Project Abyss / Mission Module

Architecture/Design Document

**Table of Contents**

1 Introduction.. 3

2 Design Goals.. 3

3 System Behavior.. 5

4 Logical View... 4

4.1 High-Level Design (Architecture) 4

4.2 Mid-Level Design. 7

4.3 Detailed Class Design. 8

5 Process View... 9

6 Use Case View... 25

Change History

**Version:** 0.1

**Modifier:** Renz Leo Nicolas Dela Cruz

**Date:** November 3rd, 2023q

**Description of Change:**

* Initial Creation

**Version:** 0.5

**Modifier:** Renz Leo Nicolas Dela Cruz

**Date:** November 3rd, 2023

**Description of Change:**

* Document Completion

**Version:** 0.6

**Modifier:** Renz Leo Nicolas Dela Cruz

**Date:** November 17th, 2023

**Description of Change:**

* Changed document text “quest” to “mission”
* Still need to change diagrams to missions
* Change to add bullet points to “Description of Change”

**Version:** 0.7

**Modifier:** Renz Leo Nicolas Dela Cruz

**Date:** November 17th, 2023

**Description of Change:**

* Updating function runtime diagrams and some descriptions. Green means done

**Version:** 0.9

**Modifier:** Renz Leo Nicolas Dela Cruz

**Date:** December 7th, 2023

**Description of Change:**

* Updating function runtime diagrams and some descriptions
* Added more to Design Goals
* Updated Mid-Level Design diagram
* Updated Detailed Class Design diagram
* Added blueprint prototypes and their events
* Changed green back to black
* Updated High-Level Design diagram

# **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 Mission 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 Mission System Module system are:

* This design should allow designers to have easy access to all necessary properties to create desired effects.
* This design should keep the Missions and its separate components easy to understand and should keep everything modular and simple.
* This design should make it easy for players to progress and view their progress throughout gameplay.
* This design should be intuitive to players upon the first few interactions.

# **3 System Behavior**

The Mission Module revolves around a Data Table that contains all missions and their information. The MissionLogSubsystem holds that information and organizes all of them into individual categories of availability such as completed or currently active. The players get access to some of this information through the PC terminal. The PC terminal is where the players can see available missions and their information, accept missions, and hand in missions. When a mission is accepted, AMissionBase actor is created in the world by the subsystem which is how objectives get tracked and progressed.

# **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)**

A diagram of a computer program

Description automatically generated with medium confidence

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.

## **Mid-Level Design of the Mission Module**

A black screen with white text

Description automatically generated

The Mission Module consists of few main classes with many structs to contain and organize mission information.

* **UMissionLogSubsystem** – Main class that organizes the entire system. Has reference to MissionDetails DataTable that is set-up in blueprint. Contains functions that updates mission organization such as accepting/completing missions. Also contains a function to update mission objectives if the right Objective ID is called using a delegate. This is how other objects will update active missions.
* **UDataTable** – A data table call Mission Details that contains all missions in the game. These missions are contained in structs called FMissionDetails.
* **FMissionDetails** – A struct containing information about the mission.
  + MissionName – The Name of the Mission.
  + LogDescription – The Description of the Mission.
  + TrackingDescription – The tracking description of the mission (not in use)
  + Type – Of EMissionType. The type of mission
  + Stage – Contains multiple FStageDetails that contains details of each stage.
  + Biome – Of EBiome. The biome that the mission takes place in
  + DepthUpgradeRequired – Of EDepthUpgradeRequired. The depth upgrade that the players need to unlock the mission
  + PrerequisiteMissions – The Missions IDs of the missions needed to unlock this mission.
* **FStageDetails** – A Struct containing information about a stage of a mission.
  + StageName – The name of the Stage
  + Description – The Description of the Stage
  + Objectives – Contains multiple FObjectiveDetails that contain details of each objective.
  + MoneyReward – The amount of money the stage will reward the players (Likely to transfer to FMissionDetails)
* **FObjectiveDetails** – A struct containing information an objective in a stage.
  + ObjectiveName – Name of the objective
  + Description – Description of the objective
  + Type – The type of objective
  + ObjectiveID – The ID that is called by objects in the world that tell AMissionBase actors to increment their objective count if the IDs match
  + Quantity – The amount of how much the objective needs to be done
  + IsOptional – If the mission is optional or not
* **EObjectiveType** – The type of objective (currently only Location and Picture types)
* **EMissionType** – Type of mission being either being a main or side mission
* **EBiome** – Type of Biome the missions takes place in. The four locations being Shallows, Crystal, Volcanic Waste, and Abyss.
* **EDepthUpgradeRequired** – The depth upgrade required to unlock the mission
* **AMissionBase** – These are active missions in the world created by the MissionLogSubsystem. Tracks and updates objective progress in the game
* **UMissionObjectiveIDComponent** – A custom component that contains Objective IDs has a function called by the owning actor that tells the MissionLogSubsystem to call the IDs for missions to update their matching objectives.
* **WBP\_MissionTrackerPrototype** – A widget on screen that displays the current tracked mission’s objective progress. (Blueprint Prototype)
* **WBP\_MissionObjectiveWIdget** – A widget contained in the mission tracker; this contains the details of a specific objective. (Blueprint Prototype)
* **BP\_MissionTrackerComponentPrototype** – A component attached to BP\_DiverCharacter that simply spawns WBP\_MissionTrackerPrototype. (Blueprint Prototype)
* **UMissionUserWidget** – This is what players interact with to accept/submit missions. *More information for this class will be found in the PC Module Design Document*.

## A screenshot of a computer screen Description automatically generated**4.3 Detailed Class Design of the Mission Module**

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

## UMissionLogSubsystem Functions

### Constructor

A black screen with white text

Description automatically generated

This is just a slightly modified constructor which grabs the Mission Data Table file and stores it as a reference and initializes the unavailable main and side missions.

1. Loads the data table containing all information of all created missions and stores it as a pointer called MissionTable.
2. Create a temporary variable being a TArray of type FName called MissionTableNames and assign it to the return value of MissionTable->GetRowNames().
3. In a ranged based for loop with auto i as the range declaration and MissionTableNames being the range expression…
4. If i is a Main mission. Add i to the AvailableMainQuests array.
5. Else if i is a Side mission. Add i to the AvailableSideQuest array.

### void AddNewMission(FName MissionID)

A screenshot of a computer

Description automatically generatedThis function looks into the available missions then creates an active mission of the matching mission ID that was given if conditions were met.

1. Checks if the mission that this function was given is not currently active or already completed.
2. Spawns an AMissionBase actor and holds its pointer.
3. Add the given MissionID to the CurrentActiveMissions array.
4. Sets the mission actor’s Mission ID to the Mission ID given in the function.
5. Call mission actor’s SetMissionDetails() to initialize the mission.
6. Adds the mission actor’s pointer to the CurrentMissions array.
7. If the mission is a main mission, remove the mission ID from available main missions since its now an active mission.
8. Else if the mission is a side mission, remove the mission ID from available side missions since its now an active mission.

### void CompleteMission(AMissionBase\* CompletedMission)

A screenshot of a computer program

Description automatically generated

This function takes the given mission Actor and sets its Mission ID as completed and then destroying the object as it is no longer tracked thus not being required. Then calls CheckPrequisites() to see if missions need to be reorganized.

1. Create a temporary copy of CompletedMission’s MissionID called CompletedMissionID for readability.
2. Removes the CompletedMissionID from CurrentActiveMissions[].
3. If CompletedMission was a Main mission, CompletedMissionID enters CompletedMissions[].
4. Else if CompletedMission was a Side mission, CompletedMissionID re-enters AvailableSideMissions[].
5. Removes CompletedMission from CurrentMissions[].
6. CompletedMission gets destroyed.
7. The MissionWasComplete delegate gets broadcasted with CompletedMissionID.
8. Calls CheckPrequisites().

### bool QueryActiveMissions(FName MissionID)

(Due to simplicity. No diagram was created.)

This function simply returns true or false depending on if the given MissionID is in the CurrentActiveMissions array.

### void TrackMission(FName MissionID)

(Due to simplicity. No diagram was created.)

This function sets the currently tracked mission to the given mission ID. Then MissionTrackingChange delegate gets broadcasted with that mission ID.

### void NewGameMissionSetup()

A screenshot of a computer

Description automatically generated

This function is meant to get called when a new game has started. Moving all the starting missions into the available array.

1. Create a temporary variable of FMissionDetails pointer called CurrentMissionDetails.
2. In a for loop where i = 0 and while i is less the number of UnavailableMainMissions…
3. Create a temporary variable of type FName called CurrentMissionID and assign it to UnavailableMainMissions’ element location at i,
4. Assign CurrentMissionDetails to MissionTable’s row that is named the same as the CurrentMissionID.
5. If the CurrentMissionDetails’ perquisite missions are empty…
6. Add CurrentMissionID to the AvailableMainMissions array.
7. Remove CurrentMissionID from the UnavailableMainMissions array.
8. Subtract 1 from i to access the correct element in UnavailableMainMissions next loop.
9. Leaving the first loop and entering a new loop where i = 0 and while i is less the number of UnavailableSideMissions…
10. Create a temporary variable of type FName called CurrentMissionID and assign it to UnavailableSodeMissions’ element location at i,
11. Assign CurrentMissionDetails to MissionTable’s row that is named the same as the CurrentMissionID.
12. If CurrentMissionDetails’ DepthUpgradeRequired is Upgrade\_0...
13. Add CurrentMissionID to the AvailableSideMissions array.
14. Remove CurrentMissionID from the UnavailableSideMissions array.
15. Subtract 1 from i to access the correct element in UnavailableSideMissions next loop.

### void LoadMissionsFromSave()

[WIP] In the current version of this Module, save and load is planned to be implemented but is not currently thought about thoroughly.

### void CheckPrequisites()

A screenshot of a computer program

Description automatically generated

This function checks all unavailable missions and their prerequisite missions. If a mission’s prerequisites are in the completed missions array, that mission will move to the available missions array. (Currently does not support Side Missions being checked due to players not having depth upgrades in the current build.)

1. In a for loop where i = 0 and while i less than the number of Unavailable Main Missions…
2. Create a temporary variable of type FName called CurrentMissionID and assign it to UnavailableMainMissions’ element location at i,
3. Create a temporary variable of type FMissionDetails called CurrentMissionDetails and assign it to a reference to MissionTable’s row that is named the same as the CurrentMissionID.
4. Create a temporary variable being a TArray of type FName called missionPrerequisites and assign it to CurrentMissionDetails’ PrequisiteMissions
5. If missionPrerequisites is empty. Continue/Loop as this is a bug that should not happen.
6. Create a temporary variable of type bool called prerequisitesComplete and set it to true.
7. In a ranged based for loop with FName QPID as the range declaration and missionPrequisites being the range expression…
8. If the CompletedMissions array does not contain an FName equal to QPID, set prerequisitesComplete to false and break out of the for loop.
9. After the ranged based for loop, if prerequisitesComplete is true…
10. Add CurrentMissionID to the AvailableMainMissions array.
11. Remove CurrentMissionID from the UnavailableMainMissions array.
12. Subtract 1 from i to access the correct element in UnavailableMainMissions.
13. After the for loop, Call Broadcast() on the PrerquisiteMet delegate for the PC to receive.

### void CallObjectiveID(FString ObjectiveID)

(Due to simplicity. No diagram was created.)

This function calls Broadcast on the ObjectiveIDCalled delegate imputing the given FString. This is for all AMissionBase actors to hear if they need to update the given objective.

### AMissionBase\* GetMissionPtr(FName MissionID)

A screenshot of a computer program

Description automatically generated

This function gets the Mission actor of the given MissionID if the actor exists. If not, return a null pointer.

1. In a ranged based for loop with auto i as the range declaration and CurrentMissions being the range expression…
2. If i’s MissionID is the same as the given MissionID of the function, return i.
3. Outside the loop, return nullptr as no Mission actor was found.

### void ResetArrays()

(Due to simplicity. No diagram was created.)

This function empties all arrays in MissionLogSubsystem. Planned to be used when returning back to the main menu screen. (Currently not in use)

## UMissionBase Functions

### void SetMissionDetails()

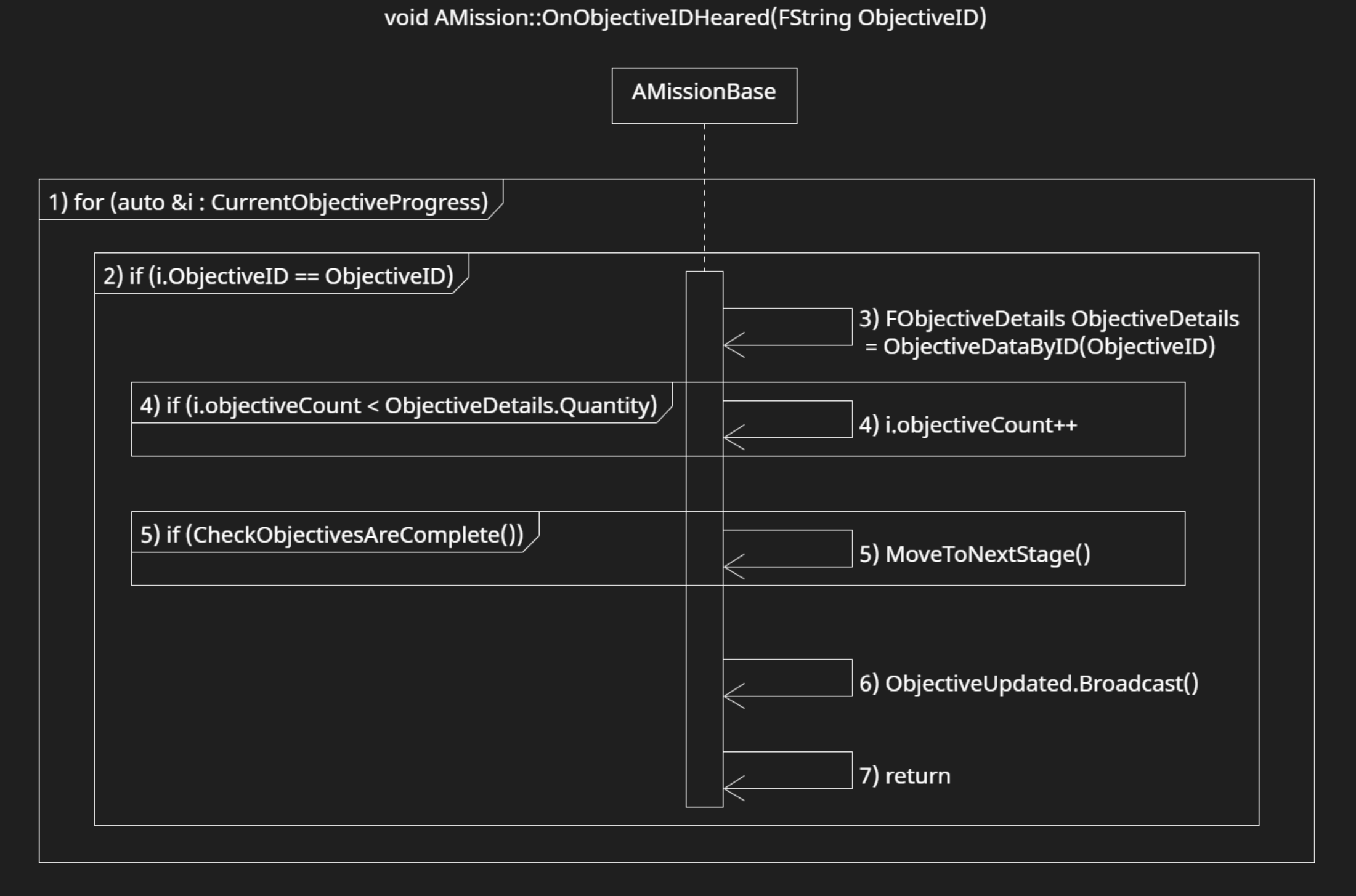
A screenshot of a computer program

Description automatically generated

This function sets the variables for the actor. This is called when the actor has been created by the MissionLogSubsystem or when entering the next stage.

1. Create a temporary variable of type UMissionLogSubsystem\* called MissionLog and assign it to the subsystem pointer through UGameplayStatics.
2. Assign MissionDetails to MissionLog’s MissionTable’s row that is named the same as the MissionID.
3. Assign CurrentStageDetails to MissionDetails’ stage at CurrentStage.
4. Clear CurrentObjectiveProgress. This is if we are going from one stage to the next.
5. Create a temporary variable being a TArray of type FObjectiveDetails called Objectives and assign it to CurrentStageDetails’s Objectives.
6. In a ranged based for loop with auto i as the range declaration and Objectives being the range expression…
7. Create a temporary variable of Type FMissionObjectiveProgress call a and assign its ObjectiveName to i’s ObjectiveName, a’s ObjectiveID to i’s ObjectiveID, and a’s objectiveCount to 0.
8. Add a to the CurrentObjectiveProgress array.

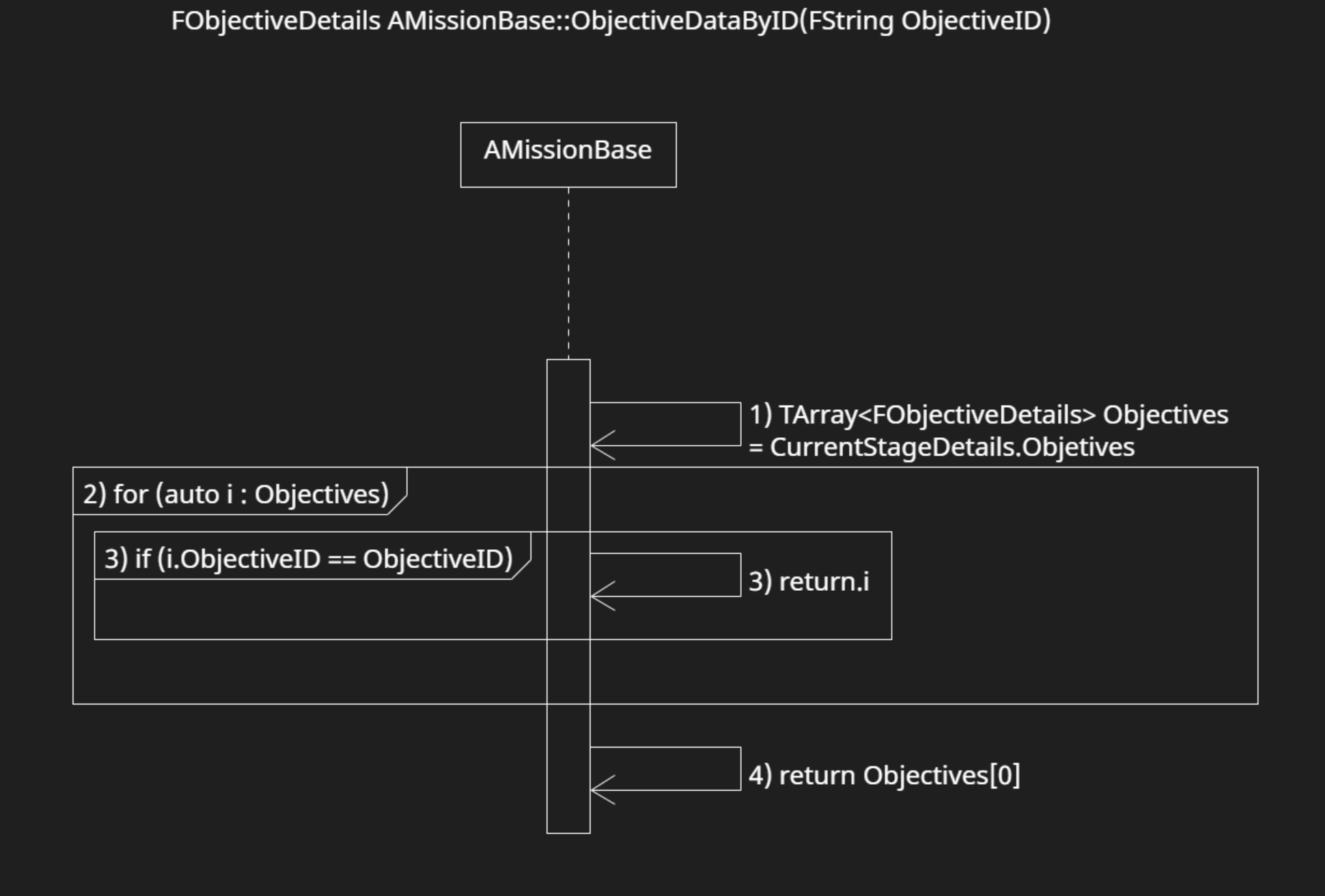
### void OnObjectiveIDHeared(FString ObjectiveID)



This function finds the given ObjectiveID ObjectiveProgress. If it does, then increase that objective’s count by 1. If all objectives are complete, call MoveToNextStage() to get to the next stage if it exists. Then brodcasts the ObjectivesUpdated delegate for mission tracking UI to update. Then returns so the loop does not continue.

1. In a ranged based for loop with auto &i as the range declaration and CurrentObjectiveProgress being the range expression…
2. If i’s ObjectiveID is equal to the given ObjectiveID…
3. Create a temporary variable of type FObjectiveDetails called ObjectiveDetails and assign it the return value of ObjectiveDataByID(ObjectiveID).
4. If i’s objectiveCount is less ObjectiveDetails’ Quantity, increment objective count by 1.
5. If CheckObjectivesAreComeplete() returns true, call MoveToNextStage().
6. Broadcast the ObjectiveUpdated delegate for UI to update.
7. Return to not continue the loop.

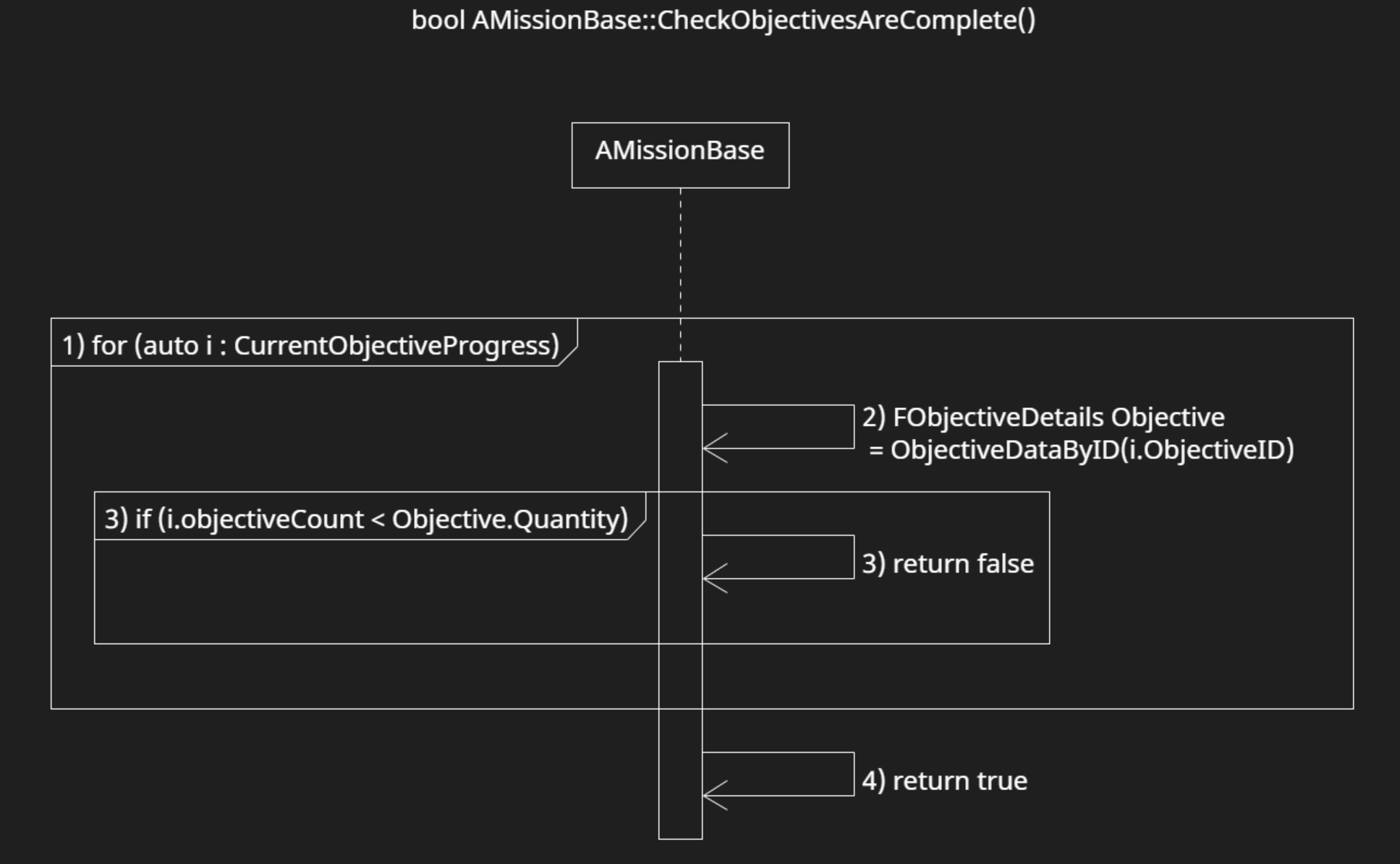
### FObjectiveDetails ObjectiveDataByID(FString ObjectiveID)



This function uses the ObjectiveID that was given and uses that to find the specific ObjectiveDetails containing the matching ObjectiveID in the current stage. If the function does not find the matching ObjectiveDetails, it returns the first ObjectiveDetails of the array which should not happen. That’s an error based on input.

1. Create a temporary variable being a TArray of type FObjectiveDetails called Objectives and assign it to CurrentStageDetails objectives.
2. In a ranged based for loop with auto i as the range declaration and Objectives being the range expression…
3. If i’s ObjectiveID is the same as the given ObjectiveID, return i.
4. This is an error if this occurs. Return the first objective in the Objectives array.

### bool CheckObjectiveAreComplete()



This function checks if all objectives are complete. Returns true if they are, false if they are not.

1. In a ranged based for loop with auto i as the range declaration and Objectives being the range expression…
2. Create a temporary variable of type FObjectiveDetails called Objective and assign it to the return value of ObjectiveDataByID(i.ObjectiveID)
3. If i’s ObjectiveCount is less than Objective’s Quantity, return false.
4. Outside of the for loop, return true.

### void MoveToNextStage()



This function moves the mission actor into the next stage if there is one. If not, set the IsComplete bool to true.

1. If CurrentStage is less than the the number of Stages in MissionDetails -1, Increment CurrentStage by 1 and call SetMissionDetails().
2. Else, set IsComplete to true and Broadcast the MissionWasComplete delegate in the MissionLogSubsystem inputing this mission’s mission ID

### void BeginPlay()



This is a small override to the BeginPlay() function. Calls AddDynamic on the ObjectiveIDCalled delegate in the MissionLogSubsystem. This is so OnObjectiveIDHeard() gets called when the delegate gets broadcasted.

1. Call Super::BeginPlay()
2. Create a temporary variable of type UMissionLogSubsystem pointer called MissionLog and assign it the MissionLog subsystem through UGameplayStatics.
3. Call AddDynamic on the ObjectiveIDCalled delegate and input this actor this actors function pointer, OnObjectiveIDHeard.

## UObjectiveIDComponent Functions

### void BrodcastObjectiveIDs()

A screenshot of a computer program

Description automatically generated

This function calls MissionLog’s CallObjectiveID() function as many time as number of ObjectiveIDs the component contains and using those ObjectiveIDs.

1. Create a temporary vriable of type UMissionLogSubsystem pointer called MissionLog and assign it the MissionLog subsystem through UGameplayStatics.
2. If the ObjectiveIDs array is empty, add a debug message to screen (For Devs) and return. Hitting this means there is an error.
3. In a ranged based for loop with FString ObjectiveID as the range declaration and ObjectiveIDs being the range expression, call CallObjectiveIDs in MissionLog

## WBP\_MissionTrackerPrototype Events

Due to the nature of blueprints and these blueprints being prototypes, functions will be given a brief description.

### Update

A screenshot of a computer screen

Description automatically generated

This event first sets its mission actor reference. Then sets the Objective Updated delegate from the referenced mission actor to call the OnObjectiveUpdated event. After, sets objectives and their count onto the screen. It gets called whenever the mission actor this widget is tracking gets one of its objectives updated, when a mission was completed, or when a different mission is tracked.

### OnObjectiveUpdated

A screenshot of a computer screen

Description automatically generated

This event simply calls the update event whenever an objective from the missions its tracking gets updated.

### MissionTrackingChanged

A screenshot of a computer screen

Description automatically generated

This event gets called when the current tracked mission gets changed. Sets the new mission actor’s pointer to the widgets mission actor reference then calls Update. If the actor pointer is null, it destroys the widget. That should not happen as it is an error.

### MissionWasCompleted

A screenshot of a computer

Description automatically generated

This event gets called when a mission gets completed. If the current mission actor reference is null, its tries to get the first actor in current missions then calls update using the new mission. There are no missions, it sets the text to “No missions being tracked”.

### Event Tick

A screenshot of a computer

Description automatically generated

This event simply forces the widget to be atop all the other widgets.

### Event Construct

A screenshot of a computer

Description automatically generated

A screenshot of a computer screen

Description automatically generatedA screenshot of a computer screen

Description automatically generatedThis event is called upon the widget’s creation. It first binds delegates from Mission Log Subsystem to the MissionTrackingChanged and MissionWasCompleted events in the tracker. Then checks for the mission “T0001” and tries to create it (this is for testing purposes and will not be final), After, it does Update’s event functionality starting at Bind Event to Objective Updated.

## WBP\_MissionObjectiveWidget Events

Due to the nature of blueprints and these blueprints being prototypes, functions will be given a brief description.

### Event Pre Construct

A screenshot of a computer

Description automatically generated A screenshot of a computer

Description automatically generatedThis event first gets the right objective by details by matching the Objective Name with the given ObjectiveProgress. Then sets the text with the objective description with the current objective count and objective quantity.

## BP\_MissionTrackerComponentProtoType Events

Due to the nature of blueprints and these blueprints being prototypes, functions will be given a brief description.

### Event Begin Play

A screenshot of a computer

Description automatically generated

This event simply creates the mission tracker prototype widget and adds it to viewport.

## Intended Interactions

### Accepting Missions

A screenshot of a computer

Description automatically generated

This is for when a player accepts a mission through the PC terminal. (More details on specific functions are in the PC module design document)

1. Player interacts with the PC Terminal by pressing ‘E’ on the PC.
2. In the PC Terminal, the player clicks on the mission user widget app on screen.
3. The missions tab displays available missions from the MissionLogSubsystem
4. The player accepts the mission by clicking accept on the selected mission.
5. This calls AddNewMission() in MissionLogSubsystem.
6. That creates a Mission Actor.

### Updating Objectives in Active Missions

A black screen with white text

Description automatically generated

This is for updating objectives at any point during gameplay.

1. The owning Actor of a UObjectiveIDComponent calls BrodcastMissionIDs() on the component.
2. In a ranged based for loop with FString ObjectiveID as the range declaration and ObjectiveIDs being the range expression…
3. That calls CallObjectiveID() in MissionLogSubsystem
4. That calls Broadcast on the ObjectiveIDCalled delegate.
5. This tells all Mission actors to call OnObjectiveIDHeared() which updates their objectives if the they have a matching ObjectiveID.

### Handing in Missions

A screenshot of a computer

Description automatically generated

This is when a player hands in the Mission through the PC.

1. Player interacts with the PC Terminal by pressing ‘E’ on the PC.
2. In the PC Terminal, the player clicks on the mission user widget app on screen.
3. The missions tab displays in progress missions from the MissionLogSubsystem
4. The player submits the mission by clicking submit on the selected mission.
5. This calls CompleteMission() in MissionLogSubsystem.
6. That destroys the Mission Actor.

# **6 Use Case View**

## Creating a Mission in the data table

This is for creating a mission in DT\_MissionDetails.

### Step 1

A screenshot of a computer

Description automatically generated

Create a new mission by clicking add at the top of the window.

### Step 2

A screenshot of a computer

Description automatically generated

Rename the Row Name which is also the MissionID into something unique. Needs to be unique to work.

## Setting-Up Mission Variables

Each Mission has multiple variables that will be described below.

### Mission Details

A screenshot of a computer

Description automatically generated

* **Mission Name** – The name of the Mission. It is different from MissionID as this is shown to the player while MissionID is mainly for development purposes.
* **Log Description** – The Description of the Mission. Use this to describe the give a quick summary of the mission. Or Whatever you want. You are the designer. Maybe give flavour text if you want. This gets shown in the mission application.
* Tracking Description – Current does nothing at this current version. Likely to be removed in the future. **Do not use**.
* **Type** – This is the type of mission it is. Currently, it is only main or side missions. More could be added into the future.
* **Stage** – This is an array of multiple stages. Each stage has a Stage Name, Description, an array of Objectives, and Money Reward. Stages will be expanded further below.
* Biome – This is to tell the player where this mission takes place. Currently does nothing as of the current build.
* **Depth Upgrade Required** – This if for side missions. This is a requirement that the player need to unlock this side mission. Due to player not having depth upgrades as of the current build, anything above Upgrade\_0 locks the mission away.
* **Prerequisite Missions** – This is for main missions. This is an array of FNames. This is for containing a list of MissionIDs the are prerequisites to unlock this mission. When those missions are completed, this mission will become available to the player. This is **important** as having none will make it available at the start of the game or cause issues if this mission was added later if there is already a save the game is loading from.

### Stage Details

A black and white striped background

Description automatically generated

* Stage Name – The name of the stage. Currently, does not display anywhere so not important to fill as of the current version.
* Description – The description of the stage. Currently, does not display anywhere so not important to fill as of the current version.
* **Objectives** – This is an array of multiple objectives. Each objective has an Objective Name, Description, Type, Objective ID, Quantity, and checks if its optional. This will be expanded further below. Each stage requires at least 1 objective, or no progression can be done.
* Money Reward – Currently does nothing. Meant to reward player with currency on completion. Not important to fill at current version.

### Objective Details

A black and white background

Description automatically generated with medium confidence

* **Objective Name** – The name of the objective. This needs to be unique from other objective names in this mission or the current objective tracker will not properly track.
* Description – The description of the objective. Currently, does not display anywhere so not important to fill as of the current version.
* Type – The type of objective. Currently there are only two types being Location and Picture. This currently does nothing other than organize so not important to fill as of the current version.
* **Objective ID** – This is **VERY IMPORTANT** as this how an objective gets called and tracked. This mission’s Objective ID and Objective ID of the ObjectiveIDComponent (Currently called MissionIDComponent in code and BP) needs to be exact or this objective wont increment.
* **Quantity** – The number of times the objective needs to be done. Needs to be above 0 or it could cause issues to move to the next stage.
* Is Optional – This sets the mission to be optional or not. Currently does not do anything so not important to fill as of the current version.

## Setting-Up MissionObjectiveIDComponent

A screen shot of a computer

Description automatically generated

Each fish will contain a MissionObjectiveIDComponent. Just go to the component and to the Mission Variable tab. Start adding as many IDs as you need. Make sure they exactly the same as the Objective IDs in the mission you wish to increment, or it will not be called.

A screenshot of a computer

Description automatically generated

If you are more savvy with Unreal Engine, you can attach MissionObjectiveIDComponent to any object and make it call Broadcast Mission IDs function for different purposes. The example above takes it so when the player collides with this trigger box, it calls its objective IDs letting mission actors increment any matching objectives.