# **「던전 마스터」 UI 아키텍처 설계 문서**

문서 버전: 2.9

작성일: 2025년 8월 4일

**v2.9 변경 내역:**

* 4.2.6: UIManager와 EventChannel이 연동되는 통합 시나리오 테스트 예시를 추가했습니다.
* 부록 A: DebugEventHistory 코드의 성능 가이드 주석에 구체적인 한계 상황 및 비활성화 옵션을 명시했습니다.
* 부록 B: UIManager\_Addressable\_Full의 XML 주석에 <remarks> 태그를 추가하여 가독성을 극대화했습니다.

**v2.8 변경 내역:**

* 4.2.4: UIManager의 패널 전환 스트레스 테스트 예시를 추가했습니다.
* 4.2.5: EventChannelSO의 다중 발행 부하 테스트 예시를 추가했습니다.
* 부록 A: DebugEventHistory 코드에 디버깅 기능 비활성화 옵션에 대한 가이드 주석을 추가했습니다.
* 부록 B: UIManager\_Addressable\_Full의 private 메서드에도 요약 주석을 추가하여 일관성을 높였습니다.

## **1. 개요**

### **1.1. 문서의 목적**

본 문서는 「던전 마스터」 프로젝트의 UI 시스템을 구축하기 위한 **최종 구현 표준 아키텍처**를 정의하는 것을 목적으로 합니다. 이 설계안은 UI Toolkit의 기능을 최대한 활용하여, 장기적으로 유지보수와 확장이 용이하며, 기획자-프로그래머 간의 협업 효율을 극대화하는 견고한 기반을 마련하는 데 중점을 둡니다.

### **1.2. 핵심 철학**

본 UI 아키텍처는 프로젝트의 핵심 개발 원칙인 \*\*데이터 주도 설계(Data-Driven Design)\*\*와 \*\*느슨한 결합(Loose Coupling)\*\*을 따릅니다. 모든 UI 구성 요소는 독립적으로 작동하고, 데이터의 변화에 반응하며, 다른 시스템에 대한 직접적인 의존성을 최소화하도록 설계됩니다.

## **2. 핵심 아키텍처 원칙**

### **2.1. 아키텍처 패턴: MVVM (Model-View-ViewModel)**

UI 시스템의 근간으로 **MVVM 패턴**을 채택합니다. 이는 UI의 시각적 표현(View)과 데이터 및 로직(Model, ViewModel)을 명확하게 분리하고, UI Toolkit의 데이터 바인딩 기능을 완벽하게 활용하기 위한 최적의 선택입니다.

#### **역할 구성**

* **View (뷰)**
  + **역할:** UI의 구조와 시각적 표현을 담당합니다.
  + **구현:** UXML 파일로 UI 요소의 계층 구조를 정의하고, USS 파일로 스타일을 지정합니다. 씬에서는 UIDocument 컴포넌트가 이들을 렌더링합니다. View의 C# 코드 비하인드는 사용자 입력을 ViewModel으로 전달하고, ViewModel의 데이터를 바인딩하는 최소한의 역할만 수행합니다.
* **Model (모델)**
  + **역할:** 애플리케이션의 핵심 데이터와 비즈니스 로직을 포함합니다. UI와는 완전히 독립적입니다. 「던전 마스터」의 특성을 반영하여 모델을 두 가지 계층으로 구분합니다.
  + **구현:**
    1. **청사진 모델 (Blueprint Model):** UnitData, EquipmentData 등 **ScriptableObject(SO)** 애셋. 이는 몬스터의 기본 스탯, 장비의 고유 옵션 등 **영구적이고 불변하는 원본 데이터**를 정의하는 '청사진' 역할을 합니다.
    2. **런타임 모델 (Runtime Model):** 던전 플레이 중에 생성되는 InGameUnit, PlayerStats와 같은 일반 C# 클래스 인스턴스. 이들은 SO 청사진을 바탕으로 생성되며, 현재 체력, 적용된 버프/디버프 등 **일시적이고 상태가 변하는 런타임 데이터**를 소유합니다.

[⚠️ 중요] ScriptableObject 데이터 보호 원칙

ScriptableObject는 게임의 원본 데이터인 '청사진'입니다. 런타임 중에는 이 데이터를 절대 직접 수정해서는 안 됩니다. 런타임에 SO를 수정하면 플레이 모드를 종료하기 전까지 해당 애셋의 값이 영구적으로 변경되어 예측 불가능한 버그를 유발합니다. 항상 게임 시작 또는 유닛 생성 시점에 SO의 데이터를 일반 클래스 인스턴스(런타임 모델)로 복사하여 사용해야 합니다.

* **ViewModel (뷰모델)**
  + **역할:** View와 Model 사이의 중재자입니다. Model의 데이터를 View가 표시하기 좋은 형태로 가공하고, View의 상태(예: 현재 선택된 아이템)와 View가 수행해야 할 로직(예: 버튼 클릭 시 실행될 명령)을 관리합니다.
  + **구현:** HealthBarViewModel, InventoryViewModel 과 같은 C# 클래스. ViewModel은 런타임 모델(예: InGameUnit)의 데이터를 참조하여 View에 필요한 속성(예: string HealthText = "150/200", float HealthPercentage = 0.75)을 노출합니다.

*<p align="center">그림 1: 「던전 마스터」에 최적화된 MVVM 데이터 흐름</p>*

### **2.2. 시스템 통신: ScriptableObject 기반 이벤트 채널**

시스템 간, 그리고 UI와 다른 시스템 간의 모든 통신은 **ScriptableObject 기반 이벤트 채널(Event Channel)** 패턴을 사용합니다. 이는 정적(static) 이벤트 클래스 방식보다 추적, 디버깅, 확장이 용이하며 프로젝트의 데이터 주도 설계 원칙과 일관성을 유지합니다.

* **구현:**
  + EventChannelSO 라는 이름의 SO 애셋을 생성합니다. 이 애셋은 C#의 Action 델리게이트를 포함합니다.
  + 이벤트를 발생시키는 시스템(Publisher)은 이 SO 애셋의 참조를 갖고 Raise() 메서드를 호출합니다.
  + 이벤트에 반응하는 시스템(Subscriber)은 인스펙터 등을 통해 동일한 SO 애셋의 참조를 할당받고, OnEnable에서 리스너를 등록, OnDisable에서 해제합니다.

### **2.3. 전역 관리자: 제한적 싱글톤 패턴**

게임 전체에 유일하게 존재해야 하는 핵심 관리자는 싱글톤(Singleton) 패턴으로 구현합니다. 단, 싱글톤의 남용은 코드의 결합도를 높이므로 반드시 필요한 경우에만 제한적으로 사용합니다.

* **GameManager:** 게임의 전반적인 상태(예: Playing, Paused, GameOver)를 관리합니다.
* **UIManager:**
  + **핵심 역할:** UI '화면(Screen)' 또는 '패널(Panel)'의 **생명주기(Life Cycle) 관리**에만 집중합니다.
  + **책임:**
    - UI 패널 애셋(UXML)을 로드하고 인스턴스화합니다.
    - 요청에 따라 특정 패널을 보여주거나(Show) 숨깁니다(Hide).
    - UI 패널 스택(Stack)을 관리하여 '뒤로가기'와 같은 내비게이션 기능을 지원합니다.
  + **주의:** UIManager는 개별 UI 패널 내부의 버튼, 텍스트 등 특정 요소의 내용이나 상태를 직접 제어하지 않습니다. 이 책임은 각 패널에 연결된 고유의 ViewModel에 위임됩니다.

### **2.4. 아키텍처 종합: 핵심 데이터 흐름**

위에 기술된 MVVM 패턴, 이벤트 채널, 싱글톤 관리자는 개별적인 기술이 아니라, 하나의 통합된 아키텍처를 구성하는 핵심 요소들입니다. 이 아키텍처의 가장 중요한 데이터 흐름은 다음과 같이 요약할 수 있습니다.

**ScriptableObject 청사진 → 런타임 모델 인스턴스 → ViewModel → View**

이 흐름은 데이터(SO), 상태를 가지는 로직(런타임 모델), 그리고 표현(View)을 명확하게 분리하여, 프로젝트의 확장성과 유지보수성을 극대화하는 기술적인 기반이 됩니다.

## **3. 필수 구현 가이드**

이 장에서는 프로젝트의 모든 UI가 따라야 할 핵심적이고 필수적인 구현 방법을 안내합니다.

### **3.1. 이벤트 채널(Event Channel) 구현: 제네릭 및 디버깅 강화**

제네릭을 활용하여 재사용성을 높이고, 리스너가 없는 이벤트를 호출할 경우 경고 로그를 출력하며, 에디터 전용 이벤트 히스토리 기능을 추가하여 디버깅 편의성을 강화합니다.

**EventChannelSO.cs (제네릭 이벤트 채널)**

using UnityEngine;  
using UnityEngine.Events;  
  
public abstract class EventChannelSO<T> : ScriptableObject  
{  
 public event UnityAction<T> OnEventRaised;  
  
 public void RaiseEvent(T value)  
 {  
#if UNITY\_EDITOR  
 // 에디터에서는 이벤트 호출 기록을 남깁니다.  
 DebugEventHistory.Add(this, value);  
#endif  
  
 if (OnEventRaised != null)  
 {  
 OnEventRaised.Invoke(value);  
 }  
 else  
 {  
 Debug.LogWarning($"Event channel '{this.name}' was raised, but has no listeners.");  
 }  
 }  
}

**Void.cs (파라미터 없는 경우를 위한 구조체)**

public struct Void { }

**VoidEventChannelSO.cs (파라미터 없는 이벤트 채널)**

using UnityEngine;  
  
[CreateAssetMenu(menuName = "Game/Events/Void Event Channel")]  
public class VoidEventChannelSO : EventChannelSO<Void>  
{  
 public void RaiseEvent() => RaiseEvent(new Void());  
}

#### **3.1.1. 디버깅 및 진단: 이벤트 히스토리**

대규모 프로젝트에서는 어떤 순서로 이벤트가 발생했는지 추적하는 것이 디버깅에 큰 도움이 됩니다. 아래는 에디터에서만 동작하는 간단한 이벤트 히스토리 로거 예시입니다.

**DebugEventHistory.cs (에디터 전용)**

#if UNITY\_EDITOR  
using System.Collections.Generic;  
using UnityEngine;  
using System;  
  
public static class DebugEventHistory  
{  
 public static event Action OnHistoryChanged;  
 public static bool IsRecording = true; // 기록 활성화/비활성화 옵션  
  
 // [성능 가이드] 에디터 플레이 중 Update 루프 등에서 수천~수만 단위의 이벤트가 폭증할 경우,  
 // 에디터 성능에 심각한 영향을 줄 수 있습니다. 이런 상황이 예상된다면 이 값을 10~20 정도로 더 줄이거나,  
 // IsRecording 플래그를 false로 설정하여 기록 자체를 비활성화하는 것을 강력히 권장합니다.  
 private static readonly int MAX\_HISTORY = 50;  
 public static readonly List<(ScriptableObject channel, object value)> History = new List<(ScriptableObject, object)>();  
  
 public static void Add(ScriptableObject channel, object value)  
 {  
 if (!IsRecording) return;  
  
 if (History.Count >= MAX\_HISTORY)  
 {  
 History.RemoveAt(0);  
 }  
 History.Add((channel, value));  
 OnHistoryChanged?.Invoke(); // 변경 사항을 구독자에게 알림  
 }  
}  
#endif

#### **3.1.2. 이벤트 히스토리 시각화 (선택적 확장)**

디버깅 효율을 극대화하기 위해, DebugEventHistory의 내용을 보여주는 전용 에디터 윈도우를 구현할 수 있습니다. 이는 필수 구현 사항은 아니지만, 프로젝트 후반부 디버깅 단계에서 매우 유용한 툴이 될 수 있습니다. (구현 예시는 **부록 A** 참조)

### **3.2. UIManager 기본 구현**

이 UIManager는 Addressables 없이도 완전히 동작하는 핵심 구현입니다. 인스펙터에 미리 할당된 UI 패널들의 활성화 상태를 관리합니다.

using System.Collections.Generic;  
using UnityEngine;  
  
public class UIManager : MonoBehaviour  
{  
 public static UIManager Instance { get; private set; }  
  
 // 인스펙터에서 관리할 UI 패널들의 GameObject 참조  
 [SerializeField] private List<UIPanel> uiPanels;  
 private Dictionary<string, UIPanel> panelDictionary;  
 private Stack<UIPanel> panelStack = new Stack<UIPanel>();  
  
 private void Awake()  
 {  
 if (Instance != null && Instance != this) { Destroy(gameObject); return; }  
 Instance = this;  
 DontDestroyOnLoad(gameObject);  
  
 panelDictionary = new Dictionary<string, UIPanel>();  
 foreach (var panel in uiPanels)  
 {  
 panelDictionary[panel.PanelId] = panel;  
 panel.gameObject.SetActive(false);  
 }  
 }  
  
 public void ShowPanel(string panelId)  
 {  
 if (panelDictionary.TryGetValue(panelId, out UIPanel panel))  
 {  
 panel.Show();  
 panelStack.Push(panel);  
 }  
 else  
 {  
 Debug.LogWarning($"UIManager: Panel with ID '{panelId}' not found.");  
 }  
 }  
  
 public void HideCurrentPanel()  
 {  
 if (panelStack.Count > 0)  
 {  
 UIPanel panel = panelStack.Pop();  
 panel.Hide();  
 }  
 }  
}  
  
// 각 UI 패널이 상속받거나 포함할 기본 클래스  
public abstract class UIPanel : MonoBehaviour  
{  
 public string PanelId;  
 public virtual void Show() { gameObject.SetActive(true); }  
 public virtual void Hide() { gameObject.SetActive(false); }  
}

### **3.3. MVVM 데이터 바인딩 및 생명주기 관리 (IDisposable 패턴)**

복잡한 ViewModel이 여러 이벤트를 구독하거나 다른 리소스를 할당할 경우를 대비해, IDisposable 인터페이스를 구현하여 명시적으로 리소스를 해제하는 패턴을 적용합니다.

**HealthViewModel.cs (IDisposable 구현)**

using System;  
using System.ComponentModel;  
using System.Runtime.CompilerServices;  
  
public class HealthViewModel : INotifyPropertyChanged, IDisposable  
{  
 public event PropertyChangedEventHandler PropertyChanged;  
  
 // ViewModel이 구독하는 외부 이벤트들  
 private readonly VoidEventChannelSO \_onPlayerDeathEvent;  
 private readonly VoidEventChannelSO \_onGamePausedEvent;  
  
 private float currentHealth;  
 private float maxHealth;  
  
 public float HealthPercentage => (maxHealth > 0) ? (currentHealth / maxHealth) : 0;  
 public string HealthText => $"{currentHealth:F0} / {maxHealth:F0}";  
  
 public HealthViewModel(VoidEventChannelSO onPlayerDeathEvent, VoidEventChannelSO onGamePausedEvent)  
 {  
 // 생성자에서 필요한 이벤트 채널을 주입받고 구독합니다.  
 \_onPlayerDeathEvent = onPlayerDeathEvent;  
 if (\_onPlayerDeathEvent != null)  
 {  
 \_onPlayerDeathEvent.OnEventRaised += OnPlayerDeath;  
 }  
  
 \_onGamePausedEvent = onGamePausedEvent;  
 if (\_onGamePausedEvent != null)  
 {  
 \_onGamePausedEvent.OnEventRaised += OnGamePaused;  
 }  
 }  
  
 public void UpdateHealth(float newCurrentHealth, float newMaxHealth)  
 {  
 bool changed = false;  
 if (currentHealth != newCurrentHealth) { currentHealth = newCurrentHealth; changed = true; }  
 if (maxHealth != newMaxHealth) { maxHealth = newMaxHealth; changed = true; }  
  
 if (changed)  
 {  
 NotifyPropertyChanged(nameof(HealthPercentage));  
 NotifyPropertyChanged(nameof(HealthText));  
 }  
 }  
  
 private void NotifyPropertyChanged([CallerMemberName] string propertyName = "")  
 {  
 PropertyChanged?.Invoke(this, new PropertyChangedEventArgs(propertyName));  
 }  
  
 private void OnPlayerDeath() { /\* 플레이어 사망 관련 로직 처리 \*/ }  
 private void OnGamePaused() { /\* 게임 일시정지 관련 로직 처리 \*/ }  
  
 public void Dispose()  
 {  
 // PropertyChanged 이벤트를 구독하는 모든 리스너를 제거합니다.  
 PropertyChanged = null;  
  
 // 생성자에서 구독했던 모든 외부 이벤트 리스너를 명시적으로 해제합니다.  
 // 이를 누락하면 ViewModel 객체가 메모리에서 해제되지 않아 메모리 누수가 발생할 수 있습니다.  
 if (\_onPlayerDeathEvent != null)  
 {  
 \_onPlayerDeathEvent.OnEventRaised -= OnPlayerDeath;  
 }  
 if (\_onGamePausedEvent != null)  
 {  
 \_onGamePausedEvent.OnEventRaised -= OnGamePaused;  
 }  
 }  
}

**HealthBarController.cs (ViewModel 생명주기 관리)**

using UnityEngine;  
using UnityEngine.UIElements;  
  
public class HealthBarController : MonoBehaviour  
{  
 private HealthViewModel viewModel;  
 private ProgressBar healthBar;  
 private Label healthLabel;  
  
 // 인스펙터에서 이벤트 채널 할당  
 [SerializeField] private VoidEventChannelSO \_onPlayerDeathEvent;  
 [SerializeField] private VoidEventChannelSO \_onGamePausedEvent;  
  
 void OnEnable()  
 {  
 var root = GetComponent<UIDocument>().rootVisualElement;  
 healthBar = root.Q<ProgressBar>("health-bar");  
 healthLabel = root.Q<Label>("health-label");  
  
 // 의존성 주입: ViewModel 생성 시 필요한 이벤트 채널 전달  
 viewModel = new HealthViewModel(\_onPlayerDeathEvent, \_onGamePausedEvent);  
 viewModel.UpdateHealth(80, 100);   
  
 BindViewModel();  
 UpdateUI();  
 }  
  
 void OnDisable()  
 {  
 // 컨트롤러가 비활성화될 때 ViewModel의 Dispose를 호출하여 모든 리소스를 정리합니다.  
 viewModel?.Dispose();  
 }  
  
 private void BindViewModel()  
 {  
 if (viewModel == null) return;  
 viewModel.PropertyChanged += OnPropertyChanged;  
 }  
  
 private void OnPropertyChanged(object sender, System.ComponentModel.PropertyChangedEventArgs args)  
 {  
 UpdateUI();  
 }  
  
 private void UpdateUI()  
 {  
 if (healthBar != null) healthBar.value = viewModel.HealthPercentage \* 100;  
 if (healthLabel != null) healthLabel.text = viewModel.HealthText;  
 }  
}

### **3.4. UI 네이밍 컨벤션**

일관된 명명 규칙은 디자이너와 프로그래머 간의 협업을 원활하게 하고, 코드의 가독성을 높이며, UQuery 사용 시 오류를 줄여줍니다.

* **UXML 요소 이름 (Name 속성):**
  + **규칙:** [ElementType]-[SpecificName]-[OptionalSuffix]
  + **형식:** 소문자 케밥 케이스 (kebab-case)
  + **예시:**
    - button-start-game
    - label-player-name
    - container-character-stats
    - progress-bar-health
    - list-view-inventory-items
* **USS 클래스 이름 (Class 속성):**
  + **규칙:** BEM(Block\_\_Element--Modifier)과 유사한 접근 방식을 따르거나, 기능 기반으로 그룹화합니다.
  + **형식:** 소문자 케밥 케이스 (kebab-case)
  + **예시:**
    - .button (기본 버튼 스타일)
    - .button--primary (주요 행동 버튼)
    - .button--danger (삭제/취소 버튼)
    - .text-title (큰 제목 텍스트)
    - .text-body (본문 텍스트)
    - .hidden (요소를 숨기기 위한 유틸리티 클래스)

## **4. 고급 주제 및 확장 계획**

이 장에서는 프로젝트의 규모가 커지거나 특정 최적화가 필요할 때 선택적으로 적용할 수 있는 고급 패턴과 확장 전략을 다룹니다.

### **4.1. UIManager 확장: Addressables 기반 동적 로드**

프로젝트의 메모리 사용량을 최적화해야 할 경우, Addressable Asset System을 활용하여 UIManager를 확장할 수 있습니다. 이는 UI 패널을 필요할 때만 동적으로 로드하고, 필요 없을 때 메모리에서 해제하는 방식입니다. 아래는 핵심 로직 예시이며, 전체 구현은 **부록 B**를 참조하십시오.

**UIManager\_Addressable.cs (핵심 로직 예시)**

using System.Collections.Generic;  
using System.Threading.Tasks;  
using UnityEngine;  
using UnityEngine.AddressableAssets;  
using UnityEngine.ResourceManagement.AsyncOperations;  
  
public class UIManager\_Addressable : MonoBehaviour  
{  
 // ... 싱글톤 및 변수 선언 ...  
 [SerializeField] private UIPanelRegistrySO \_panelRegistry;  
 private Dictionary<UIPanelKey, (GameObject instance, AsyncOperationHandle<GameObject> handle)> \_loadedPanels;  
  
 // ... Awake() ...  
  
 public async Task ShowPanel(UIPanelKey key)  
 {  
 if (\_loadedPanels.ContainsKey(key))  
 {  
 \_loadedPanels[key].instance.SetActive(true);  
 return;  
 }  
  
 AssetReference reference = \_panelRegistry.GetPanelReference(key);  
 if (reference == null || !reference.RuntimeKeyIsValid())  
 {  
 Debug.LogError($"Panel with key '{key}' not found or invalid.");  
 return;  
 }  
  
 AsyncOperationHandle<GameObject> handle = Addressables.InstantiateAsync(reference, transform);  
 await handle.Task;  
  
 if (handle.Status == AsyncOperationStatus.Succeeded)  
 {  
 \_loadedPanels[key] = (handle.Result, handle);  
 handle.Result.SetActive(true);  
 }  
 }  
}

#### **4.1.1. Addressable 키 관리 전략**

string 형식의 키를 직접 사용하는 것은 오류 발생 가능성이 높고 유지보수가 어렵습니다. 이를 해결하기 위해 **Enum과 ScriptableObject 기반 레지스트리**를 함께 사용하는 것을 강력히 권장합니다.

**UIPanelKey.cs (Enum 키 정의)**

public enum UIPanelKey  
{  
 None,  
 MainMenu,  
 Settings,  
 Inventory,  
 Shop  
}

**UIPanelRegistrySO.cs (SO 레지스트리 예시)**

using System.Collections.Generic;  
using UnityEngine;  
using UnityEngine.AddressableAssets;  
  
[CreateAssetMenu(fileName = "UIPanelRegistry", menuName = "Game/UI/Panel Registry")]  
public class UIPanelRegistrySO : ScriptableObject  
{  
 [System.Serializable]  
 public class UIPanelEntry  
 {  
 public UIPanelKey PanelKey; // string 대신 Enum 사용  
 public AssetReference PanelReference;  
 }  
  
 public List<UIPanelEntry> Panels;  
 private Dictionary<UIPanelKey, AssetReference> \_panelDictionary;  
  
 public void Initialize()  
 {  
 \_panelDictionary = new Dictionary<UIPanelKey, AssetReference>();  
 foreach (var panel in Panels)  
 {  
 if (!\_panelDictionary.ContainsKey(panel.PanelKey))  
 {  
 \_panelDictionary.Add(panel.PanelKey, panel.PanelReference);  
 }  
 }  
 }  
  
 public AssetReference GetPanelReference(UIPanelKey key)  
 {  
 \_panelDictionary.TryGetValue(key, out AssetReference reference);  
 return reference;  
 }  
}

**사용법:** UIManager는 이 SO 레지스트리를 참조하고, ShowPanel(UIPanelKey.MainMenu)와 같이 타입-세이프(type-safe)한 Enum 키를 사용하여 패널을 로드합니다. 이는 오타를 방지하고 코드 자동 완성을 지원하여 개발 효율을 크게 향상시킵니다.

### **4.2. 단위 테스트 전략**

MVVM 아키텍처의 가장 큰 장점 중 하나는 **테스트 용이성**입니다. ViewModel은 순수 C# 클래스이므로, Unity Test Framework (NUnit 기반)를 사용하여 쉽게 단위 테스트를 작성할 수 있습니다.

#### **4.2.1. ViewModel 로직 및 이벤트 테스트 예시**

**HealthViewModelTests.cs**

using NUnit.Framework;  
using System.Collections.Generic;  
using System.ComponentModel;  
  
public class HealthViewModelTests  
{  
 [Test]  
 public void UpdateHealth\_WhenCalled\_UpdatesPropertiesCorrectly()  
 {  
 // Arrange  
 var viewModel = new HealthViewModel(null, null);  
 // Act  
 viewModel.UpdateHealth(50, 200);  
 // Assert  
 Assert.AreEqual(0.25f, viewModel.HealthPercentage);  
 Assert.AreEqual("50 / 200", viewModel.HealthText);  
 }  
  
 [Test]  
 public void UpdateHealth\_WhenDataChanges\_RaisesPropertyChangedEvents()  
 {  
 // Arrange  
 var viewModel = new HealthViewModel(null, null);  
 viewModel.UpdateHealth(100, 100);  
   
 var receivedEvents = new List<string>();  
 viewModel.PropertyChanged += (sender, args) => { receivedEvents.Add(args.PropertyName); };  
  
 // Act  
 viewModel.UpdateHealth(50, 200);  
  
 // Assert  
 Assert.AreEqual(2, receivedEvents.Count);   
 CollectionAssert.Contains(receivedEvents, nameof(viewModel.HealthPercentage));  
 CollectionAssert.Contains(receivedEvents, nameof(viewModel.HealthText));  
 }  
}

#### **4.2.2. ViewModel 상태 및 컬렉션 테스트 예시**

**InventoryViewModel.cs (가상 인벤토리 ViewModel)**

using System.Collections.Generic;  
using System.Linq;  
  
public class InventoryViewModel  
{  
 public List<string> Items { get; private set; } = new List<string>();  
 public int ItemCount => Items.Count;  
 public bool IsFull => Items.Count >= 10;  
  
 public bool AddItem(string item)  
 {  
 if (IsFull) return false;  
 Items.Add(item);  
 return true;  
 }  
}

**InventoryViewModelTests.cs**

using NUnit.Framework;  
  
public class InventoryViewModelTests  
{  
 [Test]  
 public void AddItem\_WhenInventoryNotFull\_AddsItemAndReturnsTrue()  
 {  
 // Arrange  
 var viewModel = new InventoryViewModel();  
  
 // Act  
 bool result = viewModel.AddItem("Health Potion");  
  
 // Assert  
 Assert.IsTrue(result);  
 Assert.AreEqual(1, viewModel.ItemCount);  
 Assert.Contains("Health Potion", viewModel.Items);  
 }  
  
 [Test]  
 public void AddItem\_WhenInventoryIsFull\_DoesNotAddItemAndReturnsFalse()  
 {  
 // Arrange  
 var viewModel = new InventoryViewModel();  
 for (int i = 0; i < 10; i++) { viewModel.AddItem($"Item {i}"); }  
  
 // Act  
 bool result = viewModel.AddItem("Super Potion");  
  
 // Assert  
 Assert.IsFalse(result);  
 Assert.AreEqual(10, viewModel.ItemCount);  
 Assert.IsFalse(viewModel.Items.Contains("Super Potion"));  
 }  
}

#### **4.2.3. UIManager 및 비동기 로직 테스트 예시 (보강)**

Addressables와 같은 비동기 로직은 Unity Test Framework의 [UnityTest] 어트리뷰트와 IEnumerator를 사용하여 테스트할 수 있습니다.

**UIManager\_AddressableTests.cs (보강된 예시)**

using NUnit.Framework;  
using System.Collections;  
using System.Threading.Tasks;  
using UnityEngine;  
using UnityEngine.TestTools;  
using UnityEngine.AddressableAssets;  
using System.Reflection;  
  
public class UIManager\_AddressableTests  
{  
 private UIManager\_Addressable\_Full uiManager;  
 private GameObject uiManagerGo;  
 private UIPanelRegistrySO registry;  
  
 [SetUp]  
 public void Setup()  
 {  
 uiManagerGo = new GameObject();  
 uiManager = uiManagerGo.AddComponent<UIManager\_Addressable\_Full>();  
 registry = ScriptableObject.CreateInstance<UIPanelRegistrySO>();  
   
 FieldInfo registryField = typeof(UIManager\_Addressable\_Full).GetField("\_panelRegistry", BindingFlags.NonPublic | BindingFlags.Instance);  
 registryField.SetValue(uiManager, registry);  
 }  
  
 [TearDown]  
 public void Teardown()  
 {  
 Object.Destroy(uiManagerGo);  
 Object.Destroy(registry);  
 }  
  
 [UnityTest]  
 public IEnumerator ShowPanel\_WhenKeyIsInvalid\_LogsErrorAndCompletes()  
 {  
 // Arrange (레지스트리에 아무것도 등록하지 않음)  
 LogAssert.ignoreFailingMessages = true; // Addressables 초기화 경고 무시  
  
 // Act  
 Task task = uiManager.ShowPanel(UIPanelKey.MainMenu);  
 while (!task.IsCompleted)  
 {  
 yield return null;  
 }  
  
 // Assert  
 LogAssert.Expect(LogType.Error, "Panel with key 'MainMenu' not found or invalid.");  
 Assert.IsTrue(task.IsCompletedSuccessfully);  
 }  
   
 // 참고: 아래 테스트는 실제 Addressable 에셋과 빌드 구성이 필요하여,  
 // 로컬 테스트보다는 CI/CD 환경에서의 자동화된 통합 테스트에 더 적합합니다.  
 [UnityTest]  
 public IEnumerator FullSequence\_ShowHideUnload\_WorksCorrectly()  
 {  
 // Arrange: 테스트용 Addressable 에셋 참조 설정 (사전 준비 필요)  
 // var panelReference = new AssetReference("Assets/Path/To/TestPanel.prefab");  
 // registry.Panels.Add(new UIPanelRegistrySO.UIPanelEntry { PanelKey = UIPanelKey.MainMenu, PanelReference = panelReference });  
 // registry.Initialize();  
  
 // Act 1: Show  
 Task showTask = uiManager.ShowPanel(UIPanelKey.MainMenu);  
 while (!showTask.IsCompleted) { yield return null; }  
 // Assert 1: 패널이 로드되고 활성화되었는지 확인  
  
 // Act 2: Hide  
 uiManager.HideCurrentPanel();  
 // Assert 2: 패널이 비활성화되었는지 확인  
  
 // Act 3: Unload  
 uiManager.UnloadPanel(UIPanelKey.MainMenu);  
 // Assert 3: 패널이 메모리에서 해제되었는지 확인  
   
 yield return null;  
 }  
}

#### **4.2.4. UIManager 및 패널 스택 테스트 예시 (보강)**

기본 UIManager의 핵심 로직인 패널 스택(Stack) 관리 기능을 테스트하는 예시입니다.

**UIManagerTests.cs (보강된 예시)**

using NUnit.Framework;  
using UnityEngine;  
using System.Collections.Generic;  
using System.Reflection;  
  
public class UIManagerTests  
{  
 private UIManager uiManager;  
 private GameObject uiManagerGo;  
 private FieldInfo panelStackField;  
 private FieldInfo panelDictField;  
 private UIPanel panelA, panelB;  
  
 [SetUp]  
 public void Setup()  
 {  
 uiManagerGo = new GameObject();  
 uiManager = uiManagerGo.AddComponent<UIManager>();  
  
 panelA = new GameObject("PanelA").AddComponent<UIPanel>();  
 panelA.PanelId = "PanelA";  
 panelB = new GameObject("PanelB").AddComponent<UIPanel>();  
 panelB.PanelId = "PanelB";  
  
 // Reflection을 사용하여 private 필드에 테스트 데이터 주입  
 panelStackField = typeof(UIManager).GetField("panelStack", BindingFlags.NonPublic | BindingFlags.Instance);  
 panelDictField = typeof(UIManager).GetField("panelDictionary", BindingFlags.NonPublic | BindingFlags.Instance);  
   
 var panelDict = new Dictionary<string, UIPanel>  
 {  
 { "PanelA", panelA },  
 { "PanelB", panelB }  
 };  
 panelDictField.SetValue(uiManager, panelDict);  
 }  
  
 [TearDown]  
 public void Teardown()  
 {  
 Object.Destroy(uiManagerGo);  
 Object.Destroy(panelA.gameObject);  
 Object.Destroy(panelB.gameObject);  
 }  
  
 [Test]  
 public void ShowAndHide\_PanelStack\_ManagesSequenceCorrectly()  
 {  
 // Arrange  
 var panelStack = (Stack<UIPanel>)panelStackField.GetValue(uiManager);  
   
 // Act & Assert 1: Show A  
 uiManager.ShowPanel("PanelA");  
 Assert.AreEqual(1, panelStack.Count);  
 Assert.AreEqual(panelA, panelStack.Peek());  
 Assert.IsTrue(panelA.gameObject.activeSelf);  
  
 // Act & Assert 2: Show B  
 uiManager.ShowPanel("PanelB");  
 Assert.AreEqual(2, panelStack.Count);  
 Assert.AreEqual(panelB, panelStack.Peek());  
 Assert.IsTrue(panelB.gameObject.activeSelf);  
  
 // Act & Assert 3: Hide B  
 uiManager.HideCurrentPanel();  
 Assert.AreEqual(1, panelStack.Count);  
 Assert.AreEqual(panelA, panelStack.Peek());  
 Assert.IsFalse(panelB.gameObject.activeSelf);  
  
 // Act & Assert 4: Hide A  
 uiManager.HideCurrentPanel();  
 Assert.AreEqual(0, panelStack.Count);  
 Assert.IsFalse(panelA.gameObject.activeSelf);  
 }  
   
 [Test]  
 public void PanelSwitch\_StressTest()  
 {  
 // Arrange  
 var panelStack = (Stack<UIPanel>)panelStackField.GetValue(uiManager);  
 int switchCount = 100;  
  
 // Act & Assert  
 for (int i = 0; i < switchCount; i++)  
 {  
 uiManager.ShowPanel("PanelA");  
 Assert.AreEqual(i + 1, panelStack.Count);  
 uiManager.HideCurrentPanel();  
 Assert.AreEqual(i, panelStack.Count);  
 }  
   
 Assert.AreEqual(0, panelStack.Count);  
 }  
}

#### **4.2.5. 이벤트 채널 발행/구독 테스트 예시 (보강)**

아키텍처의 핵심인 이벤트 채널이 올바르게 동작하는지 검증하는 테스트입니다.

**EventChannelTests.cs**

using NUnit.Framework;  
using UnityEngine;  
using UnityEngine.Events;  
  
public class EventChannelTests  
{  
 private VoidEventChannelSO testEventChannel;  
 private int listener1CallCount;  
 private int listener2CallCount;  
  
 [SetUp]  
 public void Setup()  
 {  
 testEventChannel = ScriptableObject.CreateInstance<VoidEventChannelSO>();  
 listener1CallCount = 0;  
 listener2CallCount = 0;  
 }  
  
 [Test]  
 public void RaiseEvent\_WhenSubscribed\_ListenerIsCalled()  
 {  
 // Arrange  
 UnityAction<Void> listener = (v) => { listener1CallCount++; };  
 testEventChannel.OnEventRaised += listener;  
  
 // Act  
 testEventChannel.RaiseEvent();  
  
 // Assert  
 Assert.AreEqual(1, listener1CallCount);  
  
 // Cleanup  
 testEventChannel.OnEventRaised -= listener;  
 }  
  
 [Test]  
 public void RaiseEvent\_WithMultipleSubscribers\_AllListenersAreCalled()  
 {  
 // Arrange  
 UnityAction<Void> listener1 = (v) => { listener1CallCount++; };  
 UnityAction<Void> listener2 = (v) => { listener2CallCount++; };  
 testEventChannel.OnEventRaised += listener1;  
 testEventChannel.OnEventRaised += listener2;  
  
 // Act  
 testEventChannel.RaiseEvent();  
  
 // Assert  
 Assert.AreEqual(1, listener1CallCount);  
 Assert.AreEqual(1, listener2CallCount);  
  
 // Cleanup  
 testEventChannel.OnEventRaised -= listener1;  
 testEventChannel.OnEventRaised -= listener2;  
 }  
  
 [Test]  
 public void RaiseEvent\_WhenOneUnsubscribes\_OthersStillReceiveEvent()  
 {  
 // Arrange  
 UnityAction<Void> listener1 = (v) => { listener1CallCount++; };  
 UnityAction<Void> listener2 = (v) => { listener2CallCount++; };  
 testEventChannel.OnEventRaised += listener1;  
 testEventChannel.OnEventRaised += listener2;  
  
 // Act 1: Unsubscribe listener1  
 testEventChannel.OnEventRaised -= listener1;  
 testEventChannel.RaiseEvent();  
  
 // Assert 1  
 Assert.AreEqual(0, listener1CallCount);  
 Assert.AreEqual(1, listener2CallCount);  
  
 // Cleanup  
 testEventChannel.OnEventRaised -= listener2;  
 }  
  
 [Test]  
 public void RaiseEvent\_StressTest\_DoesNotThrowException()  
 {  
 // Arrange  
 int raiseCount = 1000;  
 UnityAction<Void> listener = (v) => { listener1CallCount++; };  
 testEventChannel.OnEventRaised += listener;  
  
 // Act & Assert  
 Assert.DoesNotThrow(() =>  
 {  
 for (int i = 0; i < raiseCount; i++)  
 {  
 testEventChannel.RaiseEvent();  
 }  
 });  
   
 Assert.AreEqual(raiseCount, listener1CallCount);  
  
 // Cleanup  
 testEventChannel.OnEventRaised -= listener;  
 }  
}

#### **4.2.6. 통합 시나리오 테스트 예시 (신규)**

시스템들이 서로 연동되는 통합 시나리오를 테스트하여 아키텍처의 견고함을 검증합니다.

**IntegrationTests.cs**

using NUnit.Framework;  
using UnityEngine;  
  
public class IntegrationTests  
{  
 // 가상의 게임 시스템  
 public class ScoreSystem  
 {  
 public int Score { get; private set; }  
 private readonly VoidEventChannelSO \_eventChannel;  
  
 public ScoreSystem(VoidEventChannelSO eventChannel)  
 {  
 \_eventChannel = eventChannel;  
 \_eventChannel.OnEventRaised += OnEnemyDefeated;  
 }  
  
 private void OnEnemyDefeated(Void v) => Score += 10;  
   
 public void Unsubscribe() => \_eventChannel.OnEventRaised -= OnEnemyDefeated;  
 }  
  
 [Test]  
 public void EnemyDefeated\_EventRaised\_ScoreSystemUpdatesScore()  
 {  
 // Arrange  
 var enemyDefeatedEvent = ScriptableObject.CreateInstance<VoidEventChannelSO>();  
 var scoreSystem = new ScoreSystem(enemyDefeatedEvent);  
  
 // Act  
 enemyDefeatedEvent.RaiseEvent();  
  
 // Assert  
 Assert.AreEqual(10, scoreSystem.Score);  
   
 // Act 2  
 enemyDefeatedEvent.RaiseEvent();  
   
 // Assert 2  
 Assert.AreEqual(20, scoreSystem.Score);  
  
 // Cleanup  
 scoreSystem.Unsubscribe();  
 }  
}

## **5. 결론 및 권장 사항**

본 문서에서 정의한 UI 아키텍처는 「던전 마스터」의 복잡한 시스템 요구사항을 지원하고, 장기적인 프로젝트의 건전성을 보장하기 위해 설계되었습니다. 이 아키텍처를 따름으로써 얻을 수 있는 핵심 이점은 다음과 같습니다.

* **모듈성(Modularity):** 각 UI 패널과 시스템은 독립적으로 개발 및 테스트가 가능합니다.
* **유지보수성(Maintainability):** 데이터, 로직, 뷰의 명확한 분리로 인해 코드 수정이 용이하고 부작용이 적습니다.
* **확장성(Scalability):** 새로운 UI 기능이나 시스템을 추가할 때, 기존 코드에 미치는 영향을 최소화하며 유연하게 확장할 수 있습니다.
* **팀 협업 효율 증대:** 기획자와 프로그래머가 각자의 역할에 집중하며 병렬적으로 작업할 수 있는 환경을 제공합니다.

성공적인 UI 시스템 구축을 위해 아래의 핵심 권장 사항을 반드시 준수해야 합니다.

1. **이벤트 채널 표준화:** 모든 시스템 간 통신은 예외 없이 **ScriptableObject 기반 이벤트 채널**을 사용합니다.
2. **MVVM 데이터 흐름 준수:** **'SO 청사진 → 런타임 인스턴스(Model) → ViewModel → View'** 의 데이터 흐름을 엄격히 지켜, 영구 데이터와 임시 데이터를 명확히 분리합니다.
3. **관리자 역할 제한:** **UIManager**의 역할은 UI 패널의 생명주기 관리에 한정하고, 패널 내부의 로직은 각자의 ViewModel이 책임지도록 합니다.

## **6. UI 프로토타이핑 및 입력 시스템 구축**

이 섹션은 추상적인 아키텍처와 실질적인 게임플레이 사이의 간극을 메우는 과정입니다. 최종 아트 에셋 없이 UI의 구조와 흐름을 신속하게 구성하고, 사용자와의 상호작용을 위한 견고한 입력 시스템을 구축하는 데 중점을 둡니다.

### **6.1. 그레이박싱(Greyboxing)을 통한 UI 흐름 설계**

그레이박싱은 최종 디자인이나 아트 에셋이 완성되기 전에, 간단한 형태와 색상을 사용하여 UI 레이아웃과 화면 전환 흐름을 빠르게 설계하고 테스트하는 프로토타이핑 기법입니다. 이를 통해 복잡한 아트 작업에 시간을 들이기 전에 기능과 사용성을 먼저 검증할 수 있습니다.

* **방법론:**
  + UI Builder를 사용하여 VisualElement를 마치 사각형 블록처럼 활용하여 주요 UI 패널(메인 메뉴, 상점, 인벤토리 등)의 레이아웃을 구성합니다.
  + Label과 Button 같은 기본 컨트롤을 배치하여 핵심적인 상호작용 요소를 표현합니다.
  + 아직 구현되지 않은 기능은 버튼을 클릭했을 때 콘솔에 로그를 출력하는 방식으로 임시 처리합니다. (예: Debug.Log("상점 UI 열기 버튼 클릭됨");)
* **목표:**
  + 주요 화면 간의 내비게이션(예: 메인 메뉴 → 설정 → 뒤로가기)이 논리적으로 올바른지 검증합니다.
  + 각 화면에 필요한 정보와 버튼이 적절히 배치되었는지 확인합니다.
  + 팀원(기획자, 아티스트, 프로그래머)이 완성된 아트 없이도 UI의 전체적인 구조와 흐름에 대해 논의하고 피드백을 주고받을 수 있는 기반을 마련합니다.

### **6.2. Unity 입력 시스템(Input System) 연동**

Unity의 새로운 입력 시스템은 키보드, 마우스, 게임패드 등 다양한 입력 장치를 일관된 방식으로 처리할 수 있는 유연한 프레임워크를 제공합니다. "액션(Action)" 기반 접근법을 통해, 특정 키(예: 'ESC' 키)가 아니라 추상적인 행동(예: 'UI 취소')에 로직을 연결하므로 코드의 재사용성과 확장성이 크게 향상됩니다.

#### **6.2.1. 입력 액션 에셋 설정**

1. **패키지 설치:** Package Manager에서 Input System 패키지를 설치하고 활성화합니다.
2. **액션 에셋 생성:** Assets > Create > Input Actions를 통해 입력 액션 에셋을 생성합니다.
3. **액션 맵과 액션 정의:**
   * **UI 액션 맵:** UI와 관련된 모든 입력을 관리하기 위한 UI 액션 맵을 생성합니다.
   * **액션 정의:** UI 맵 안에 Navigate (방향키, WASD, 조이스틱), Submit (Enter, A버튼), Cancel (ESC, B버튼) 등의 액션을 정의하고 각 액션에 실제 키나 버튼을 \*\*바인딩(Binding)\*\*합니다.
4. **C# 클래스 생성:** 액션 에셋의 인스펙터에서 Generate C# Class 옵션을 활성화하여, 코드에서 액션을 쉽게 참조할 수 있는 래퍼 클래스를 생성합니다.

#### **6.2.2. UI와 입력 액션 연결**

생성된 입력 액션을 UI 시스템과 연결하여 실제 동작을 구현합니다. 전역적인 UI 입력을 처리하는 별도의 관리자(예: UIInputManager)를 두는 것이 좋습니다.

**UIInputManager.cs (예시)**

using UnityEngine;  
using UnityEngine.InputSystem;  
  
public class UIInputManager : MonoBehaviour  
{  
 private PlayerInputActions inputActions;  
  
 private void Awake()  
 {  
 inputActions = new PlayerInputActions();  
 }  
  
 private void OnEnable()  
 {  
 inputActions.UI.Enable();  
 inputActions.UI.Cancel.performed += OnCancelPerformed;  
 }  
  
 private void OnDisable()  
 {  
 inputActions.UI.Cancel.performed -= OnCancelPerformed;  
 inputActions.UI.Disable();  
 }  
  
 private void OnCancelPerformed(InputAction.CallbackContext context)  
 {  
 // '취소' 입력이 감지되면 UIManager를 통해 현재 패널을 닫습니다.  
 // 이는 아키텍처 원칙(느슨한 결합)을 지키는 좋은 예시입니다.  
 // UIInputManager는 UIManager의 존재만 알면 되며,  
 // 어떤 패널이 열려있는지와 같은 구체적인 상태는 알 필요가 없습니다.  
 UIManager.Instance.HideCurrentPanel();  
 }  
}

이러한 접근 방식은 입력 처리 로직을 UI의 특정 부분에서 분리하여, 모든 UI 패널에 일관된 '뒤로가기' 기능을 쉽게 적용할 수 있게 해줍니다.

## **부록 (Appendix)**

### **부록 A: 이벤트 히스토리 시각화 에디터 윈도우 예시**

아래 코드는 DebugEventHistory에 기록된 이벤트들을 실시간으로 확인하기 위한 최적화된 에디터 윈도우 예시입니다.

**DebugEventHistory.cs (수정)**

#if UNITY\_EDITOR  
using System.Collections.Generic;  
using UnityEngine;  
using System;  
  
public static class DebugEventHistory  
{  
 public static event Action OnHistoryChanged;  
 public static bool IsRecording = true; // 기록 활성화/비활성화 옵션  
  
 // [성능 가이드] 에디터 플레이 중 Update 루프 등에서 수천~수만 단위의 이벤트가 폭증할 경우,  
 // 에디터 성능에 심각한 영향을 줄 수 있습니다. 이런 상황이 예상된다면 이 값을 10~20 정도로 더 줄이거나,  
 // IsRecording 플래그를 false로 설정하여 기록 자체를 비활성화하는 것을 강력히 권장합니다.  
 private static readonly int MAX\_HISTORY = 50;  
 public static readonly List<(ScriptableObject channel, object value)> History = new List<(ScriptableObject, object)>();  
  
 public static void Add(ScriptableObject channel, object value)  
 {  
 if (!IsRecording) return;  
  
 if (History.Count >= MAX\_HISTORY)  
 {  
 History.RemoveAt(0);  
 }  
 History.Add((channel, value));  
 OnHistoryChanged?.Invoke(); // 변경 사항을 구독자에게 알림  
 }  
}  
#endif

**EventHistoryWindow.cs (최적화된 버전)**

#if UNITY\_EDITOR  
using UnityEditor;  
using UnityEngine;  
using System.Linq;  
  
public class EventHistoryWindow : EditorWindow  
{  
 private Vector2 scrollPosition;  
  
 [MenuItem("Window/Analysis/Event History Viewer")]  
 public static void ShowWindow()  
 {  
 GetWindow<EventHistoryWindow>("Event History");  
 }  
  
 // 창이 활성화될 때 이벤트 구독  
 private void OnEnable()  
 {  
 DebugEventHistory.OnHistoryChanged += Repaint;  
 }  
  
 // 창이 비활성화될 때 이벤트 구독 해제  
 private void OnDisable()  
 {  
 DebugEventHistory.OnHistoryChanged -= Repaint;  
 }  
  
 private void OnGUI()  
 {  
 // 녹화 토글 UI 추가  
 DebugEventHistory.IsRecording = EditorGUILayout.Toggle("Enable Recording", DebugEventHistory.IsRecording);  
  
 if (GUILayout.Button("Clear History"))  
 {  
 DebugEventHistory.History.Clear();  
 Repaint();  
 }  
  
 EditorGUILayout.LabelField($"Event History (Last {DebugEventHistory.History.Count})", EditorStyles.boldLabel);  
 scrollPosition = EditorGUILayout.BeginScrollView(scrollPosition);  
   
 // OnGUI에서는 데이터를 그리기만 하고, Repaint는 이벤트 발생 시에만 호출됨  
 foreach (var (channel, value) in DebugEventHistory.History.AsEnumerable().Reverse())  
 {  
 string channelName = (channel != null) ? channel.name : "N/A";  
 string valueStr = (value != null && !value.GetType().Name.Contains("Void")) ? value.ToString() : "[Void]";  
 EditorGUILayout.LabelField($"[{channelName}]", $"{valueStr}");  
 }  
  
 EditorGUILayout.EndScrollView();  
 }  
}  
#endif

### **부록 B: UIManager\_Addressable 상세 구현 예시**

아래는 Hide, Unload, OnDestroy 로직을 포함한 UIManager\_Addressable의 전체 구현 예시입니다.

**UIManager\_Addressable\_Full.cs**

using System.Collections.Generic;  
using System.Threading.Tasks;  
using UnityEngine;  
using UnityEngine.AddressableAssets;  
using UnityEngine.ResourceManagement.AsyncOperations;  
  
/// <summary>  
/// Addressables를 사용하여 UI 패널을 동적으로 로드하고 관리하는 UIManager입니다.  
/// </summary>  
/// <remarks>  
/// [핵심 안전성] Addressables.ReleaseInstance는 이미 해제된 핸들에 대해 호출해도  
/// 오류 없이 안전하게 무시되므로, 중복 해제에 대한 걱정 없이 사용할 수 있습니다.  
/// </remarks>  
public class UIManager\_Addressable\_Full : MonoBehaviour  
{  
 public static UIManager\_Addressable\_Full Instance { get; private set; }  
  
 [SerializeField] private UIPanelRegistrySO \_panelRegistry;  
 private Dictionary<UIPanelKey, (GameObject instance, AsyncOperationHandle<GameObject> handle)> \_loadedPanels;  
 private Stack<UIPanelKey> \_panelStack;  
  
 /// <summary>  
 /// UIManager와 패널 레지스트리를 초기화합니다.  
 /// </summary>  
 private void Awake()  
 {  
 if (Instance != null && Instance != this) { Destroy(gameObject); return; }  
 Instance = this;  
 DontDestroyOnLoad(gameObject);  
  
 \_panelRegistry.Initialize();  
 \_loadedPanels = new Dictionary<UIPanelKey, (GameObject, AsyncOperationHandle<GameObject>)>();  
 \_panelStack = new Stack<UIPanelKey>();  
 }  
  
 /// <summary>  
 /// 지정된 키에 해당하는 UI 패널을 로드하고 보여줍니다.  
 /// </summary>  
 public async Task ShowPanel(UIPanelKey key)  
 {  
 if (\_loadedPanels.TryGetValue(key, out var loadedPanel))  
 {  
 loadedPanel.instance.SetActive(true);  
 \_panelStack.Push(key);  
 return;  
 }  
  
 AssetReference reference = \_panelRegistry.GetPanelReference(key);  
 if (reference == null || !reference.RuntimeKeyIsValid())  
 {  
 Debug.LogError($"Panel with key '{key}' not found or invalid.");  
 return;  
 }  
  
 AsyncOperationHandle<GameObject> handle = Addressables.InstantiateAsync(reference, transform);  
 GameObject instance = await handle.Task;  
  
 if (handle.Status == AsyncOperationStatus.Succeeded)  
 {  
 \_loadedPanels[key] = (instance, handle);  
 \_panelStack.Push(key);  
 instance.SetActive(true);  
 }  
 else  
 {  
 Debug.LogError($"Failed to load panel with key '{key}'.");  
 }  
 }  
  
 /// <summary>  
 /// 현재 활성화된(스택의 최상단) 패널을 숨깁니다.  
 /// </summary>  
 public void HideCurrentPanel()  
 {  
 if (\_panelStack.Count > 0)  
 {  
 UIPanelKey key = \_panelStack.Pop();  
 if (\_loadedPanels.TryGetValue(key, out var panelInfo))  
 {  
 panelInfo.instance.SetActive(false);  
 }  
 }  
 }  
  
 /// <summary>  
 /// 지정된 키에 해당하는 UI 패널을 메모리에서 해제합니다.  
 /// </summary>  
 public void UnloadPanel(UIPanelKey key)  
 {  
 if (\_loadedPanels.TryGetValue(key, out var panelInfo))  
 {  
 Addressables.ReleaseInstance(panelInfo.handle);  
 \_loadedPanels.Remove(key);  
 }  
 }  
  
 /// <summary>  
 /// 게임 종료 시 로드된 모든 Addressable 인스턴스를 정리합니다.  
 /// </summary>  
 private void OnDestroy()  
 {  
 // 게임 종료 시 로드된 모든 Addressable 인스턴스를 해제합니다.  
 foreach (var panelInfo in \_loadedPanels.Values)  
 {  
 // UnloadPanel을 통해 개별적으로 해제되었을 수 있지만,  
 // ReleaseInstance는 중복 호출을 안전하게 처리합니다.  
 Addressables.ReleaseInstance(panelInfo.handle);  
 }  
 \_loadedPanels.Clear();  
 }  
}