# **UI Toolkit 마스터하기: 구조, 스타일, 로직 (v1.3)**

## **서론: 왜 구조, 스타일, 로직을 분리해야 하는가?**

Unity의 UI Toolkit은 웹 개발의 현대적인 접근 방식에서 영감을 받아, UI의 세 가지 핵심 요소를 명확하게 분리하는 철학을 기반으로 합니다.

* **구조 (UXML):** UI가 어떤 요소들로 이루어져 있고, 어떻게 배치되어 있는지 정의하는 뼈대입니다.
* **스타일 (USS):** 구조에 시각적인 디자인(색상, 폰트, 여백 등)을 입히는 역할입니다.
* **로직 (C#):** 사용자의 입력에 반응하고, 게임 데이터와 상호작용하며 UI를 동적으로 제어하는 두뇌입니다.

이러한 '관심사의 분리(Separation of Concerns)'는 '던전 마스터'와 같이 복잡한 시스템을 가진 프로젝트에서 다음과 같은 결정적인 이점을 제공합니다.

1. **협업 효율 극대화:** UI/UX 디자이너는 프로그래머의 코드 수정 없이 UXML(구조)과 USS(스타일) 파일을 직접 수정하며 디자인을 개선할 수 있습니다.
2. **유지보수 용이성:** 특정 버튼의 색상을 변경하기 위해 C# 코드를 헤맬 필요 없이, 해당 USS 파일만 수정하면 됩니다. 로직과 디자인이 분리되어 있어 코드의 가독성과 유지보수성이 크게 향상됩니다.
3. **재사용성 및 확장성:** 공통된 스타일(예: 버튼 스타일)이나 UI 구조(예: 팝업창 템플릿)를 만들어 프로젝트 전체에서 일관되게 재사용할 수 있습니다.

본 문서는 이 세 가지 요소를 '던전 마스터'의 실제 UI 개발에 어떻게 적용하는지 구체적인 예시와 함께 상세히 설명합니다.

## **4.1. 구조 (UXML): UI의 뼈대 설계하기**

UXML은 HTML과 유사한 마크업 언어로, UI 요소들의 계층 구조를 정의합니다. UI Builder를 사용하면 코드를 직접 작성하지 않고도 시각적으로 이 구조를 설계할 수 있습니다.

### **실전 예제: '베이스 카드 강화' UI 구조 설계**

'던전 마스터' 기획서(4.6.1)에 명시된 '베이스 카드 강화' UI를 UXML로 설계해 보겠습니다. 이 UI는 카드 이미지, 현재 스탯, 강화 버튼 등으로 구성됩니다.

**1. UI Builder로 기본 구조 잡기**

* Window > UI Toolkit > UI Builder를 실행합니다.
* 라이브러리(Library)에서 VisualElement, Image, Label, Button을 뷰포트(Viewport)로 드래그하여 아래와 같은 계층 구조를 만듭니다.
* **핵심:** 각 요소에 C#에서 접근할 수 있도록 인스펙터(Inspector)에서 명확한 **이름(Name)** 을 부여합니다.

**2. UXML 코드 예시**

UI Builder에서 시각적으로 작업하면 아래와 같은 UXML 코드가 생성됩니다.

<!-- UpgradeCardView.uxml -->  
<ui:UXML xmlns:ui="UnityEngine.UIElements">  
 <!-- 전체 패널을 감싸는 컨테이너 -->  
 <ui:VisualElement name="upgrade-panel" class="upgrade-panel">  
   
 <!-- 카드 정보 섹션 -->  
 <ui:VisualElement name="card-info-section" class="card-info-section">  
 <ui:Image name="card-image" class="card-image" />  
 <ui:Label name="card-name-label" class="card-name-label" text="고블린 주술사"/>  
 </ui:VisualElement>  
  
 <!-- 스탯 정보 섹션 -->  
 <ui:VisualElement name="stats-section" class="stats-section">  
 <ui:Label name="attack-stat-label" class="stat-label" text="공격력: 10 (+1)"/>  
 <ui:Label name="defense-stat-label" class="stat-label" text="방어력: 5"/>  
 <ui:Label name="health-stat-label" class="stat-label" text="생명력: 100 (+10)"/>  
 </ui:VisualElement>  
  
 <!-- 강화 버튼 및 재화 정보 -->  
 <ui:VisualElement name="upgrade-action-section" class="upgrade-action-section">  
 <ui:Label name="cost-label" class="cost-label" text="비용: 주화 100개"/>  
 <ui:Button name="upgrade-button" class="upgrade-button" text="강화" />  
 </ui:VisualElement>  
  
 </ui:VisualElement>  
</ui:UXML>

**Tip:** class 속성은 여러 요소에 동일한 스타일을 적용하기 위해 사용되며, 다음 USS 섹션에서 자세히 다룹니다.

## **4.2. 스타일 (USS): UI에 생명 불어넣기**

USS는 CSS와 유사한 스타일시트 언어로, UXML로 만든 구조에 디자인을 적용합니다.

### **핵심 개념: 선택자(Selector)와 속성(Property)**

* **선택자:** 어떤 요소에 스타일을 적용할지 지정합니다.
  + **이름 선택자 (#):** 특정 이름의 단일 요소 (예: #upgrade-button)
  + **클래스 선택자 (.):** 특정 클래스를 가진 모든 요소 (예: .stat-label)
  + **타입 선택자:** 특정 타입의 모든 요소 (예: Button)
* **속성:** 색상, 폰트 크기, 여백, 정렬 등 시각적 스타일을 정의합니다.

### **실전 예제: '베이스 카드 강화' UI 스타일링**

위에서 만든 UXML에 USS를 적용하여 전문적인 UI로 만들어 보겠습니다.

**1. USS 파일 생성 및 연결**

* Assets > Create > UI Toolkit > Style Sheet로 UpgradeCardView.uss 파일을 생성합니다.
* UI Builder의 StyleSheets 패널에서 + 버튼을 눌러 생성된 USS 파일을 추가합니다.

**2. USS 코드 예시**

/\* UpgradeCardView.uss \*/  
  
/\* 전체 패널 스타일 \*/  
.upgrade-panel {  
 width: 300px;  
 background-color: rgb(40, 40, 50);  
 padding: 15px;  
 border-radius: 10px;  
 border-width: 2px;  
 border-color: rgb(100, 100, 120);  
}  
  
/\* 카드 이미지 \*/  
.card-image {  
 width: 100px;  
 height: 150px;  
 background-color: rgb(20, 20, 30);  
 border-radius: 5px;  
}  
  
/\* 카드 이름 레이블 \*/  
.card-name-label {  
 -unity-font-style: bold;  
 font-size: 20px;  
 color: white;  
 margin-top: 10px;  
}  
  
/\* 스탯 레이블 공통 스타일 \*/  
.stat-label {  
 font-size: 16px;  
 color: rgb(200, 200, 200);  
 margin-bottom: 5px;  
}  
  
/\* 강화 버튼 스타일 \*/  
.upgrade-button {  
 width: 100%;  
 height: 40px;  
 font-size: 18px;  
 background-color: rgb(80, 150, 255);  
 border-radius: 5px;  
 -unity-font-style: bold;  
 color: white;  
 transition-property: background-color, scale; /\* 애니메이션 대상 속성 \*/  
 transition-duration: 0.2s; /\* 애니메이션 지속 시간 \*/  
}  
  
/\* 버튼에 마우스를 올렸을 때 (Hover) 상태 \*/  
.upgrade-button:hover {  
 background-color: rgb(120, 180, 255);  
 scale: 1.05 1.05; /\* 살짝 커지는 효과 \*/  
}  
  
/\* 버튼을 클릭했을 때 (Active) 상태 \*/  
.upgrade-button:active {  
 background-color: rgb(60, 120, 220);  
}

### **반응형 레이아웃: Flexbox 활용**

UI Toolkit은 Flexbox 레이아웃 모델을 사용하여 다양한 화면 크기에 유연하게 대응합니다. 부모 VisualElement에 Flexbox 속성을 설정하여 자식 요소들의 정렬, 방향, 크기 조절 방식을 제어할 수 있습니다.

* flex-direction: 자식 요소의 배치 방향 (row: 수평, column: 수직)
* justify-content: 주 축 방향의 정렬 방식 (center, space-between 등)
* align-items: 교차 축 방향의 정렬 방식 (center, flex-start 등)
* flex-grow: 남는 공간을 차지하며 늘어날지 여부 (0: 안 늘어남, 1: 늘어남)

/\* 예시: 카드 정보 섹션을 가로로 배치 \*/  
.card-info-section {  
 flex-direction: row; /\* 자식들을 가로로 배치 \*/  
 align-items: center; /\* 세로 중앙 정렬 \*/  
}  
  
.card-name-label {  
 margin-left: 15px; /\* 이미지와 이름 사이에 여백 추가 \*/  
}

## **4.3. 로직 (C#): UI를 살아 움직이게 만들기**

C# 스크립트는 UI의 두뇌 역할을 합니다. UXML로 정의된 요소에 접근하여 데이터를 표시하고, 사용자 입력(버튼 클릭 등)을 처리하며, 게임의 다른 시스템과 통신합니다.

### **핵심 원칙: 기술 아키텍처와의 연동**

'던전 마스터 기술 아키텍처 제안서'에서 강조한 **이벤트 기반 아키텍처**를 따르는 것이 매우 중요합니다. UI 로직이 게임의 핵심 로직(예: 재화 차감, 스탯 계산)을 직접 수행해서는 안 됩니다.

잘못된 접근:

강화 버튼 클릭 -> UI 스크립트가 직접 플레이어 재화 차감 -> UI 스크립트가 직접 카드 데이터 수정

**올바른 접근 (이벤트 기반):**

1. 강화 버튼 클릭 -> UI 스크립트가 **"카드 강화 요청" 이벤트**를 발행(Broadcast)합니다.
2. CardManager나 PlayerDataManager 같은 핵심 시스템이 이 이벤트를 구독(Listen)하고 있다가, 요청을 받아 처리합니다. (재화 확인, 스탯 강화 등)
3. 핵심 시스템은 처리가 완료되면 **"카드 정보 갱신됨" 이벤트**를 발행합니다.
4. UI 스크립트는 이 "갱신됨" 이벤트를 구독하고 있다가, 새로운 데이터로 화면의 Label들을 업데이트합니다.

이 방식은 UI와 게임 로직을 완벽하게 분리하여, 각 시스템을 독립적으로 개발하고 테스트할 수 있게 해줍니다.

### **실전 예제: '베이스 카드 강화' UI 로직 구현**

using UnityEngine;  
using UnityEngine.UIElements;  
  
public class UpgradeCardViewController : MonoBehaviour  
{  
 // --- UI 요소 참조 ---  
 private VisualElement root;  
 private Image cardImage;  
 private Label cardNameLabel;  
 private Label attackStatLabel;  
 // ... 다른 스탯 레이블들  
 private Button upgradeButton;  
 private Label costLabel;  
  
 // --- 데이터 ---  
 private BaseCardData currentCard;   
   
 // --- 이벤트 채널 (ScriptableObject) ---  
 // 인스펙터에서 할당  
 [SerializeField] private CardEventChannelSO onUpgradeCardRequested;  
 [SerializeField] private CardEventChannelSO onCardDataUpdated;  
  
  
 /// <summary>  
 /// 컴포넌트가 활성화될 때 UI 요소 쿼리 및 이벤트 콜백을 등록합니다.  
 /// </summary>  
 void OnEnable()  
 {  
 // 1. UI 요소 쿼리 및 캐싱  
 root = GetComponent<UIDocument>().rootVisualElement;  
 cardImage = root.Q<Image>("card-image");  
 cardNameLabel = root.Q<Label>("card-name-label");  
 attackStatLabel = root.Q<Label>("attack-stat-label");  
 upgradeButton = root.Q<Button>("upgrade-button");  
 costLabel = root.Q<Label>("cost-label");  
  
 // 2. 이벤트 콜백 등록  
 upgradeButton.clicked += OnUpgradeButtonClicked;  
 onCardDataUpdated.OnEventRaised += UpdateUI; // 게임 시스템으로부터 오는 데이터 갱신 이벤트를 구독  
 }  
  
 /// <summary>  
 /// 컴포넌트가 비활성화될 때 메모리 누수 방지를 위해 이벤트 콜백을 해제합니다.  
 /// </summary>  
 void OnDisable()  
 {  
 if (upgradeButton != null) upgradeButton.clicked -= OnUpgradeButtonClicked;  
 if (onCardDataUpdated != null) onCardDataUpdated.OnEventRaised -= UpdateUI;  
 }  
  
 /// <summary>  
 /// UI에 표시할 카드 데이터를 설정하고 화면을 갱신하는 메서드입니다.  
 /// </summary>  
 public void SetCardData(BaseCardData cardData)  
 {  
 currentCard = cardData;  
 UpdateUI(cardData);  
 }  
  
 /// <summary>  
 /// 카드 데이터에 기반하여 UI를 업데이트하는 메서드입니다.  
 /// </summary>  
 private void UpdateUI(BaseCardData cardData)  
 {  
 // 현재 UI가 보고 있는 카드와 동일한 데이터일 때만 업데이트  
 if (currentCard == null || cardData.ID != currentCard.ID) return;  
   
 // cardImage.sprite = cardData.CardSprite; // 실제 스프라이트 할당  
 cardNameLabel.text = cardData.CardName;  
 attackStatLabel.text = $"공격력: {cardData.Attack} (+{cardData.AttackGrowth})";  
 // ... 다른 스탯들도 업데이트  
 costLabel.text = $"비용: 주화 {cardData.UpgradeCost}개";  
 }  
  
 /// <summary>  
 /// 강화 버튼이 클릭되었을 때 호출될 콜백 메서드입니다.  
 /// </summary>  
 private void OnUpgradeButtonClicked()  
 {  
 if (currentCard == null) return;  
  
 // 4. 게임 시스템에 '강화 요청' 이벤트를 발행합니다.  
 onUpgradeCardRequested.RaiseEvent(currentCard);  
   
 // UI는 여기서 직접 게임 데이터를 수정하지 않습니다.  
 }  
}  
  
// 예시용 데이터 클래스  
public class BaseCardData   
{  
 public string ID;  
 public string CardName;  
 public Sprite CardSprite;  
 public float Attack;  
 public float AttackGrowth;  
 public int UpgradeCost;  
 // ... 기타 데이터  
}

### **UQuery를 이용한 요소 접근**

* root.Q<T>("이름"): 지정된 이름과 타입(Button, Label 등)을 가진 **첫 번째** 요소를 찾습니다. 가장 일반적으로 사용됩니다.
* root.Query<T>("이름", "클래스"): 더 복잡한 조건으로 여러 요소를 찾을 때 사용합니다.
* **성능 Tip:** OnEnable이나 Start에서 한 번만 쿼리하여 변수에 결과를 저장(캐싱)해두고, Update와 같이 매 프레임 호출되는 곳에서 쿼리하는 것을 피해야 합니다.

## **결론: 통합된 워크플로우**

UI Toolkit을 마스터한다는 것은 UXML, USS, C#이 어떻게 상호작용하는지 이해하는 것입니다.

1. **UI Builder**로 **UXML** 구조를 시각적으로 설계하고 요소에 이름을 부여합니다.
2. **USS** 스타일시트를 작성하여 클래스와 이름을 기준으로 UI에 일관된 디자인과 반응형 레이아웃을 적용합니다.
3. **C# 스크립트**에서 UQuery로 필요한 요소들을 찾아 캐싱하고, 이벤트 콜백을 등록합니다.
4. 사용자 입력이 발생하면, C# 로직은 게임의 핵심 시스템에 **이벤트**를 보내 작업을 요청합니다.
5. 핵심 시스템이 데이터를 변경하고 **응답 이벤트**를 보내면, C# 로직은 이 이벤트를 받아 UI를 최신 상태로 업데이트합니다.

이 워크플로우를 따르면 '던전 마스터'의 복잡하고 다양한 UI들을 체계적이고 효율적으로 개발할 수 있으며, 장기적으로 유지보수와 확장이 용이한 고품질 프로젝트를 완성할 수 있을 것입니다.

## **부록: 실무 효율을 위한 고급 팁 및 트러블슈팅**

### **1. 핵심 워크플로우 다이어그램**

팀원 온보딩 및 개념 이해를 돕기 위한 기본 워크플로우입니다.

UI Builder에서 시각적 설계 → UXML/USS 파일로 저장 → C# 스크립트에서 UQuery로 요소 참조 → 이벤트 콜백 등록 및 로직 연동

### **2. UXML/USS 확장성 가이드**

* **UXML 템플릿 재사용:** TemplateContainer 요소를 사용하면 다른 UXML 파일을 현재 UXML 파일 내에 삽입할 수 있습니다. 공통 '팝업창 헤더'나 '아이템 슬롯'을 별도 UXML로 만들어 여러 UI에서 재사용하면 일관성 유지와 중복 작업 방지에 효과적입니다.
* **전역 테마 및 다크 모드:** base-style.uss에 기본 구조와 레이아웃을, dark-theme.uss, light-theme.uss에 테마 관련 스타일만 정의합니다. 아래 코드로 테마를 동적으로 전환할 수 있습니다.  
  // 인스펙터에서 Dark/Light 테마 USS 에셋을 할당  
  [SerializeField] private StyleSheet darkTheme;  
  [SerializeField] private StyleSheet lightTheme;  
    
  public void SetTheme(bool isDark)  
  {  
   var root = GetComponent<UIDocument>().rootVisualElement;  
   root.styleSheets.Remove(isDark ? lightTheme : darkTheme);  
   root.styleSheets.Add(isDark ? darkTheme : lightTheme);  
  }  
  **⚠️ 주의:** 런타임에서 스타일시트를 동적으로 변경할 경우, 일부 복잡한 UI에서는 변경 사항이 즉시 반영되지 않을 수 있습니다. 이 경우 UI Document 컴포넌트를 비활성화했다가 다시 활성화하면 강제로 UI를 다시 그리게 할 수 있습니다.

### **3. EventChannelSO 패턴 심화**

* **기본 발행/구독 패턴:** UI가 이벤트를 발행하고, 다른 시스템이 수신하는 가장 기본적인 형태입니다. (본문 4.3절 코드 참조)
* **Addressables와 연동한 동적 UI 로딩:** Addressables를 사용하면 필요할 때만 UI 에셋을 비동기적으로 로드하여 메모리를 효율적으로 관리할 수 있습니다.  
  // 인스펙터에서 로드할 UXML 에셋의 어드레서블 주소와 이벤트 채널을 할당  
  [SerializeField] private string upgradePanelAddress;  
  [SerializeField] private CardEventChannelSO onShowUpgradePanel;  
  private VisualElement root;  
    
  void OnEnable()  
  {  
   root = GetComponent<UIDocument>().rootVisualElement;  
   onShowUpgradePanel.OnEventRaised += ShowUpgradePanel;  
  }  
    
  private async void ShowUpgradePanel(BaseCardData cardToShow)  
  {  
   // 1. Addressables를 통해 UXML(VisualTreeAsset)을 비동기 로드  
   var handle = Addressables.LoadAssetAsync<VisualTreeAsset>(upgradePanelAddress);  
   await handle.Task;  
    
   if (handle.Status == AsyncOperationStatus.Succeeded)  
   {  
   // 2. 로드된 UXML을 인스턴스화하여 UI 트리에 추가  
   VisualElement panelInstance = handle.Result.Instantiate();  
   root.Add(panelInstance);  
    
   // 3. 인스턴스화된 패널의 컨트롤러에 데이터 전달 및 초기화  
   panelInstance.Q<UpgradeCardViewController>().SetCardData(cardToShow);  
   }  
  }

### **4. 성능 최적화 및 디버깅**

* **자주 묻는 문제 해결 (Troubleshooting):**
  + **Q: Play Mode에서 USS 스타일이 적용되지 않아요.**
    - **A:** UI Document 컴포넌트의 Panel Settings 에셋이 할당되었는지, 해당 에셋에 스타일시트가 연결되었는지 확인하세요.
  + **Q: 특정 요소에만 스타일을 주고 싶은데, 다른 요소까지 전부 바뀌어요.**
    - **A:** 너무 광범위한 선택자(예: Button 타입 선택자) 대신, 더 구체적인 클래스(.my-button)나 이름(#unique-button) 선택자를 사용하세요.
* **성능 최적화 심화:**
  + **UI 프로파일러 활용:** Window > Analysis > Profiler를 열고 UI Details 모듈을 확인하세요. Geometry Rebuild 항목에서 불필요한 스파이크가 발생한다면, 레이아웃을 자주 변경하는 요소가 원인일 수 있습니다.
  + **동적 UI 갱신 비용 최소화:** 목록 아이템처럼 요소를 동적으로 추가/제거하는 대신, 미리 생성해두고 display 스타일을 Flex(보이기)와 None(숨기기)으로 전환하는 것이 훨씬 빠릅니다. 이는 UI 계층 구조 변경으로 인한 레이아웃 재계산을 방지합니다.