INTRODUCTION

If you have played The Legend of Zelda: Ocarina of Time (1998), you will recognize this scene.

In this game, there is a scene where Link (the boy in green clothes) places the three Spiritual Stones (green, red, and blue gems) on a pedestal inside the Temple of Time. Upon placing the three stones, multiple objects in the scene react. For instance, a door in front of him opens, revealing an entrance to the Chamber of Time; the Triforce (shiny triangle on top of the door) lights up; and epic music plays.

GOAL

Our goal is to recreate, in a very simple manner, this memorable game mechanic by exploring a use case for two new concepts: static fields and events.

ACTIVITY 1 - Completing Script Implementation

- **Step 1.** Download the file "**spiritualStones.unitypackage**" from the Module: Software Design Patterns.

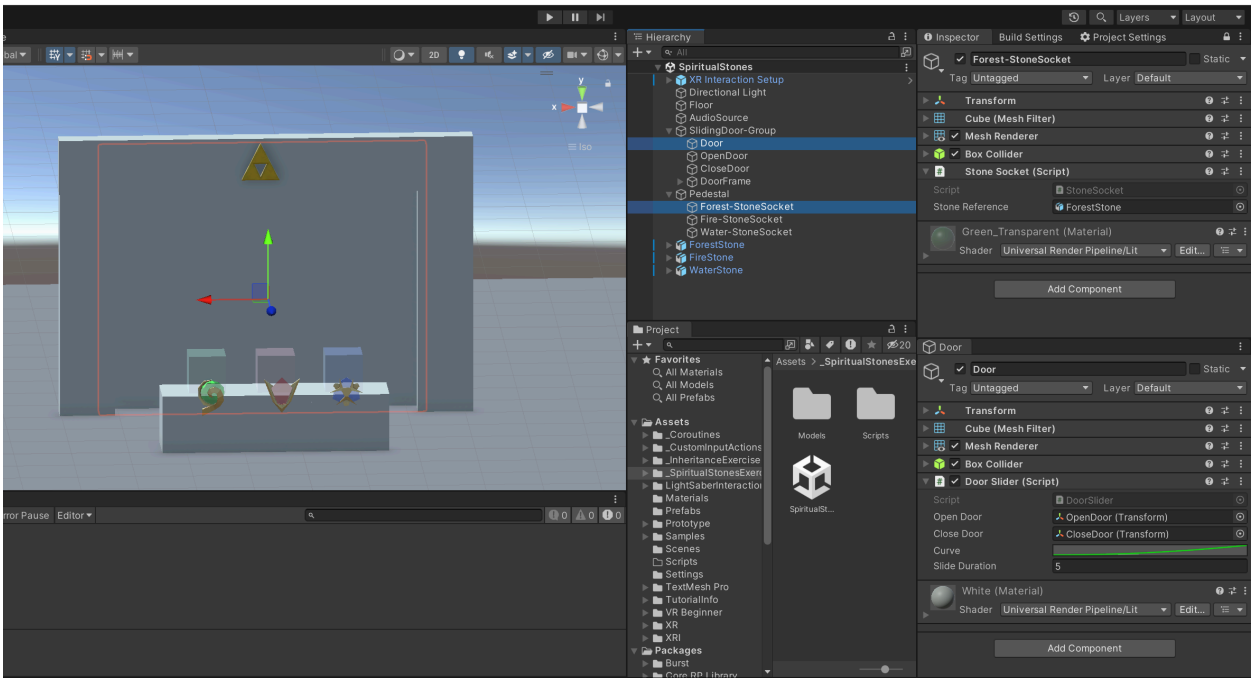
- **Step 2.** Examine the scene's construction.

  Two scripts are included: `SlidingDoor` and `StoneSocket`.

  The `SlidingDoor` script is already implemented. Its primary function is to open and close the door when its corresponding public functions are called. Pay close attention to how the `AnimationCurve` class, coroutines, and the `Vector3.Lerp(arg0, arg1, arg2)` function are used. These are widely used tools that will be beneficial to you.

  The `StoneSocket` script requires completion. This script registers when a stone enters or exits the socket. You need to modify the script so that each slot accepts only a specific type of rock. For example, the red socket should only accept the fire stone. Consider using Tags, the name of the script or enums to compare the value of the stone entering and the one that the socket expects.

- **Step 3.** You also need to track the number of stones placed and open the door when all three stones are in the correct spots. Consider creating a third script, such as a "StoneSocketManager," to handle this, or determine whether you can use a static field. Refer to this video to learn about static classes and fields. [C# Statics in Unity! - Intermediate Scripting Tutorial]



A Journey into the Observer Pattern

ACTIVITY 2 - Events

In this activity, you will implement and compare three different system architectures to examine how scripts interact. The objective is to see what problems **events** solve and how to use them in practice.

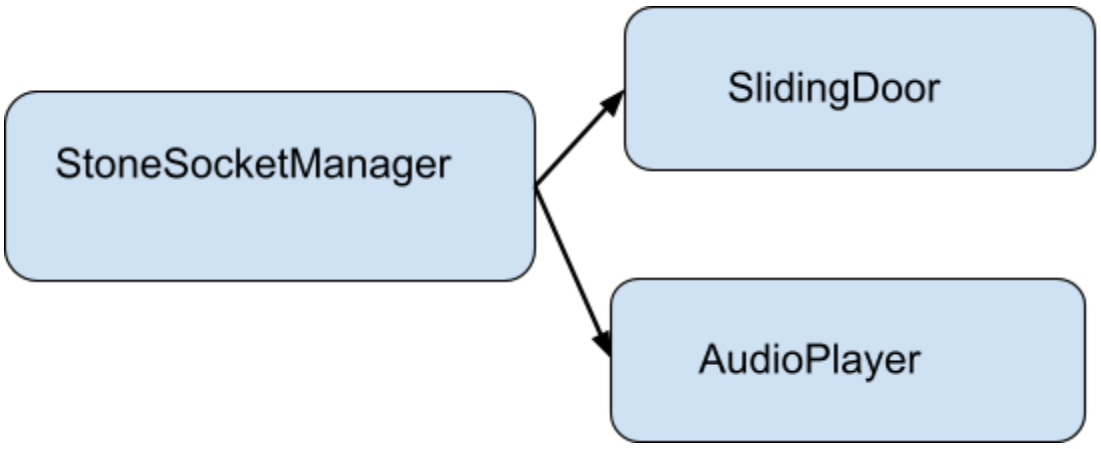
ARCHITECTURES

1. **Tightly Coupled Architecture**

One class has a dependency on many classes. It knows a lot about different classes.

In this architecture, the `StoneSocket` script directly references/depends on the `SlidingDoor` and `AudioPlayer` scripts.

A drawback of this approach is that if we want to add more objects, like the Triforce, to react when all the stones are placed, we must add another dependency to the `StoneSocket` class. This increases the class's responsibilities and coupling, making it harder to maintain and potentially leading to code that is difficult to understand and modify. Adding more dependencies like this makes the `StoneSocket` script less reusable and more tightly coupled to the specific objects it interacts with.

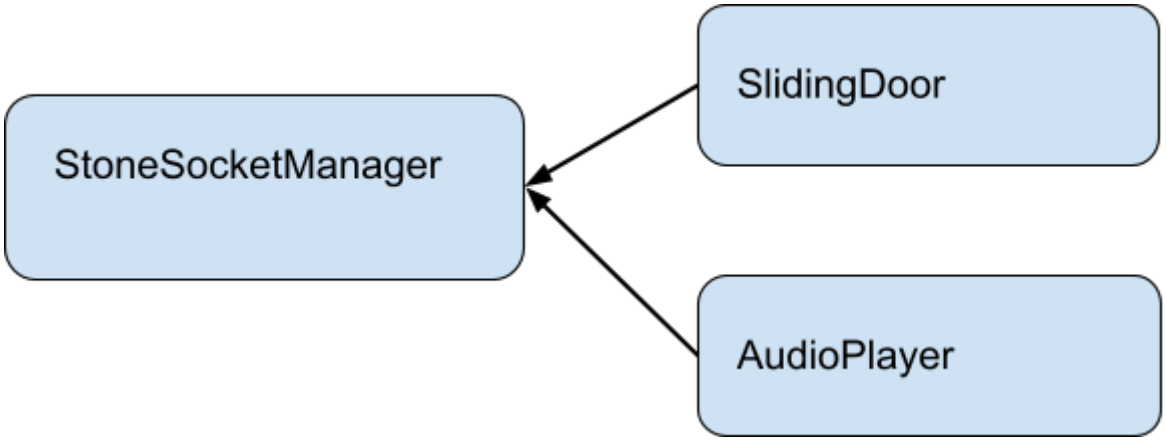


2. Inverted Dependency Architecture

In this architecture, instead of the `StoneSocket` depending on the `SlidingDoor` and `AudioPlayer`, the `SlidingDoor` and `AudioPlayer` depend on the `StoneSocket`. This inversion of dependency can be beneficial.

In this example, we are in a better position in terms of decoupling. `StoneSocket` no longer depends on `SlidingDoor` and `AudioPlayer`. This makes `StoneSocket` more reusable and less prone to changes if we add or remove other objects that need to react to the stone placement.

However, the drawback is that `SlidingDoor` and `AudioPlayer` now need a way to know when all three stones have been placed. A naive approach would be to constantly check a status flag in the `StoneSocket` within their `Update` functions. This is inefficient because the condition of placing all three stones is likely to happen only once (or a very small number of times) during the entire game. Constantly checking in `Update` wastes resources. A better solution is needed.



3. **Inverted Dependency using Events Architecture**

With the Inverted Dependency using Events architecture, we maintain the benefits of loose coupling (like in the previous example). However, instead of constantly checking if all stones are placed (which would be inefficient), we now only run code (the actions of the door opening and sound playing) *when the actual event of placing the last stone happens*. This makes it much more efficient because we're not wasting resources checking every frame in `Update`. We only react when the specific event we're interested in occurs.

How Events work: The Observer Pattern

- **Subject sends/emits/triggers the event:** The subject (or observable) is the entity that triggers or emits the event.
- **Observers/Listeners subscribe to the event:** Observers (or listeners) register their interest in the event with the subject.
- **Subject notifies the observers:** When the event occurs, the subject notifies all subscribed observers.
- **Observers/Listeners react to the event:** The observers, upon being notified, execute their corresponding event handler functions.