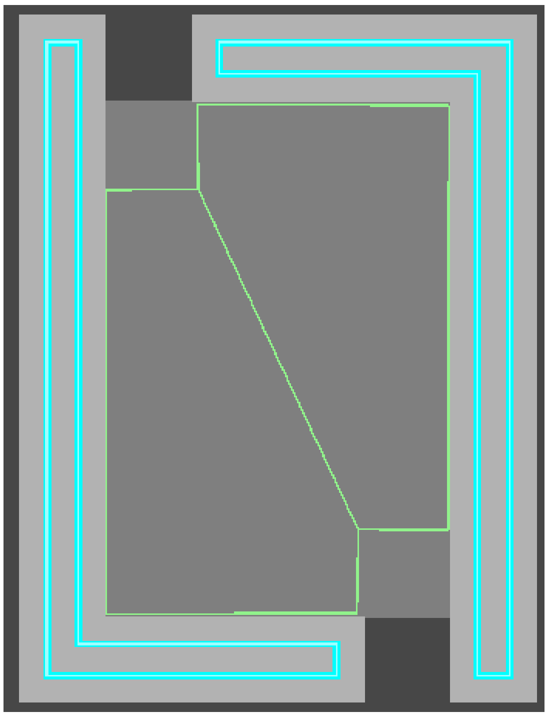
Entities&Objects
Children of this gameobject are the entities and objects in the room that are always there. This does not include enemies, as those are spawned only when the player enters the room. It does include objects such as traps, boxes and treasure chests. These objects are always part of the room, whenever the prefab is used.

Entities&Objects Spawners
This object contains the spawn points of the dynamically placed entities and objects. This is separate from the other entities and objects because these spawners can have multiple options and an associated spawn chance, meaning they may spawn nothing. There is more information in 6.1.5.

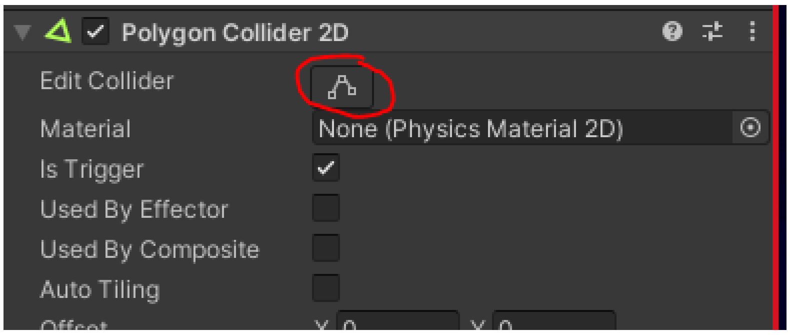
Enemy Spawners
Similar to 'Entities&Objects Spawners', 'Enemy Spawners' contains the spawn points for the enemies that spawn when the player enters the room. Each spawn point has a list of waves in which it needs to spawn its enemies. When a wave is cleared, the room makes the spawners spawn the next wave. Not all waves need to be filled for every spawner. There is more information in 6.1.5.

6.1.2. Trigger

The trigger is the area within the green lines in the picture below. When the player touches it, it will close the doors of the room. The trigger leaves about a player's length and width worth of room at each entrance so that the player doesn't have any unexpected interactions with the doors, e.g. getting stuck, glitching through or getting locked out rather than in.



The trigger area can be edited for each new room with the marked button below that can be found when going into the 'Trigger' object's inspector.

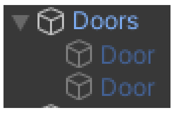


Polygon Collider 2D

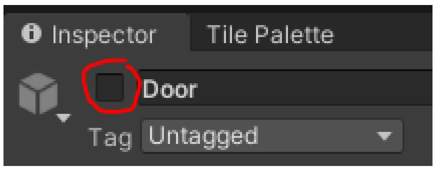
Edit Collider	
Material	None (Physics Material 2D)
Is Trigger	✓
Used By Effector	
Used By Composite	
Auto Tiling	

6.1.3. Doors

A door object requires a collision box, a tilemap and a renderer to function correctly. The door needs to be added to the 'Doors' list of the 'Room' script in the inspector of the room object. The name of the door object is irrelevant.

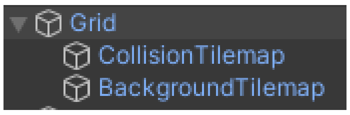


The door needs to be disabled in the editor to properly open and close.

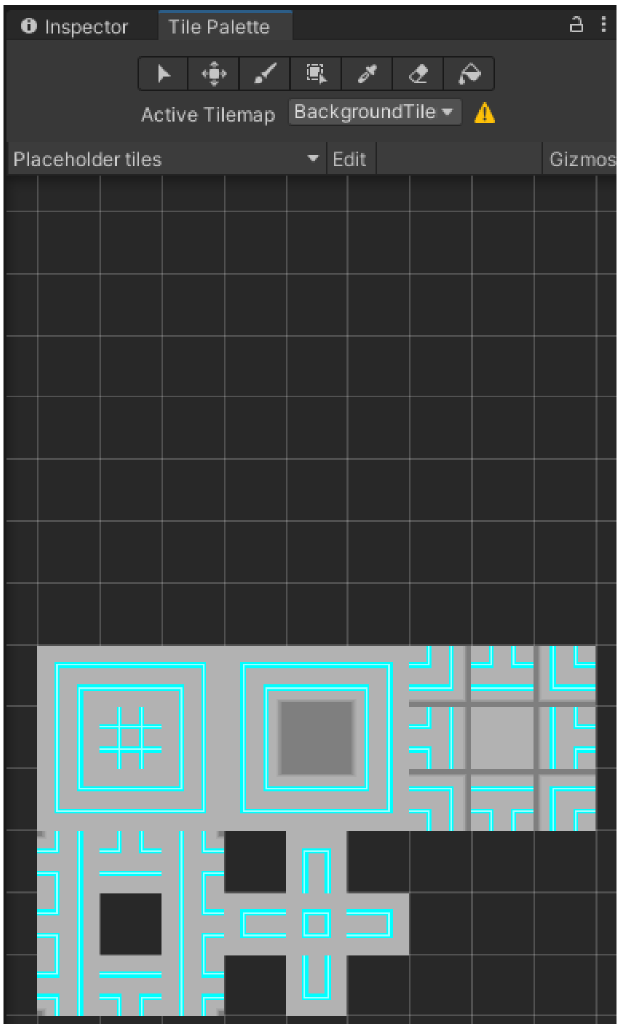


6.1.4. Tilemaps

Tilemaps are used to create the general geometry of the room. The ‘CollisionTilemap’ contains all the terrain that requires collision like walls. The ‘BackgroundTilemap’ is used to create the floor of the room.



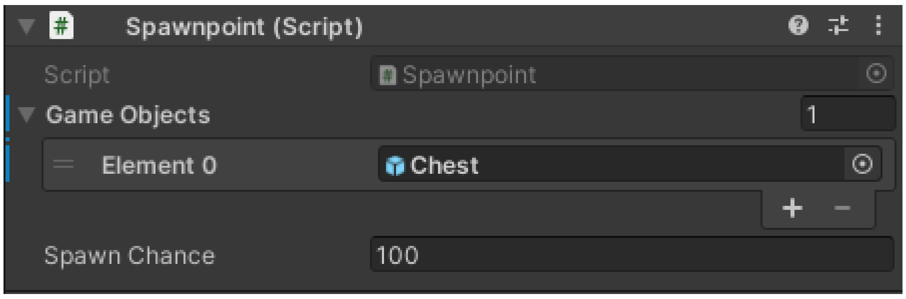
Tilemaps were chosen because they are easy to use when designing a new room. It's as easy as selecting a tilemap to draw on, then selecting a block of a given texture in the Tile Palette and drawing the room on the screen.



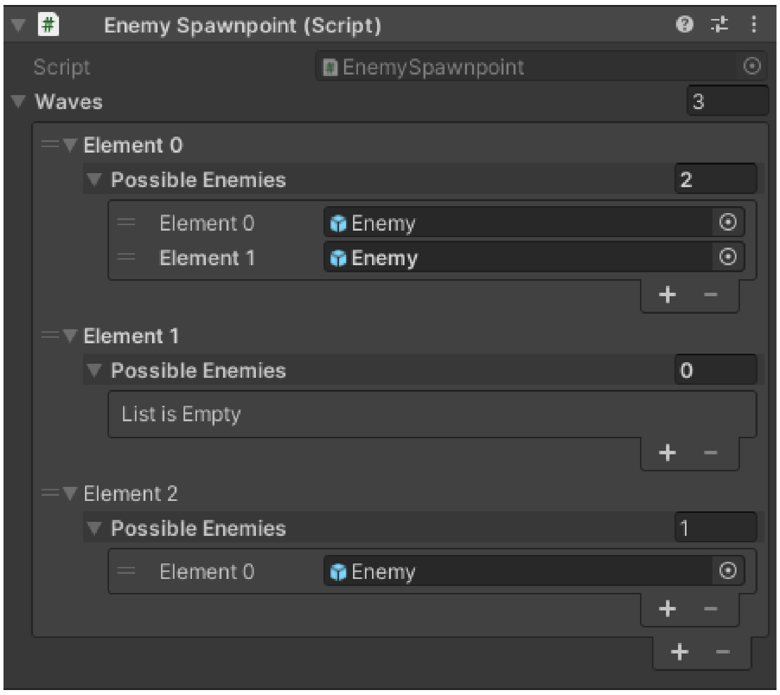
6.1.5. Spawnpoints

Spawnpoints are used to spawn entities and objects into a room. There are two types of spawnpoints: a ‘Spawnpoint’ and an ‘EnemySpawnpoint’. The prefabs that are added to the spawnpoints don’t need to be added to the object tree in advance.

Spawnpoint
The ‘Spawnpoint’ object is used to spawn objects and entities that are placed in the room while generating the level. This is separate from the predefined list of objects and entities to allow the room designer to add some random elements to the room. To achieve this, the spawnpoint can have multiple options to choose from. If the list of gameobjects contains more than one element, a random element is chosen. The spawnpoint also has a spawn chance variable to add some randomization to the spawned objects and entities. A good use for this spawn system would be chests, traps and obstacles.


Spawnpoint (Script)
Script: Spawnpoint
Game Objects: 1
Element 0: Chest
Spawn Chance: 100

EnemySpawnpoint
The ‘EnemySpawnpoint’ is used to spawn enemies in the room upon entering. These spawnpoints have a list of waves and possible enemies. When spawning a wave the spawnpoint looks in the list of possible enemies within the wave to select a random one, as seen in the first wave in the example below *(Element 0)*. An EnemySpawnpoint spawns up to one enemy per wave. It is allowed to leave waves without enemies. As you can see below, the second wave of this spawnpoint doesn’t contain any enemies.


Enemy Spawnpoint (Script)
Script: EnemySpawnpoint
Waves: 3
Element 0
Possible Enemies: 2
Element 0: Enemy
Element 1: Enemy
Element 1
Possible Enemies: 0
List is Empty
Element 2
Possible Enemies: 1
Element 0: Enemy

6.2. Generation

The generation works by adding the rooms one by one. The algorithm chooses a random door from the room it wants to place and then chooses a random door from an already placed room to connect it to. It then checks if the room fits in that position. If it does, it gets placed and the doors of the room that weren't used get added to the unused doors list so that they can be used to add rooms to in the future. If it doesn't fit it will try all the doors of the correct orientation one by one. If none of them fit it will choose a different door from the room we want to place and try that.

After the algorithm has placed the specified amount of rooms, it will try to connect as many of the unused doors to each other as possible by placing corridors. It first tries to connect them straight but that only works if 2 doors align perfectly. Then it tries to add in corridors with a corner in them.

Finally the doors that are left are removed and replaced with a wall. This is done on the tilemap of the room itself.

6.2.1 Corridors

Corridors are placed on 2 tilemaps that are attached to the FloorGeneratorObject. The corridors aren't based on prefabs like the rooms but are generated by the FloorGenerator.

6.2.2. Collision check

To check if a room will fit at a position we loop through all the positions of the tiles in the room and check if that tile is already occupied. For the corridors we use the GetTile function on the corridor tilemap. For rooms we spawn an overlapbox at the location and see if it hits anything.
To check if a corridor will fit we only check the middle part of the corridor because it makes the code faster and more readable. This does mean that the walls of the corridor might overlap with the walls of a room or another corridor. With our current tileset this causes a visual glitch.

6.2.3. How to use

To use the FloorGenerator you drag a FloorGenerator prefab into the scene. You add your favorite rooms to the roomlists and specify how many of each list should be spawned. Then you want to change the tileset for the corridors to match the tileset of the rooms you used. You can also specify the minimum and maximum length of the corridors.

6.3. Transitions

Level transitions are initiated when the player triggers a transition object at the end of a level, or when certain menu options are selected. These menu options are the start option in the main menu and the quit option in the pause menu.

When the level transition is executed, the player is usually moved to the next scene in the build order (File → Build Settings to view and edit). The only exception to this is when the player quits to the main menu, then scene 0 is loaded. All level transitions are handled by the SceneLoader.

6.3.1. SceneLoader

The SceneLoader component is used to transition to another scene. This component also activates and deactivates the loading screen. The loading screen lasts for as long as the scene is loading its Awake and Start functions. During this time the Sceneloader will disable player movement and reset the player's position to the default coordinates.

6.3.2. Persistence

The following components are persistent between scenes for their own reasons, which are explained below:

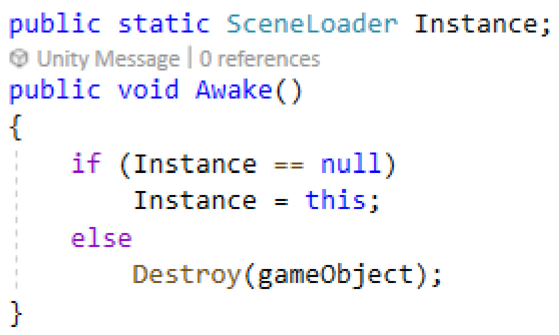
Player
Player is persistent because the contents of Player (inventory, health, etc.) can change during the level. These changes obviously shouldn't be lost between levels. Player is not persistent between runs and thus destroyed when quitting to the main menu.

SceneLoader
The SceneLoader needs to be persistent between scenes because it still needs to function once the scene has been loaded. The loading process unloads previous scenes by default, so the SceneLoader would be unloaded during the process, and be unable to run code in the new scene. This needs to be possible because the Player's movement is disabled at the start of loading and needs to be re-enabled when loading is done.

LoadingScreen
The LoadingScreen component is activated and deactivated by SceneLoader. If this component were not persistent between levels it could not be turned off at the right time during a level switch.

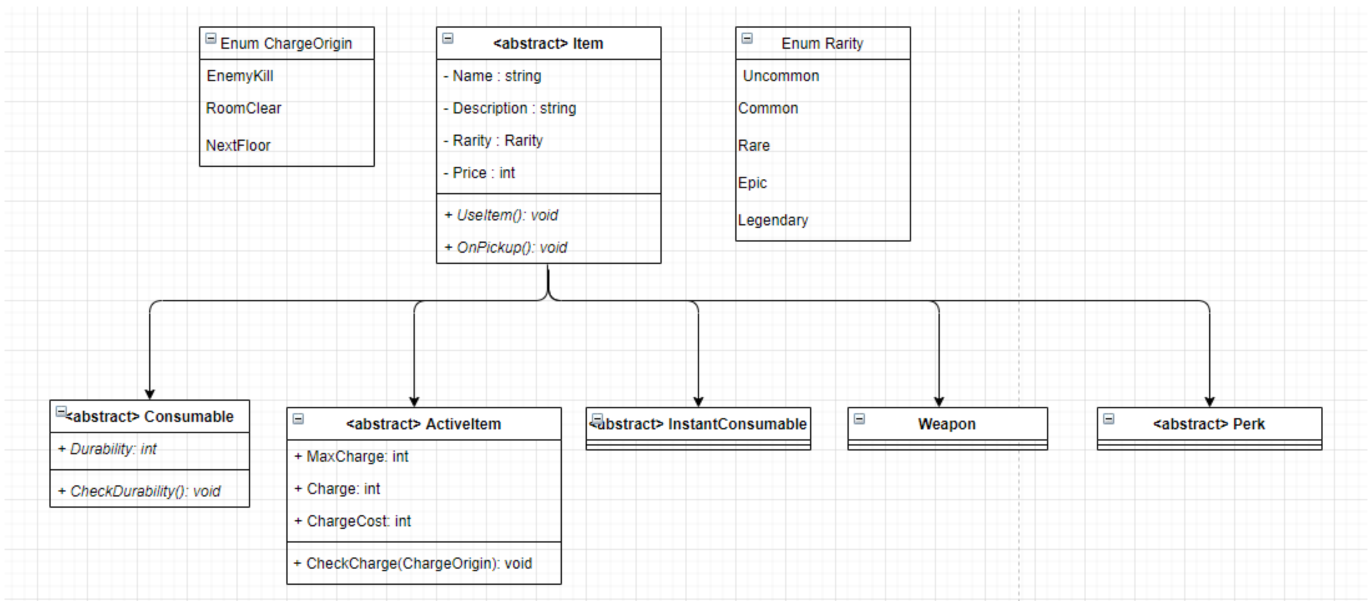
Camera
The camera is persistent because it needs to be connected to the player using a serialized field. It would be very hard to connect it at runtime, so the best solution we found was to take the camera along with the player.

If these persistent objects are only created in the first scene that needs it, developers would always have to start at the main menu and move through scenes until the desired one was reached. To avoid this, the objects can be added to every scene. However, doing this would cause duplication, as they would be carried over from the previous scene ***<u>and</u>*** created in the new scene. To avoid this constant duplication of objects, the singleton pattern is used. Our implementation creates a public static instance of the class during the *Awake()* method unless it already exists. In that case, it destroys the new object, leaving only the already existing one.



```
public static SceneLoader Instance;
Unity Message | 0 references
public void Awake()
{
    if (Instance == null)
        Instance = this;
    else
        Destroy(gameObject);
}
```

7. Items



An item is an object which can be picked up and/or utilized by an entity that contains an inventory. The item framework allows for a variety of item variations through the use of abstract classes. Each class contains a set of core variables related to all items.

name	The name of the item, used in hud overlay and on pickup message
description	The description of the item, used in hud overlay
rarity	The rarity of the item for shops and chest drops
price	An integer used for the cost to purchase this item from a shop

7.1. Consumable

A consumable is an abstract version of Item, which contains an additional variable for handling durability. Whenever the event for using the item is fired off, the durability is lowered and the item is removed from the inventory if it reaches 0.

7.2. ActiveItem

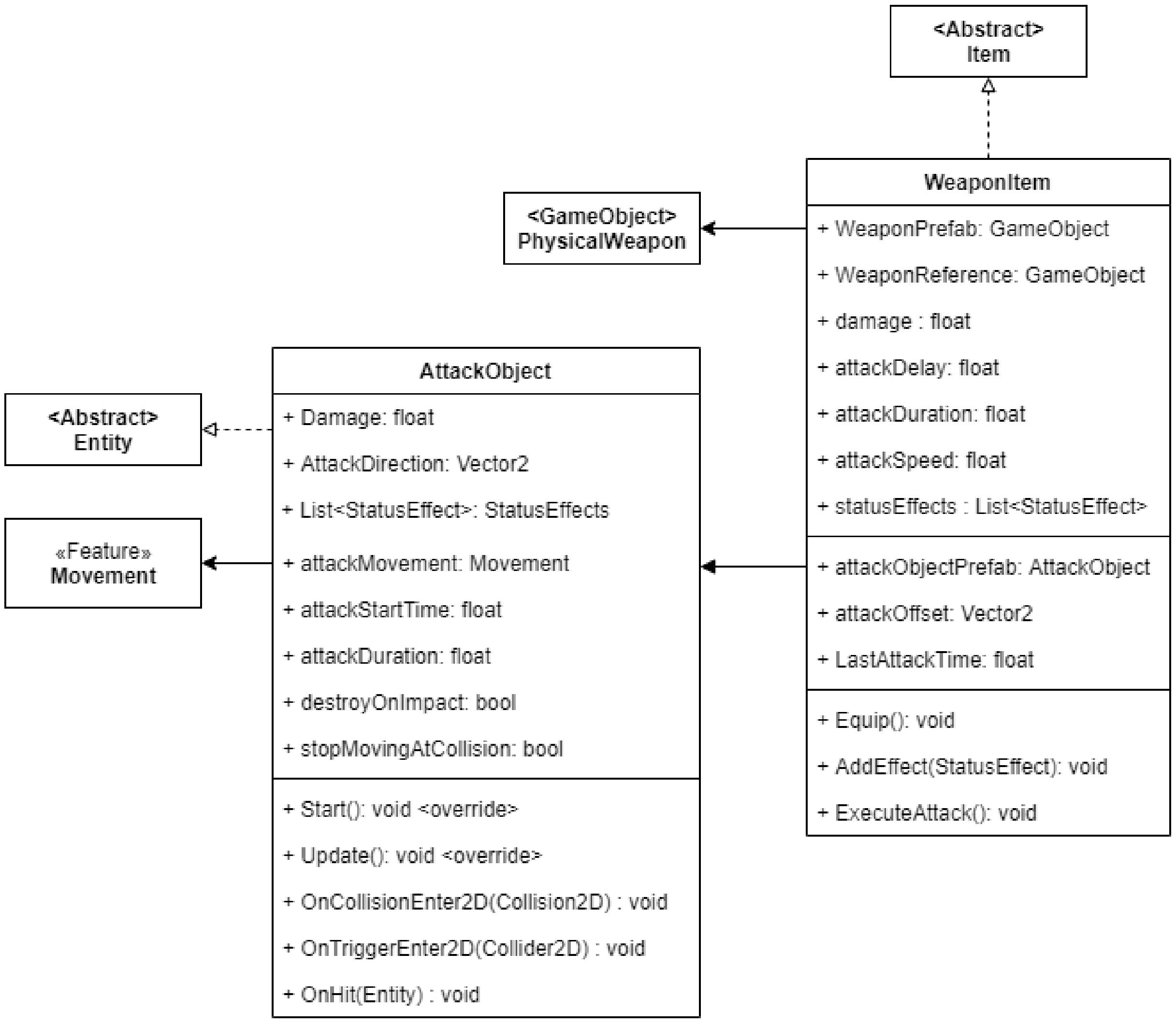
An active item is an abstract version of Item, which contains a series of additional variables for handling the use of activatable items.

charge	The amount of times an active item can be used/filled
chargeCost	The amount of charges required in order to active the item

7.3. InstantConsumable

Instant consumables are an abstract version of Item, with slight alterations. Instead of being stored within the inventory, instant consumables have their “*use*” effect run instantly after pickup and are not stored within the inventory.

7.4. Weapon



<Abstract>
Item
WeaponItem
+ WeaponPrefab: GameObject
+ WeaponReference: GameObject
+ damage : float
+ attackDelay: float
+ attackDuration: float
+ attackSpeed: float
+ statusEffects : List<StatusEffect>
+ attackObjectPrefab: AttackObject
+ attackOffset: Vector2
+ LastAttackTime: float
+ Equip(): void
+ AddEffect(StatusEffect): void
+ ExecuteAttack(): void
<GameObject>
PhysicalWeapon
AttackObject
+ Damage: float
+ AttackDirection: Vector2
+ List<StatusEffect>: StatusEffects
+ attackMovement: Movement
+ attackStartTime: float
+ attackDuration: float
+ destroyOnImpact: bool
+ stopMovingAtCollision: bool
+ Start(): void <override>
+ Update(): void <override>
+ OnCollisionEnter2D(Collision2D) : void
+ OnTriggerEnter2D(Collider2D) : void
+ OnHit(Entity) : void
<Abstract>
Entity
«Feature»
Movement

Weapons exist in two forms in the game. First, as an item in the inventory of a player, npc or Chest. This item can also show up as an item on the floor of the level that can be picked up. The second form is an equipped object by an entity within the world that can use it’s AttackObject to interact with other objects within the game world.

Let’s first look at creating a new weapon, and then dive into the systems.

7.4.1. Creating a weapon

To create a weapon from scratch:

- Create a weapon prefab in Prefabs/Weapons. This contains all stats of the weapon through the use of the WeaponItem component (or a class that inherits WeaponItem).
  * The sprite of this prefab is the sprite that the weapon has when lying on the floor.

- Add an existing or newly created (see below) PhysicalWeapon prefab to the weapon in the “Weapon prefab” field.
  * The sprite of this prefab is the sprite that the weapon has when it’s an Entity’s active weapon.

- Add an existing or newly created (see below) AttackObject prefab and the desired AttackObject sprite to the weapon prefab through the relevant fields in the "Weapon Attack Properties" Section of the WeaponItem component.
  * You can also change the Attack Offset Distance, this determines how far away from the weapon the Attack should start.

7.4.2. Creating a PhysicalWeapon

To create a new PhysicalWeapon:

- Create a new prefab in Prefabs/PhysicalWeapons. Give it a sprite that you want to show in-game when the weapon is held by an Entity.

7.4.3. Creating an AttackObject

To create a new AttackObject:

- Create an AttackObject prefab in Prefabs/PhysicalWeapons/AttackObjects. This contains movement details of the attack, except its Force, which is determined by the Weapon that spawns the AttackObject. This prefab also sets whether the attack should stop moving on collision or return itself to the object pool on impact.
  * Make sure to add the movement feature script to the MovementFeature field in the entity (AttackObject or a child) as seen in the picture in section 5.4.5.
  * The sprite of an AttackObject is set by the Weapon, but its colour filter and hitbox can't be (at the moment).

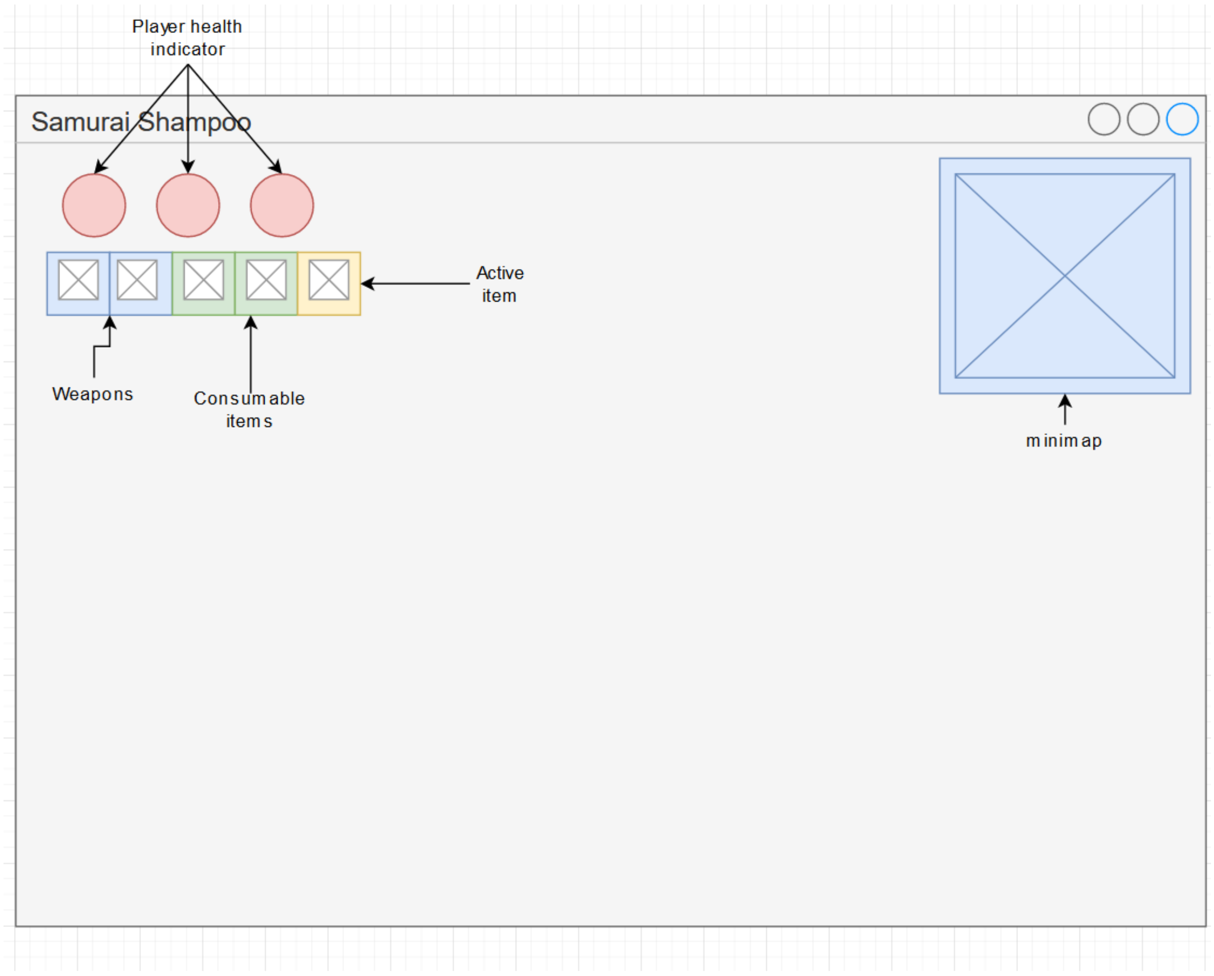
7.4.4. WeaponItem

All the weapon‘s main characteristics are stored in an instance of the WeaponItem class. It contains a blueprint of the physical game object in the form of the WeaponPrefab. When the weapon is equipped by an NPC or the player, an instance is created of the WeaponPrefab and stored in the WeaponReference variable. This reference will be used to destroy the physical weapon object when the item is removed from the inventory. WeaponItems also have a set of variables that determine the weapon‘s behaviour and characteristics. (Damage, status effects, attack speed, etc.)

7.4.5. Windup Animations

Some weapons will have a windup animation before the AttackObject appears in the game world. Initially this mainly applies to enemies, as the player needs to be able to anticipate an incoming attack. For enemies the windup animation uses their state machine to switch to a Windup-state before switching to the attack state. Note that the duration of the Windup-state needs to be larger or equal to the duration of the animation. Otherwise the windup animation will still be playing while the AttackObject is already visible.

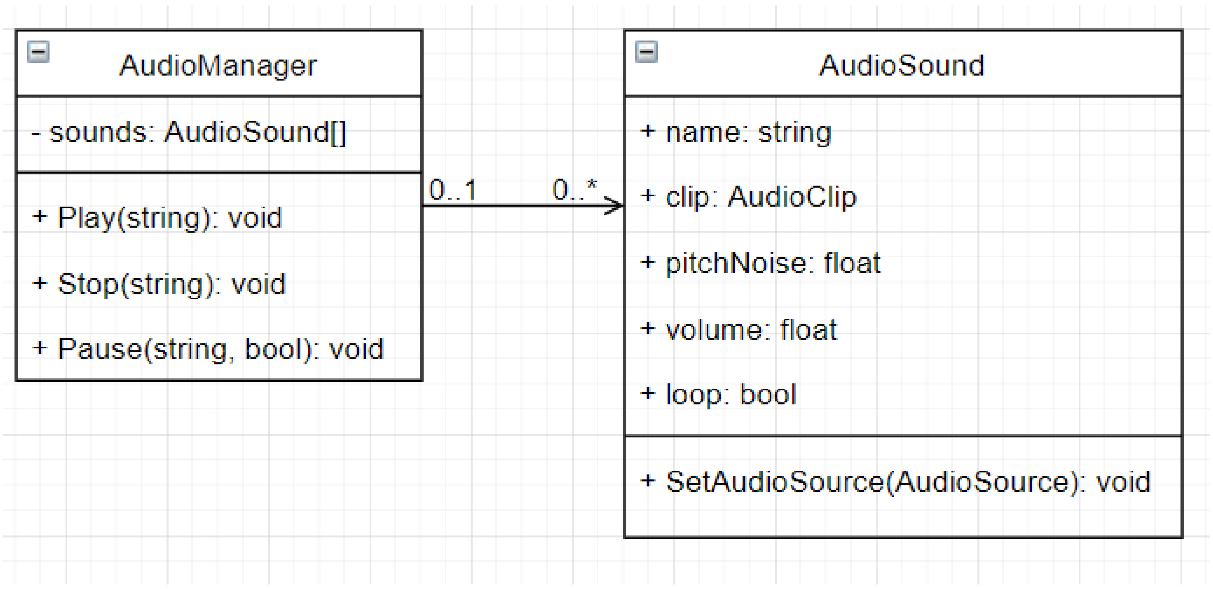
8. UI



When the level is initialized, the player object calls the health and inventory UI elements and fills it with the items and health the player has. The amount of health indicators increase or decrease dynamically based on the amount of maxhealth and health the player has.
Every time the player gets damaged or gets healed, it calls the health UI and changes the amount of health indicators it should show.
Every time the player swaps weapon, or picks up a weapon or item, the player inventory feature calls the UI and asks it to update it to represent the current player inventory.

9. Aesthetics

9.1. Audio

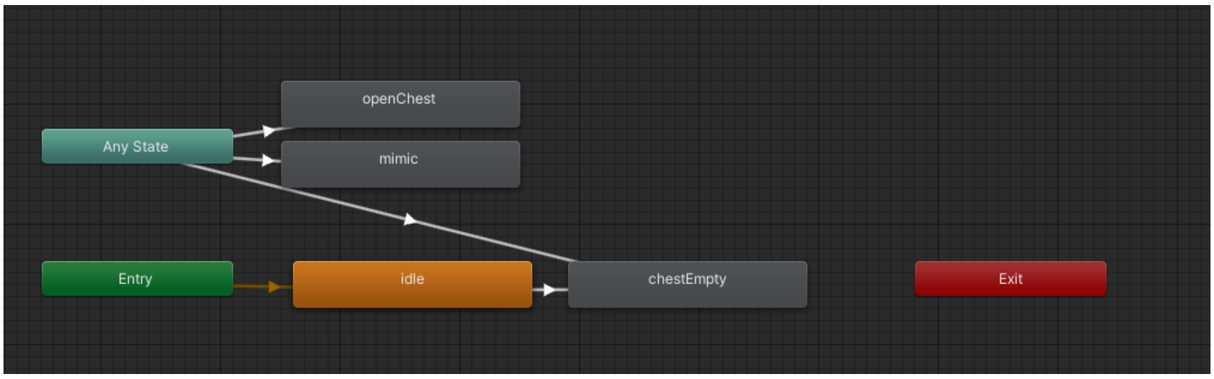


The audio of this game is handled by a single class. The audio manager allows for adding an indefinite amount of sounds to the game which can be played through any other class. Through the inspector, a person can selectively add a new audio clip (AudioSound) and give it specific information. This allows for fast and customizable implementation of sounds into our game. Any class can play, stop or pause any sound as long as they reference an already implemented sound.

```
AudioManager.Instance.Play("TestSound");
```

With this easy and flexible system, we can reduce redundancy and omit the need to add individual audio listener components, sound clips and settings to each object that should play a sound.

9.2. Animations



Animations are handled through the default unity animator.