ChatGPT

moddable-study 專案單元測試導入指南

本指南將逐步說明如何為 GitHub 專案 `angus945/moddable-study`（Unity 專案，具備類 RimWorld 的資料驅動模組化架構）導入 **Unit Test**，並提供完整的測試規劃。重點將放在 **Definition** 資料解析模組的現有程式碼回填測試，優先涵蓋定義檔解析、定義註冊與繼承驗證流程，最終拓展至整體模組化設計的測試。內容包括：

1. **測試框架選擇與整合方式** – 使用 Unity Test Framework 的策略與配置流程
2. **Definition 模組的測試回填步驟** – 拆解解析、註冊、驗證（繼承）流程並撰寫對應單元測試
3. **依賴管理與重構建議** – 針對緊密耦合部分提供介面抽象、移除靜態依賴與依賴注入建議
4. **測試資料處理策略** – 運用 stub/mock/fake 模擬外部資源（如定義檔 JSON/XML、模組目錄等）的方案
5. **範例測試案例** – 以一個具代表性的 Definition Loader 或解析器為例，提供可執行的單元測試樣板
6. **測試目錄與 Assembly Definition 設計** – 規劃測試檔案的目錄結構，區分 Editor/PlayMode 測試，並設定對應的 Assembly Definition Files (asmdef)

請按照以下步驟進行，確保測試導入過程清晰可循，讓 Unity 開發者能無縫套用至本專案架構。

測試框架選擇與整合方式

選擇測試框架： 我們將採用 Unity 官方提供的 **Unity Test Framework (UTF)** 作為單元測試框架。UTF 與 NUnit 測試框架整合 [1] 並內建於 Unity 編輯器的 Test Runner 中，不需額外引入第三方框架。UTF 支援 **Edit Mode（編輯器模式）** 和 **Play Mode（運行模式）** 測試，可覆蓋編輯器擴充功能與遊戲執行時兩方面的代碼 [2] [3]。

安裝與啟用： 在 Unity Package Manager 中確認已安裝 **Unity Test Framework** 套件（一般在 Unity 2019.2+ 預設已包含）。若尚未安裝，搜尋並安裝 "Unity Test Framework"。安裝後，於 **Window > General > Test Runner** 開啟測試執行視窗。

測試 Assembly 組態： 根據 Unity 規範，需將 Edit Mode 與 Play Mode 測試分置不同的測試 Assembly 中 [1]。建議使用 Test Runner 提供的快捷創建功能：

- 在 **Test Runner** 視窗中選擇 **EditMode** 分頁，點擊 **Create EditMode Test Assembly Folder**。這將於專案的 Assets 中建立一個 **Tests** 資料夾及相應的 `.asmdef` 檔，預設命名如 Tests.asmdef [4]。該 asmdef 檔已自動包含必要的參考：
- **References** 列表包含 `nunit.framework.dll`（NUnit 測試框架） [5]。
- **Assembly Definition References** 列表包含 `UnityEngine.TestRunner` 及 `UnityEditor.TestRunner`（提供 Unity 測試運行所需 API） [5]。
- **Include Platforms** 預設只勾選 **Editor**，表示此測試 Assembly 僅在編輯器下執行（Edit Mode 測試） [6]。
- 同理，在 **PlayMode** 分頁點擊 **Create PlayMode Test Assembly Folder**，建立對應的 PlayMode 測試資料夾與 asmdef。PlayMode 測試 Assembly 通常將 **Include Platforms** 設為 **Any Platform**（或特定執行平台），不僅限於 Editor [7]。**注意：** PlayMode 測試 asmdef 應該引用 `UnityEngine.TestRunner`（不需要 `UnityEditor.TestRunner`）且讓測試程式碼能引用專案的 runtime 程式集。

創建好測試 Assembly Definition 後，請檢查並調整以下配置：

- **引用被測專案程式集：** 在測試 asmdef 的 **Assembly Definition References** 中，加入本專案中待測程式碼的 Assembly。例如，如果專案的核心程式碼已定義在 `ModInfrastructure.Core` 或 `Implement.Core` 等 asmdef，將它們加入引用。這確保測試程式可存取被測類別與方法 [8] [9]。根據檔案顯示，專案至少有 `Implement.Core.asmdef` [10] 以及對應的 `ModInfrastructure.Core`（可能以 GUID 參考）組件，需一併加至測試 Assembly 的引用設定。

- **平台與腳本後端：** 對 EditMode 測試的 asmdef，保持 **Include Platforms** 僅為 **Editor** 即可，確保其中的測試不會包含在最終玩家構建中 [5]。對 PlayMode 測試 asmdef，可將 **Include Platforms** 設為 **Any Platform**（或選擇所需平台）。Unity 預設透過區隔平台設定來區分 EditMode/PlayMode 測試 [6]。

- **測試工具引用：** 如需使用 Unity 特定的測試輔助 API（例如 `UnityEngine.TestTools` 提供的 `LogAssert` 等），請確認在 **Assembly Definition References** 中包含 `UnityEngine.TestRunner`（對應 runtime 測試功能）以及 `UnityEditor.TestRunner`（僅 EditMode 測試可用） [5] [11]。透過這些引用，可使用 Unity 提供的各種斷言輔助，例如攔截日誌輸出等。

完成上述設定後，專案結構將如：

```
Assets/
  Scripts/              <-- 原專案程式
    ModInfrastructure.Core/ ...
    Implement.Core/ ...
  StreamingAssets/Mods/ <-- 定義檔與模組檔案
  Tests/                <-- 測試程式碼
    Editor/            <-- 編輯器模式測試
      moddableStudy.EditorTests.asmdef
      DefinitionTests.cs (等測試腳本)
    PlayMode/          <-- 運行模式測試（如需要）
      moddableStudy.PlayModeTests.asmdef
     ...
```

註： 一般而言，Definition 資料解析邏輯屬於純粹的資料處理，不依賴 UnityEngine 物件，適合編寫 **Edit Mode 單元測試**。除非有需要測試實際運行時的行為（如載入 AssetBundle、ScriptableObject 等），否則大部分 Definition 模組的測試可放在 EditMode 下執行，加速測試執行效率。

Definition 模組的測試回填步驟

模組概述： Definition 模組負責載入和解析遊戲定義資料，包括：從各個 mod 資料夾讀取定義檔（XML/JSON 等）、合併與反序列化為 C# 物件、套用繼承與覆蓋規則，最後註冊到全域資料庫以供遊戲檢索使用。依據現有程式碼，Definition 模組由多個組件組成，例如：

- `ModDefinitionLoader`：負責讀取模組檔案中的定義檔 (例如 Characters.xml, Things.xml) 並合併到一個統一的 XML 文件 [12] [13]。它會處理重複定義的覆寫 [14]。
- `ModDefinitionPatcher`：（若存在）應用定義的修改/移除 Patch。

- `ModDefinitionDeserializer`：將最終的 XML 定義文件反序列化為對應的 Definition 類別實例（如 `CharacterDef`, `ThingDef` 等）列表。
- `ModDefinitionInheritor`：處理定義之間的繼承關係，解析父子定義並產出繼承後的最終定義 [15] [16]。
- `DefinitionDatabase`：全域靜態資料庫，存放處理完的定義，以類型和 defID 索引供查詢 [17] [18]。

針對上述流程，我們將拆解「**解析→註冊→驗證**」三個階段，逐步回填測試。

測試解析流程

目標： 確認 **定義檔解析** 的相關功能正確運作，包括從檔案讀取合併、解析 XML 節點為物件等。這部分涵蓋 `ModDefinitionLoader` 和 `ModDefinitionDeserializer` 的測試。

1. **準備測試資料：** 建立最小的定義檔內容，以模擬遊戲中的定義 XML。例如，可使用專案 **Core 模組** 提供的測試檔案片段作為範本。以角色定義為例，我們準備一小段 XML 字串，其中包含一個抽象父定義和一個具體子定義：

```
<Defs>
  <CharacterDef isAbstract="true">
    <defID>BaseWarrior</defID>
    <health>150</health>
    <speed>4</speed>
  </CharacterDef>
  <CharacterDef inheritsFrom="BaseWarrior">
    <defID>Knight</defID>
    <health>0</health> <!-- 將透過繼承取得父類的生命值 -->
    <speed>3</speed>  <!-- 覆寫父類速度 -->
  </CharacterDef>
</Defs>
```

這段 XML 模擬了一個抽象的 `BaseWarrior` 和繼承自它的 `Knight`。`Knight` 的 `health` 設為 0，代表應由父類繼承得到值，而 `speed` 則自行定義覆蓋。

注意： 在實際專案的 Characters.xml 中，`Knight` 繼承 `BaseWarrior`，預期繼承後的健康值為 150（父類值） [19] [20]。我們的測試資料即反映這種情形。

1. **測試 ModDefinitionLoader 合併邏輯：** 如果要單獨測試 `ModDefinitionLoader` 的檔案讀取與 XML 合併，需模擬檔案系統環境。可以採取兩種方式：

2. **方法A：建構假檔案系統 (fake filesystem)**：在單元測試中動態建立臨時檔案。例如利用 `System.IO.File.WriteAllText` 將上述 XML 內容寫入臨時路徑，然後呼叫 `ModDefinitionLoader.LoadDefinition(tempFilePath, xdoc)` 方法。測試呼叫後，檢查傳入的 `XDocument xdoc` 是否正確合併了定義節點，以及返回值是否為 true。特別地，若測試包含兩個內容不同但具有相同 defID 的檔案，可驗證較後載入的檔案覆蓋了較早的定義 [14]。例如：

```
// Arrange
var loader = new ModDefinitionLoader();
XDocument mergeDoc = XDocument.Parse("<Defs></Defs>"); // 初始空的 Defs 根節點
```

```
File.WriteAllText("Defs1.xml", xmlContent1);
File.WriteAllText("Defs2.xml",
xmlContent2); // xmlContent2 中含有與 xmlContent1 相同 defID 的定義

// Act
loader.LoadDefinitions(new[]{ "Defs1.xml", "Defs2.xml" }, mergeDoc);

// Assert
var allDefIDs = mergeDoc.Root.Elements().Select(e => e.Element("defID")?.Value).ToList();
Assert.IsFalse(allDefIDs.Contains(duplicateDefIdFromContent1), "舊的重複定義未正確被覆蓋");
```

如上所示，`LoadDefinitions` 會依序讀取檔案並合併節點，其中透過 `RemoveExisting` 方法確保重複的 `<defID>` 僅保留後載入的定義 [14]。測試應驗證合併結果沒有重複的定義，且日誌正確記錄了 override 行為（可配合 `LogAssert` 驗證有無 `"Overriding Definition: ..."` 警告）[21]。

3. **方法B：改造 Loader 以接受字串輸入**：若不希望在測試中真的產生檔案，可考慮對 `ModDefinitionLoader` 進行小幅重構，例如添加一個可以直接接受 XML 字串或 `XDocument` 的載入方法。在生產代碼中，該方法仍可呼叫原有從檔案讀取的邏輯。在測試中則使用字串直接解析，避免檔案 I/O。例如新增一個 `LoadDefinitionFromString(string xml, XDocument mergeDoc)` 供測試呼叫。這樣可輕鬆傳入 `xmlContent1` 等字串進行解析，結果應等價於從檔案載入。

如果修改原始碼不可行，方法A 利用臨時檔案亦是可接受的權衡，因為定義檔通常不大，I/O 開銷有限。

1. **測試 ModDefinitionDeserializer 反序列化：** `ModDefinitionDeserializer` 在 `LoadModsDefinition()` 時被用來將合併後的 XML 文件實例化為 C# 物件 [22]。具體流程是先呼叫 `RegisterDeserializers()` 註冊所有 `IDefinitionDeserializer` [22]，然後 `InstanceDefinitions()` 遍歷 XML 中的節點，利用對應的 Deserializer 將其轉換為 `Definition` 子類實例。

我們可以對這部份進行單元測試： - **Deserializer 映射測試：** 確認每種 `<XyzDef>` 節點都有相應的反序列化器被註冊。例如專案定義了 `CharacterDefDeserializer` 來處理 `<CharacterDef>` [23]、`ThingDefDeserializer` 處理 `<ThingDef>` 等。呼叫 `deserializer.RegisterDeserializers()` 後，對私有映射表（如果可存取）進行檢查，或間接測試：給定特定節點名稱，`InstanceDefinitions` 能正確生成對應類型。比如構造一個只含 `<CharacterDef>` 節點的 `XDocument`，執行 `InstanceDefinitions`，驗證結果字典中含有鍵 `typeof(CharacterDef)` 且其列表數量正確為1。 - **欄位解析測試：** 測試單個 `DefinitionDeserializerBase<T>` 子類對 XML 元素的解析是否準確。以 `CharacterDefDeserializer` 為例：它應該讀取節點內的 `<health>` 和 `<speed>` 文字並轉成整數賦值給 `CharacterDef` 實例 [24]。我們可以直接實例化 `CharacterDefDeserializer` 來測試其 `Deserialize(XElement)` 方法：

```
// Arrange
XElement knightElement = XElement.Parse(@"
 <CharacterDef inheritsFrom='BaseWarrior'>
   <defID>Knight</defID>
   <label>騎士</label>
```

```
    <description>重裝戰士</description>
    <health>0</health>
    <speed>3</speed>
 </CharacterDef>");
var deser = new CharacterDefDeserializer();
// Act
Definition defObj = deser.Deserialize(knightElement);
var knightDef = defObj as CharacterDef;
// Assert
Assert.IsNotNull(knightDef);
Assert.AreEqual("Knight", knightDef.defID);
Assert.AreEqual(0, knightDef.health);
Assert.AreEqual(3, knightDef.speed);
Assert.AreEqual("BaseWarrior", knightDef.inheritsFrom);
Assert.IsFalse(knightDef.IsAbstract);
```

此測試直接檢驗：反序列化後的 `CharacterDef` 對象，其各屬性與 XML 對應欄位匹配正確，包括繼承字串 `inheritsFrom` 和 `IsAbstract` 標記（預期為預設 `false` 未被設定時） [25] 。同理，可為 `ThingDefDeserializer` 等編寫類似測試，確保數值、字串、巢狀物件等欄位皆能正確解析。

- **批次實例化測試：** 使用前面合併好的整份 `XDocument` （包含多個定義節點），測試 `InstanceDefinitions` 的整體行為。例如我們將 Characters.xml 和 Things.xml 合併的 XDocument 傳入，執行 `var defDict = deserializer.InstanceDefinitions(xdoc)`, 然後檢查：
  - `defDict` 的 Key 包含 `typeof(CharacterDef)` 和 `typeof(ThingDef)` 等類型。
  - 各 Key 對應的 List 長度正確（例如核心模組中 CharacterDef 應有4個非抽象定義：Knight, Wizard, Archer, Peasant [26] ；ThingDef 應有5個非抽象定義 [27] ，在繼承前的原始反序列化階段可能仍包含抽象定義）。
  - 可以進一步驗證某幾個重點定義物件的屬性是否與原始 XML 對應：例如 `KnightDef.label == "騎士"`, `BaseWarriorDef.IsAbstract == true` 等。

若 `InstanceDefinitions` 直接傳回的是 **包含抽象定義** 的列表（因為繼承處理尚未過濾），需注意繼承處理在下一步才排除抽象項。但我們仍可檢視抽象定義有無被正確建立，以及繼承字串皆已設定妥當。

總之，解析流程的測試重點在於**資料正確轉換**：XML -> 物件。測試應覆蓋常見情形（完整欄位與缺省欄位的處理）、錯誤情形（例如 defID 缺失是否記錄錯誤）等。

測試定義註冊流程

目標： 確認 **定義註冊** 到資料庫的功能，以及全域存取是否正常。這涉及 `DefinitionDatabase` 的使用，以及 `ModManager.LoadModsDefinition()` 最後階段 [28] [29] 將定義放入資料庫的邏輯。

1. **隔離 DefinitionDatabase 測試：** 由於 `DefinitionDatabase` 是一個靜態類別，內部維護 `Dictionary<Type, List<Definition>>` 作為存儲 [17] 。首先，為避免靜態狀態干擾不同測試，用例應在 **測試初始化 (SetUp)** 階段或每次測試開始時呼叫 `DefinitionDatabase.Clear()` [30] 清空資料庫。幸運的是，專案已提供 `Clear()` 方法重置內部狀態，務必在測試中善加利用。

2. **測試 SetDefinitions 正確性：** `DefinitionDatabase.SetDefinitions(Dictionary<Type, List<Definition>> defs)` 用於批量添加定義 [31] 。可模擬前一步的反序列化結果，建立一個字

典，鍵為某 Definition 基類型，值為一組該類型的定義物件清單。傳入後，驗證資料庫內部儲存是否符合預期：

3. 調用後，透過 `DefinitionDatabase.GetDefinitions()` 抓取全域字典副本 [32] 。確認每個鍵對應的清單包含我們加入的所有定義，沒有遺漏或重複。
4. 如果 `SetDefinitions` 是採用 **增量合併**（從程式碼看，其對於不存在的鍵會新建，已存在的則 AddRange 附加 [33] ），需測試連續兩次 SetDefinitions 的行為：如第一次加入一些定義，第二次加入同類型的另一批定義，最終資料庫該類型下應包含兩批的合集。

5. 測試對重複 defID 的處理：理論上在 Loader 階段已避免重複，但若透過 SetDefinitions 重複加入，當前實作不會檢查重複而是盲目 AddRange。因此，我們可以在測試中關注不要重複加入相同物件兩次，以免干擾其他測試。總之，此部分重點在於確認資料結構操作正常。

6. **測試 GetDefinition 查詢：** `DefinitionDatabase.GetDefinition<T>(string defID)` 提供依類型和ID檢索定義的介面 [18] 。對於前面加入的測試定義，可逐一嘗試查詢：

7. 傳入存在的 defID，應回傳相應物件。例如前述 `Knight` 定義加入後，`DefinitionDatabase.GetDefinition<CharacterDef>("Knight")` 應返回其 `CharacterDef` 對象 (非 null)，且 `health` 、 `speed` 等屬性與加入時相同或已被繼承處理更新。
8. 傳入不存在的 defID，應返回 null [34] 。可測試一兩個不存在的ID，確保沒有例外拋出而是優雅地返回 null。

9. 邊界測試：不同類型的定義使用相同 defID 時， `GetDefinition<Subtype>` 不會錯誤混淆。例如如果存在同名的角色和物品定義（不同類別），`GetDefinition<CharacterDef>("ExampleID")` 只會搜尋角色列表，不會誤返回物品。同樣地， `GetDefinition<ThingDef>("ExampleID")` 只能找到物品定義。這在測試上可透過插入兩類定義同名來驗證隔離性。

10. **整合註冊流程測試：** 將前述解析流程取得的 `definitions` 字典直接傳入 `DefinitionDatabase.SetDefinitions` ，模擬 `ModManager.LoadModsDefinition()` 最後的行為 [28] 。然後使用 `GetDefinition` 驗證幾個關鍵定義能成功取出。例如：

11. 取得 `Knight` 的定義物件，確認其各欄位值最終狀態正確（特別是繼承後值）。例如 `Knight.health` 是否為 150 而非 0， `Knight.speed` 是否為 3 [35] 。
12. 確認抽象定義無法取出或不存在於資料庫中。依照繼承處理邏輯，抽象定義在處理後應該被排除，不加入資料庫 [36] [37] 。我們可嘗試 `GetDefinition<CharacterDef>("BaseWarrior")` ，期望得到 null（或定義列表中根本沒有 BaseWarrior）。若發現抽象定義也被存入，則指出測試失敗，促使改善（例如在繼承處理時過濾）。
13. 針對跨模組的定義，若專案支援，可在測試中模擬多個模組的定義合併，確保不同來源的定義都能註冊，且 defID 沒有衝突或衝突已被覆寫。

透過上述測試，我們能確信定義資料庫的註冊與查詢機制穩定可靠，使後續遊戲邏輯能正確取得定義資料。

測試驗證流程（繼承與完整性驗證）

目標： 確認 **定義的繼承與驗證** 流程，這是 Definition 模組的關鍵功能。包括測試繼承處理是否按預期應用父類屬性覆蓋子類預設值，以及最終結果的完整性（如抽象定義不出現在結果中、各項屬性值正確）。

1. **繼承處理邏輯簡述：** `ModDefinitionInheritor` 會自動註冊所有 `IDefinitionInheritor` 子類（如 `CharacterDefInheritor` , `ThingDefInheritor` ） [38] 來分別處理不同類型的定義。對於每種類型：

2. 若有專門的 Inheritor，則使用其邏輯；否則採用默認邏輯 [39] [40] 。默認邏輯會將所有抽象定義過濾掉並處理每個具體定義的繼承欄位 [41] [42] 。
3. 專門 Inheritor 則可能利用工具類 `DefinitionInheritanceUtils` 進行更精細的屬性繼承。例如 `CharacterDefInheritor` 使用 `InheritNumericProperty` 讓子定義在數值為0時繼承父定義的值 [43] 。

我們需要測試繼承後的定義清單是否滿足以下要求： - **抽象定義剔除：** 繼承處理後輸出的定義列表不應包含任何 `IsAbstract=true` 的定義 [36] 。這意味著例如 BaseWarrior, BaseMage 等不會出現在最終 `processedDefinitions` 中 [26] 。 - **屬性繼承/覆蓋：** 子定義的屬性若在子節點中給定值，應保留該值；若未給值或為預設值（如數值0或空字串），應從父定義繼承非預設值 [43] [44] 。同時，複合物件或列表型屬性應正確合併（例如 tag 列表、weaponProps 物件的欄位繼承） [45] [46] 。 - **繼承鏈多層處理：** 對於存在多層繼承的情況，系統應遞迴套用直到根基類。例如 `IronSword` 繼承 `BaseMeleeWeapon`，而後者又繼承 `BaseWeapon` [47] ；測試要驗證 `IronSword` 的最終屬性包含了從 `BaseWeapon` 間接繼承的部分（如 tags 列表含 weapon,equipment 標籤 [48] [49] ）。 - **無效繼承處理：** 如果子定義引用了不存在的父類（不當的 inheritsFrom），系統應能檢測並發出警告，而不致於崩潰 [50] [51] 。這點可通過檢查日誌或返回結果驗證：例如構造一個 `inheritsFrom="NonExistDef"` 的定義，執行繼承處理後確認： * 日誌出現 `"Parent definition 'NonExistDef' not found..."` 警告 [51] （可用 `LogAssert.Expect` 捕捉）。* 該子定義仍出現在結果中但保持原狀（除了 label/desc 等可能部分繼承不到之外），**或**（依實作）直接略過該子定義。從程式碼看目前實作遇不到父類僅是 log 警告後仍保留子定義 [52] 。所以測試應確保即便父類缺失，`ProcessInheritance` 依然返回子定義，只是沒有套用父屬性。同時驗證錯誤計數有增加（若繼承器有錯誤統計）。

1. **設計繼承測試用例：** 以核心模組提供的測試定義為依據來設計驗證點（Expected Outcome） [53] ：
2. **數量驗證：** 進行繼承處理後，確認最終具體定義數量正確。如以 Characters.xml 測試資料，最終應有 4 個 `CharacterDef`（Knight, Wizard, Archer, Peasant） [26] ；以 Things.xml 測試資料，應有 5 個 `ThingDef`（IronSword, WoodenBow, MagicStaff, HealthPotion, Coin） [27] 。
3. **屬性繼承驗證：** 驗證幾個具體案例：
    - `Knight`：繼承自 BaseWarrior，健康值應為 150（父值），速度為 3（自身覆蓋） [19] [20] 。
    - `Wizard`：繼承自 BaseMage，健康值應為 70（自身覆蓋，比父類80低），速度為 6（父值，因子節點未提供 speed） [54] [19] 。
    - `Archer`：繼承自 BaseCharacter，健康值 90、速度7（均覆蓋父類100/5） [55] 。
    - `Peasant`：無繼承關係，保持定義值健康60、速度3 [56] 。
    - `IronSword`：繼承自 BaseMeleeWeapon：
    - damage 未在子定義顯式給出，因此繼承父值20；但子節點的 `<weaponProps>` 裡提供了 `<damage>30</damage>` 覆蓋父武器屬性傷害 [57] 。
    - 標籤 tags 應為父(BaseMeleeWeapon: melee + 間接父BaseWeapon: weapon,equipment + BaseItem: item) **合併** 上子定義新增的 "metal" [49] 。測試應確認最終 `IronSword.tags` 包含 `"weapon","equipment","melee","metal"` [20] 。
    - `WoodenBow`：繼承自 BaseRangedWeapon：
    - 父類 damage15，子定義覆寫為12 [58] ；父類 weaponProps.type="Ranged", damage=20, range=8；子 weaponProps 覆寫 range 為10 [59] （type與damage應繼承父值Ranged/20）。測試檢查：最終 `WoodenBow.weaponProps.type == "Ranged"` [60] ； `weaponProps.damage == 20` ； `weaponProps.range == 10` 。
    - tags 合併：父有 "ranged"，子新增 "wood","hunting"，最終應含三者。
    - `MagicStaff`：繼承自 BaseWeapon：
    - 父 tags: weapon,equipment；子 tags: magic,staff -> 合併四個標籤。
    - 父 weaponProps: type=Melee,damage=15,range=1；子 weaponProps 完全提供了 type=Magic, damage=35, range=6，應**覆蓋**父 weaponProps（最終不應保留父屬性）。測試應確認 MagicStaff.weaponProps.type為"Magic"，damage為35，range為6。

- `HealthPotion`：繼承自 BaseConsumable（stack10, tag: consumable）；子覆寫 stack=5，新增 tag: healing,potion -> 驗證 stack 最終為5，tags 合併含 consumable,healing,potion。
- `Coin`：無繼承，保持自身定義：damage0, stack100, tags: currency,valuable。

為實現上述測試，我們可以重用專案內的定義資料（Characters.xml, Things.xml）。在 EditMode 測試中，可直接讀取 `Assets/StreamingAssets/Mods/Core/Defs/` 目錄下的測試檔案內容（透過 `File.ReadAllText` 或將其加入 Unity Editor 中的 TextAsset）。然後： - 解析 XML 為 XDocument，透過 `ModDefinitionDeserializer.InstanceDefinitions` 獲得原始定義字典。 - 執行 `ModDefinitionInheritor`：先呼叫 `RegisterInheritors()` 註冊處理器 61（會透過 ReflectionUtils 找出專案中所有 `IDefinitionInheritor` 38），再呼叫 `ProcessInheritance(defDict)` 獲取處理繼承後的新字典 62。 - 拿結果字典進行上述各項驗證。由於項目預期在繼承處理後才將結果放入 `DefinitionDatabase` 63，我們可以選擇直接驗證結果字典內容，或進一步經由 DefinitionDatabase 取出驗證。

範例： 以下展示如何測試 `CharacterDef` 繼承結果（以簡化資料）：

```
[Test]
public void CharacterDefInheritance_ShouldInheritParentValues()
{
  // Arrange: 準備父子定義列表
  var baseDef = new CharacterDef { defID = "BaseWarrior", IsAbstract = true, health = 150, speed = 4 };
  var childDef = new CharacterDef { defID = "Knight", inheritsFrom = "BaseWarrior", health = 0,
speed = 3 };
  List<Definition> defs = new List<Definition> { baseDef, childDef };
  var inheritor = new CharacterDefInheritor(); // 專門處理 CharacterDef 的繼承器

  // Act: 執行繼承處理
  var processed = inheritor.ProcessInheritance(defs);
  var resultList = processed.Cast<CharacterDef>().ToList();

  // Assert: 應只有具體定義 Knight 在結果中，BaseWarrior (抽象)被過濾掉
  Assert.AreEqual(1, resultList.Count);
  var knightResult = resultList[0];
  Assert.AreEqual("Knight", knightResult.defID);
  // 繼承：因 Knight.health 原為0，應繼承父值150；Knight.speed 有自訂為3，保持不變
  Assert.AreEqual(150, knightResult.health, "Knight 未繼承到父定義的 health");
  Assert.AreEqual(3, knightResult.speed, "Knight.speed 不應受父定義影響");
}
```

如上，`CharacterDefInheritor` 內部運用了 `DefinitionInheritanceUtils.InheritNumericProperty` 來實現「子值為0則繼承父值」的邏輯 43 44。測試驗證 Knight 的 health 確實從0變為150，速度維持3不變。抽象 BaseWarrior 不在結果中，符合預期 36。

1. **完整集成測試：** 最後，可撰寫一個涵蓋 **從載入到繼承到資料庫** 的集成測試，用來模擬實際 Mod 定義載入流程：利用 `ModManager` 或各組件協同將 Mods/Core/Defs 下的所有定義載入、反序列化、繼承並註冊，然後對最終狀態進行驗收式斷言。例如：

2. **定義總數驗收：** `DefinitionDatabase.GetDefinitions()` 總數是否符合已知值（如核心模組 defs）。
3. **抽樣驗證：** 抽幾個定義核對關鍵屬性，例如 `DefinitionDatabase.GetDefinition<CharacterDef>("Knight").health == 150` [35] 等，如前所述的驗證點。
4. **錯誤日誌驗證：** 如果故意構造錯誤（如重複 defID、不存在父類），測試在期望的位置使用 `LogAssert.Expect(LogType.Warning, "...")` 或 `LogAssert.Expect(LogType.Error, "...")` 來斷言系統確實報告了問題並沒有靜默失敗 [51]。

通過上述各層級測試，能確保 Definition 模組在解析->註冊->繼承各環節表現正確，資料完整一致。

依賴管理與重構建議

在為現有程式碼補充測試時，我們常需要**鬆散耦合**現有系統以方便注入測試依賴。以下針對 moddable-study 專案提出依賴管理和重構建議，以降低測試難度並提升程式的可維護性：

- **引入介面抽象關鍵元件：** 將目前具體實現轉接為介面介入點，使我們能在測試中使用替代實現。例如：
- 檔案存取相關類別（如 `ModFinder`, `ModDefinitionLoader`）可定義對應的介面 `IModFinder`, `IDefinitionLoader`，提供方法如 `FindMods()`、`LoadDefinitions()`。實際程式中由原類別實作，單元測試中可以注入一個假實現（stub）返回預先設定的模擬資料。這麼做可避免在測試環境真的創建檔案或目錄。

- 定義處理相關類別（如 `ModDefinitionDeserializer`, `ModDefinitionInheritor`）也可考慮介面化。例如提供 `IDefinitionDeserializerService`、`IDefinitionInheritanceService` 等。如果未來更換實現（如從 XML 換成 JSON 定義）或在測試中希望跳過某些複雜步驟，都能透過替代實現達成。

- **拆解大型模組並注入依賴：** `ModManager` 構造函數目前接受多個子系統物件 [64]，說明已有良好的依賴注入傾向。可以繼續優化：

- 將 `ModDefinitionInheritor` 也透過建構子注入，而非在內部直接 new [65]。這允許測試時傳入自訂的繼承處理器（或經過修改的繼承器，例如繼承鏈中插入故意的錯誤狀況）。
- 考慮將 `DefinitionDatabase` 抽象為介面（例如 `IDefinitionRepository`），由具體類別實作全域儲存。或至少提供讓 `ModManager` 可接收一個資料庫實例，而不直接調用靜態方法。雖然靜態方便，但在測試中無法替換或隔離。如果能傳入一個假資料庫，測試可監控插入行為或避免影響全域狀態。

- 目前 `DefinitionDatabase.Clear()` 等靜態方法需手動呼叫來重置狀態 [66]。若改為非靜態，則每個測試可 new 一個資料庫實例，不會相互干擾。此外靜態集合可能導致平行執行測試時的衝突，因此非靜態更加安全。

- **移除不必要的靜態依賴：** 除了資料庫，檢查專案中是否有其他透過靜態狀態共享的部分，例如單例模式或工具類快取。如有，提供對外重置接口（如已有的 `ReflectionUtils.ClearCache()` [66]）或重構為實例方法，降低隱含狀態。靜態耦合越多，測試前後就越需要進行初始化和清理，增加出錯機率。

- **依賴注入框架考量：** 在 Unity 中可以考慮使用輕量的 DI 框架（如 Zenject、Unity's Dependency Injection 等），但對於本專案規模，或許不必上升到引入框架。手工的構造函數注入已足夠清晰 [64]。

建議保持這種模式，讓上層組織物件（如 ModManager）負責組裝依賴，而底層模組盡量使用介面和抽象類，避免直接 new 深層物件。

- **SOLID 原則與可測性：** 檢視現有架構，繼承處理部分已遵循單一職責和開放封閉（每種定義類型有獨立處理器，可擴充）[67]。可以進一步在**依賴反轉**上加強，讓高階模組（如 ModManager）依賴於抽象（介面）而非具體實現[68]。例如 ModManager 不需要知道 ModDefinitionLoader 的細節，只需調用 `IDefinitionLoader` 介面的方法，測試時可換成一個返回固定 XDocument 的 FakeLoader。這樣也隔離了檔案系統帶來的不確定性。

- **記錄與錯誤處理替代：** 測試時常需要檢查日誌輸出（如前述驗證無父定義警告）。目前使用的 `ModLogger` 在背後呼應 UnityEngine.Debug 日誌[69]。可考慮在測試環境替換 ModLogger 為一個簡單實現（實作同介面或繼承但將輸出存入緩存串列），方便測試對日誌內容斷言，而不依賴 UnityEngine 日誌系統。或者利用 Unity 提供的 `LogAssert`，不過那需要日誌實際輸出到 Unity console 才能截取。在純邏輯單元測試中，繞過真正日誌而採用虛擬日誌對象會更直接。

總而言之，**降低耦合、控制副作用** 是提升可測性的關鍵。透過引入介面、移除靜態以及依賴注入，能讓我們在編寫單元測試時自由替換底層行為，專注於測試邏輯本身，而不必啟動完整的遊戲環境或讀寫真實檔案。此外，這些重構對專案未來維護也大有裨益：更清晰的依賴關係、更模組化的組件，使得新增功能或修改行為時影響範圍更可控。

測試資料處理策略

在進行單元測試時，常需要模擬外部資源或環境，例如定義檔內容、模組目錄結構等等。以下策略有助於在不依賴真實檔案系統或 Unity 引擎的情況下，提供測試所需的資料：

- **使用 Stub/Fake 替代外部資源：** 透過前述介面抽象，我們可以為測試打造**假資源提供者**：
- **假 Mod 掃描**：實作一個 `IModFinder` stub，直接返回預設的模組清單，而不實際掃描目錄。例如返回一個只包含 "Core" 模組的列表，其中定義檔路徑指向我們準備的測試檔案。這避免依賴 `Application.streamingAssetsPath` 等環境變數。
- **假定義載入**：實作 `IDefinitionLoader` stub，將預備的字串內容轉成 XDocument 合併返回。這樣測試不用真正打開每個文件，只需在程式內定義好多個 XML 字串。甚至我們可以把前面提到的 Characters.xml, Things.xml 內容作為常值字串放入測試代碼中，用 Fake Loader 載入之。
- **假 Patch 處理**：如果有 Patch 系統，也可以提供 Fake Patcher 在測試中簡化行為（比如直接應用或者什麼也不做），以專注測試主要功能。

- **假 Assets 載入**：對於載入美術資源（`ModAssetsLoader`）或設定（`ModSettings`），單元測試通常可跳過或提供假資料結構，除非這些對功能有直接影響。大多情況下，可將這些部分從 `ModManager` 初始化中剝離或注入空實現，使測試流程不涉及 UnityEngine.Object 等無法脫離引擎的類型。

- **運用 Mock 框架驗證互動：** 在需要確認**某方法是否被呼叫**或**呼叫次數**等互動行為時，可使用 Mock 框架（如 Moq, NSubstitute）。例如要驗證 `DefinitionLoader` 正確對每個檔案都呼叫了一次載入，可用 Mock 對象替代之，並在測試後使用 `Verify` 檢查呼叫計數。不過，由於 Unity 編輯器下使用 Moq 需額外配置（導入對應 DLL），且多數情況我們可以透過檢查結果狀態來推斷互動是否發生，因此 Mock 框架可視需要選用。

- **資料驅動測試：** 考慮將多組輸入/預期輸出使用測試案例提供。NUnit 支援 `[TestCase]` 等屬性，允許我們撰寫一次測試方法，針對不同輸入資料執行多次。例如可以為 `RemoveExisting` 方法寫一組測試，提供不同組合的已存在defIDs、新元素defID，檢查結果。這種資料驅動有助涵蓋更多邊界情況。

- **隔離引擎相關 API：** 單元測試應儘量不依賴 UnityEngine 或 Editor 類別。如測試需要 Application.path 之類，可在程式中將這類值通過參數傳入，或包裝在可替換的服務中。又如前述 ModLogger，可將 UnityEngine.Debug 功能封裝，以便替換。總之，把**不穩定或外部環境**相關的呼叫集中管理，在測試中以可控方式提供輸出或輸入。

- **利用 Unity 測試工具：** 對於仍須部分依賴 Unity 的測試，Unity Test Framework 提供一些實用工具：

- `LogAssert`：可檢查執行期間的日誌。如前面提到，用 `LogAssert.Expect(LogType.Warning, "Parent definition 'NonExistDef' not found*")` 來驗證是否正確拋出警告，而無需真的實作一個 Logger stub。
- `UnityTest` **協程測試**：如果有需要等待幀更新或測試異步過程，可將測試寫成協程並標記 `[UnityTest]`。不過對於純資料邏輯通常不需要。
- `Setup` / `TearDown`：使用 `[SetUp]` 在每個測試前清理/初始化環境（如前述清空 DefinitionDatabase），`[TearDown]` 在每個測試後收尾。如果某些假目錄或臨時檔案生成了，可在 TearDown 刪除，以免影響後續測試。

測試資料管理也是不容忽視的部分。建議在測試專案中建立一套小型的測試定義檔資源，可放在 Assets/Tests/Resources 或直接以字串形式寫在代碼裡。這些測試資源應覆蓋一般和極端案例，例如：

- 正常的單一繼承層級定義、一對多繼承定義（多個子繼承同一父）。
- 多層繼承定義。
- 缺失必要欄位的定義（沒有 defID 或 inheritsFrom 指向不存在）。
- 多模組覆寫同一 defID 的情況。

將這些資料作為 **fixtures** 餵給測試，確保各種情況下系統行為都符合預期。

範例測試案例

本節以 **Definition 解析與繼承** 模組中的一個代表性組件為例，展示如何撰寫單元測試。讓我們以 **角色定義 (CharacterDef)** 的載入與繼承為場景：我們期待測試能驗證角色定義的繼承關係處理正確，符合設計預期。

假設我們已有以下定義結構（與前述 Characters.xml 範例一致）：

- `BaseWarrior`：抽象角色，health=150, speed=4
- `Knight`：繼承自 BaseWarrior，未明示 health（視為0），speed=3

依據需求，繼承處理後 Knight 應取得 health=150（從父繼承），保持 speed=3。以下是一個對應的單元測試程式碼範例：

```
using NUnit.Framework;
using ModArchitecture.Definition;            // Definition 基類
using ModArchitecture.Definition.Inheritors;    // Definition 繼承相關介面
// 假定 CharacterDef, CharacterDefInheritor 定義於 Angus 命名空間:
using Angus;                        // 包含 CharacterDef, CharacterDefInheritor 等

[TestFixture]
public class CharacterDefInheritanceTests
{
  [SetUp]
  public void SetUp()
```

```
    {
        // 每次測試前清空全域定義資料庫，確保不受前次測試影響
        DefinitionDatabase.Clear();
    }

    [Test]
    public void Knight_ShouldInheritHealth_FromBaseWarrior()
    {
        // Arrange: 構造父 (抽象) 和子定義物件
        var baseWarrior = new CharacterDef { defID = "BaseWarrior", IsAbstract = true, health = 150,
speed = 4 };
        var knight = new CharacterDef { defID = "Knight", inheritsFrom = "BaseWarrior", health = 0,
speed = 3 };

        // 將兩者加入列表，模擬反序列化出的定義集合
        var definitions = new List<Definition> { baseWarrior, knight };

        // 實例化繼承處理器（CharacterDef專用）
        IDefinitionInheritor inheritor = new CharacterDefInheritor();

        // Act: 執行繼承處理
        var processedDefs = inheritor.ProcessInheritance(definitions);
        // 轉型回 CharacterDef 以取用特定屬性
        var processedList = processedDefs.Cast<CharacterDef>().ToList();

        // Assert:
        // 只應有 Knight 一個具體定義（BaseWarrior 屬抽象，應被過濾掉）
        Assert.That(processedList, Has.Exactly(1).Items);
        var processedKnight = processedList[0];
        Assert.AreEqual("Knight", processedKnight.defID);
        // 驗證繼承效果：health 從 0 改為 150，speed 保持 3
        Assert.AreEqual(150, processedKnight.health, "Knight health should inherit from BaseWarrior
35 ");
        Assert.AreEqual(3, processedKnight.speed, "Knight speed should remain as defined in child");
    }
}
```

上面這段測試代碼做了以下事情：

- **建立測試物件 (Arrange)：** 手動 new 出 `CharacterDef` 物件來模擬反序列化結果，而不涉及 XML 讀取。`BaseWarrior` 被標記為抽象且給定屬性值，`Knight` 設定了繼承父ID且將 health 設為0表示預設值。這樣的資料準備反映了我們預期測試的場景。

- **執行待測行為 (Act)：** 直接使用 `CharacterDefInheritor` 來處理我們的清單。實作上，`CharacterDefInheritor.ProcessInheritance` 會呼叫 `DefinitionInheritanceUtils.ProcessInheritance` 並應用 `ApplyCharacterDefSpecificInheritance` 來處理 health 和 speed [70] [71]。我們不需關心內部細節，只需拿結果。

- **驗證結果 (Assert)：**

- 使用 NUnit 斷言確認結果清單長度以及內容。`Has.Exactly(1).Items` 確保只有一個結果（Knight）。
- 檢查 Knight 的 defID 是否還是 "Knight"，防止意外修改。
- 檢查 Knight.health 是否等於150，並在訊息中引用了我們先前分析的預期值出處 [35] 以佐證這一數值的正確性。
- 檢查 Knight.speed 是否仍為3，說明沒有被父類覆蓋。

此測試涵蓋了**繼承處理的核心情況**：子定義缺省值繼承父定義，以及抽象定義的過濾。它相對獨立，不依賴 Unity 引擎，因此可快速執行並定位問題。如果此測試不通過，我們會知道是繼承邏輯出了問題（例如 `InheritNumericProperty` 判斷0的機制或抽象過濾機制有誤）。

進一步範例： 類似地，我們可以為 `ThingDefInheritor` 撰寫測試，驗證較複雜的列表合併與物件屬性繼承。例如： - 建立 BaseWeapon (抽象), BaseMeleeWeapon (抽象, inherits BaseWeapon), IronSword (具體, inherits BaseMeleeWeapon) 三者物件。填充 BaseWeapon.tags = {"weapon","equipment"}, BaseMeleeWeapon.tags 繼承加 {"melee"}, IronSword.tags 加 {"metal"}；還有 weaponProps 中 BaseWeapon.type=Melee, BaseMeleeWeapon.weaponProps.damage=25, IronSword.weaponProps.damage=30 等。 - 執行 ThingDefInheritor.ProcessInheritance，驗證結果 IronSword: - tags 是否包含 weapon,equipment,melee,metal 四項。 - weaponProps.damage 是否為子定義的30，weaponProps.type 是否繼承為 "Melee"，range 是否繼承父值。 - 同時驗證抽象 BaseWeapon, BaseMeleeWeapon 不在結果清單中等。

編寫這類測試時，可參考前述 Things.xml 的繼承關係預期 [20] [49] [58] 來設計驗證條件。

透過逐步擴充上述範例測試，最終我們可以涵蓋： - **單一屬性繼承**（如數值型的 health, damage 繼承）。 - **列表合併**（tags, component 列表等）。 - **複合對象**（weaponProps 等部分字段繼承）。 - **邏輯分支**（子有給值 vs 無給值，不同情況繼承與否）。

這些測試案例將有效防止將來對定義格式或繼承邏輯的修改引入迴歸。同時，由於我們使用了簡化的輸入物件，測試執行快速且不依賴環境，大量案例也可在幾秒內跑完。

測試目錄與 Assembly 定義規劃

為了在 Unity 專案中良好地組織與執行測試，我們需要規劃適當的目錄結構和 Assembly Definition 檔配置。遵循 Unity 的慣例，可以採取以下方案：

- **測試目錄結構：** 將所有測試程式碼放在 Assets/Tests/ 路徑下。這個目錄名不是強制的，但是 Unity 社群普遍使用的習慣，有助辨識。子目錄可細分為：
- Assets/Tests/Editor/ : 放置 **Edit Mode 測試**。例如 `DefinitionTests.cs`, `ModManagerTests.cs` 等針對編輯器執行的單元測試。
- Assets/Tests/PlayMode/ : （如需要）放置 **Play Mode 測試**。如果某些測試需在遊戲運行時環境（如測試場景中的實例化、MonoBehaviour互動），就會寫在這裏。對於純資料邏輯，不一定需要此部分。

- 如測試數量龐大，可再依功能模組細分子資料夾，如 Tests/Editor/Definition/, Tests/Editor/ModLoading/ 等。

- **Assembly Definition Files (asmdef)：** 為測試目錄各自建立 Assembly 定義，以隔離測試程式集並聲明對其它組件的引用：

- Assets/Tests/Editor 下建立 **moddable-study.Tests.Editor.asmdef**（名稱可自定義）。設定：
    - **Platforms** 僅選擇 Editor（不勾選任何其他平台），表示此組件專屬 Editor 執行 [72]。
    - **Assembly Definition References** 增加對被測專案程式集的引用，如 `Implement.Core`, `ModInfrastructure.Core` 等。也加入 Unity 測試相關引用：`UnityEngine.TestRunner` 和 `UnityEditor.TestRunner`（後者只有 Editor 下可用）[5]。
    - Unity 預設透過 Test Runner 創建的 asmdef 已經包含 `nunit.framework` 的引用 [5]（可在 **Assembly References** 欄位看到），若無則需手動加入對 NUnit 框架DLL的引用（通常GUID引用由 Test Runner 自動處理）。
    - 確保 **Auto Referenced** 為 **false**（通常預設如此），以免這個測試組件被非測試組件引用到。
    - 該 asmdef 檔可設定成 Editor Only，因此不會包含在遊戲發行的build中。
- Assets/Tests/PlayMode 下建立 **moddable-study.Tests.PlayMode.asmdef**（如需要PlayMode測試）。設定：
    - **Platforms** 可選擇 All Platforms 或特定運行平台，但**不要**包含 Editor。All Platforms 意味著這個組件將在PlayMode下執行測試 [7]。
    - 引用的組件包括被測程式集，`UnityEngine.TestRunner`（不用加 UnityEditor.TestRunner，因為在運行時不可用），以及 `nunit.framework`。
    - 另外需要在 **Optional Unity References** 中勾選 **TestAssemblies** [9]。這讓Unity在編譯player時也包含該測試組件（僅在我們需要在player環境執行測試時使用，如做自動化或CI在無Editor環境跑PlayMode測試）。
- **編輯器 & 遊戲腳本分離：** 測試程式碼也要注意編輯器專用API不要出現在PlayMode測試組件中。例如我們在 Editor 測試可以使用 `UnityEditor.AssetDatabase` 讀取資源，但這類代碼不能編譯進PlayMode測試（因為目標平台可能是獨立執行檔）。確保 Editor 測試和 PlayMode 測試各自在自己的 asmdef 中，各自引用正確的 UnityEngine/UnityEditor API。
- **使用範圍 (Assembly Definition References)：** 測試組件需要能訪問被測組件的**內部成員**時，有兩種方式：
- **[InternalsVisibleTo]**：在被測組件的 Assembly Info 中加入InternalsVisibleTo屬性指向測試組件名稱。
- **Testable Assembly**：Unity 2020+ 的 asmdef 有個選項 **"Testables"**，可直接設定哪個組件可被測試組件視為朋友。透過在被測 asmdef 的 Testables 清單加入自己的 runtime asmdef，則該組件內部成員對測試組件可見。

這樣可避免不得已將成員改為 public 才能測試的情形。根據需要調整，本指南建議盡量透過公開API來測試，但在必要時上述措施也可採用。

- **運行與持續整合：** 設置好目錄與 asmdef 後，可在 Unity Editor 的 Test Runner 中看到我們的測試分類列出（EditMode 和 PlayMode 分開顯示）。可隨時運行以確保通過。未來可將測試運行整合到持續整合管線，如使用 Unity Test Runner CLI 在每次提交自動執行所有測試，保障程式品質。

透過合理的目錄與組件定義規劃，我們做到： - 測試程式碼與生產程式碼相互隔離，避免不小心將測試代碼包含進遊戲。 - 清晰區分 Editor/PlayMode 測試，符合 Unity Test Framework 對不同環境的要求 [73] [74]。 - 易於擴充：未來新增其他模組的測試，可以在 Tests 資料夾按模組新增子資料夾和修改asmdef引用即可，不影響其他部分。

經由上述指南步驟，Unity 開發者可按部就班地在 **moddable-study** 專案中導入單元測試。從啟用測試框架、建立測試專案結構，到針對 Definition 模組撰寫詳盡的測試用例，再到調整架構以利測試與維護，我們確保了每一步都有清晰的說明與範例佐證 [18] [20] 。

透過優先為 **Definition 資料解析** 等核心模組回填測試，我們可以及早發現架構中的潛在問題（例如繼承邏輯邊緣情況、跨模組定義衝突等），並在重構時有測試保駕護航，防止回歸。隨著測試覆蓋率的提升，整體模組化設計的可靠性與可擴充性將大大增強，這對專案長期發展至關重要。祝您在專案中順利導入單元測試，打造更健壯的遊戲系統！

參考來源： 專案原始碼 [17] [14] 、專案文件 [20] 以及 Unity 官方文檔 [5] [72] 等。上述內容希望對您有所幫助，讓您能成功將本指南的步驟應用到實際開發中。祝測試順利進行！

---

[1] [4] [5] [6] [7] [11] Workflow: How to create a new test assembly | Test Framework | 1.4.0
https://docs.unity.cn/Packages/com.unity.test-framework@1.4/manual/workflow-create-test-assembly.html

[2] [3] How to run automated tests for your games with the Unity Test Framework
https://unity.com/how-to/automated-tests-unity-test-framework

[8] [9] [72] [73] [74] Edit Mode vs. Play Mode tests | Test Framework | 1.1.33
https://docs.unity3d.com/Packages/com.unity.test-framework@1.1/manual/edit-mode-vs-play-mode-tests.html

[10] Implement.Core.asmdef
https://github.com/angus945/moddable-study/blob/50c3fa62a28abab469bab4df42565ff98908e9b3/Assets/Script/Implement.Core/Implement.Core.asmdef

[12] [15] [16] [22] [28] [29] [65] ModManager.cs
https://github.com/angus945/moddable-study/blob/50c3fa62a28abab469bab4df42565ff98908e9b3/Assets/Script/ModInfrastructure.Core/ModManager/ModManager.cs

[13] [14] [21] ModDefinitionLoader.cs
https://github.com/angus945/moddable-study/blob/50c3fa62a28abab469bab4df42565ff98908e9b3/Assets/Script/ModInfrastructure.Core/ModManager/ModDefinitionLoader.cs

[17] [18] [30] [31] [32] [33] [34] DefinitionDatabase.cs
https://github.com/angus945/moddable-study/blob/50c3fa62a28abab469bab4df42565ff98908e9b3/Assets/Script/ModInfrastructure.Core/Definition/DefinitionDatabase.cs

[19] [20] [26] [27] [35] [53] [60] README.md
https://github.com/angus945/moddable-study/blob/50c3fa62a28abab469bab4df42565ff98908e9b3/Assets/StreamingAssets/Mods/Core/Defs/README.md

[23] [24] CharacterDefDeserializer.cs
https://github.com/angus945/moddable-study/blob/50c3fa62a28abab469bab4df42565ff98908e9b3/Assets/Script/Implement.Core/DefinitionDeserializer/CharacterDefDeserializer.cs

[25] DefinitionDeserializerBase.cs
https://github.com/angus945/moddable-study/blob/50c3fa62a28abab469bab4df42565ff98908e9b3/Assets/Script/ModInfrastructure.Core/ModManager/Deserializer/DefinitionDeserializerBase.cs

[36] [37] [61] [62] [63] [67] [68] README.md
https://github.com/angus945/moddable-study/blob/50c3fa62a28abab469bab4df42565ff98908e9b3/Assets/Script/ModInfrastructure.Core/ModManager/Inheritor/README.md

38 39 40 41 42 51 52 ModDefinitionInheritor.cs

https://github.com/angus945/moddable-study/blob/50c3fa62a28abab469bab4df42565ff98908e9b3/Assets/Script/ModInfrastructure.Core/ModManager/Inheritor/ModDefinitionInheritor.cs

43 70 71 CharacterDefInheritor.cs

https://github.com/angus945/moddable-study/blob/50c3fa62a28abab469bab4df42565ff98908e9b3/Assets/Script/Implement.Core/DefinitionInheritor/CharacterDefInheritor.cs

44 DefinitionInheritanceUtils.cs

https://github.com/angus945/moddable-study/blob/50c3fa62a28abab469bab4df42565ff98908e9b3/Assets/Script/ModInfrastructure.Core/Utils/DefinitionInheritanceUtils.cs

45 46 47 48 49 57 58 59 Things.xml

https://github.com/angus945/moddable-study/blob/50c3fa62a28abab469bab4df42565ff98908e9b3/Assets/StreamingAssets/Mods/Core/Defs/Things.xml

50 DefinitionInheritorBase.cs

https://github.com/angus945/moddable-study/blob/50c3fa62a28abab469bab4df42565ff98908e9b3/Assets/Script/ModInfrastructure.Core/ModManager/Inheritor/DefinitionInheritorBase.cs

54 55 56 Characters.xml

https://github.com/angus945/moddable-study/blob/50c3fa62a28abab469bab4df42565ff98908e9b3/Assets/StreamingAssets/Mods/Core/Defs/Characters.xml

64 66 69 Test.cs

https://github.com/angus945/moddable-study/blob/50c3fa62a28abab469bab4df42565ff98908e9b3/Assets/Script/Test.cs