Project: Abyss / Inventory Module

Architecture/Design Document

**Table of Contents**

1 Introduction.. 3

2 Design Goals.. 3

3 System Behavior.. 4

4 Logical View... 4

4.1 High-Level Design (Architecture)… 4-6

4.2 Mid-Level Design… 7

4.3 Detailed Class Design… 8

5 Process Views... 9-17

6 Use Case Views... 18

Change History

**Version:** 0.1

**Modifier:** Austin Morris

**Date:** 11/28/2023

**Description of Change:** Started the module with basic overviews.

**Version:** 0.2

**Modifier:** Austin Morris

**Date:** 12/6/2023

**Description of Change:** Finished module.

# **1 Introduction**

This document describes the architecture and design for “Project: Abyss”, a game being developed by Astralwater Interactive. Embark on a groundbreaking cooperative underwater adventure with "Project Abyss," where the uncharted depths of Hollow Earth reveal a hidden world teeming with enigmatic flora, fauna, and ancient mysteries. As intrepid researchers contracted by a major tech conglomerate, you and your partner dive into the abyssal unknown armed with cutting-edge technology. However, beneath the waves, the shadows of corporate indifference threaten your every move.

The purpose of this document is to describe the architecture and design of the Inventory Module application in a way that addresses the interests and concerns of all major stakeholders. For this application the major stakeholders are:

● Developers – they want an architecture that will minimize complexity and development effort.

● Project Manager – the project manager is responsible for assigning tasks and coordinating development work. He or she wants an architecture that divides the system into components of roughly equal size and complexity that can be developed simultaneously with minimal dependencies. For this to happen, the modules need well-defined interfaces. Also, because most individuals specialize in a particular skill or technology, modules should be designed around specific expertise. For example, all UI logic might be encapsulated in one module. Another might have all game logic.

● Maintenance Programmers – they want assurance that the system will be easy to evolve and maintain on into the future.

# **2 Design Goals**

The design priorities for the Inventory system are:

● Allow the player to have an inventory to hold equipment.

● Let the inventory equipment be used with player input.

● Make the inventory modular to ensure that it can be placed on non player objects, like storage.

● Allow seamless transfer between two different inventories.

# **3 System Behavior**

The inventory module is built with modularity in mind, ensuring that the player will be able to store items in a hotbar, while making sure that the inventory can also be applied to any form of storage we want in the game.

In our case, we will have one main storage chest for the player to store equipment once purchased from the shop. The player can click and drag items from their hotbar into the storage, and vice-versa.

# **4 Logical View**

The logical view describes the main functional components of the system. This includes modules, the static relationships between modules, and their dynamic patterns of interaction.

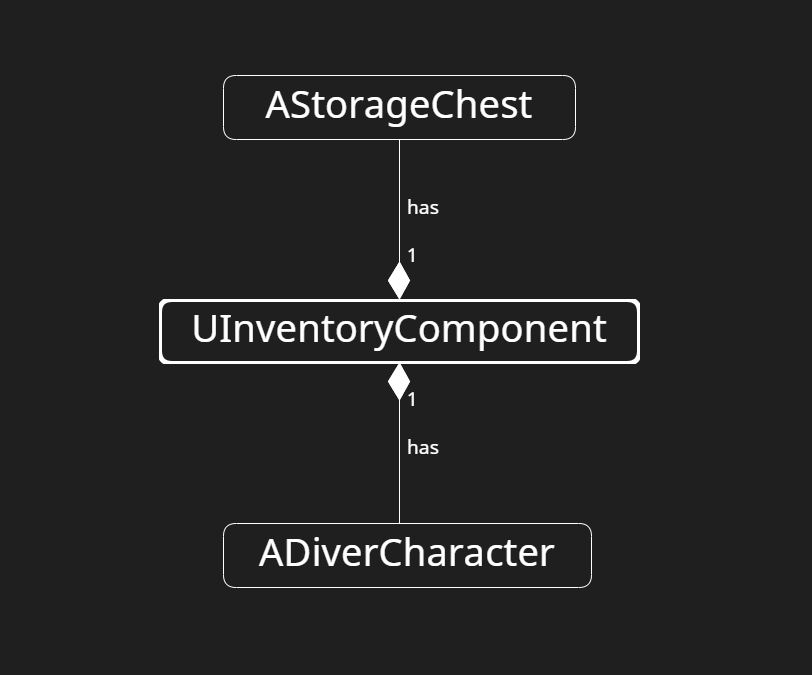
In this section the modules of the system are first expressed in terms of high level components (architecture) and progressively refined into more detailed components and eventually classes with specific attributes and operations.

## **4.1 High-Level Design (Architecture of the Entire system)**

The high-level view consists of…

* **Player System** is the central component that manages the player's actions, interactions, and progression throughout the game. It includes the player character, input handling. Key responsibilities involve controlling the player's movements, handling inputs, and triggering movement states such as swimming.
* **Photography System** is responsible for implementing the mechanics related to underwater photography. It includes the Photography Camera and associated components. This system allows players to take pictures during dives, manage a limited film capacity, store pictures in a gallery, and upload relevant ones to the Collection journal, contributing to the player's knowledge about underwater creatures.
* **Creature System** manages the behavior, interactions, and characteristics of underwater creatures. This system includes various types of fish, their AI, and the logic governing their movement, appearance, and responses to the player's actions. The system ensures a dynamic and realistic underwater ecosystem, providing challenges and opportunities for the player.
* **AI System** governs the artificial intelligence of both hostile and non-hostile entities in the game. It includes the behaviors and decision-making processes of underwater creatures, as well as any AI-driven challenges or enemies the player may encounter during dives.
* **Missions System** oversees the organization of missions into distinct categories, monitors the progression of objectives, and governs the allocation of rewards and unlocks. Accessible through the PC Terminal, players can seamlessly navigate through available, active, and completed missions. The PC Terminal serves as the central hub for mission management in the form of an email system, allowing players to accept new missions and submit completed ones, providing a comprehensive interface for tracking and advancing in the game's narrative.
* **Health Component System** manages the player's health. It includes components that handle damage, healing, and any other health-related mechanics. This system ensures that the player's survival is a key consideration during dives and encounters with underwater creatures.
* **Stress Component System** adds a psychological horror element to the game. It manages the player's stress levels based on in-game events, environmental factors, and encounters. High stress may affect performance or trigger adverse effects, adding an immersive layer to the overall experience.
* **Boat System** encompasses the functionality related to the player's boat. This includes navigation between dive locations, boat inventory, and serving as a safe zone between dives. The boat system also facilitates equipment upgrades, PC Terminal interaction, and overall progression.
* **DiveCage System** is used as a later game upgrade that can be used to speed up travel time to get to deeper locations.
* **Equipment System** is the main item system in the game. Players will be able to buy and use equipment from the shop, including a camera, deployable light sources, and more. Some equipment is consumable, and some is permanent.
* **Divebot System** encompasses the functionality to the controllable Divebot available in the game. The Divebot will be possessed on death as a spectator mechanic, and can be possessed on command to allow remote control for scouting purposes. This will include a custom movement component to assist in executing some of these tasks.
* **Upgrades System** enables the enhancement and customization of the player's equipment, boat, and dive-related tools. Players can earn or purchase upgrades to improve their camera, dive cage, and other components, providing a sense of progression and empowerment.
* **Shop System** allows players to buy in-game equipment and upgrades. It provides a marketplace where players can spend earned resources to acquire new equipment, upgrade existing gear, or replenish consumables.
* **PC System** refers to the personal computer within the game. It serves as the hub for managing various aspects, such as reviewing your research journal, viewing photographs, accessing the in-game shop, and planning future dives. The PC system contributes to the player's overall agency and strategic decision-making.
* **Inventory System** refers to the inventory component that is attached to the player and storage chest. The player will be able to swap equipment to and from the storage chest.

## **4.2 Mid-Level Design of the Inventory Module**

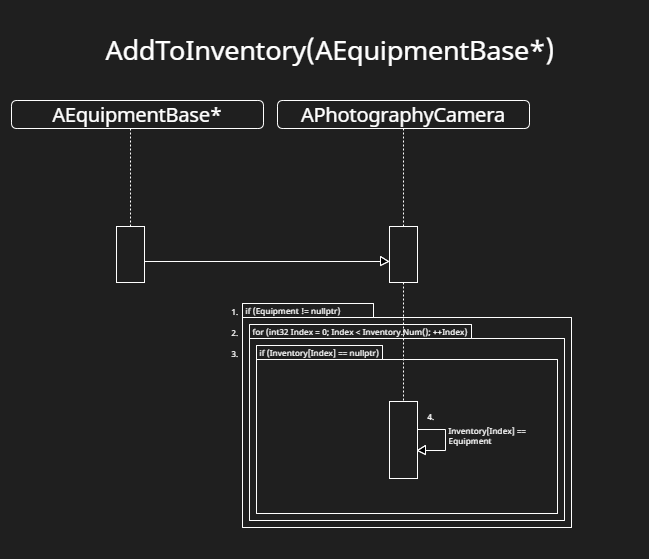


The inventory system is modular and self contained and contains a few key functionalities.

* **AddToInventory()** - Adding equipment to the inventory in the earliest available slot.
* **RemoveFromInventory()** - Removing equipment from the inventory, using an index.
* **SwapEquipment()** - Swaps two equipment between each slot.
* **SwapEquipmentBetweenInventories()** - Manages equipment swapping between two inventories.

## **4.3 Detailed Class Design of the Inventory Module**

# **5 Process View of the Inventory Module**

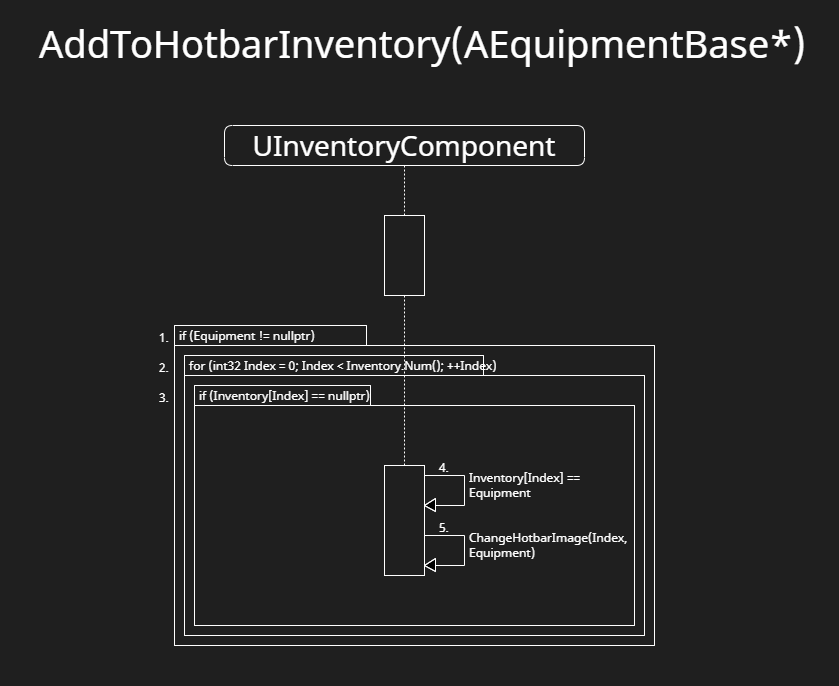
**AddToInventory(Int32 Index)**

This function is called when adding an equipment to an inventory. It always adds it to the earliest available empty slot.

1. If the equipment being added isn’t nullptr, proceed.
2. This loop goes through every inventory space, and within each iteration,
3. Checks if a slot is null at the current inventory index
4. If it is null, add an equipment at the current slot.

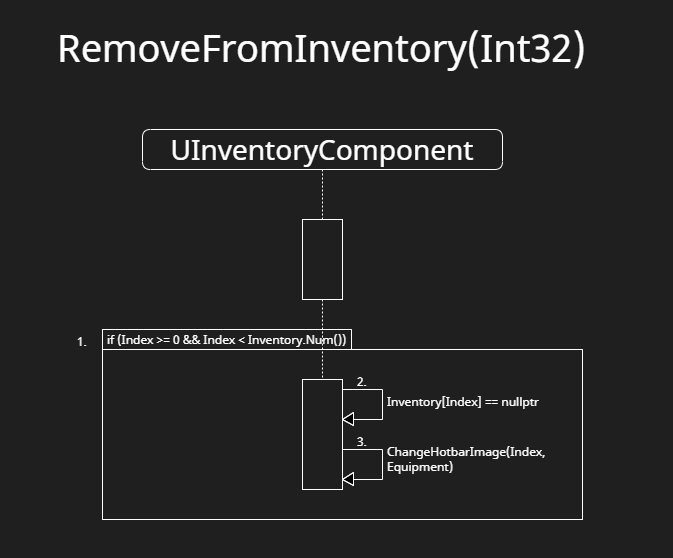
If it isn’t null, and no slots are available, nothing happens.

**AddToHotbarInventory(Int32 Index)**



This function is called when adding an equipment to the player inventory. It always adds it to the earliest available empty slot.

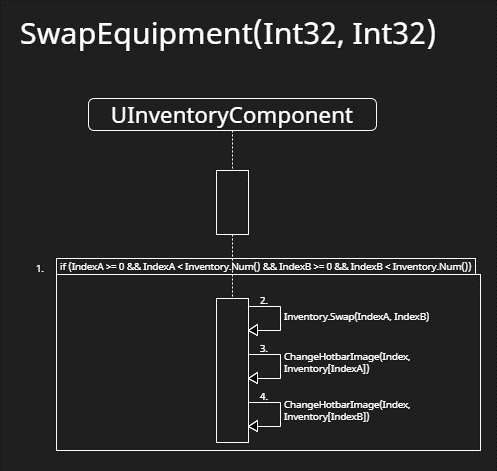
1. If the equipment being added isn’t nullptr, proceed.
2. This loop goes through every inventory space, and within each iteration,
3. Checks if a slot is null at the current inventory index
4. If it is null, add an equipment at the current slot.
5. Changes the hotbar image for the player widget depending on the index, as well as what equipment is being added.

**RemoveFromInventory(Int32 Index)**

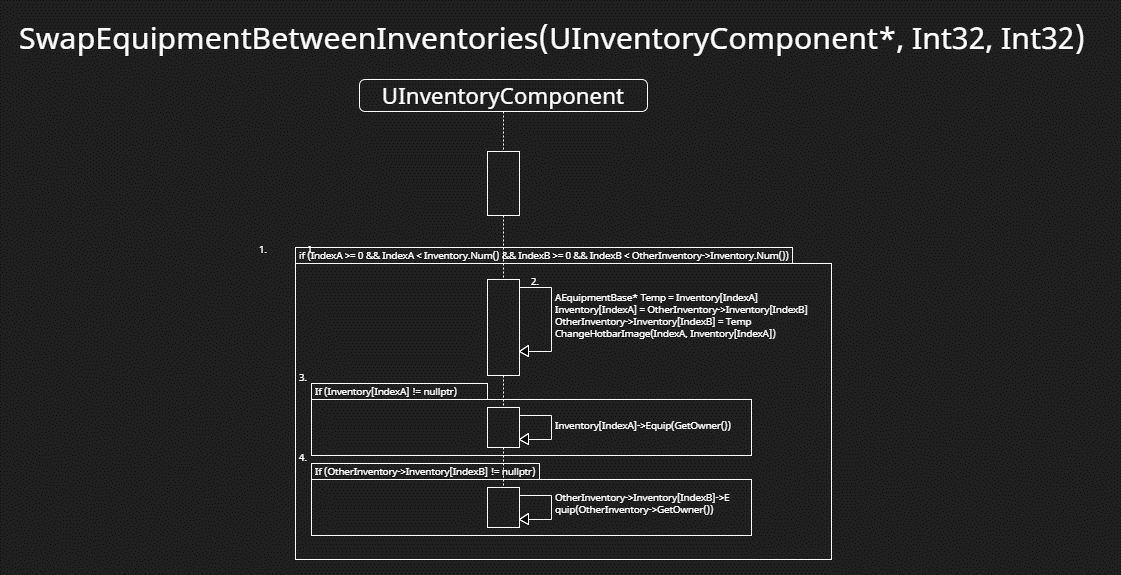
This function is called when removing an item from an inventory. This method deletes the item altogether.

1. If the passed in index is valid for the inventory, proceed
2. Sets the equipment at the passed in index to nullptr.
3. Changes the hotbar image to be empty.

**SwapEquipment(Int32 IndexA, Int32 IndexB)**

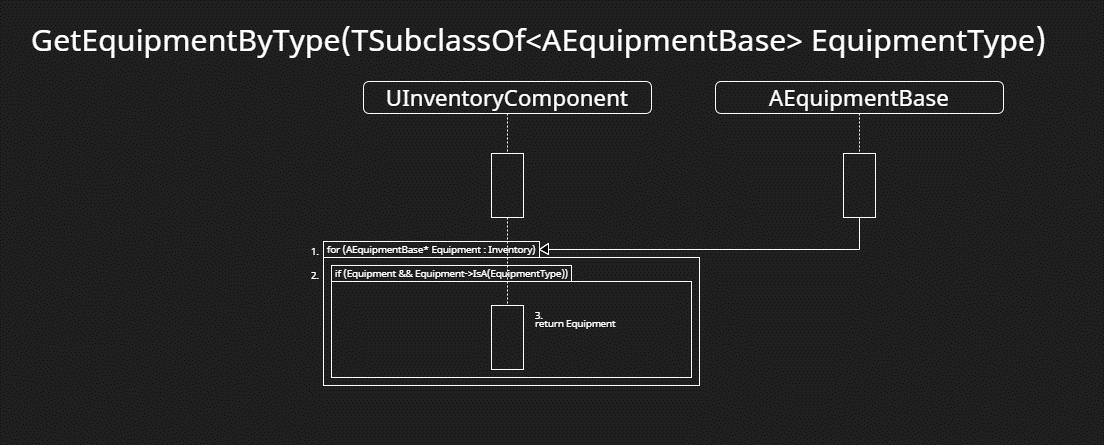
****This function is called when swapping items in an inventory, from one slot to another.

1. If the passed in index is valid for the inventory, proceed
2. Swaps the slots using the passed in indices
3. Changes the hotbar image of IndexA
4. Changes the hotbar image of IndexB

**SwapEquipmentBetweenInventories(UInventoryComponent\* OtherInventory, Int32 IndexA, Int32 IndexB)**

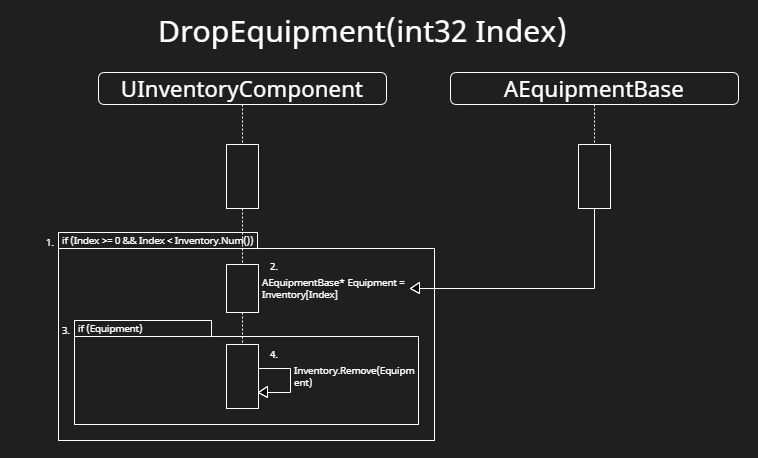
This function is called when swapping items from one inventory to another, from one slot to another slot in the other inventory.

1. If the passed in index is valid for the inventory, proceed
2. Performs swapping logic that is necessary for separate inventory changes.
3. If the passed in equipment isn’t null, equip it to the current inventory owner.
4. If the passed in equipment isn’t null, equip it to the current inventory owner.

** GetEquipmentByType(TSubclassOf<AEquipmentBase> EquipmentType)**

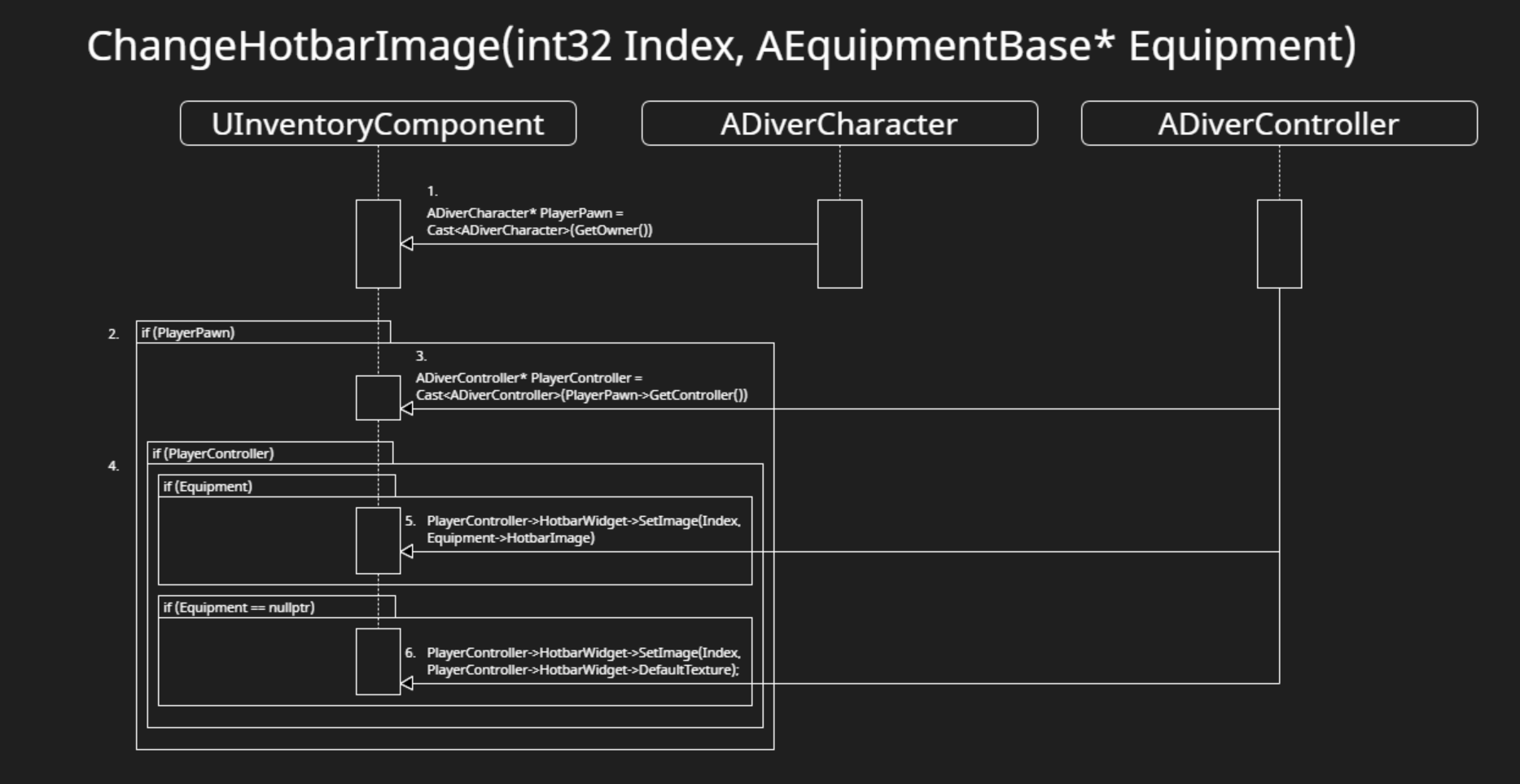
This function is used to get specific equipment in the inventory using a equipment type/subclass.

1. Ranged based loop on all equipment.
2. If the equipment in the ranged based loop matches with the type of EquipmentType, proceed.
3. Return the equipment.

**DropEquipment(Int32 Index)**

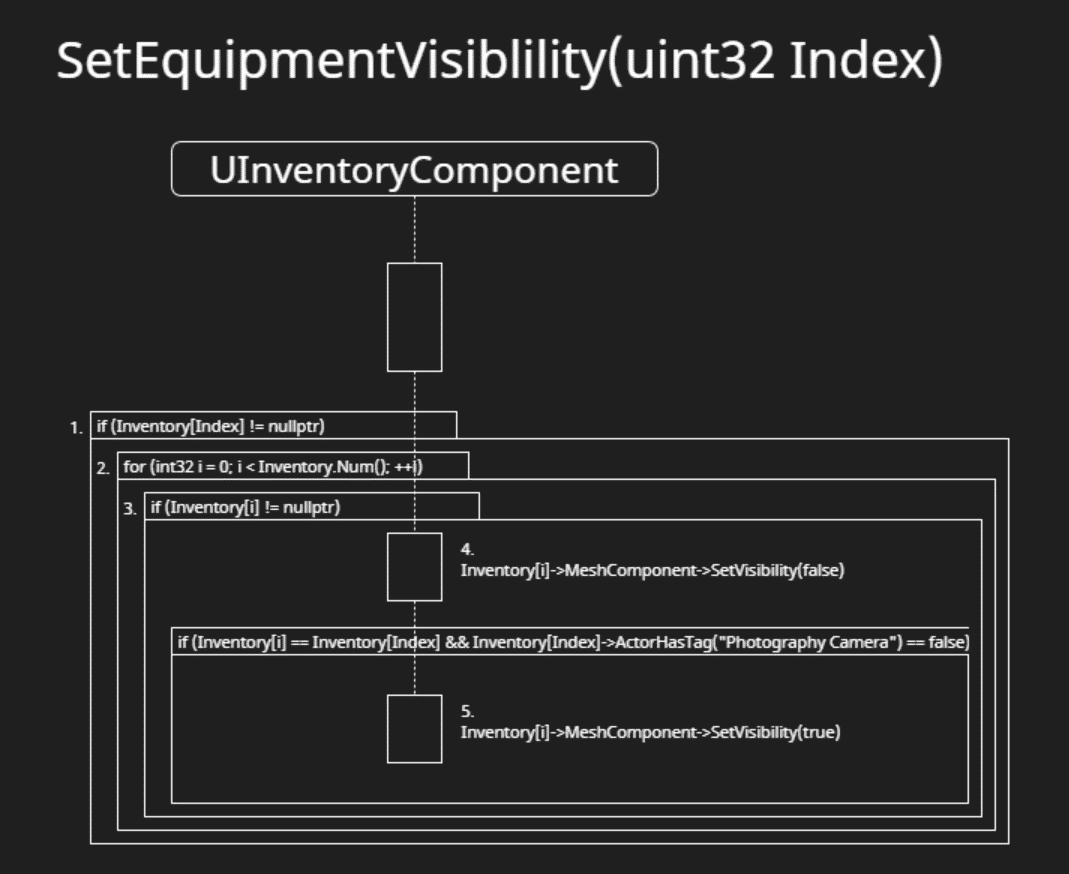
This function is called when dropping an equipment. For this version of implementation, it just removes it from the array.

1. If the passed in index is valid for the inventory, proceed
2. Assigns temp equipment to the current index.
3. If Equipment is valid
4. Removes the selected equipment from the inventory array.

**ChangeHotbarImage(Int32 Index, AEquipmentBase\* Equipment)**

This function is called to change the images of the players hotbar based on the passed in equipment.

1. Casts to the player pawn using GetOwner().
2. If player pawn is valid, proceed.
3. Casts to the controller from the player pawn.
4. If the controller & equipment is valid, proceed.
5. Sets the image of the hotbar slot to the passed in equipments hotbar image.
6. If the equipment is not valid, set the image to the default blank texture.

**SetEquipmentVisibility(uint32 Index)**

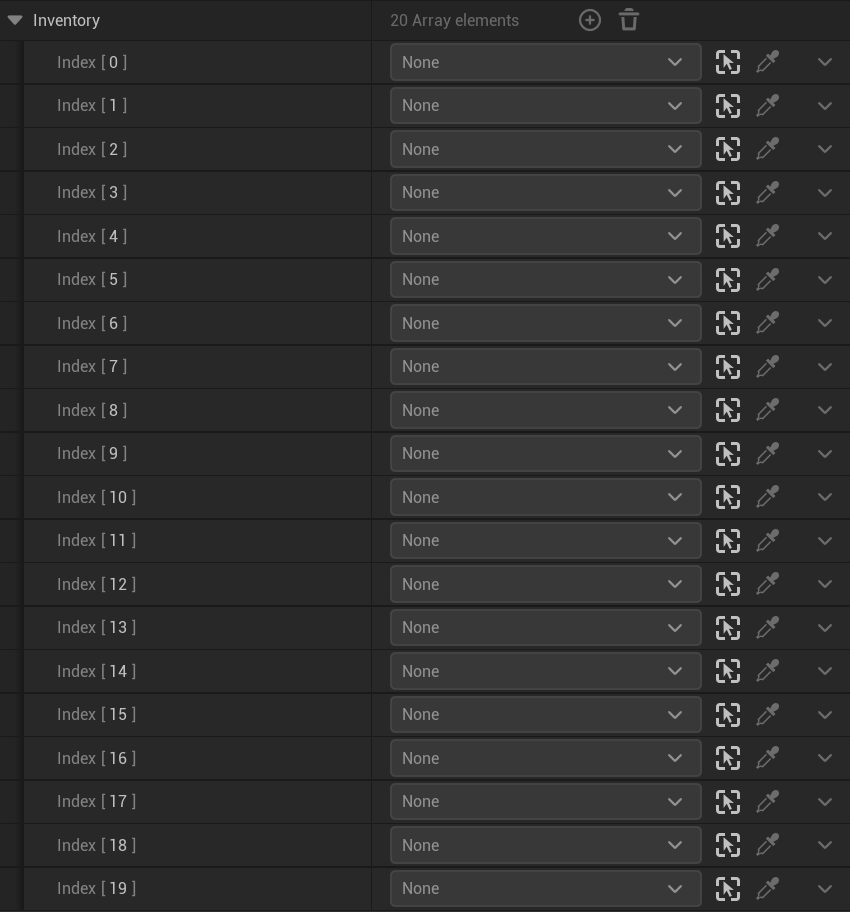
This function is called when swapping current selected slot in player hotbars. It changes the visibility of the equipment based on the index passed in.

1. If the current equipment at the passed in index is valid, proceed
2. For loop for the whole inventory
3. If inventory at the current index in the for loop is not null, set visibility to false.
4. If inventory at the current index in the for loop is equal to the passed in inventory index and the equipment isn’t the camera, set visibility to true.

# **6 Use Case View**

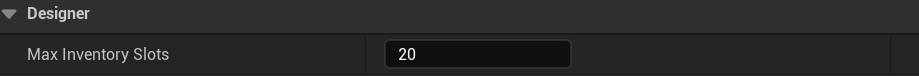
Below are the designer variables which can be found within the InventoryComponent when placed on an actor.

Inventory Array



This is a blueprintreadonly for designers and bug fixing to see current inventory without using breakpoints. It cannot be directly edited.

MaxInventorySlots



This int can be changed to easily adjust the size of an actors inventory.