Bachelor's thesis

Degree programme: Information and Communications Technology

2020

Duy Vu Dinh Pham

GAME ARCHITECTURES IN UNITY PROJECTS

TURKU AMK
TURKU UNIVERSITY OF APPLIED SCIENCES

BACHELOR'S THESIS | ABSTRACT

TURKU UNIVERSITY OF APPLIED SCIENCES

Degree programme: Information and Communications Technology

November 2020 | 45

Duy Vu Dinh Pham

GAME ARCHITECTURES IN UNITY PROJECTS

Game architecture is an important aspect of game development but rarely mentioned. A game architecture may be a factor for the success or failure of a game project. Video games are complex pieces of software and game architectures are guidelines for programmers to understand that complexity. This thesis is an attempt to give an overview of game architectures for Unity projects and provide a general view of architecture implementation in the Virpa2 project as a practical example. Virpa2 is an educational game that teaches children about fire safety. The project utilizes game technologies to enhance the learning experience and deliver the best possible result.

The thesis presents how a game architecture can be built in a game application by specifically utilizing ScriptableObject, a built-in data type in Unity engine. The Virpa2 project is a case study to experiment and test the core idea of architecture design. In the end, the result of the game architecture implementation in the Virpa2 project is considered successful since it fulfilled all the technical requirements of its game design and provided a coherent guideline for developers to follow and contribute to the game.

KEYWORDS:

Game, software architecture, C#, Unity, design pattern, event-driven programming, data-driven programming

CONTENTS

FIGURES

CODE SNIPPETS

LIST OF ABBREVIATIONS

API	Application Programming Interface
AR	Augmented Reality
ECS	Entity Component System
GUID	Globally Unique Identifier
NPC	Non-Playable Character
OOP	Object Oriented Programming
RAM	Random Access Memory
RPG	Role Playing Game
UI	User Interface

1 INTRODUCTION

Video games are composed of several components: graphics, audio, animation, environment and scripting. Those components are computer programs provided by game engines and exposed as APIs (Application Programming Interface). Game developers mostly work with those APIs and combine their functionalities to create video games. By combining them, we inherently define their relationship and form some kind of structure or system. Those programs, the relationships between them and the human discipline to follow as well as maintain the design are defined as software architecture (Bass, et al., 2013, p. 4). Since video games are more or less software, we have the term game architecture to express the intent of software architecture in the context of video game development.

Usually, the codebase of a project with bad architecture falls into one of these two categories:

- Spaghetti codebase: codes are written in unstructured and spontaneous manner (Pizka, 2004, p. 3). Parts of the game reference each other without any clear intention and gradually form into a woven thread of codes. This makes the codebase extremely hard to inspect and understand. Whenever a new feature built upon available functionalities is introduced to the codebase, developers have to unravel the related parts and make an assumption about their connection. The more ambiguous the assumption is, the higher the risk that leads to the situation where changes in one functionality may accidentally break others. In the best scenarios, those broken functionalities reveal themselves right after the changes were made and this is manageable to fix. In the worst cases, they lurk and wait until the new changes happen to touch them. This time it is hard to tell exactly which modification in the history broke them and, therefore, a considerable amount of time will be invested to trace down the suspect.
- Rigid codebase: unlike Spaghetti codebase, this one usually has a structure. The main problem is that it does not embrace nor anticipate changes. Yet changes are inevitable in software. Different architecture designs are considered for the project and the most suitable is chosen in the beginning phase. Then the implementation is conducted and tons of features are built upon it. A new idea shows up a few weeks later and requires implementation that the architecture is

not well prepared for. In order to implement that new idea, coders have to tweak the architecture to facilitate the new implementation and of course tons of features that depend on it as well. This makes adding new features to the game such a tedious task and if it is ever once taken carelessly the whole structural codebase potentially ends up being just like the spaghetti one.

Bad architectures are always painful to work with but that does not mean projects with good architecture design are always joyful and glorious. Good architecture requires a lot of effort to design as well as discipline to maintain its elegant, well-organized structure for the whole development cycle. (Nystrom, 2014, p. 9) The major reason that leads to bad architecture is the way functionalities rely on each other, one thing changes could affect anything that depends on it. Good architectures advocate modularity to solve this problem but nevertheless introduce another layer of abstraction and indirection that sometimes put taxing on performance and obscurity on the code logic (Nystrom, 2014, p. 14).

The purpose of this thesis is to provide information about how game architectures are applied in Unity projects and discuss about the tradeoffs of their implementations. Chapter 2 introduces basic blocks to build a game architecture in Unity and some popular methods which are widely adopted by the community. Chapter 3 introduces a designer-friendly and data-oriented method that revolves around ScriptableObject. Chapter 4 is a report of the Virpa2 project that is based upon ScriptableObject architecture.

2 BASIC BLOCKS OF A GAME ARCHITECTURE

This chapter introduces basic concepts to build a game architecture and opens a discussion about their pros and cons.

2.1 Initialization

Game applications often need some crucial functionalities to be set up before they can be used. In general programming practice, initialization codes are usually put in a constructor and then get invoked upon its creation. (McShaffry, 2012, pp. 130-133) However, when working in Unity environment, developers mostly utilize MonoBehaviour class-based components to control programmable behaviors of a GameObject. Unity takes over the control of MonoBehaviour**'**s creation and provides a set of callback functions as an alternative (Dickinson, 2017, p. 29). Although MonoBehaviour can be instantiated by calling **new MonoBehaviour()** however that method produces undefined behavior and is strongly against convention. Upon scene loading, Unity engine iterates through all GameObjects within the current active scene to collect all script components inherit from MonoBehaviour and then invoke the pre-defined callback functions available in that script. Unlike other game engines, Unity finds and store function pointers to those callbacks instead of applying inheritance overriding. This means the engine merely cares about the function's name and its presence in that script including empty functions (Dickinson, 2017, p. 45).

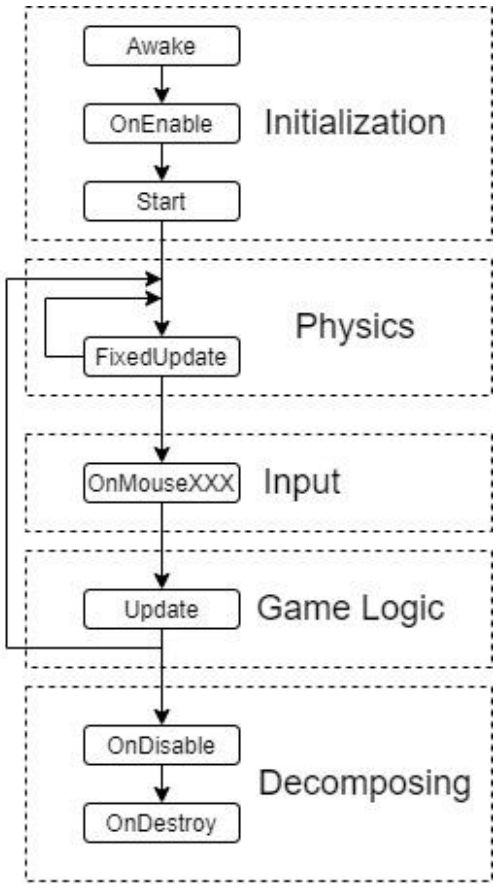

Figure 1. Simplified Order of Execution based on Unity documentation (Unity Technologies, n.d.).

By convention, codes in **Awake()** are responsible for initialization tasks and the ones depend on the initialized are put in any callback invoked after **Awake()**, usually in **Start()** as demonstrated in Figure 1. If the initialization codes complete

their execution within a single frame, they are safe to be used by others after **Awake()**. (Dickinson, 2017, p. 29) This does not only apply to the initialization flow in the same class but also across multiple classes since all the **Awake()**s in the same scene are called and completed before any **Start()**. Class A can reference class B and utilize B's functionalities in **Start()** without any concern about whether those functionalities are ready to use, as shown in Figure 2.

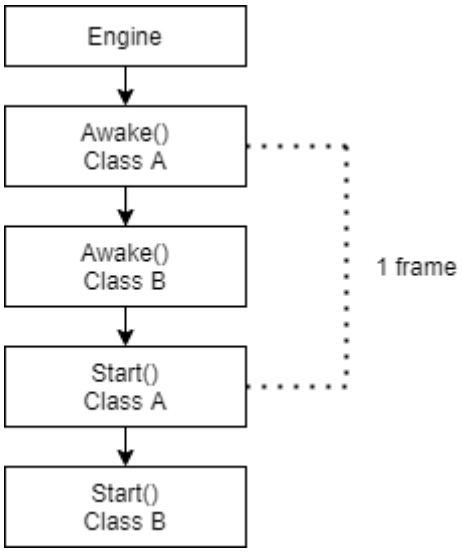

Figure 2. Single frame execution flow.

However, this setup fails its purpose in scenarios where initialization tasks are spread across multiple frames. To tackle this problem, initialization codes can apply an event-based solution where initialization processes trigger events upon their completion.

Even though the execution order of callbacks is deterministic, the MonoBehaviour**'**s is not (Dickinson, 2017, p. 29). This means that the callback functions of GameObject A may run before or after GameObject B's counterparts and that order changes randomly in every game run. Therefore, sequential initialization is not possible in this setup.

One more problem with this callback-based setup is that initialization codes tend to scatter across the scene. Unity triggers all the MonoBehaviour callbacks it can find in a scene so any GameObject could participate in the initialization invocation process. Since GameObjects can be anywhere in the scene, the initialization is decentralized and in turn, becomes difficult to trace in both Editor and codes. One solution is to create a MonoBehaviour script that handles all the initializations and assign its Script Execution Order to take place before any other MonoBehaviour. Then attach that script to any

GameObject which acts as a central initializer. However, with this setup, the initialization process becomes scene-dependent and hence requires a strategy for data persistency between scenes.

2.2 Data persistency

A game is usually comprised of multiple scenes. When a scene is unloaded, all of its GameObject instances are destroyed (Unity Technologies, n.d.). This makes the data belonged to a MonoBehaviour not persistent during scene transitions. This behavior ensures scenes are always loaded in a clean slate. In many cases, it is desired. However, it's quite common that initialized data and states need to carry on across multiple scenes and objects (Murray, 2014, p. 4). **Static** data appears to be useful in this scenario. However, it also has some major limitations:

- They are not serializable (Unity Technologies, n.d.).
- They do not show up in Inspector.
- They can not have constructors. (Skeet, 2013)
- They advocate code-driven practice since all changes must be made by codes.
- Their execution order is non-deterministic (especially the moment when the scene is first loaded).
- They do not support inheritance and polymorphism (Skeet, 2013).

One attempt is to use **[RuntimeInitializeOnLoadMethod]** attribute. The initialization process is still centralized since all initialization codes can be put in one script. Most importantly, this attribute does not require a class to inherit from MonoBehaviour. The data's life cycle, therefore, becomes scene-independent. Unity automatically calls all functions marked with this attribute anywhere in the project.

Code Snippet 1. Initilization using [RuntimeInitializeOnLoadMethod] attribute.

```
public class Initializer
{
    [RuntimeInitializeOnLoadMethod]
    private static void Initialize()
    {
        // ... codes perform initialization
    }
}
```

However, the functions that execute initialization codes have to be **static**. Therefore, if those functions happen to perform any operation on data (which is most likely), that data also has to be **static** (Skeet, 2013)**.** Therefore, this method inherits the limitations of **static** as mentioned above.

By solving some of **static** classes's problems and yet maintaining the singular global accessibility and persistency between scene transitions, Singleton pattern come into favor of Unity developers (Murray, 2014, p. 5).

2.3 The Singleton

Singleton is one of the most controversial patterns in software development. It is even considered anti-pattern due to its false sense of convenience.

According to Gamma et al. (1994), the intent of Singleton is:

> *"Ensure a class has one instance, and provide a global point of access to it."*

Singleton internally uses **static** data and wraps it under an instance. Therefore, Singleton achieves data persistency but still retains some attributes of a regular class instance (Murray, 2014, p. 5). Singleton in Unity comes with many variants but they usually share the same structure as demonstrated in Listing 2.

Code Snippet 2. Singleton implementation in Unity.

```
public class Singleton : MonoBehaviour
{
    private static Singleton instance;
    public  static Singleton Instance { get => instance; }

    public int SomeData;

    private void Awake()
    {
        if (instance != null && instance != this)
        {
            Destroy(this.gameObject);
        }
        else
        {
            instance = this;
            DontDestroyOnLoad(this.gameObject);
        }
    }
}
```

Firstly, any class that needs to implement the Singleton pattern must inherit MonoBehaviour so it is scene-dependent. Secondly, it destroys any duplicate instance if detected. This ensures the initialized instance will not be overwritten when a new scene that has the same Singleton class is loaded. Finally, if data need to persist across multiple scenes, **DontDestroyOnLoad()** function is required to push that instance to a Unity's special scene called DontDestroyOnLoad. This scene only shows up in play mode and stay active for the whole game session to display GameObjects marked with DontDestroyOnLoad (Unity Technologies, n.d.). In Build mode, the scene does not exist but the persistency of those objects still is retained (Unity Technologies, n.d.). With this structure, Singleton provides some advantages:

- It is easy to setup. Classes that want to be a Singleton could simply follow the code structure above or inherit a generic template for reusability.
- It is easy to use. Any class can access data from Singleton globally by calling **Singleton.Instance**.
- The initialization process is controllable as opposed to pure **static**. In addition, lazy initialization is also possible.
- Data can persist between scene loads.
- Instance's data is open for serialization since they are not required to be **static**.
- Singleton supports interfaces and inheritance since the class itself is not **static**.

However, Singleton has major problems:

- It restricts the class to only have one single instance. Which greatly reduces flexibility and testability.
- It is troublesome to debug. Anywhere in the codebase can reference the global instance. Whenever a bug related to the Singleton appears, developers have to go through the whole codebase to investigate which parts touch that instance and potentially cause the bug. (Nystrom, 2014, p. 74)
- It makes codes harder to understand and follow by inherently hiding dependencies. **Singleton.Instance** can be conveniently called anywhere in the class definition without field declaration. In order to find out which Singleton the class depends on, developers have to manually go through every single line of code in the class. (Nystrom, 2014, p. 74)
- It is not designer-friendly. If Singleton is applied to keep data persistent across multiple scenes, that Singleton itself has to destroy any duplicate in other scenes. Therefore, all the serialized references of those scenes' instance will be lost. This makes Singleton unusable in Editor.

2.4 Preload scene

Preload scene is a widely adopted method to solve initialization and data persistency among Unity community. The idea is that the game application loads a scene which only contains initialization codes (hence the name preload) and passes the initialized data or services to other scenes by using Singleton. After all the initializations have been completed, the next game scene is loaded and all initialized parts are ready for the rest of the game via a Singleton as depicted in Figure 3. The preload scene is more extensible as compared to using scripts for initialization. For example, the UI (User Interface) can be utilized to perform a loading screen or visual effect to indicate the progress of initialization.

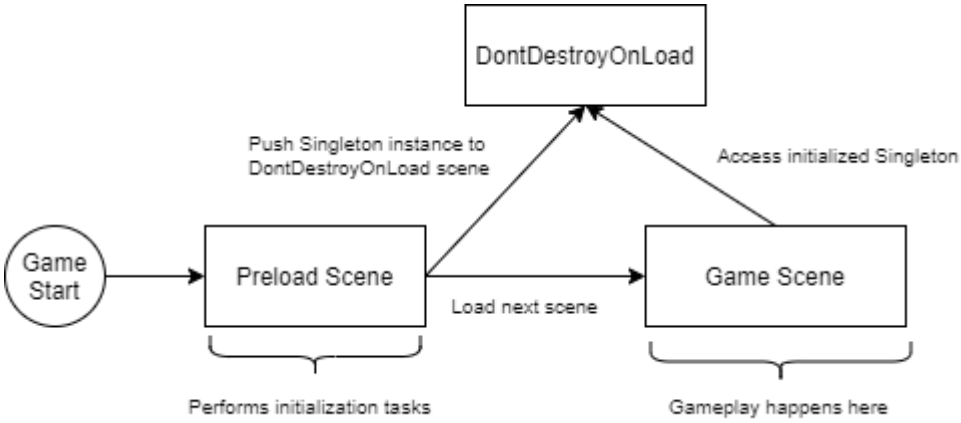

Figure 3. Preload scene execution flow.

However, preload scene has a huge drawback to the development phase. The preload scene must be the first one loaded in Hierarchy during play mode in Editor. In order to test a specific scene, developers have to manually find and open the preload scene and then change the code handles initialization to load that scene after all initialization tasks are completed. Instead of hitting the “Play” button to test that scene immediately, developers have to manually jump between scenes to make a test.

One solution to automate this process is to make a script that automatically loads the preload scene and attach it to every game scene of the project. The game scene has to run before the preload scene in order for those methods to be executed. This throws away one of the most important purposes for using the preload scene: all the data and services are completely initialized in one scene before jumping to another. Additionally, using this method may lead to different behaviors between Build mode and Play mode. In Build mode, the preload scene has index 0 in the Build Setting and therefore is inherently loaded before any other scenes (Unity Technologies, n.d.). The deterministic initialization process works as intended. In Play mode, boilerplate codes and workaround are needed to simulate this behavior if automation is wanted. The workaround for this is to use pre-processors to change behavior based on the working environment but this opens a possibility for future pitfalls since Unity’s Play mode has a completely different flow from the Build mode.

Another option is to utilize the **[RuntimeInitializeOnLoadMethod]** that gets executed before scene load. With this method, the preload scene is automatically loaded before any game scene. However, it does not know which one to load next. In the first solution, the game scene is loaded first and then the preload scene. The preload scene does not

need to know which scene to load next. In this case, the preload scene must contain information about the next scene. Hence the developer has to change that information manually whenever there is a scene that needs testing in Editor which is exactly the original problem. Automation is now rendered useless.

Code Snippet 3. Script to load Preload scene in Play mode.

```
public class TestScene
{
#if UNITY_EDITOR
    [RuntimeInitializeOnLoadMethod(RuntimeInitializeLoadType.BeforeSceneLoad)]
    static void LoadPreloadScene()
    {
        SceneManager.LoadScene("Preload Scene");
    }
#endif
}
```

Note that both attempts above are for Editor play mode only to avoid tedious workflow and they all fail. However, the preload scene setup still works properly in Build mode since Unity always loads scene with index 0 in Build Setting.

Overall using preload scene may benefit in some cases but it is still troublesome nevertheless. A scene should function on its own without any dependency on other scenes. This encourages decoupling and reduces the complexity of the game by breaking the game into modular testable scenes.

2.5 Managers

Game functionalities are usually separated into dedicated sub-systems called managers (Murray, 2014, p. 11). This makes codebase easier to reason about since developers can always expect what a manager does, based on its specialization. For example, any functionalities related to audio should go to AudioManager, how scoring works should only be ScoreManager's responsibility, UIManager will take care of UI elements for the game as shown in Figure 4. These managers are belonged to the whole game itself rather than to any specific scene so it is quite common to make them Singleton for global accessibility. Therefore, they inherit all the pros and cons of a Singleton. An alternative to minimize the need for Singleton will be discussed in a later chapter.

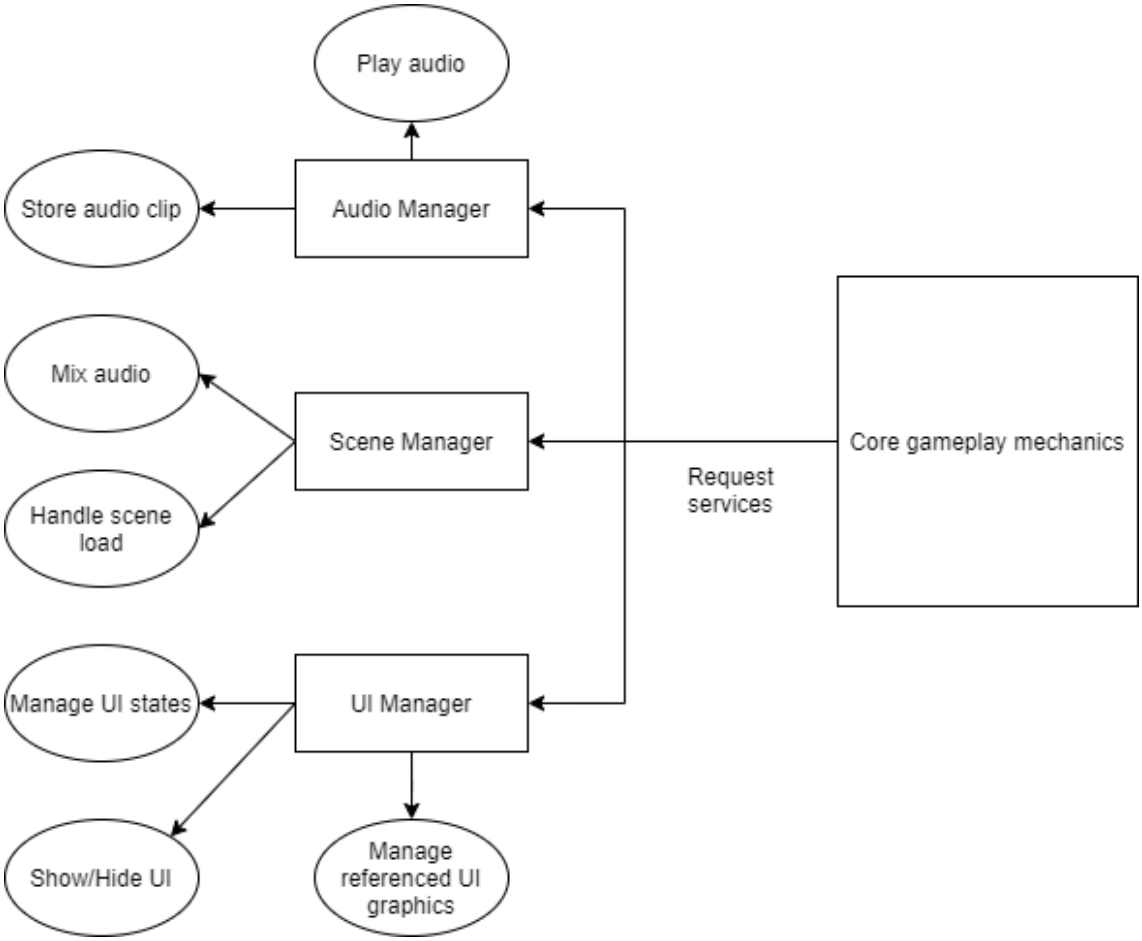

Figure 4. Managers in a game.

3 SCRIPTABLEOBJECT ARCHITECTURE

ScriptableObject Architecture is originally introduced by Ryan Hipple in Unite 2017 talk (Unity Technologies, 2017). His method of solving dependencies, events and shared data revolves around utilizing ScriptableObject. By separating data and logic, the system becomes debuggable, testable and changeable. This chapter introduces ScriptableObject and basic tools to build the foundation of a ScriptableObject Architecture.

3.1 What is ScriptableObject?

ScriptableObject is a UnityEngine built-in class. They have all characteristics of regular C# classes such as instantiating objects, containing data and methods (Unity Technologies, n.d.). Unlike regular classes, ScriptableObject's instances have different representations based on how they are instantiated:

- If instances are created at run time by calling **ScriptableObject.CreateInstance** or **GameObject.Instantiate()**, they are stored in RAM (Random Access Memory) just as regular instances. Calling **new ScriptableObject()** is technically possible but produces undefined behaviors.
- If instances are created in Editor by using **[CreateAssetMenu]** attribute, they are serialized and stored as Unity asset files (Unity Technologies, n.d.). At run time, those asset gets deserialized into a regular object and stored in memory.

Furthermore, ScriptableObject internally has the same C++ implementation as MonoBehaviour so they share most of the functionalities including the ability of being shown in Inspector (Unity Technologies, 2016). However, there are some distinctions:

- ScriptableObject can not be attached to a GameObject which means its life cycle does not belong to any scene (Unity Technologies, n.d.).
- ScriptableObject's callbacks such as **Start()**, **Update(), OnCollisionXXX,** etc **...** are not invoked as MonoBehaviour's counterparts.
- ScriptableObject can not use coroutines.

Due to the lack of proper documentation, ScriptableObject becomes confusing and less accessible to the community. Here are a few notes about ScriptableObject to help clarify some of its undocumented behaviors:

- Even though **Awake()**, **OnEnable(), OnValidate()** are invoked as opposed to other callbacks, they do not work reliably. In fact, most of the ScriptableObject's callbacks are recommended to not be used at all.
- Changes in ScriptableObject asset persist between scene loads since ScriptableObject's life cycle is not bound to any GameObject. Additionally, changes even persist after exiting Play mode in Editor. (Unity Technologies, 2017) This is useful for keeping tweaked settings during Play mode. To make data not persistent every run, the **[NonSerializable]** attribute could be used for fields that should start in a clean slate. However, assets literally do not exist in the context of Build mode so data persistency after game sessions does not apply in the build version (Unity Technologies, n.d.).
- ScriptableObject assets are not loaded if they are not referenced in scenes which behaves exactly like other Unity assets. (Unity Technologies, n.d.)
- If a ScriptableObject asset holds any reference to an object from a scene. That reference will be shown as TypeMismatch in Inspector even though the reference is still valid.
- ScriptableObject instances instantiated during runtime are not automatically saved as asset files.

3.2 Shared and non-shared states

In OOP (Object Oriented Programming), a class is usually composed of data and methods. This paradigm tends to be used to model self-contained objects. The data describe what the object has and the methods describe what it does, together they define what an object is. However, most of the time, classes need to reference others to access external data and perform an action based on it. For example, a class called **PlayerInput** which is responsible for handling input from the keyboard as shown in Listing 4.

Code Snippet 4. Sample script to handle input from players.

```
public class PlayerInput : MonoBehaviour
{
    private Vector2 movementAxis;
    public  Vector2 MovementAxis { get => this.movementAxis; }

    private void Update()
    {
        movementAxis.x = Input.GetAxis("Horizontal");
        movementAxis.y = Input.GetAxis("Vertical");
    }
}
```

And **PlayerController** class reads the **MovementAxis** data from **PlayerInput** and moves the player accordingly, as shown in Listing 5.

Code Snippet 5. Script handles player's action.

```
public class PlayerController : MonoBehaviour
{
    [SerializeField]
    private PlayerInput input;

    private void Update()
    {
        Move(input.MovementAxis);
    }

    public void Move(Vector2 direction)
    {
        // codes control player's movement based on directional input
    }
}
```

PlayerController merely reads the input value from **PlayerInput** and does not intervene in how the input is processed. This creates a clear separation between handling input and player's mechanics. However, games nowadays require more than one type of input devices, such as Xbox, Playstation and Wii controllers. To support those devices, the **PlayerController** gets 3 more dependencies and it needs to change the logic to detect and adapt each type of input device as shown in Listing 6. Suddenly handling input now becomes **PlayerController**'s responsibility too.

Code Snippet 6. PlayerController handles new input devices.

```
public class PlayerController : MonoBehaviour
{
    [SerializeField]
    private PlayerInput input;
    [SerializeField]
    private XboxInput xboxInput;
    [SerializeField]
    private PSInput psInput;


    private void Update()
    {
        // check which device is active and handle movement
    }
}
```

A better solution is to transfer that responsibility to **PlayerInput** and the **PlayerController** remains unchanged and unnoticed about the new devices. All it cares about is one single value: **MovementAxis**. Fortunately, the Unity input system abstracts the low-level devices. Input data from many types of physical devices can be retrieved via **Input.GetAxis()**. However, those input data are read-only which means they are not extensible for mobile devices.

Mobile games do not have physical devices for handling input. Their input data are retrieved via the player's physical interaction with the virtual buttons on the screen. (Madhav, 2013, p. 105) If the logic for handling those virtual buttons is put into **PlayerInput** then the class itself is responsible for handling both inputs from physical devices and virtual ones. The problem that comes with this approach is that the virtual devices come with UI components and other settings specific to them. This increases the complexity of **PlayerInput** because the class is forced to combine the logic for physical and virtual devices not to mention handling the UI for the virtual ones. However, the virtual input handling cannot be put into a separate class either since **PlayerController** has to reference it in order to use it and every new type of virtual input will become a new component of **PlayerController**.

But what if the **MovementAxis** variable is changeable outside of **PlayerInput**. The script for handling virtual device can reference **PlayerInput** and modify the **MovementAxis** data directly. A new way of handling input is added to the system without producing any major modification. **PlayerInput** just needs to make **movementAxis** field public and the **PlayerController** remains the same.

However, this breaks the data encapsulation principle of Object Oriented Programming. **PlayerInput** does not own input data anymore and any class can modify its data freely. This is one of the basic principle of data-oriented design. (DeLoura, 2000, p. 3)

3.3 Variables

In the example above, any script that needs access to **MovementAxis** has to reference the whole PlayerInput class although its only interest is **MovementAxis** data. Additionally, **MovementAxis** is defined within **PlayerInput** so this conceptually makes that data bound to **PlayerInput** and in turn, makes its lifetime bound to a GameObject. In Ryan's idea of ScriptableObject Architecture (Unity Technologies, 2017), those data should be disassembled into primitive types and contained by a ScriptableObject instance as shown is Listing 7.

Code Snippet 7. ScriptableObject wraps around a primitive float.

```
[CreateAssetMenu(fileName = "FloatVariable.asset")]
public class FloatVariable : ScriptableObject
{
	public float Value;
}
```

The input handling example can be extended a bit further with ScriptableObject Variable:

Code Snippet 8. ScriptableObject wraps around a primitive Vector2.

```
[CreateAssetMenu(fileName = "Vector2Variable.asset")]
public class Vector2Variable : ScriptableObject
{
       public Vector2 Value;
}
```

Code Snippet 9. Use Vector2Variable to decouple input handling's data and logic.

```
public class PhysicalInput : MonoBehaviour
{
    [SerializeField]
    private Vector2Variable movementAxis;

    private void Update()
    {
        movementAxis.Value.x = Input.GetAxis("Horizontal");
        movementAxis.Value.y = Input.GetAxis("Vertical");
    }
}
```

The **PlayerInput** does not own the **MovementAxis** anymore instead it uses the shared **Vector2Variable** as demonstrated in Listing 9. It also sensible to change the class name to **PhysicalInput** since all values from **Input.GetAxis()** are physical devices's inputs. By sharing **MovementAxis**, any new input handler can be added to the system without modifying others as shown in Listing 10.

Code Snippet 10. VirtualInput handler is added without changing PlayerController.

```
public class VirtualInput : MonoBehaviour
{
    [SerializeField]
    private Vector2Variable movementAxis;

    [SerializeField]
    private GameObject UIPanel;

    private void Update()
    {
        movementAxis = CalculateVirtualJoystickPosition();
    }
}
```

```
public class PlayerController : MonoBehaviour
{
    [SerializeField]
    private Vector2Variable movementAxis;

    private void Update()
    {
        Move(movementAxis.Value);
    }
}
```

The Vector2Variable is injected into those MonoBehaviour instances from Inspector. One more benefit of creating a separation between these input handlers is that they can be easily enabled or disabled on demand.

3.4 Runtime Sets

Besides UI, environment, sound and decorator objects, a scene also contains a number of gameplay elements called game entities such as the player's character, NPCs (Non-Playable Character), collectible items, interactable objects. Most of the time they need to be tracked or managed by a top-level system to compose meaningful gameplay. (Gregory, 2009, p. 689) For example, a level requires the player to slay all the monsters in the level before advancing to another. How does the system know if all the monsters have been slain? One solution to this is to constantly find and check every monster in the scene by calling **Object.Find()** every frame which is extremely costly for performance (Dickinson, 2017, p. 63). The list of monsters, however, can be cached upon the object's creation to provide faster lookup and iteration. Then this opens a question of which object should contain that list.

- A MonoBehaviour: this requires developers to manually drag the reference of the list to every monster in the scene because prefab workflow can not be applied in this case. Prefabs are not allowed to reference scene instances (Unity Technologies, n.d.).
- A Singleton: list of entities gets referenced by a hard-coded global instance. No manual work is needed which is a great help in scenarios where a game can have up to thousands of entities in one scene. However, this method suffers from all disadvantages of using Singleton as mentioned in the previous chapter.
- A ScriptableObject: solves all the problems that come with both methods above. ScriptableObject instance can be used in prefabs. It also exists before any scene

load so any MonoBehaviour can safely use that list. ScriptableObject instances are created in Editor hence different lists can be used during development. Changes can also be made via Inspector instead of code modification.

ScriptableObject **Runtime Set** is an extended version of **Variable**. Instead of wrapping a single primitive value, it contains a collection (Unity Technologies, 2017).

Code Snippet 11. Generic RuntimeSet implementation.

```
public class RuntimeSet<T> : ScriptableObject
{
    private HashSet<T> set;

    public void Add(T item) => set.Add(item);
    public void Remove(T item) => set.Remove(item);
    public bool Contain(T item) => set.Contains(item);
}
```

The set that stores those game entities can be any type of data structure. In this case, a **HashSet** indicates that the set only contains unique entities. Furthermore, the class itself is a template hence any new types of entities can be implemented easily. In fact, they can be left empty as demonstrated in Listing 12.

Code Snippet 12. New type of RuntimeSet can be easily implemented.

```
[CreateAssetMenu(fileName = "MonsterSet.asset")]
public class MonsterSet : RuntimeSet<Monster>
{
}
```

A **Monster** script that handles the logic for a monster can reference a RuntimeSet<Monster> asset and add itself to the set on **OnEnable()** as shown in Listing 13. This creates a safe and fast protocol to cache references of game entities scattered across a scene.

Code Snippet 13. Monster's subcription to the set.

```
public class Monster : MonoBehaviour
{
    [SerializeField]
    private MonsterSet set;

    private void OnEnable() => set.Add(this);
    private void OnDisable() => set.Remove(this);
}
```

In order to know if all monsters are gone in the level, a handler script can reference the set and count the number of active monsters. Additionally, the set is a pure data container hence it can serve all kinds of purposes. Other systems can reference the set and operate its own logic to create new gameplay. Although this is not thread-safe, the Unity main loop executes codes on a single thread so data is safe to be shared this way (Unity Technologies, n.d.). To utilize multi-threaded operations on data, the Unity has in-house ECS (Entity Component System) architecture for that purpose. However, ECS is out of the scope of this thesis.

3.5 Plugable event system

Occasionally, the game system needs to be reactive based on what is happening or changing in the game world. Polling the data changes is not quite effective since games are interactive and many things happen at the same time. Constantly monitoring every game element, especially in **Update()** callback, can dramatically slow down the performance (Dickinson, 2017, p. 52). An event system is a good solution for this. Just like shared data, events can be also be shared to indicate an event that happens in the entire game world which is called Game Event (McShaffry, 2012, p. 308). These events are free to be invoked or subscribed to by any class and hence effectively decouple the messaging system. They can be put into one single static class and become a central event system for the entire game (Penzentcev, 2015, pp. 37-39). However, this system can be improved with ScriptableObject. Instead of cramming all GameEvents into one script, each ScriptableObject instance will hold a GameEvent as shown in Listing 14.

Code Snippet 14. Generic GameEvent implementation.

```
public class GameEvent<T> : ScriptableObject
{
    private List<Action<T>> callbacks = new List<Action<T>>();

    public void Raise(T arg)
    {
        for (int i = callbacks.Count - 1; i >= 0; i--)
            callbacks[i](arg);
    }

    public void Subcribe(Action<T> action) => callbacks.Add(action);
    public void Unsubcribe(Action<T> action) => callbacks.Remove(action);
}
```

By using ScriptableObject, events have access to some advantages:

- They can be customized with **CustomEditor** since **ScriptableObject**s are serializable by default (Unity Technologies, 2017).
- They can store extra data as opposed to pure events. For example, tracing callbacks data.
- They are pluggable in Inspector without code modification (Unity Technologies, 2017).
- They enforce the dependency declaration and hence improve code tracing.

4 THE VIRPA2 PROJECT ARCHITECTURE

4.1 Introduction

Virpa2 is an educational game that teaches children about fire safety. Virpa2 was funded by Palosuojelurahasto. The game combines AR (Augmented Reality) and traditional elements of an RPG (Role Playing Game) mobile game to enhance and gamify the learning experience. Players will play as students in a school building where most of the rooms are locked. In order to explore the three-floor building, players have to unlock the rooms by scanning typical fire safety signs in the real world. Each room has a specific topic about one of the safety signs. By answering the dedicated questions to those rooms, players will gain score and precious rewards as well as climbing up the leader board.

In the first game design phase, the game had many unclear mechanics and the gameplay had to be developed along with the implementation. One of the most important task was to design and build an architecture able to account for regular changes as well as unexpected ones. Furthermore, the technical implementation required the game to have a structure enabling connections between game implementation and the server backend, data transfer and statistics, and the communication between RPG mode and AR mode.

The purpose of this chapter is to present implementations of various parts of Virpa2 which are based on ScriptableObject essential blocks in Chapter 3 and how it is applied to tackle technical requirements which will be introduced in the next section.

4.2 Technical Requirements

Firstly, the game was split up into several scenes. Those were; three scenes according to three floors of the building, a scene for using the phone camera to scan real-world objects, and a scene for the main menu of the game. The reasons for this separation are coherent workflow and game performance. By separating the game into scenes, the overall structure was improved and team members could also contribute to the game without blocking each other. The collaboration becomes more fluid by reducing the chance of blocking members when the whole game is divided into multiple scenes.

Therefore, the Virpa2 project applied multiple scenes and prefab workflow to create flexible collaboration. Another reason for using multiple scenes was about performance. The map of the game was large but mobile devices usually have limited resources as compared to other platforms, such as computers and consoles. If the game would not had been chunked into multiple scenes, the memory space in a mobile device to accommodate could have been quickly used up and the frame rate would had drop. By splitting the game into multiple scenes, the architecture had to take data persistency and scene management into account.

Secondly, the game required scanning real fire safety signs via the phone camera. Unity has a built-in plugin for image recognition which is AR Foundation. However, the plugin proved to be insufficient for the task. The accuracy of the scanned signs needed to be absolute since a faulty result could ruin the educational purpose. Therefore, a neural network was implemented to overcome that failure. The architecture was decided to be modular and extendable to integrate with the neural network API and communicate with the backend server.

Thirdly, the game required expose functionalities to Editor so that everyone, not just programmers, could participate creating game mechanics. To do that, the architecture was designed to be flexible and serializable interface for data and functions to be shown in the Inspector.

Finally, the game included several subsystems such as: initializing data and services, saving and loading game state, database system, logging players' actions and sending them to the backend server, a scoring system that handles different scenarios, and finally managers that handle game entities in scenes. The architecture harmonically coordinated these subsystems and avoided dependency between them so that adding a new subsystem to the game did not interfere with others nor increase the complexity of the architecture.

4.3 Framework setup and plugins

Virpa2 project was built with Unity version 2019.3.9f1. It had 6 plugins but only 3 of them were essential tools to build the foundation of the game architecture:

- Odin Inspector: this library improves the workflow of Editor customization. However, the most important functionality that was used extensively in the Virpa2

project is the Odin serialization system. The Odin Serializer can serialize interfaces and abstract classes as well as complicated data structures, such as **Dictionary**. Therefore, the plugin plays a crucial role in building a clean architecture.

- Guild-based References: provides **GuidComponent** script to automatically generate persistent GUID (Globally Unique Identifier) for objects that need state saving in scenes.
- ScriptableObject Architecture: this plugin is not an architecture itself but rather a library that already defines and implements basic blocks such as **Variable**, **RuntimeSet** and **GameEvent as mentioned in Chapter 3**. They have debug Inspector and Editor features which are essential to speed up the Virpa2 development process. In this plugin, **RuntimeSet** is renamed to **Collection** but it still shares the same principle with the original **RuntimeSet**.

4.4 Input Handling

The movement of the player's character was controlled by 2 virtual joysticks. The left one was for moving the character relative to the forward direction of the camera and the right one was for rotating the camera. Their relationship and definition are depicted in Figure 5.

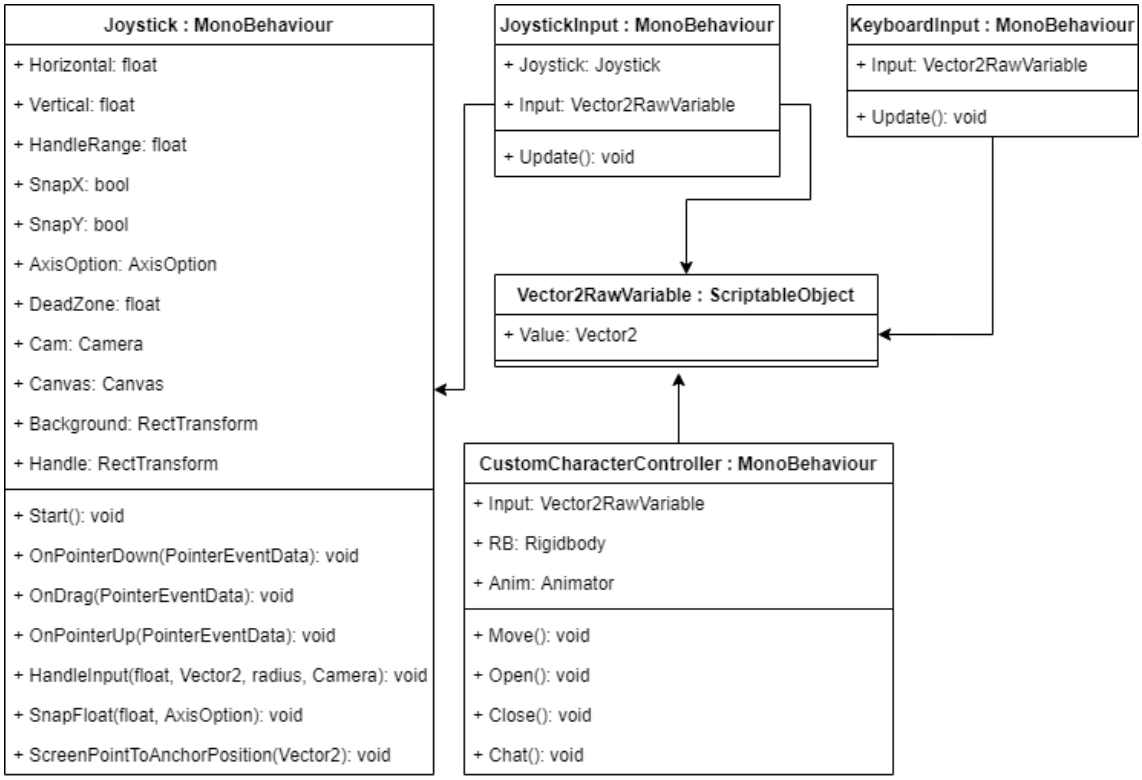

Figure 5. Handling input for player's movement and camera rotation.

The **Joystick** was responsible for handling the player's touch actions on the screen and calculating input value based on those touch positions then adjusting the sprite of the joystick accordingly. The **JoystickInput** extracted **Direction** value from **Joystick** and wrote that value to variable **Input**. Then **KeyboardInput** read input from **Input.GetAxis()** and wrote that value to variable **Input**. The **Vector2Variable** was an intermediary that contains the input value for both reading and writing operations. A new method for handling input can be added without affecting the relationship between input reader (**CustomCharacterController**) and input writers (**JoystickInput** and **KeyboardInput**). The right joystick for handling camera rotation shared the same principle.

4.5 Game Manager

GameManager was a central system that references other subsystems as depicted in Figure 6. It served 2 purposes: being a unified interface for calling most of the functionalities in the game and acting as a coordinator that combines functionalities from multiple subsystems. This made exposed functionalities in Inspector convenient for designers since most functionalities can be found from **GameManager**. In addition, it

encouraged decoupling subsystems by being a middleware for handling composite actions. Subsystems did not need to reference each other to make new functionalities that may beyond their responsibility.

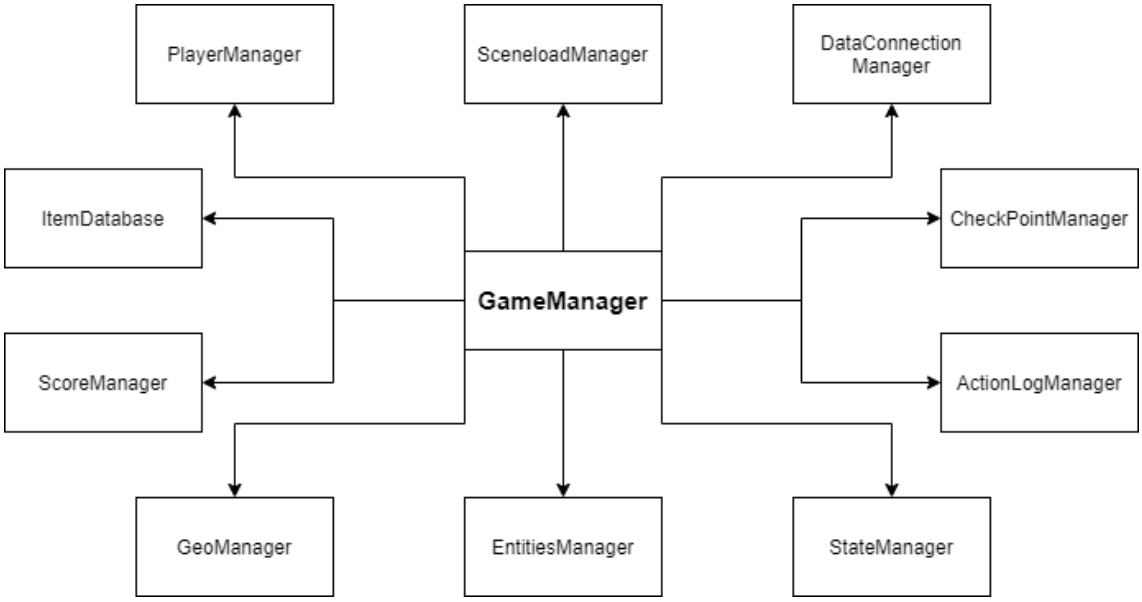

Figure 6. A central GameManger that references subsystems.

The **GameManager** itself was a ScriptableObject which means it belongs to the whole game rather than any specific scene. Its lifetime also binds to the game application rather than a specific GameObject. Since a ScriptableObject instance can be shown in Inspector, its setting can be modified without changing internal code. **GameManager**'s functionalities were exposed to Inspector and can be invoked as demonstrated in Figure 4. This allowed everyone to participate in the game development process and enforces modular design in programming practice.

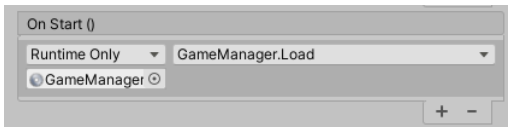

Figure 7. GameManager gets called in Inspector.

4.6 Initializables

ScriptableObject instances are not invoked automatically when the scene is loaded. Therefore, they require another script to call them. Most of the subsystems in Virpa2 were ScriptableObject and they required initialization before being used. This has led to another main role of **GameManager**: initialize other ScriptableObjects. In Virpa2 project, MonoBehaviours were not allowed to be a subsystem that is used by others due to its nature. The **GameManager.Initialize()** invoked all the subsystems that implement **IInitializable** interface as shown in Figure 8.

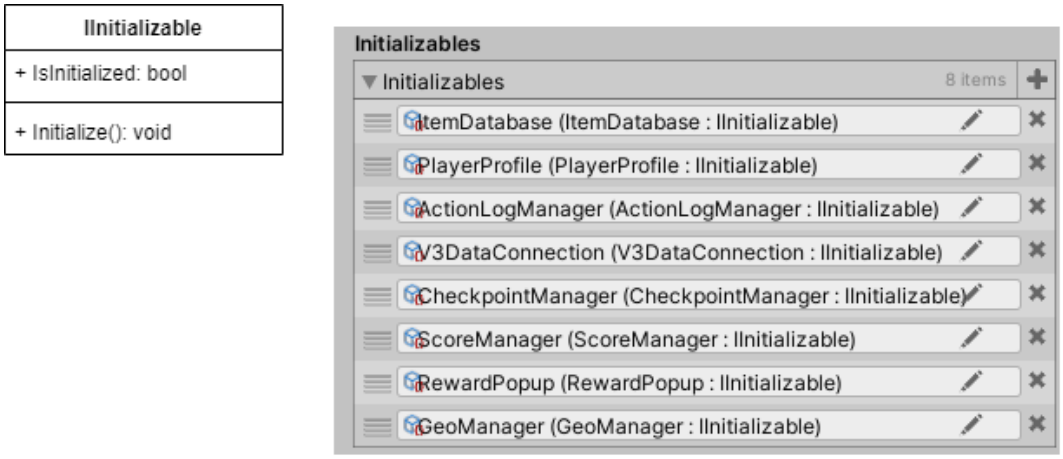

Figure 8. IInitializable interface can be dragged into GameManager's Inpsector.

To invoke **GameManager.Initialize(),** there must be GameObject that references **GameManager** instance and call that method. It was a special GameObject named Main. It was an entry point to execute codes when the scene is loaded as shown in Figure 9. This GameObject contained a MonoBehaviour script which is called **UnityCallbackBehaviour**. The script's execution order was modified to run before any other **MonoBehaviour**s to ensure its deterministic order. Upon invocation, the script triggered UnityEvents that correspond to Unity callbacks such as **Awake()**, **Start()** and **OnDestroy()**. Then the **GameManager.Initialize()** can be attached to one of these UnityEvent to start the initialization. This made the application inspectable and adjustable in the game view without digging into codes. In Virpa2 project, the initialization process happened in the first **Awake()** and within one frame. This ensured other classes can use the services and data from subsystems without any concern about the initialization process. Additionally, this created a safe protocol for initialization which in turn enables

the ability to run the game from any scene as long as that scene calls **GameManager.Initialize()**.

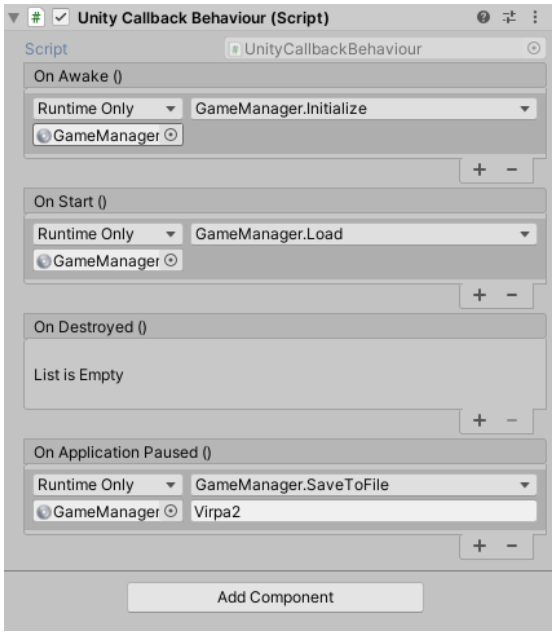

Figure 9. UnityCallbackBehaviour in Inspector.

4.7 Entities Manager

In Virpa2, there were 3 game entities that need to be managed. They were **NPC**s, **Door**s and **Requirer**s. These entities were stored in a **ScriptableObject Collection** and tracked by **EntitiesManager<T>** where T is a MonoBehaviour. Each of these **EntitiesManager** referenced a **Collection<T>** as shown in Figure 10. These managers acted as central processors that handled operations on all the active entities in the scene. One example is saving and loading state operations. They were also containers to store saved data across multiple scenes since these entities were MonoBehaviour. The **Collection**s were general for different purposes. Two of their practical applications in the project were statistic and mini map mechanics. The mini map referenced these **Collection**s to reflect player's progression in a particular scene and show it in the minimap's canvas.

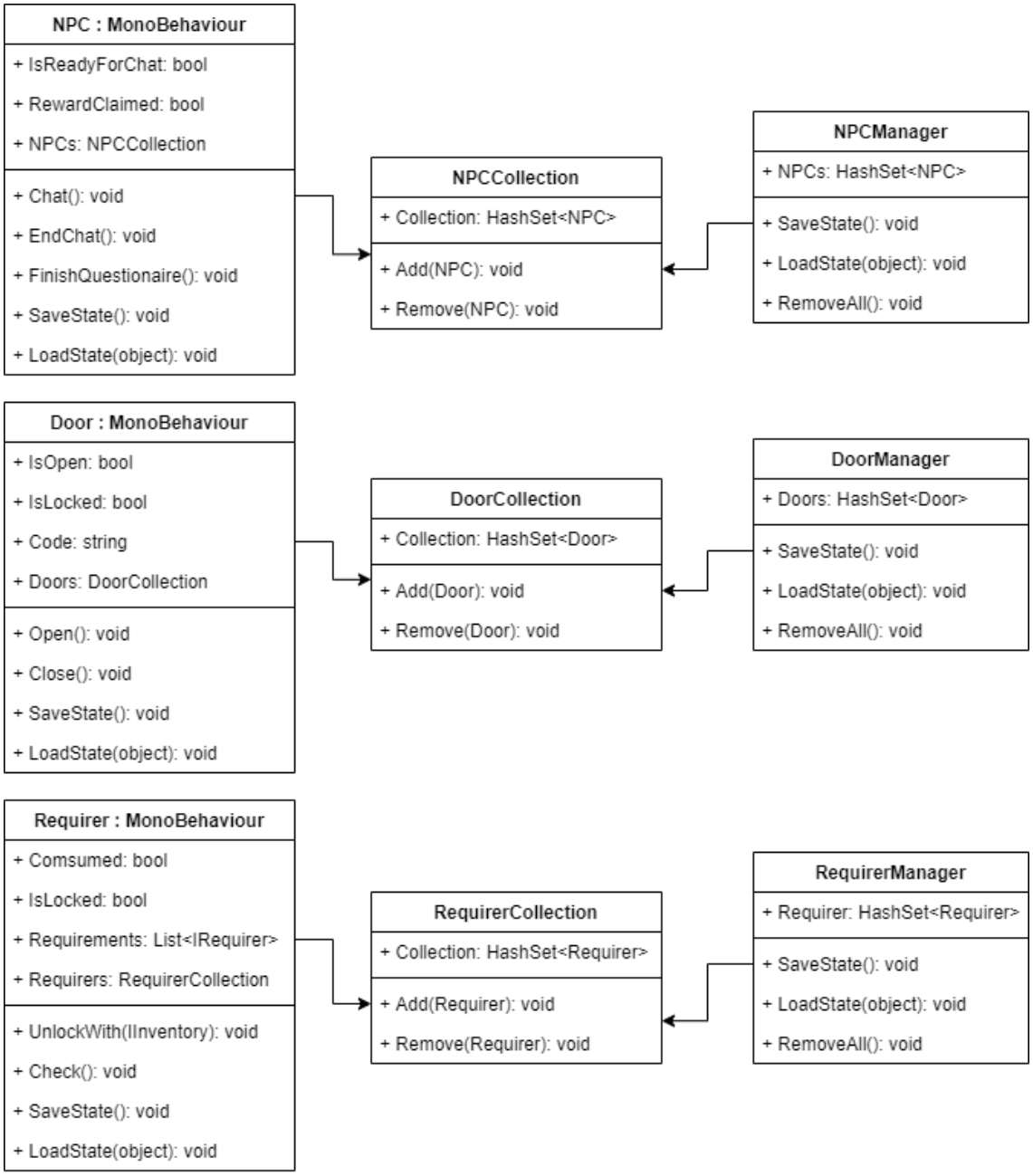

Figure 10. Collections and Managers for game entities.

4.8 State Manager

Every MonoBehaviour object that needed state saving or loading implemented **ISaveState** interface and got tracked by its corresponding **EntitiesManager**. When the

game needed to save these entities, the managers looped through all the entities in **Collection** and capture the state data returned from **SaveState()** and stored them in a **Dictionay<Guid, object>** with the key is the entitiy' GUID and the value was its captured state. The GUID was automatically assigned to an entity when it was first created in the scene by **GuidComponent** script.

There was one more layer of abstraction in the save/load system. The **ISaveState** worked great for objects that can not store the saved data themselves since their lifetime was bound to a GameObject. A ScriptableObject instance, by its nature, can store persistent data throughout the application life cycle. There came another interface to unify both scenarios, the **ISavable**.

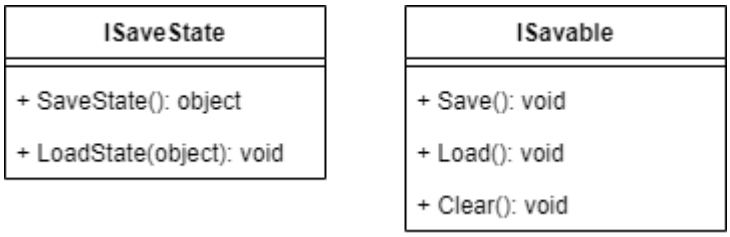

Figure 11. ISaveState and ISavable.

The **EntitiesManager**s inherited the **ISavable** interface and then got enlisted in **StateManager** along with other savable objects which can contain saved data themselves, such as **Inventory**. The **StateManager** had a unified list of **ISavable**s so that any object that needs state saving can register itself to **StateManager** and the application can invoke **StateManager.Save()** to effectively command all the savables to capture their current state as depicted in Figure 12.

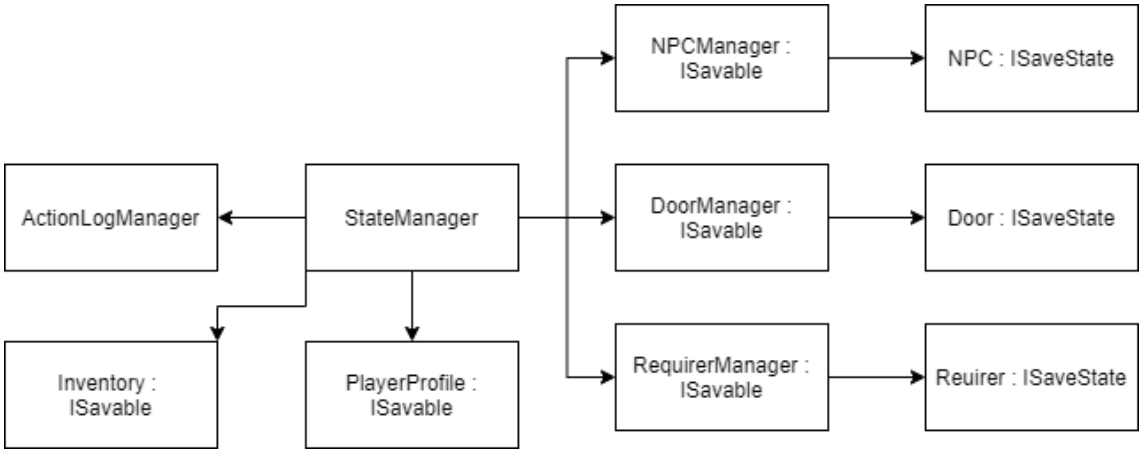

Figure 12. StateManager references other Isavables.

The **StateManager** had one more responsibility: collect the saved data of the whole application and transform it into a serializable format that can be stored in a file. The **ISavable** represented objects that have state persist across multiple scenes whereas **IComposeGameState** represented objects that want their data stored in a unified object, **GameState**, which in turn got serialized into a file.

Figure 13. IComposeGameState interface.

A **GameState** was an object that stores the serializable format of the saved data from **ISavable**s as shown in Figure 14.

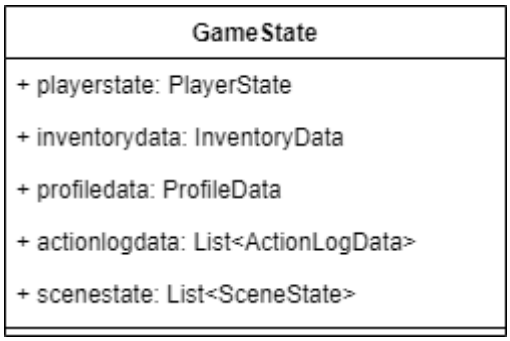

Figure 14. GameState definition.

4.9 Event System

Virpa2 had a few game events which are exposed to the whole game by ScriptableObject GameEvent instances. These assets were put in a directory and serve as a central place to distribute all GameEvent instead of a static class that stores them. Subsystems or GameObjects can reference these GameEvents to subscribe or trigger them without establishing a direct connection to any other subsystems or GameObjects as shown in Figure 15. This effectively reduced dependencies among classes and simplified the structure of game architecture.

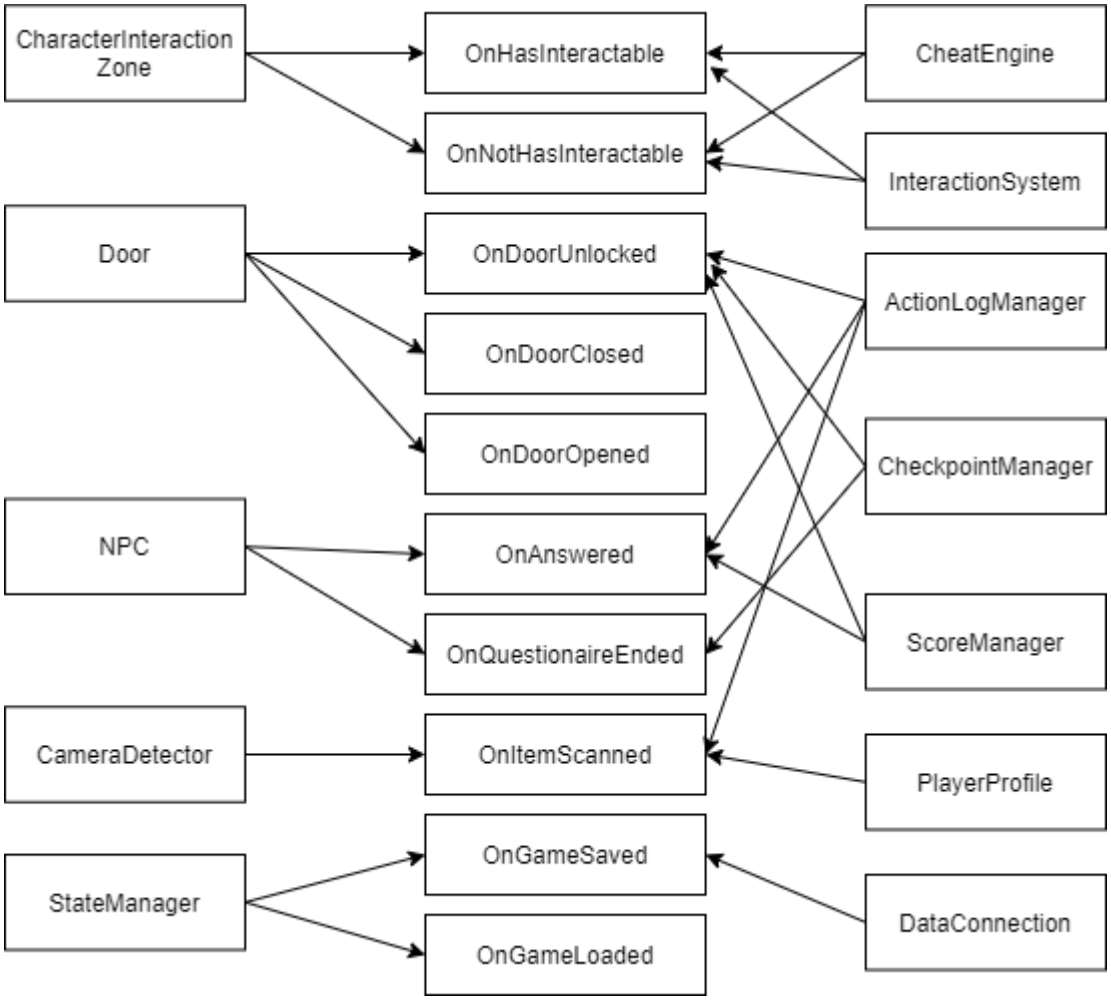

Figure 15. EventSystem connections.

Furthermore, each interactable entity also had an extra C# event type and triggers it upon being interacted. An entity triggered the GameEvent to signal the whole game that an action had been performed on that entity, and another C# event was for the interactor that was currently interacting with it. This completed the communication protocol for the entities.

4.10 Score Manager

The game had different scoring scenarios, such as when the player scans a sign in the real world, when the player unlocks a door, when the player answers a question correctly. By participating in a modular and central event system, the **ScoreManager** can listen to those interested events to give scores. Codes that performed actions, such as

Door.Unlock(), did not need to define how they would give score to the players. Instead, they triggered the GameEvent **OnDoorUnlocked** and the **ScoreManager** listened to that event and then gave scores to the player. Additionally, if a **Door** knows how to give scores, it needs to define how many scores it should give and reference **PlayerProfile** for the score transfer. This makes **Door** has extra responsibilities that it should not have and hence increases the complexity of a **Door** in particular and the architecture as a whole. By putting all the scoring mechanics into **ScoreManager**, the system was simplified and modularized.

Since **ScoreManager** was a ScriptableObject, it can expose setting to Inspector for better visualization and customization as shown in Figure 16.

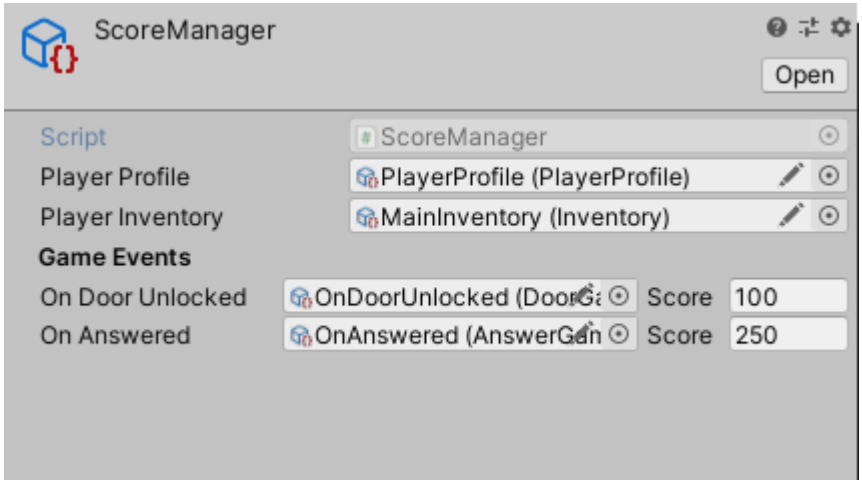

Figure 16. ScoreManager interface.

All scoring settings were defined in one place and can be easily tweaked or changed or even removed without affecting any other part of the game. The **ScoreManager** needed to be initialized in order to participate in the game system. By not getting enlisted in the **Initializables** list from **GameManager**, the **ScoreManager** was unplugged from the system without causing any interference.

4.11 Action Log

Virpa2 project was an educational game. Therefore, statistics and players' activities were important. The game needed to track when players unlock doors with scanned signs,

whether players go into the rooms to answer questions from NPCs, how many answers are correct and so on. These activity records were called action logs and represented by **ActionLog** type as shown in Figure 17.

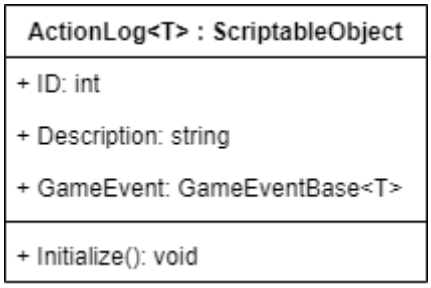

Figure 17. ActionLog's definition.

The ID was used to uniquely identify between different types of ActionLog, such as **ActionLog<AsnwerData>** and **ActionLog<Door>**. It was also used to match with the server backend action log's ID. The GameEvent field was the event that the ActionLog create log data when the event is triggered. The data included the timestamp and description of the logged action. Each **ActionLog<T>** was a ScriptableObject so they can be created in Editor and tweaked in Inspector as shown in Figure 18.

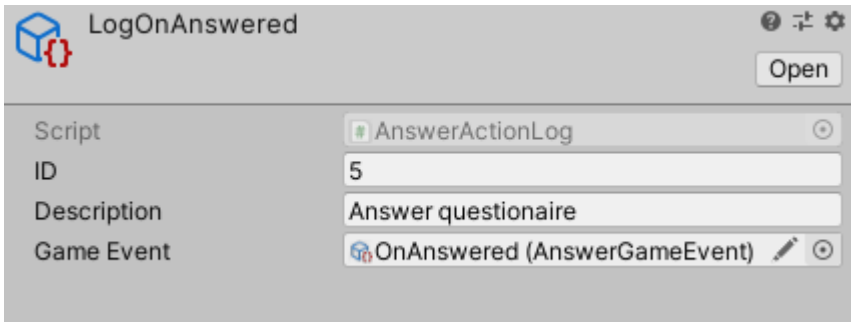

Figure 18. Example of ActionLog<AnswerData>.

The **ActionLogManager** was a central place to contain and initialize action logs. Just like **ScoreManager**, the **ActionLogManager** shared the same philosophy: being independent and modular as much as possible. It provided a user interface for choosing which action should be logged or whether the whole action log subsystem is a part of the game as depicted in Figure 19.

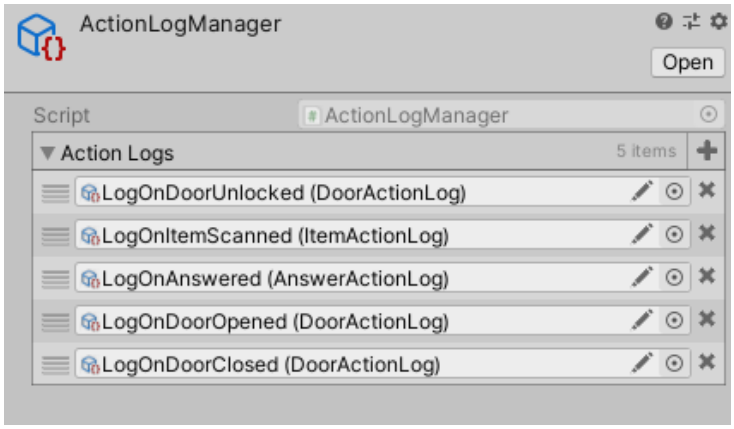

Figure 19. ActionLogManager's user interface.

4.12 Item Database

In Virpa2, players scanned fire safety signs in the real world and get items as rewards. These items could be corresponding sign items, stars or apparels. In a traditional item database, the definitions of these items are usually hardcoded in the database which means in order to create a new entry, developers have to add a new item blueprint in the database script and then recompile the project. This makes the item database code-driven and eliminates the designer's participation in this part. Since item data is pure data so it can be put in a file, get parsed in runtime and populated as entries in the database. This makes the system more data-driven and does not require script recompilation. In Virpa2, the item database applied ScriptableObject extensively. Each item had a blueprint definition and an item instance will be created from those blueprints at runtime. The **Item** shared the exact data definition with **ItemBlueprint** as shown in Figure 20. In Editor, **ItemBlueprint** instances were used as draggable data to help developers design game mechanics. In runtime, these **ItemBlueprint**s produced **Item** instances to be used in the game, such as requirement and **Inventory** system.

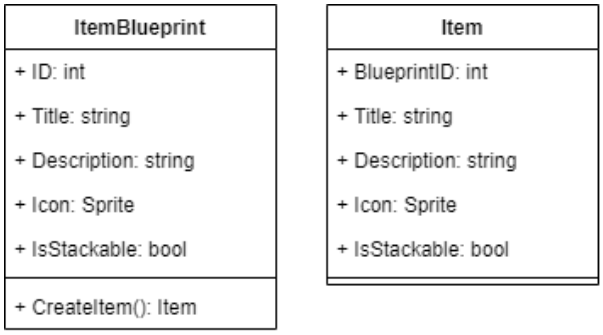

Figure 20. ItemBlueprint and Item share the same data definition.

The **ItemBlueprints** were created and changed flexibly in Editor and then got added to **ItemDatabase** instance (which is also a ScriptableObject) as shown in Figure 21.

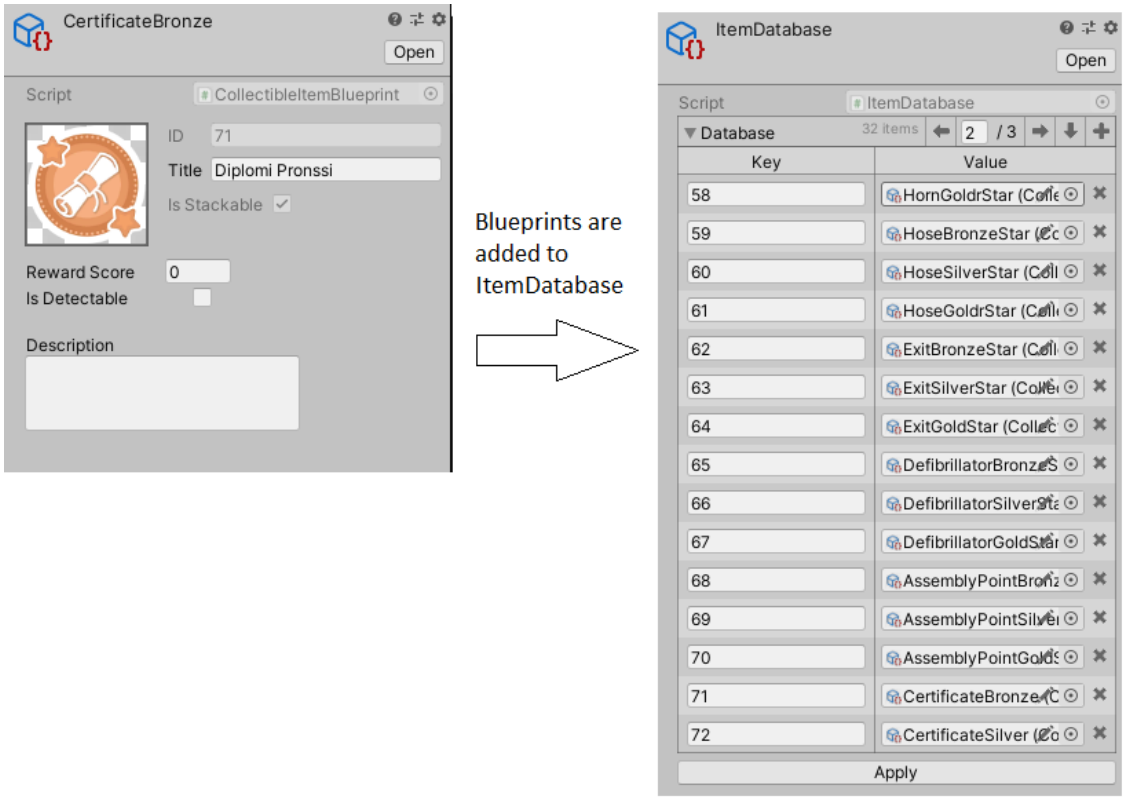

Figure 21. ItemBlueprint is defined in Editor and get added to ItemDatabase.

The **ItemDatabase** was a singleton that acted as a central database for querying items. This provided a convenient and safe way to manage items in the Virpa2 project since **ItemDatabase** can be accessed flexibly via code or Editor.

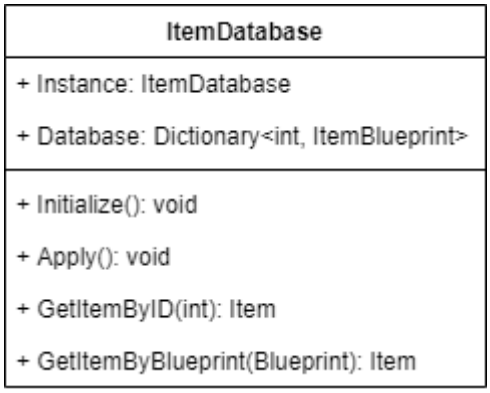

Figure 22. ItemDatabase.

5 CONCLUSION

This thesis was a part of the Virpa2 project. The main goal of this thesis was to build a scalable architecture for experimenting with a new way of utilizing ScriptableObject and satisfying the technical requirements of the project. One of the initial challenges was the time constraint. The architecture had to be built within one month along with the game design phase of the project so that when the basic features of the game are formed, the architecture is ready to accommodate them.

Fortunately, the first version of the architecture was finished in time and continuously updated afterward. In the first version, classes were attached and referenced by concrete types instead of abstract ones or interfaces since Unity's built-in serializer is not able to serialize abstract or interface classes. This created a huge limitation to the architecture since those classes can not inherit multiple interfaces hence reducing modularity. For example, a class can inherit both the **ISaveState** and **IComposeGameState** interfaces to perform saving its state between scene loads and saving the state to a file. In the first version, those interfaces were a single interface. Therefore, a class must implement both of those features even though it may need only one. Another limitation with Unity's serializer is the custom Inspector workflow. One of the aims of the Virpa2 architecture was to provide tools for artists and designers so that they can participate in the project and create new gameplay through those exposed programming tools. Creating tools via Unity custom Inspector API is time-consuming and error-prone. Therefore, Odin Inspector was integrated into the project to help express the architecture's intent better and improve workflow for creating tools. The architecture that is presented in this thesis is the second version built with the Odin Inspector plugin which both functioned and fulfilled all technical requirements of the Virpa2 project.

The architecture formed a guideline for programmers since it created patterns to follow in order for a new feature to be integrated into the project. Adding new functionalities became easier since programmers only needed to understand and followed the guideline and the rest was handled by the system. This also came with a challenge of learning. Programmers must understand the basic idea of the architecture to work with it as well as have knowledge about serialization and ScriptableObject. Therefore, instruction and education were needed to help others participating in the development process.

REFERENCES

Bass, L. J., Bass, L. & Kazman, R., 2013. *Software Architecture In Practice.* 3rd toim. s.l.:Addison-Wesley Professional.

DeLoura, M., 2000. *Game Programming Gems 1.* s.l.:Charles River Media.

Dickinson, C., 2017. *Unity 2017 Game Optimization: Optimize all aspects of Unity performance.* 2nd toim. s.l.:Packt Publishing.

Gamma, E., Helm, R., Johnson, R. & Vlissides, J., 1994. *Design Patterns: Elements of Reusable Object-Oriented Software.* 1st toim. s.l.:Addison-Wesley Professional.

Gregory, J., 2009. *Game Engine Architecture.* 1st toim. s.l.:A K Peters/CRC Press.

Madhav, S., 2013. *Game Programming Algorithms and Techniques: A Platform-Agnostic Approach.* 1st toim. s.l.:Addison-Wesley Professional.

McShaffry, M., 2012. *Game Coding Complete.* 4th toim. s.l.:Cengage Learning PTR.

Murray, J. W., 2014. *C# Game Programming Cookbook for Unity 3D.* s.l.:s.n.

Nystrom, R., 2014. *Game Programming Patterns.* 1st toim. s.l.:Genever Benning.

Penzentcev, A., 2015. *Architecture and implementation of system for serious games in Unity 3D.* s.l.:Masaryk University.

Pizka, M., 2004. *Straightening Spaghetti-Code with Refactoring?.* s.l.:s.n.

Skeet, J., 2013. *C# in Depth.* 3rd toim. s.l.:Manning Publications.

Unity Technologies, 2016. *Unite 2016 - Overthrowing the MonoBehaviour Tyranny in a Glorious Scriptable Object Revolution.* [Online]
Available at: https://youtu.be/6vmRwLYWNRo
[Accessed5 10 2020].

Unity Technologies, 2017. *Unite Austin - Game Architecture with Scriptable Objects.* [Online]
Available at: https://youtu.be/raQ3iHhE_Kk
[Accessed 3 10 2020].

Unity Technologies, ei pvm *Unity Documentation.* [Online] Available at: https://docs.unity3d.com/ [Accessed 3 10 2020].