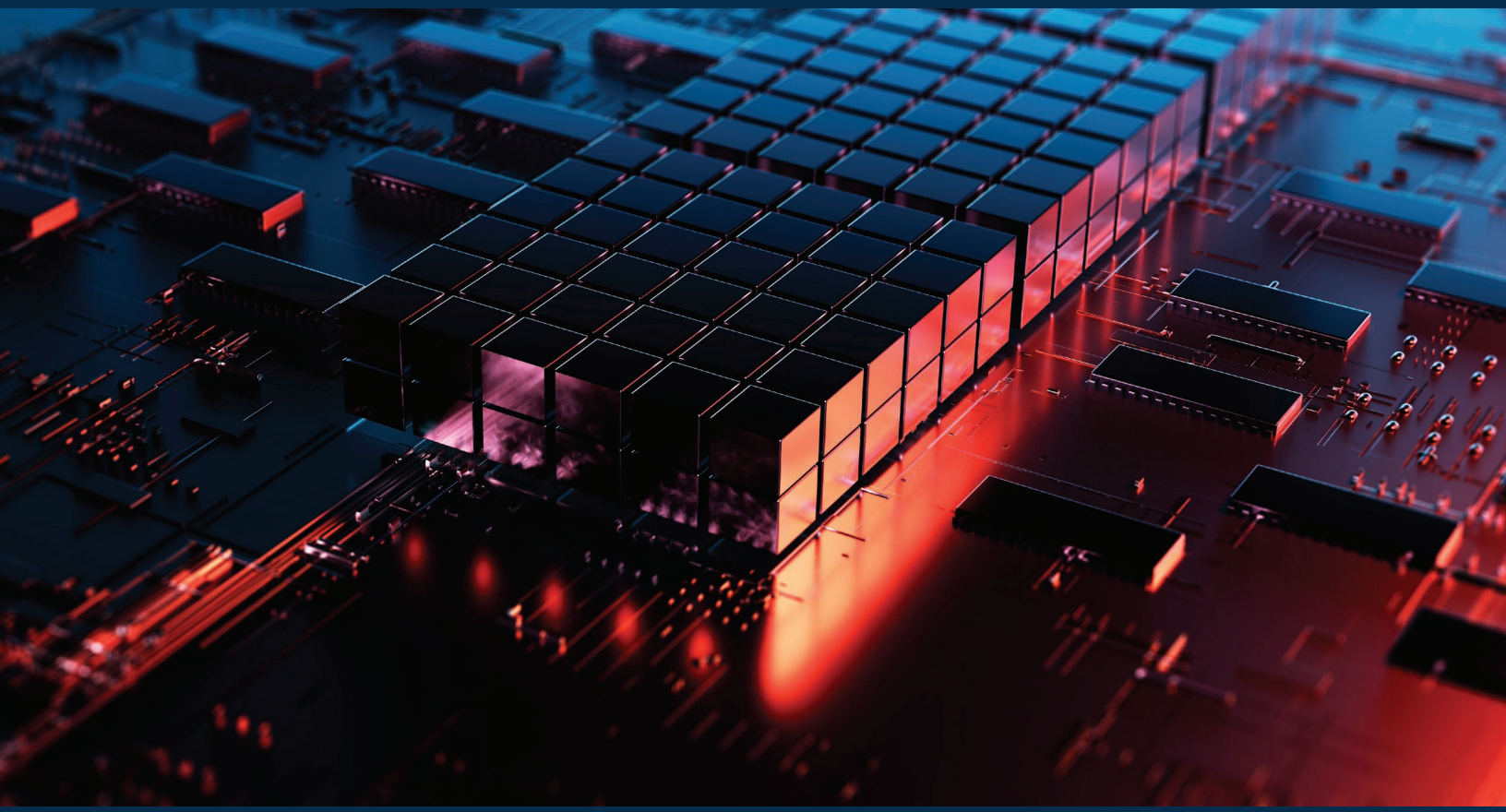
→ E-BOOK

ScriptableObject を使って Unity でモジュラーゲームアーキテクチャを作成する



Unity®

Contents

はじめに

ScriptableObject は “データコンテナ” と形容されることがよくあります。しかし、この呼び名は内容を十分に表していません。ScriptableObject を正しく活用すると、Unity ワークフローの迅速化、メモリ使用量の削減、コードアーキテクチャの単純化にも役立ちます。

このガイドでは、ScriptableObject を実制作にデプロイする際に役立つ、プロの開発者からのヒントとコツをまとめています。具体的なデザインパターンへの適用方法や、よくある落とし穴を回避する方法の例も含まれています。

ScriptableObject はデータとロジックを分離することで、クリーンなコーディング手法の促進に役立ちます。つまり、意図しない副作用を引き起こさずに変更を加えやすくなり、テストのしやすさとモジュール性が向上します。

ScriptableObject はエディターでインタラクティブに操作できるため、プログラマーとプログラマー以外の人 (アーティストやデザイナーなど) とのコラボレーションに特に役立ちます。このガイドでは、ScriptableObject を活用するいくつかのテクニックと方法について説明します。既存のワークフローの補完や、プロジェクト設定の合理化のお役に立てば幸いです。

ゲームアーキテクチャの陰の立役者、ScriptableObject について探求してみましょう。

ScriptableObject とは

ScriptableObject は、ゲームオブジェクトインスタンスではない Unity のオブジェクトです。これを使用すると、MonoBehaviour よりもオーバーヘッドが少ない、独自の変数とメソッドを持つカスタムクラスを作成できます。

ScriptableObject は Transform を持たず、Scene Hierarchy には属していません。その代わり、マテリアルや 3D モデルと同様に、プロジェクトレベルのアセットとして存在します。

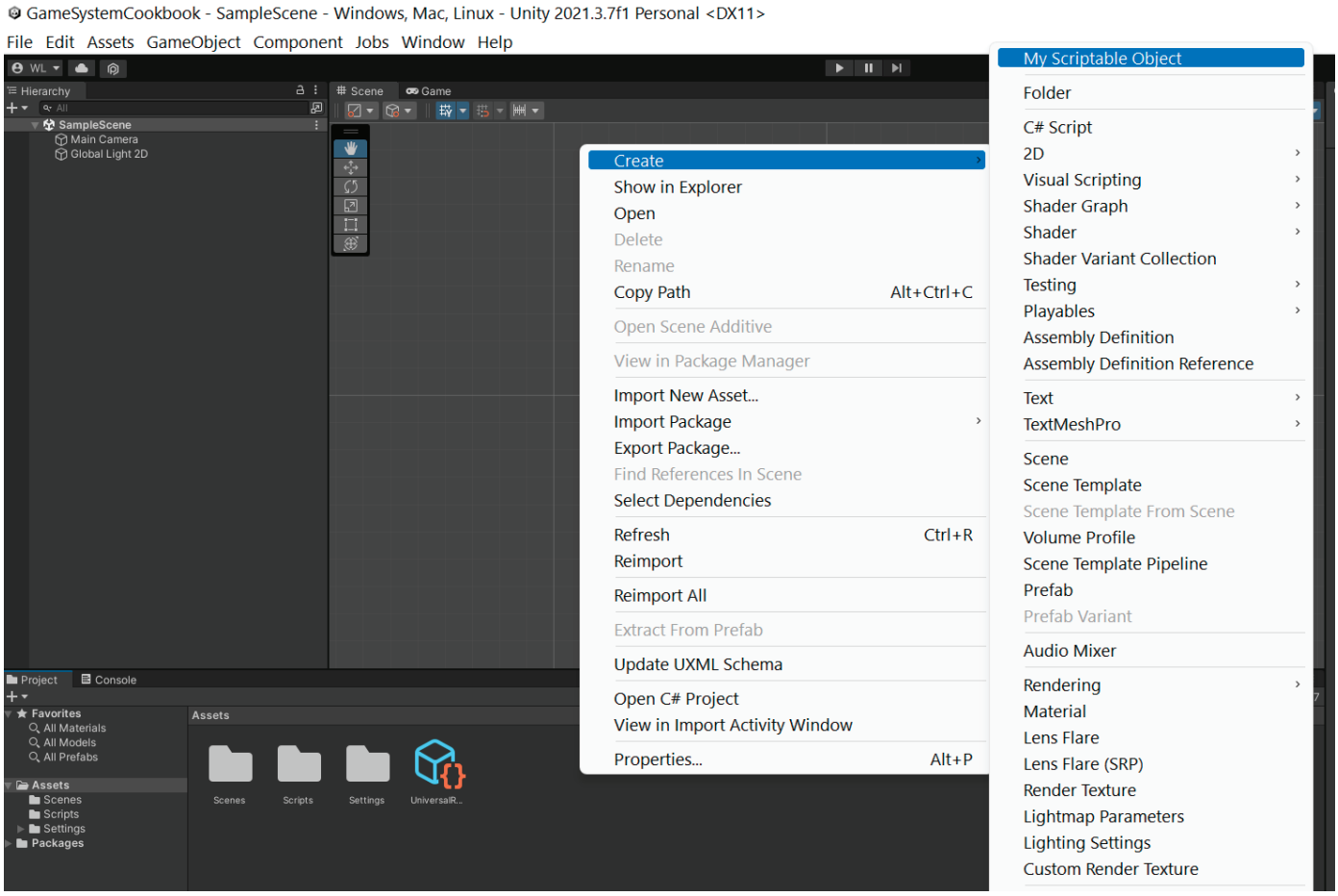
ScriptableObject は以下のように宣言できます。

```
[CreateAssetMenu(fileName="MyScriptableObject")]
public class MyScriptableObject:ScriptableObject
{
        public int someVariable;
}
```

上記のコードでわかるように、MonoBehaviour から派生するのではなく、ScriptableObject から継承します。

これをゲームオブジェクトに直接使用することはできません。代わりに、CreateAssetMenu 属性によって、メニューに追加のアクションが用意されています。

Assets > Create > MyScriptableObject を選択する (または Project ウィンドウで右クリックする) と、ScriptableObject クラスからカスタムアセットをインスタンス化できます。



ScriptableObject の作成

他のスクリプトは、フィールドを使用して、この ScriptableObject アセットを任意のシーンから参照できます。

```
public class MyMonoBehaviour :MonoBehaviour
{
    public ScriptableObject soInstance;
}
```

ScriptableObject は、ランタイムに変更する必要がないものに対して特に有用です。

Unity は ScriptableObject を第一級オブジェクトとして扱うため、以下のことが可能です:

— 変数に格納する

— ランタイムに動的に作成または破棄する

— 引数として渡す

— メソッドから返す

— データ構造に含める

— シリアル化/デシリアライズする

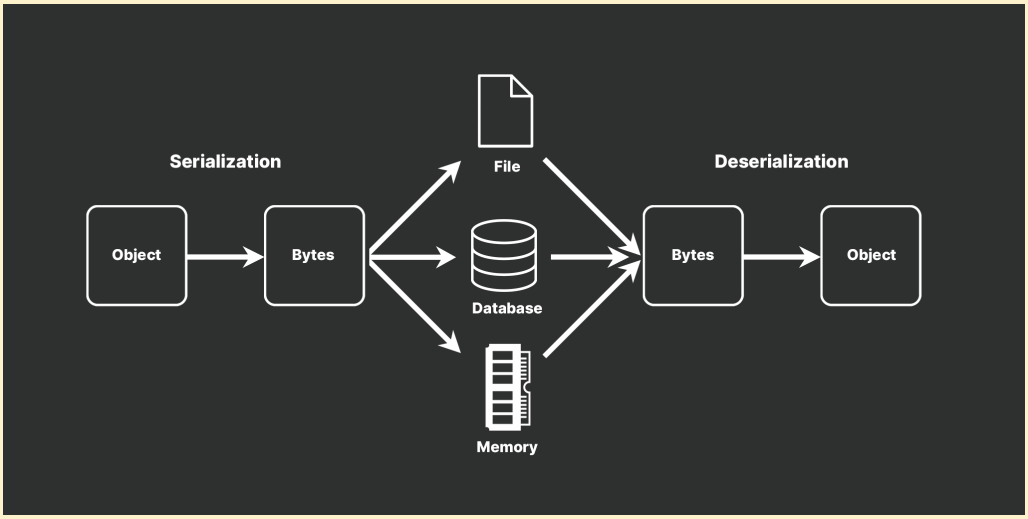
最後の箇条書きは、ScriptableObject の主要な機能の 1 つである、Inspector への表示機能に焦点を当てています。つまり、ScriptableObject のフィールドはエディター内で読みやすく、変更しやすいということです。

ランタイムに ScriptableObject の値を変更すると、ゲームアプリケーションが即座に更新されます。再生モードを終了すると、これらの値は維持されます。そのため、コードを書かずに ゲーム設定のバランス を取る必要があるゲームデザイナーにとって便利です。さらに、ScriptableObject は多くの場合、アプリケーションの実行中に変更を保存できます。これには、MonoBehaviour を単独で使用する場合と比べていくつかの利点があります。

シリアル化

シリアル化 は、データ構造やオブジェクトの状態を、後で保存および再構築できる形式へと変換するための自動プロセスです。Unity のシリアル化バックエンドは、メモリに分散しているデータを取得し、それを順番に配置します。

再構成されたデータストリームは、データベース、ファイル、またはメモリに格納できます。
“デシリアライズ” はその逆のプロセスです。



シリアル化は、オブジェクトをバイトのストリームに変換して格納したり、メモリ、データベース、またはファイルに送信したりするプロセスです。

C# 開発ではメモリレイアウトは見落としがちですが、シリアル化を利用する Unity のビルトイン機能をいくつか認識しておく必要があります:

— **保存と読み込み:.unity** シーンファイルをテキストエディターで開き、Unity が “テキスト形式でシリアル化するよう” 設定すると、シリアライザーは YAML バックエンドで実行されます。

— **Inspector ウィンドウ:**このインターフェースは、C# API と通信して、検査しているものの値を特定することはありません。代わりに、オブジェクトに自身をシリアル化するよう要求し、シリアル化されたデータを表示します。

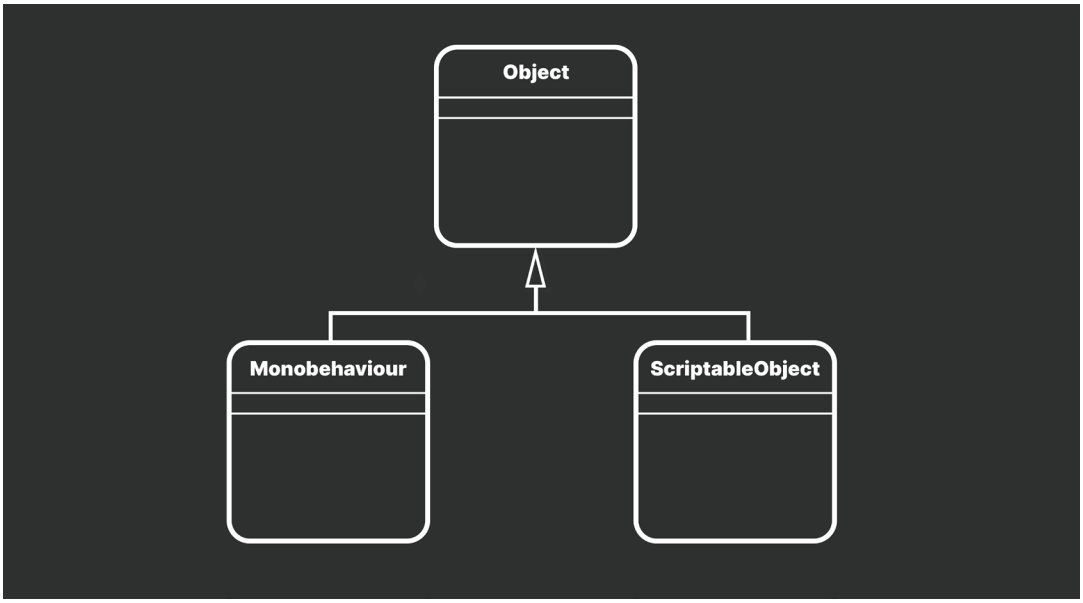
— **プレハブ:**内部的には、プレハブ はゲームオブジェクトとコンポーネントのシリアル化されたデータストリームです。プレハブは、シリアル化されたデータの上に変更を加える必要があるリストです。

— **インスタンス化**:プレハブ (またはシーン内に存在するゲームオブジェクト) をインスタンス化するときは、オブジェクトをシリアル化し、新しいオブジェクトを作成してから、その新しいオブジェクトにデータをデシリアライズします。

Unity におけるシリアル化の詳細については、こちらのブログ記事 または「How Unity’s Serialization system works」をご覧ください。(Youtube翻訳音声・字幕推奨)

ScriptableObject と MonoBehaviour の比較

ScriptableObject は一見シンプルです。API はいくつかのメソッドのみを扱います。この場合、これは良いことです。シンプルであるほどミスも減ります。



Unity オブジェクト UML

MonoBehaviour と同様に、ScriptableObject クラスは UnityEngine.Object クラスから派生します。

比較

ScriptableObject を理解する最良の方法は、おそらくその兄弟分である MonoBehaviour と比較することでしょう。この表は、それらの共通点と相違点を示しています。

MonoBehaviour	ScriptableObject
MonoBehaviour と ScriptableObject はどちらもスクリプトです。 MonoBehaviour クラスと ScriptableObject クラスは、UnityEngine.Object から派生します。	
MonoBehaviour は Unity からコールバックを受け取ります。 Update() などの MonoBehaviour の イベント関数 に従ってメソッドに名前を付けて、ゲームエンジンのプレイヤーループに接続しましょう。 例: Start、Awake、Update、OnEnable、OnDisable、OnCollisonEnter	**ScriptableObject は、Update、Start、FixedUpdate など、Unity からのほとんどのライフサイクルコールバックを受け取りません。** ScriptableObject がサポートできるイベント関数は、ランタイムの Awake、OnEnable、OnDestroy、OnDisable などに限られています。また、Inspector から OnValidate と Reset が呼び出されます。 ScriptableObject に別のメソッドを作成することもできますが、プレイヤーループによって自動的に呼び出されることはありません。
MonoBehaviour はランタイムにゲームオブジェクトにアタッチする必要があります。 ランタイムに作成する場合は、AddComponent API を使用してください。	**ScriptableObject は特定のゲームオブジェクトにアタッチされていません。** ScriptableObject はプロジェクトレベルで独自のアセットファイルに保存しましょう。その後、MonoBehaviour またはその他のスクリプトから ScriptableObject アセットを参照してください。
保存する場合は、MonoBehaviour データをシーンとプレハブに保存します。	**各 ScriptableObject インスタンスは、プロジェクトレベルで独自のファイルに保存できます。**
エディターでは、MonoBehaviour 値に対する変更は再生モード終了時にリセットされます。	**エディターで再生モードを終了しても ScriptableObject 値の変更は リセットされません。** **スタンドアロンビルドでは、ランタイムの ScriptableObject 値の変更は 保存されません。**
MonoBehaviour と ScriptableObject はどちらもシリアル化可能で、Inspector で表示できます。	

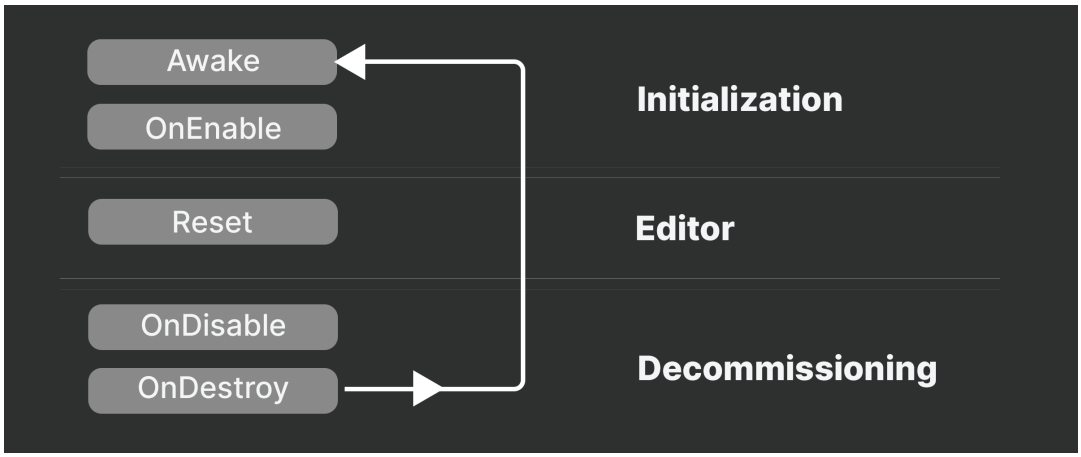
コールバックとメッセージ

ScriptableObject には、MonoBehaviour で使用できるイベント関数のサブセットがあります。ScriptableObject に独自のメソッドを作成することもできますが、自分で呼び出す必要があります。以下の表は、PlayerLoop 内で自動的に呼び出されるメソッドのみを示しています。

イベント関数 **(ランタイム)**	**実行時**
Awake	MonoBehaviour の Awake コールバックと同様に、ScriptableObject スクリプトの開始時に呼び出されます。 ゲームが起動されたとき、またはシーンが ScriptableObject アセットを参照してロードされたときにも実行されます。
OnEnable	ScriptableObject がロードまたはインスタンス化されたときに、Awake コールバックの直後に呼び出されます。 OnEnable は、ScriptableObject.CreateInstance の間、またはスクリプトの再コンパイルが成功した後に実行されます。
OnDisable	ScriptableObject がスコープ外になったときに呼び出されます。これは、ScriptableObject アセットを参照せずにシーンをロードした場合、または ScriptableObject の OnDestroy の直前にシーンをロードした場合に発生します。 また、Unity はスクリプトの再コンパイル前に OnDisable を実行します。再生モードに入ると、OnDisable は OnEnable の直前に実行されます。
OnDestroy	何かが ScriptableObject をエディター内またはコードから削除して破棄するときに呼び出されます。 ランタイムに ScriptableObject を作成した場合、OnDestroy はアプリケーションが終了したとき、またはエディタの再生モードが終了したときにも起動します。注意:オブジェクトのネイティブ C++ 部分のみが破棄されます。詳細については、「ライフサイクルと作成」を参照してください。
エディター専用の関数	**実行時**
OnValidate	OnValidate は、スクリプトがロードされたとき、または Inspector で値が変更されたときに実行されます。これを使用すると、データを一定の範囲内に収めることができます。
Reset	Reset は、Inspector コンテキストメニューのリセットボタンを押すと呼び出されます。

ScriptableObject を破棄するには、エディターから削除するか、ランタイムで Destroy/DestroyImmediate を呼び出します。

ここでは、ScriptableObject のイベント関数とライフサイクルを簡単に紹介します。これを、MonoBehaviour イベント関数の実行順序の先頭から比較してください。



ScriptableObject イベント関数と実行順序

ファイル

MonoBehaviour と ScriptableObject の最大の違いの 1 つは、データの保存方法です。

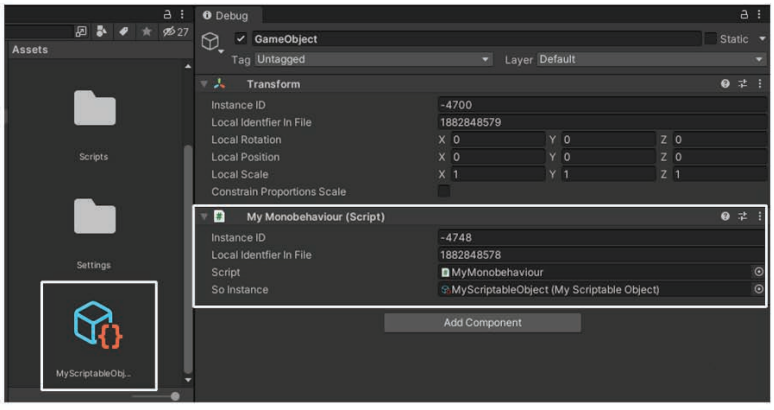
Unity は MonoBehaviour をシーンまたはプレハブファイル内にシリアル化します。保存されるデータには以下が含まれます:

— MonoBehaviour 自体

— アタッチされたゲームオブジェクト

— Transform

— アタッチされたゲームオブジェクト上のその他のコンポーネントと MonoBehaviour

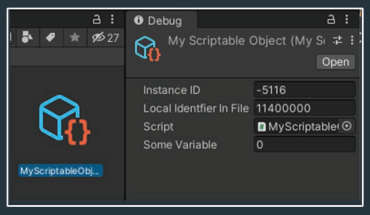


ScriptableObject と MonoBehaviour の比較

これに対し、Unity は ScriptableObject を独自のアセット ファイルに保存します。これらのファイルは、MonoBehaviour よりも小さく、より細分化されています。

モードを使用する場合:テキストを強制表示 (Project Settings > Asset Serialization) ウィンドウを選択すると、ScriptableObject アセットをテキストエディタで開くことができます。以下のように表示されます:

```
%YAML 1.1
%TAG !u! tag:unity3d.com,2011:
--- !u!114 &11400000
MonoBehaviour:
  m_ObjectHideFlags: 0
  m_CorrespondingSourceObject: {fileID: 0}
  m_PrefabInstance: {fileID: 0}
  m_PrefabAsset: {fileID: 0}
  m_GameObject: {fileID: 0}
  m_Enabled: 1
  m_EditorHideFlags: 0
  m_Script: {fileID: 11500000, guid: 3fc7a749f692b4f41923450f022d8428, type: 3}
  m_Name: MyScriptableObject
  m_EditorClassIdentifier:
  someVariable: 0

```



テキストとしてシリアル化された ScriptableObject インスタンス

YAML はマークアップ言語ではありません

Unityは、YAML 仕様のサブセットを実装する高パフォーマンスのシリアル化ライブラリを使用します。XML と JSON に関連する、軽量で読みやすい言語です。

YAML では、データはネスト状の要素の階層として整理されます。各オブジェクトには、クラス ID、ファイル ID、オブジェクト型があります。ScriptableObject は、独自のオブジェクトタイプを定義するのではなく、“MonoBehaviour” をオブジェクト型として使用します。

YAML のオブジェクトヘッダー

各オブジェクトの下には、キーと値のペアで表される、シリアル化されたプロパティがあります。

詳細については、ブログ記事「Unity のシリアライズ言語 YAML を理解する」を参照してください。

作成とライフサイクル

ScriptableObject のライフサイクルは、プロジェクト内の他のアセット (マテリアル、テクスチャなど) のライフサイクルと似ています。

前の例と同様に、スクリプトに [CreateAssetMenu] 属性を適用して、エディターにカスタムメニューアクションを追加します。オプションで、デフォルトの fileName またはメニュー項目の順序を指定できます。次のコードは、ScriptableObject アセットを作成する最も一般的な方法です。

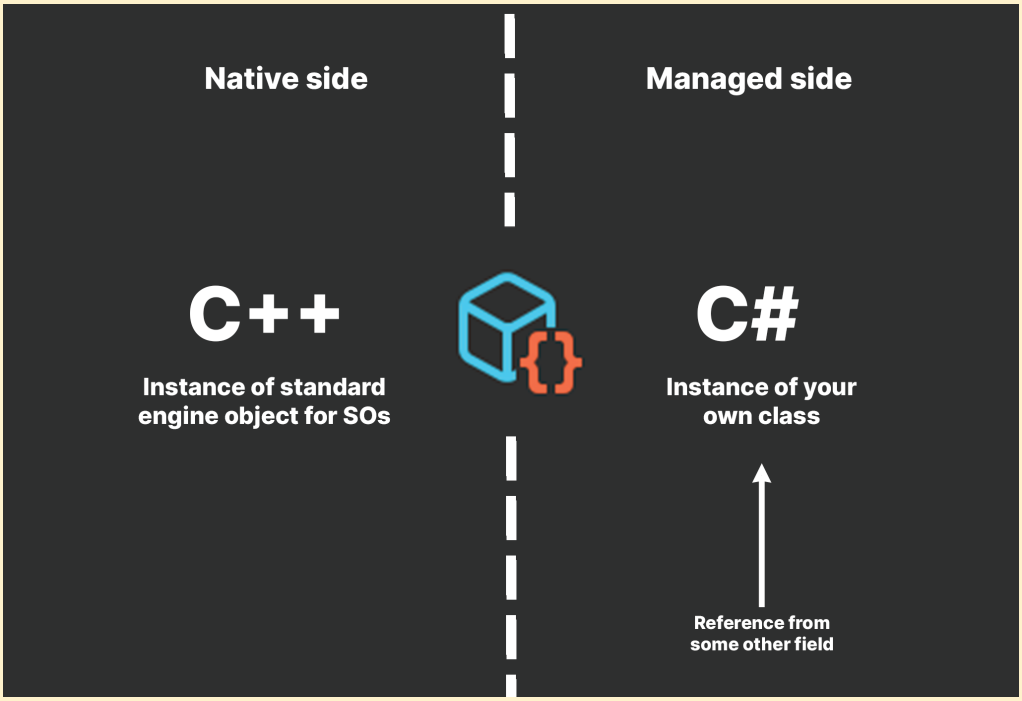
```
[CreateAssetMenu(fileName="MyScriptableObject"]
public class MyScriptableObject:ScriptableObject
{
        public int SomeVar;
}
```

ランタイムに ScriptableObject インスタンスを作成する必要がある場合は、静的 CreateInstance メソッドを呼び出すことができます:

```
ScriptableObject.CreateInstance<MyScriptableObjectClass>();
```

ScriptableObject の破棄

他の Unity オブジェクトと同様に、ScriptableObject はネイティブ C++ 部分と C# マネージ部分で構成されます。ネイティブ C++ 部分は直接破棄できますが、マネージ部分はアセットガベージコレクター（GC）によってクリアされるまで保持されます。GC のクリーンアップは、シーンを変更するか、Resources.UnloadUnusedAssets を呼び出した場合に発生します。



ScriptableObject には、ネイティブ側とマネージ側の両方があります。

ガベージコレクションの遅延を回避するには、ScriptableObject アセットへの参照を明示的に null に設定しましょう。

注意:これは、Destroy または DestroyImmediate を呼び出す前に行うことが重要です。そのようにしない場合、オブジェクトへの参照は、実際には null でなくても、エディターで名目上 “null” と表示されることがあります。GC クリーンアップは、ScriptableObject への参照がなくなったときにだけ実行されます。

ScriptableObject を作成および破棄する方法を習得したら、ゲームアプリケーションにおける創造的な使い方について探ってみましょう。

データコンテナ

ScriptableObject の最も一般的な用途は、共有された静的データ、特にランタイムで変更されないゲーム設定データのデータコンテナとしての使用です。

ScriptableObject の典型的なユースケースには、以下のようなものがあります。

— アイテムの種類、アイコン、レア度、エフェクトなどのインベントリ

— 敵、プレイヤー、アイテムの統計 (体力、ダメージ、スピード、AI パラメーターなど)

— 足音、UI サウンド、アンビエントループ用のオーディオクリップグループなどのオーディオコレクション

— 難易度設定、進行曲線、スポーンテーブルなどの設定ファイル

— 会話データ

ランタイムにこのデータを MonoBehaviour に保存することもできますが、これは非効率的な場合があります。MonoBehaviours は、ゲームオブジェクト (デフォルトではTransform も) をホストとして動作させる必要があるため、余分なオーバーヘッドが発生します。つまり、1 つの値を保存する前に、多くの未使用データを作成する必要があります。

ご自分の目で確かめるために、空の MonoBehaviour を含む新しいゲームオブジェクトを生成してください。次に、シリアル化されたオブジェクトをテキストエディターで開くと、以下のようになります。

```
GameObject:
  m_ObjectHideFlags: 0
  m_CorrespondingSourceObject: {fileID: 0}
  m_PrefabInstance: {fileID: 0}
  m_PrefabAsset: {fileID: 0}
  serializedVersion: 6
  m_Component:
  - component: {fileID: 7467558130563611466}
  - component: {fileID: 2006805663113952451}
  m_Layer: 0
  m_Name: GameObject
  m_TagString: Untagged
  m_Icon: {fileID: 0}
  m_NavMeshLayer: 0
  m_StaticEditorFlags: 0
  m_IsActive: 1
--- !u!4 &7467558130563611466
Transform:
  m_ObjectHideFlags: 0
  m_CorrespondingSourceObject: {fileID: 0}
  m_PrefabInstance: {fileID: 0}
  m_PrefabAsset: {fileID: 0}
  m_GameObject: {fileID: 338842853706088653}
  m_LocalRotation: {x: 0, y: 0, z: 0, w: 1}
  m_LocalPosition: {x: -1.1780801, y: -2.6573246, z: -0.01486098}
  m_LocalScale: {x: 1, y: 1, z: 1}
  m_ConstrainProportionsScale: 0
  m_Children: []
  m_Father: {fileID: 0}
  m_RootOrder: 0
  m_LocalEulerAnglesHint: {x: 0, y: 0, z: 0}
--- !u!114 &2006805663113952451
MonoBehaviour:
  m_ObjectHideFlags: 0
  m_CorrespondingSourceObject: {fileID: 0}
  m_PrefabInstance: {fileID: 0}
  m_PrefabAsset: {fileID: 0}
  m_GameObject: {fileID: 338842853706088653}
  m_Enabled: 1
  m_EditorHideFlags: 0
  m_Script: {fileID: 11500000, guid: 4d1457d233f8d479795a30f859537d5f,
  m_Name:
  m_EditorClassIdentifier:
```

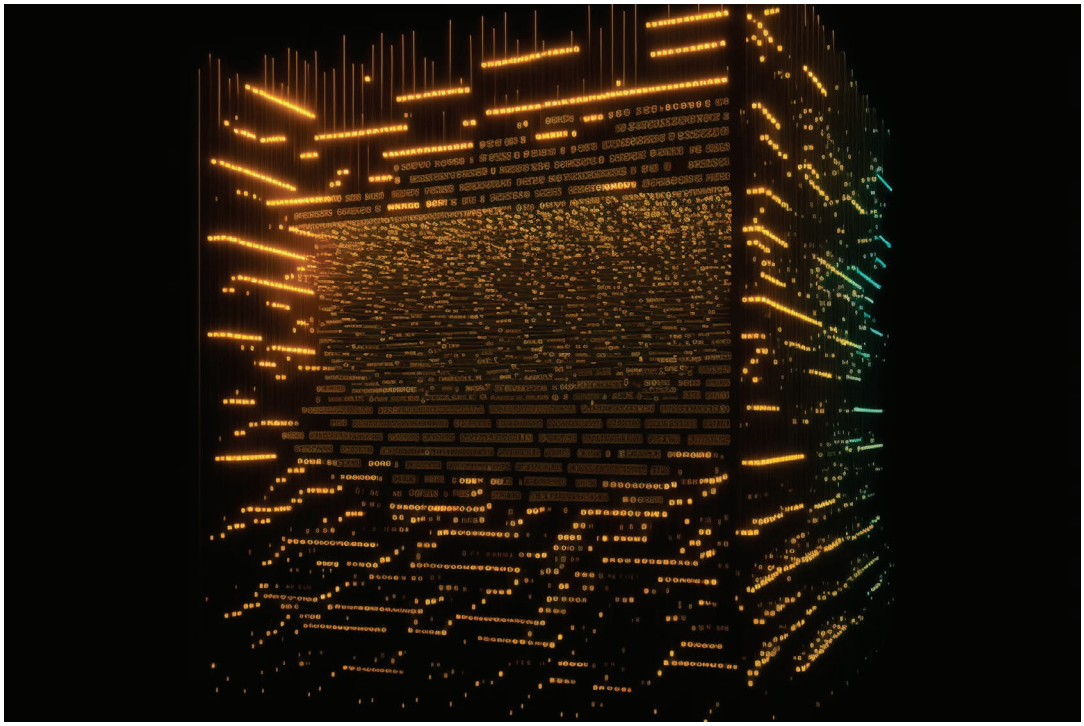
基本的な MonoBehaviour を含む新しいゲームオブジェクト

これを空の ScriptableObject と比較すると、驚くほど無駄がないものであることがわかります。

```
MonoBehaviour:
  m_ObjectHideFlags: 0
  m_CorrespondingSourceObject: {fileID: 0}
  m_PrefabInstance: {fileID: 0}
  m_PrefabAsset: {fileID: 0}
  m_GameObject: {fileID: 0}
  m_Enabled: 1
  m_EditorHideFlags: 0
  m_Script: {fileID: 11500000, guid: a2d85be56c1ae45d69604547c4544ba5,
  m_Name: PlayerID
  m_EditorClassIdentifier:
```

ScriptableObject はデータ保存のオーバーヘッドを削減します。

ScriptableObject は、このメモリフットプリントを縮小し、ゲームオブジェクトと Transform を削除します。これは大規模な商用プロジェクトにおいては有意義なことです。また、ScriptableObject にはプロジェクトレベルのデータも保存されます。これは特に、複数のシーンから同じデータにアクセスする必要がある場合に有用です。



MonoBehaviour の追加データは、最初はアプリケーションのパフォーマンスに影響しないかもしれませんが、ゲームが商用規模に成長し、オブジェクトが大幅に増えると、その影響が目立つようになります。

ScriptableObject データと永続データの比較

ScriptableObject がデータコンテナと呼ばれる場合、これは通常、ランタイムに永続的に変更を加える必要のない静的または共有設定データを指します。

ScriptableObject データに対する変更は、マテリアルやプレハブの変更と同様にエディター内でも保持されますが、アプリケーションビルドのランタイムにはこれらの変更は保存されません。ゲームプレイ中に ScriptableObject インスタンスに加えられた変更はメモリにのみ存在し、セッション終了時に失われます。

あるセッションから保存し、別のセッションにロードする必要がある永続データは通常、異なるファイル形式 (JSON、XML、MessagePack、プロトコルバッファなど) で保存されます。詳細については、後述の「デュアルシリアル化」を参照してください。

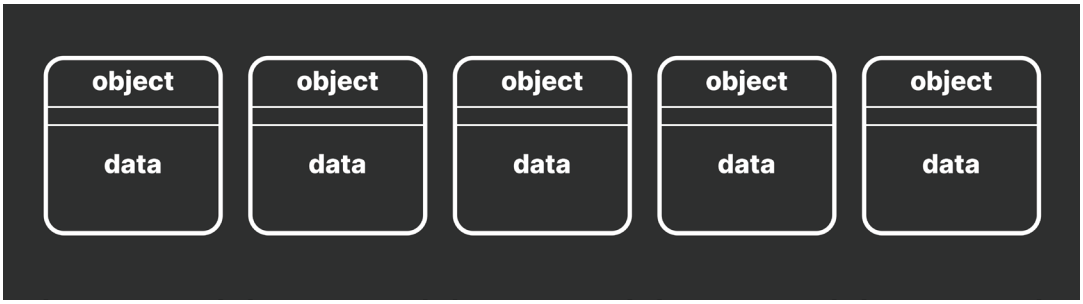
ゲームビルドのランタイムに ScriptableObject データ (ScriptableObject 変数やランタイムセットなど) を変更することは可能ですが、これらの変更は一時的なものです。新しいゲームセッションを開始すると、ScriptableObject データがビルド時の元の状態に戻ります。

ScriptableObject データを "読み取り専用" の永続データと考えてください。永続データは “読み取り/書き込み用” で、外部に保存する必要があります。

重複データの削減

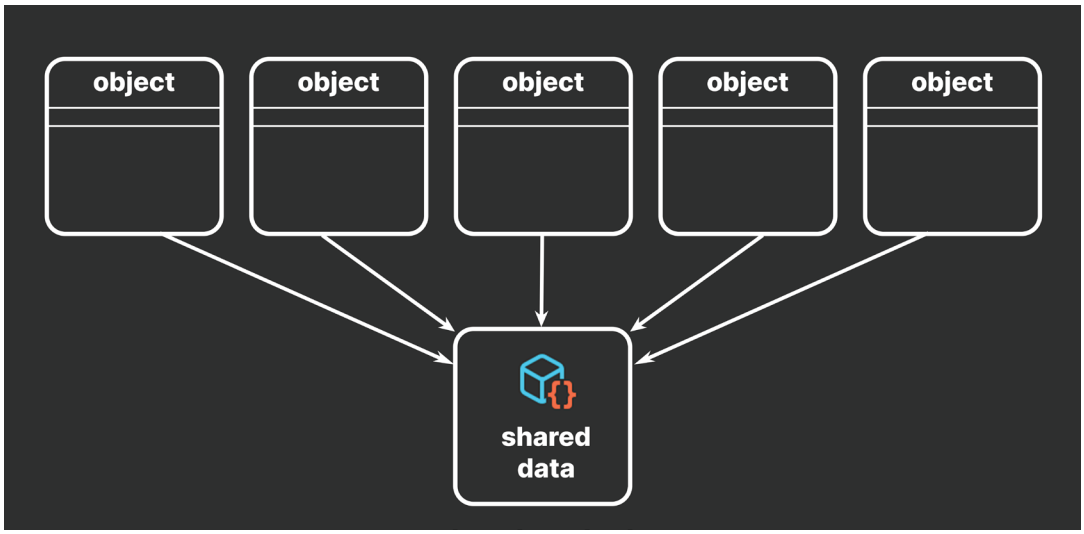
カスタムの MonoBehaviour を含むゲームオブジェクトが 1000 個あり、それぞれに複数のフィールドがあるとします。

各コンポーネントがこれらの値の独自のコピーを保持している場合、メモリ内に大量のデータが重複しています。これは、特にデータがインスタンス間で同一であり、ランタイムに変更されない場合は非効率的です。



同一のローカルデータを含むオブジェクトが多数あると、パフォーマンスの非効率性につながります。

この静的データを複製する代わりに、ScriptableObject にファネルできます。すると、1000 個のオブジェクトそれぞれがこの共有データアセットを参照できます。各オブジェクトは、データ自体をコピーするのではなく、データへの参照を保存します。



ScriptableObject を介してデータを共有する多数のオブジェクト

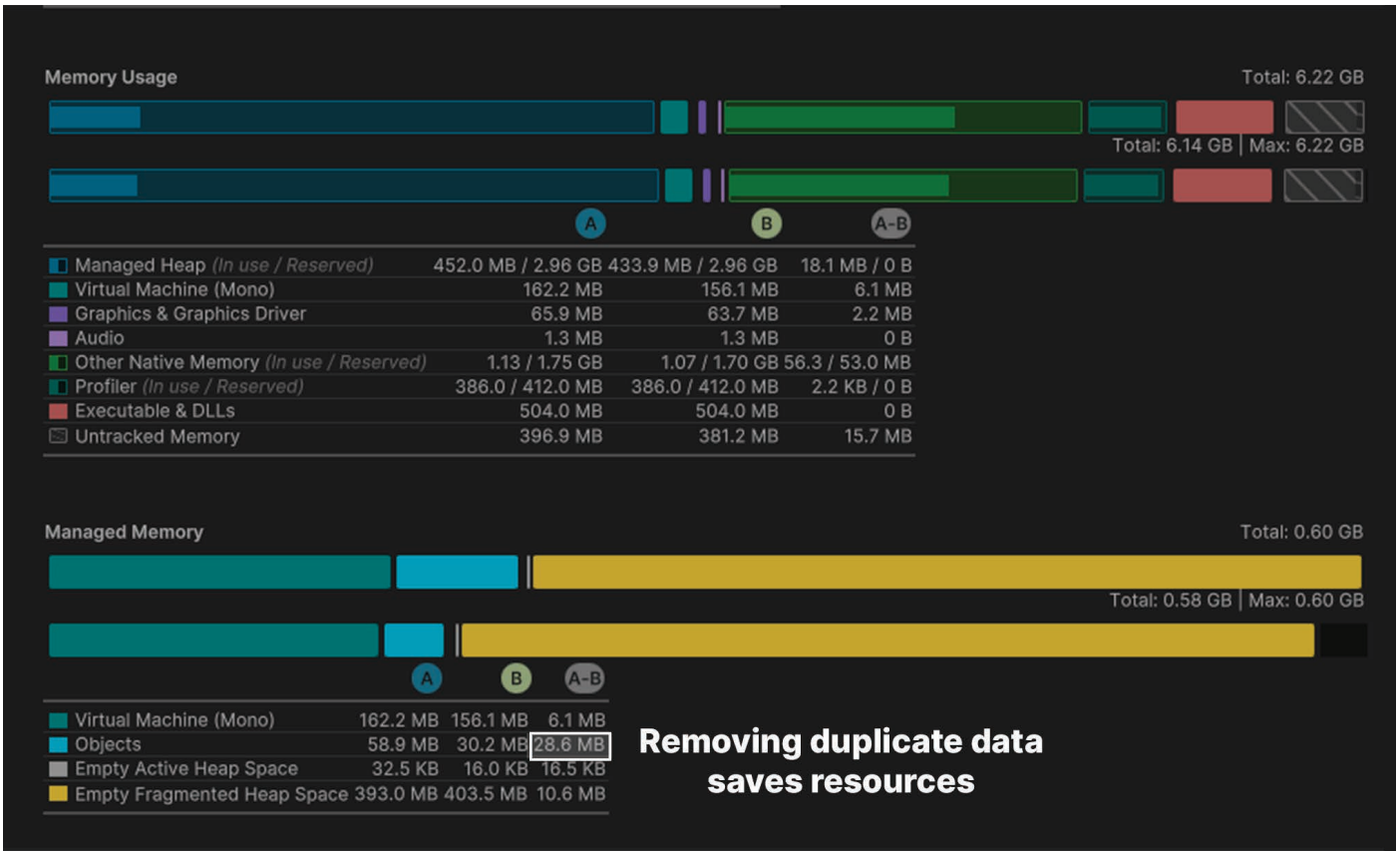
ソフトウェア設計では、これはフライウェイト (Flyweight) パターンと呼ばれる最適化です。ScriptableObject を使用してこのようにコードを再構築すれば、多くの値をコピーする必要がなくなり、メモリフットプリントも削減できます。

デザインパターン

デザインパターンは、開発者がより保守しやすく柔軟なコードを作成するために役立つので、頻繁に変化するゲーム開発の世界では有用です。

SOLID の原則とデザインパターンの詳細については、無料の *ゲームプログラミングのパターンを活用してコードをレベルアップさせる* をダウンロードしてください。

データの完全なコピーではなく、ScriptableObject への参照を使用すると、コストは比較的小さくなります。スケールアップすると、データを複製しないことによるメモリの節約量が大幅に増えます。



Memory Profiler で、重複データ (A) と共有データ (B) のメモリ使用量を比較します。

この方法で、大量の共有データを保存できます。ScriptableObject の使用を検討すべき用途:

— エディターセッション中のデータの保存

— ランタイムで使用するアセットとしてのデータの保存

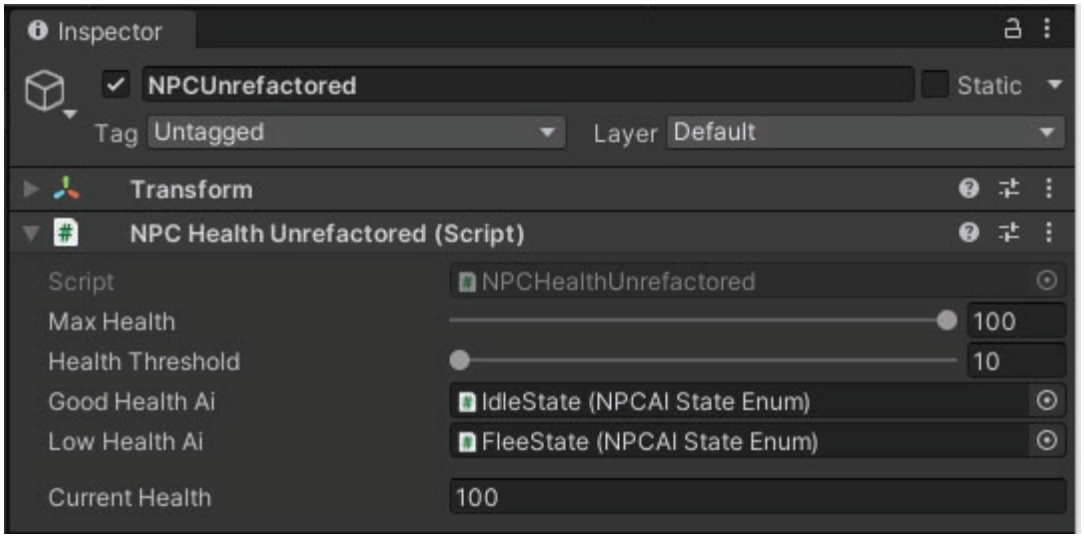
MonoBehaviour とは異なり、ScriptableObject はゲームオブジェクトにアタッチできません。代わりに、アセットとしてプロジェクトに保存されます。これは、MonoBehaviour で変更されないデータを使用するプレハブがある場合に特に便利です。

リファクタリングの例

NPC の体力を制御する MonoBehaviour について考えてみましょう。そのクラスを次のように定義できます。

```
public class NPCHealthUnrefactored :MonoBehaviour
{
    [Range(10, 100)]
    public int MaxHealth;
    [Range(10, 100)]
    public int HealthThreshold;


    public int CurrentHealth;
}
```

これは有効ですが、ランタイムで変更されないデータがあります。NPCHealthUnrefactored がアタッチされたオブジェクトが多数ある場合、不要な重複データが多くなる可能性があります。



リファクタリング前の NPCHealth MonoBehaviour

変更する必要がないデータは、ScriptableObject に移動できます。

```
[CreateAssetMenu(fileName="NPCConfig")]
public class NPCConfigSO :ScriptableObject
{
    [Range(10, 100)]
    public int maxHealth;

    [Range(10, 100)]
    public int healthThreshold;


}
```

メニューアクションを設定するには、CreateAssetMenu 属性を使用します。オプションで、デフォルトの fileName またはメニュー項目の順序を指定できます。

このガイドのコード規則

このガイドのコードサンプルの多くは、説明しやすくするために簡略化されています (例:public フィールドなど)。

本番環境では、カプセル化のためと柔軟性を高めるために、private フィールドと public プロパティを使用します。SerializeField 属性を private フィールドに適用すると、エディターの Inspector に表示されます。

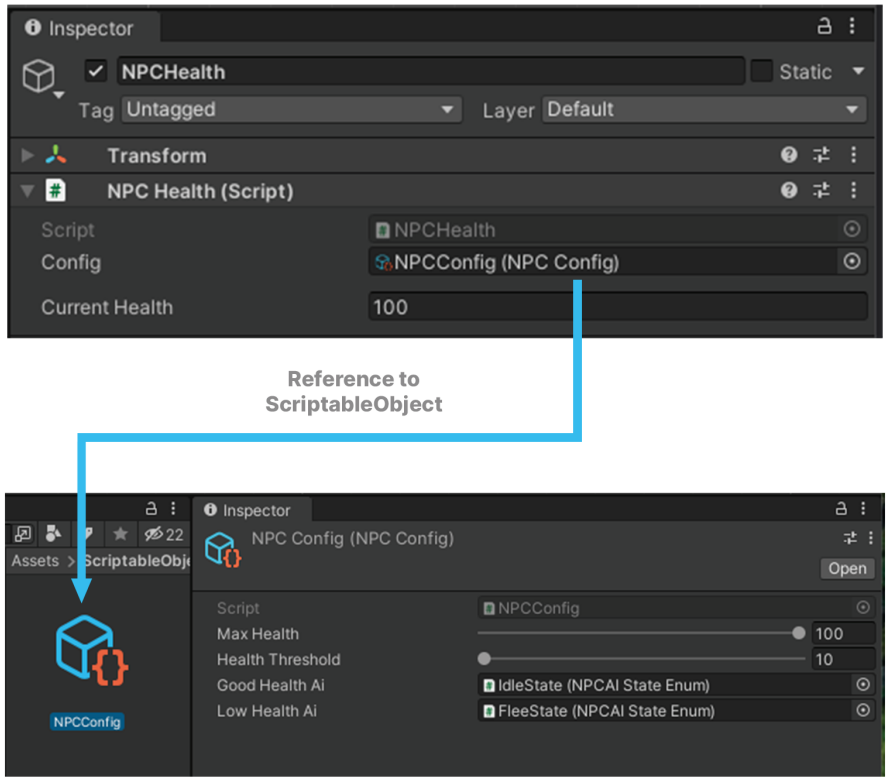
また、命名規則は、ScriptableObject のスクリプトと MonoBehaviour を区別するのに役立ちます。これを実現する 1 つの方法は、クラス名の最後に “Data” または “SO” のサフィックスを追加することです。これは必須ではありませんが、プロジェクトを整理し、曖昧さを減らすのに役立ちます。

コードベースの拡大に合わせて、コードのスタイルガイドを維持し、それに従うことをおすすめします。詳細については、*C# スタイルガイドを使用して、クリーンでスケーラブルなゲームコードを書く* を参照してください。

次に、リファクタリングされた NPCHealth コンポーネントによって次のように簡略化できます。

```
public class NPCHealth:MonoBehaviour
{
    // ScriptableObject への参照
    public NPCConfigSO Config;

    public int CurrentHealth;
}
```

MonoBehaviour に、この新しい ScriptableObject への参照が含まれるようになりました。Inspector では、データが分割されている点を除き、リファクタリング後の外観はすべて似ています。

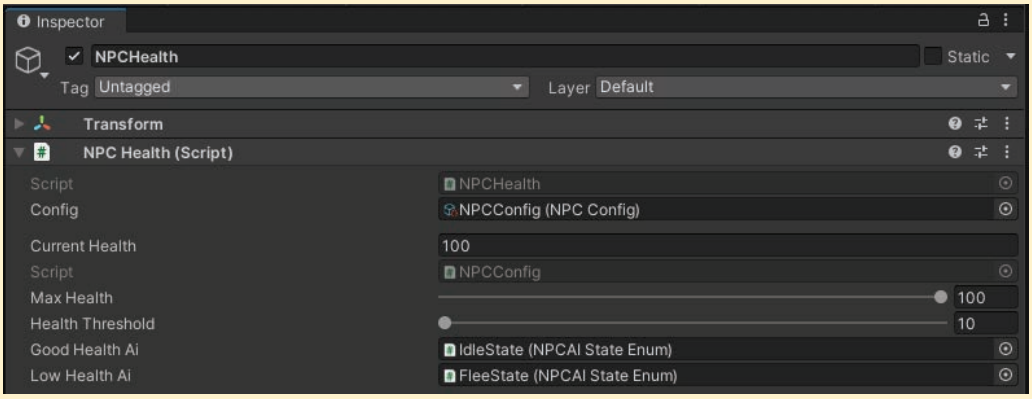


リファクタリングにより、MonoBehaviour と ScriptableObject の間でデータが分割される

カスタムインスペクター

データを ScriptableObject に分離する場合、ScriptableObject アセットとそれを参照する MonoBehaviour の 2 つの場所にデータが含まれます。

MonoBehaviour の操作をより簡単にするには、カスタムエディターの作成を検討してください。NPCHealth の例を次に示します。



カスタムインスペクターは、MonoBehaviour 内の ScriptableObject の変数を表示します。

これにより、NPCConfig の変数を、MonoBehaviour の他のプロパティーと一緒に検査できます。元の NPCHealth プレハブを選択すると、両方のオブジェクトで値を簡単に編集できます。

カスタムエディターを作成するのに必要なのは、わずか数行のコードです。

— Editor から新しいクラスを派生させ、“Editor” という名前のフォルダーに保存します。CustomEditor 属性に NPCHealth 型を適用します。

— NPCConfig ScriptableObject 用の一時エディターを確保します。

— OnInspectorGUI で、NPCHealth コンポーネントのエディターを作成します。

— 基本クラスと新しいカスタムインスペクターからインスペクターを描画します。

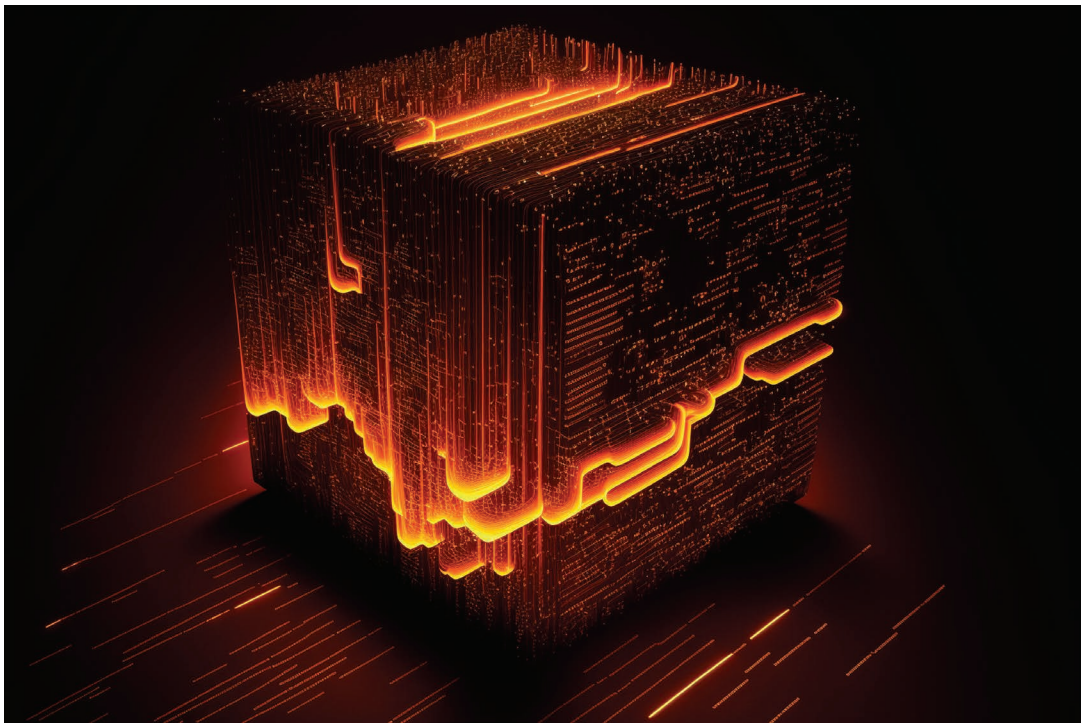
```
using UnityEditor;

[CustomEditor(typeof(NPCHealth))]
public class NPCHealthEditor :Editor
{
   private Editor editorInstance;

   private void OnEnable()
   {
      // エディターのインスタンスをリセット
      editorInstance = null;
   }

   public override void OnInspectorGUI()
   {
      // 検査対象のターゲットコンポーネント
      NPCHealth npcHealth = (NPCHealth)target;

      if (editorInstance == null)
         editorInstance = Editor.CreateEditor(npcHealth.config);

      // MonoBehaviour の変数を表示
      base.OnInspectorGUI();
```

```
        // ScriptableObject Inspector の描画
        editorInstance.DrawDefaultInspector();
    }
}
```

この例は、カスタムプロパティドロワーとエディター属性で拡張できます。これにより、ScriptableObject を扱う際のユーザー体験も向上させることができます。

アーキテクチャ上のメリット

ScriptableObject を使用すると、共有データと非共有データを明確に分離できます。共有データは ScriptableObject に保存される一方で、ゲームオブジェクトインスタンスに固有の動的なものは MonoBehaviour 内に残ります。しかし、ScriptableObject を使用したアーキテクチャは、単にメモリを節約するだけではありません。



コードアーキテクチャを再構築することには、いくつかの利点があります。

— **設計者はソフトウェア開発者からより独立して作業できます:** データとロジックを 1 つの MonoBehaviour に格納すると、開発者とゲームデザイナーが互いの作業領域を踏み越える可能性が生じます。2 人が同じプレハブまたはシーンの異なる部分を変更すると、時間の無駄となるマージ競合が発生します。共有データを小さなファイルやアセットに分割することで、これらの問題を減らすことができます。また、ScriptableObject を使用して構築することで、デザイナーは常にプログラマーに頼ることなくゲームプレイをビルドできます。

データを共有する際には、チーム間で明確なワークフローを定義する準備をしておいてください。コミュニケーションを円滑にし、境界を設けると、問題の発生を防ぐのに役立ちます。追加のエラーチェックやデータ検証が必要な場合もあります (例えば、範囲属性や OnValidate を使用して不正な値を防ぐ事など)。

— **共有データの編集はより高速で、エラーの発生も少なくなります:**共有データへの変更が一度に行われるようになります。例えば、NPC の設定を変更する必要がある場合は、1 か所だけで調整し、すべてのシーンの影響を受けるすべてのコンポーネントに変更を伝播させることができます。

これにより、多数の個別のゲームオブジェクトを手作業で大量に編集することによる潜在的なエラーが減少します。ScriptableObject にデータをオフロードすることはバージョン管理にも役立ち、チームメイトが同じシーンやプレハブで作業する際のマージの競合を防ぐことができます。

— **再生モードでのゲームプレイの調整の保存:**エディターの再生モードで、デザイナーがゲームプレイや設定を試すことができます。ただし、再生モードを終了すると Unity がシーンの一時コピーを破棄するため、MonoBehaviour に加えた変更は失われます。

ScriptableObject はアセットであるため、Unity が再生モードであるかどうかに関係なく、その値に対する変更は保存されます。これは、ランタイムに調整を行う場合に便利です。

ただし、これらの変更を元に戻す必要がある場合は、この方法も問題になります。Unity Version Control または別のバージョン管理システムに頼ることで、必要に応じていつでも作業を復元できます。詳細については、*プロジェクト整理とバージョン管理のベストプラクティス (Unity 6 版)* のガイドを参照してください。

— **シーンの読み込み時間の短縮:**シーンまたはプレハブを保存するとき、Unity はその内部にあるものをすべてシリアル化します。これには、すべてのゲームオブジェクト、それらのゲームオブジェクトにアタッチされているすべてのコンポーネント、およびすべての public フィールドが含まれます。Unityは、データの重複をチェックせずにこれを行います。

ScriptableObject にデータを移動すると、シーンとプレハブのサイズが小さくなり、読み込みと保存に著しい影響を与える可能性があります。

ScriptableObject 変数

1 つの値だけを表す ScriptableObject を使用して、共有データコンテナをさらに細かくできます。例えば、value という名前の public フィールドを 1 つ保持している、IntVariable という名前の ScriptableObject クラスを作成することが可能です。

```
using UnityEngine;
[CreateAssetMenu(menuName = "Variables/Int", order = 1)]
public class IntVariableSO :ScriptableObject
{
    public int value;
}
```

その後、MonoBehaviour で IntVariable を使用できます。そのようにして PlayerHealth クラスを構成すると、次のようになります:

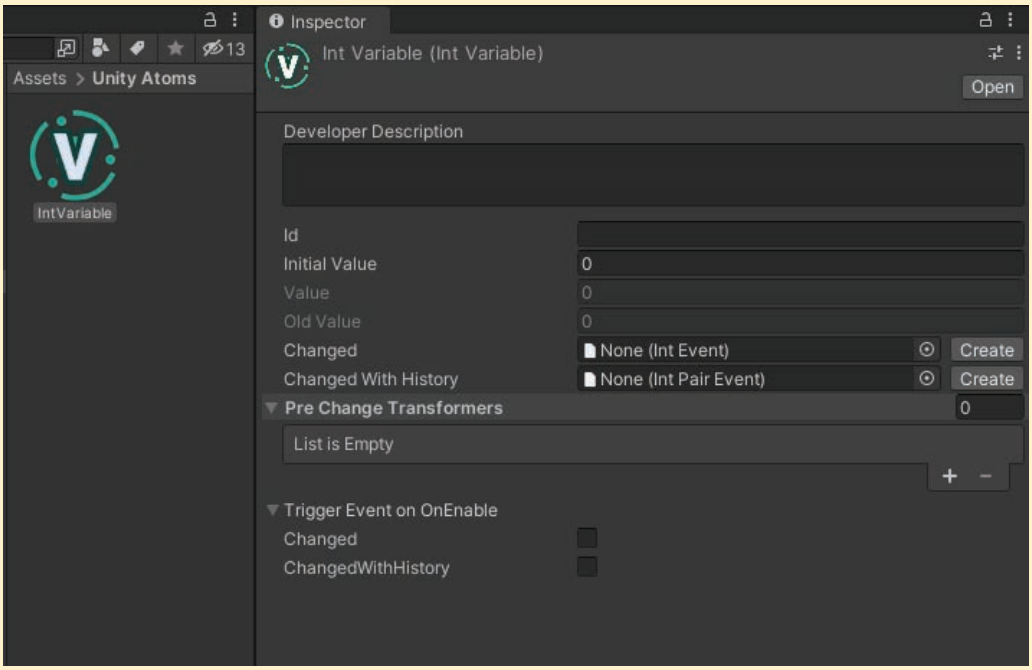
```
public class PlayerHealth :MonoBehaviour
{
    public IntVariableSO health;
}
```

通常、ScriptableObject は変更されない値を保持するものと考えられていますが、ランタイムにこのデータを更新するメソッド (および再生モードを終了するときに初期値にリセットするメソッド) を指定することもできます。このようにして、整数、浮動小数点数、ブーリアンなど、原則的に変数として機能する ScriptableObject を作成できます。

デザイナーは、ゲームロジック用にデータを確保できるため、必要なときに毎回ソフトウェア開発者がデータを準備する必要はありません。ただし、そのためには計画を立てる必要があります。オーサリングのゲームプレイデータの分割方法をデザイナーとともに決定しましょう。

重要なのは、コラボレーションの方法に一定の境界線を設けることです。例えば、プログラミングチームがインベントリシステムで使用するための ScriptableObject の初期設定を行う場合があります。その後、デザインチームはこれらを使用して、各アイテムのゲーム内のステータスや動作を埋めることができます。

エディターのスクリプトを追加すれば、この体験はほぼシームレスなものになりえます。別の可能性として、Inspector 内のフィールドを、ScriptableObject の共有値と定数の間で切り替えられるようにする方法があります。これにより、ゲームデザインチームは、インスタンスごとに ScriptableObject データをオーバーライドする自由度が高くなります。



オープンソースの Unity Atoms プロジェクトの ScriptableObject ベースの IntVariable の例

この動作をご自身のプロジェクトに実装する方法については、Unite Austin のプレゼンテーション「Game Architecture with ScriptableObjects」を参照してください。(Youtube翻訳音声・字幕推奨)

また、オープンソースの Unity Atoms プロジェクトをダウンロードして、ScriptableObject 変数の動作実装を確認することもできます。

デュアルシリアル化

Unity 内でデータをシリアル化する手法を組み合わせることができます。そうすると、ScriptableObject をエディターで操作しながら、そのデータを JSON ファイルや XML ファイルなどの別の場所に保存できます。これにより、各形式の強みを活用できます。

JSON や XMLのようなファイル形式は、ゲームのセーブデータや設定などの永続データの保存に適していますが、エディターでの操作が難しい場合があります。ただし、Unity の外で任意のテキストエディターを使用して簡単に変更できます。

一方、ScriptableObjects はエディターで良好に動作し、簡単なドラッグアンドドロップ操作で差し替えることができます。しかし、Unity の外での変更や、プレイヤーコミュニティ内での共有は容易ではありません。

シリアル化されたフォーマットを混在させることで、レベル編集や改造など、ゲームに新たな可能性が開けます。ビルド時に、スクリプトで他のファイルを ScriptableObject に変換できます。これはプレーンテキストよりも高速に読み込まれます。

一部の機密データ (ゲーム内通貨、アカウント情報など) はサーバーに安全に保管する必要がありますが、ゲームデータの一部をコミュニティに公開することで、ユーザーによる実験用のサンドボックスレベルが実現可能になるため、ゲームプレイが向上する可能性があります。

ゲームレベルのレイアウトを定義する ScriptableObject があるとします。これには、プレハブの配置や開始設定などを定義する多数の Transform が含まれている場合があります。ゲームスクリプトはこのデータを使用して各レベルを組み立てます。

ScriptableObject 内に格納されているゲームの壁と開始位置を想像してみてください。

```
[CreateAssetMenu(fileName ="LevelLayout")]
public class LevelLayout :ScriptableObject
{
    public Vector3[] wallPositions = new Vector3[2];
    public Vector3[] playerPositions = new Vector3[2];
    public Vector3[] goalPositions = new Vector3[2];
    public Vector3 ballPosition;
}
```

これにより、レベルの設定方法を定義できます。レベル管理スクリプトは、LevelLayout オブジェクトからデータを読み取り、プレハブを正しい位置にインスタンス化できます。

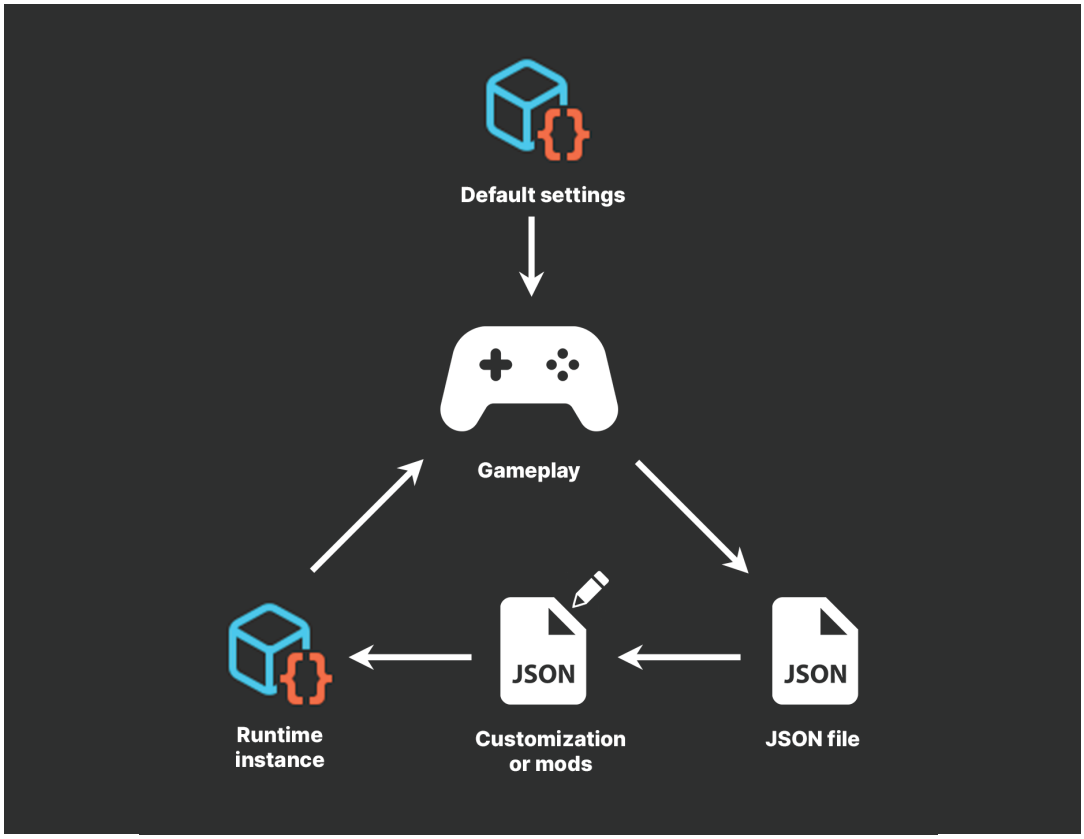
カスタムスクリプトでは、JsonUtility を使用して同じデータをディスクにエクスポートできます。その結果、ユーザーが外部ツールで変更できるテキストファイルがエディター外で作成されます。

変更済みのカスタムレベルをロードする際、ScriptableObject.CreateInstance はランタイムに ScriptableObject を生成します。次に、JSON ファイルからテキストを読み取り、ScriptableObject に入力します。この LoadLevelFromJson サンプルメソッドは、実用例を示しています。

```
using System.IO;

public class LevelManager :MonoBehaviour
{
    public ScriptableObject levelLayout;

    public void LoadLevelFromJson(string jsonFile)
    {
        if (levelLayout == null)
        {
            levelLayout = ScriptableObject.CreateInstance<LevelLayout>();
        }

        var importedFile = File.ReadAllText(jsonFile);
        JsonUtility.FromJsonOverwrite(importedFile, levelLayout);

    }
}
```

カスタムデータによって ScriptableObject の内容が置き換えられ、この外部で変更されたレベルをゲーム内の他のレベルと同じように使用できます。アプリケーション側が違いに気付くことはありません。

どのように動作するのか、サンプルプロジェクト を用いてご自身の目で確かめてみてください。変更された JSON ファイルをロードすると、このカスタマイズされたレベルは ScriptableObject のデフォルトのレベル



データをオーバーライドします。

シリアル化されたフォーマットを組み合わせて柔軟性を高める

注意:JsonUtility を使用して JSON を ScriptableObject にデシリアライズする場合は、FromJsonOverwrite メソッドを使用する必要があります。

新しいオブジェクトを作成して JSON データを読み込む代わりに、JsonUtility では既存のオブジェクトに JSON データを読み込みます。これにより、クラスまたはオブジェクトに格納されている値が、割り当てなしで更新されます。

データを保護

上記の簡単な変更の例は、ScriptableObject の活用例の 1 つです。ただし、変更用のゲームデータを公開する場合は、プレイヤーがアプリケーションの他の部分を改ざんしないように注意する必要があります。

変更してほしくないものを保護する一般的な方法は以下の通りです。

— **暗号化:**暗号化 を使用して、データファイルが容易に読み取られたり変更されたりしないようにします。これにより、ユーザーが重要なデータを変更することをより難しくすることができます。

— **デジタル署名:**フィンガープリントアルゴリズム を使用して、データファイルが改ざんされていないことを確認できます。

— **サーバー側の検証**:サーバーに格納されているデータにゲームが依存している場合は、ゲームで使用される前にサーバー上のデータを確認し、改ざんされていると思われるデータはすべて拒否できます。

単一のアプローチに絶対的な安全策はなく、プレイヤーがゲームにバグや脆弱性を持ち込まないよう、これらの手法を組み合わせて使用するのが一般的に望ましい方法です。

Extendable enum (拡張可能な列挙体) パターン

ゲーム開発では、繰り返し発生する問題や同様の問題を解決するタスクが必要になることがよくあります。幸いなことに、デザインパターン によって、既にそれらを “経験済み” のソフトウェアエンジニアたちの集合知を活用することができます。

デザインパターンは、より大規模でスケーラブルなアプリケーションをビルドするのに役立つ一般的なソリューションです。これにより、コードの可読性が向上し、コードベースがよりクリーンになります。デザインパターンを活用すると、リファクタリングとテストにかかる時間を削減できます。

デザインパターンは、以下のような一般的な問題を解決するためのテンプレートだと思ってください。

— 大量のデータを効率的に保存する

— 異なるゲームシステム間のオブジェクトを通信させる

— ランタイムに動的に動作を切り替える

ScriptableObject は、これらのパターンのいくつかを実装するのに役立ちます。データコンテナとして機能する様子は既に説明しましたが、ScriptableObject は単に値や設定を保存する以上の役割を果たすことができます。次のいくつかのセクションでは、ScriptableObject を使用してデータ保存以外の用途で使用する方法を探ります。

enum のようなカテゴリ

実際、ScriptableObject は何も含まれていなくても有用です。空の ScriptableObject を作成すると、他の ScriptableObject との比較にしか使われない場合でも、それには依然として有用性があることに気づくでしょう。

ゲームアプリケーションで、次のように空の GameItemSO ScriptableObject から多数のアセットを作成するとします。

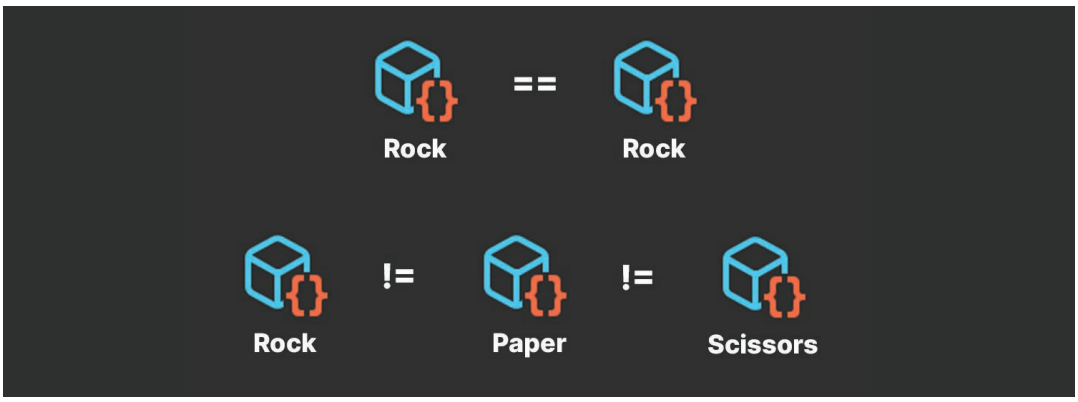
```
Using UnityEngine;


[CreateAssetMenu(fileName="GameItem")]
public class GameItemSO :ScriptableObject
{


}
```

空の ScriptableObject は enum として機能します。

これにより、プロジェクト内で任意の数のアセットを生成できます。データが含まれていない場合でも、ScriptableObject 自体は enum のようなカテゴリやアイテム型を表すことができます。

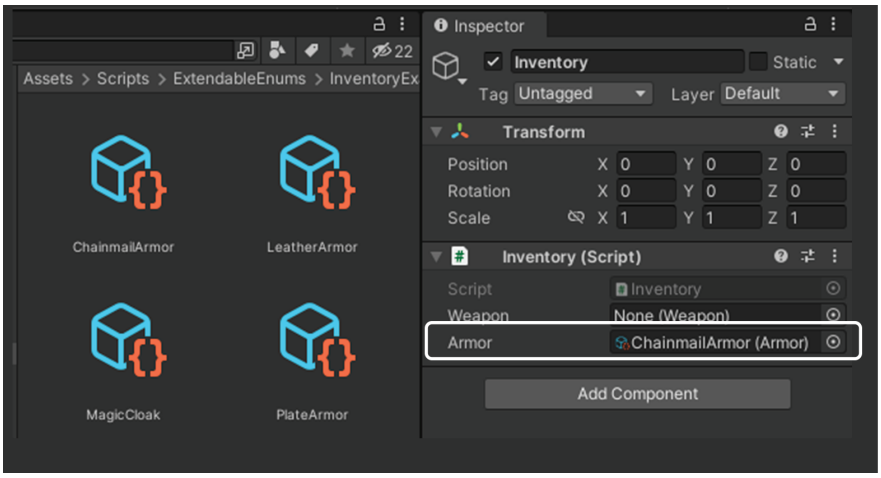
2 つの変数は同じ ScriptableObject を参照していますか。そうであれば、アイテムの型は同じです。そうでない場合は、アイテムの型が異なります。

つまり、特別なダメージ効果 (冷気、熱気、電気、魔法など) や、ゼロサムゲーム におけるじゃんけん指定を定義する ScriptableObject を用意することも可能です。



ScriptableObject の比較

アプリケーションがゲームプレイ内アイテムを装備するためにインベントリシステムを必要とする場合、アイテムタイプや武器スロットを表す ScriptableObject を設定できます。Inspector のフィールドは、ドラッグアンドドロップで設定できるインターフェースとして機能します。



Inventory slots categorized by ScriptableObject

ScriptableObject ベースのカテゴリをドラッグアンドドロップする

アーティストにとって使いやすい UI のおかげで、デザイナーは開発者からの追加サポートなしにゲームプレイデータを修正および拡張できます。デザインチームにゲームプレイデータを維持する手段と責任を与えると、全員が自分の最も得意なことに集中できるようになります。

動作の拡張

ScriptableObject を enum として使用すると、データを追加して拡張したい時に特に有用です。通常の enum とは異なり、ScriptableObject には追加のフィールドとメソッドを含めることができます。

じゃんけんの GameItem を改造したものがこちらです。ScriptableObject アセット自体は引き続き enum のようなカテゴリを定義していますが、今回は空ではありません。

```
public class GameItem :ScriptableObject
{
    public GameItem weakness;
    public bool IsWinner(GameItem other)
    {
        return other.weakness == this;
    }
}
```

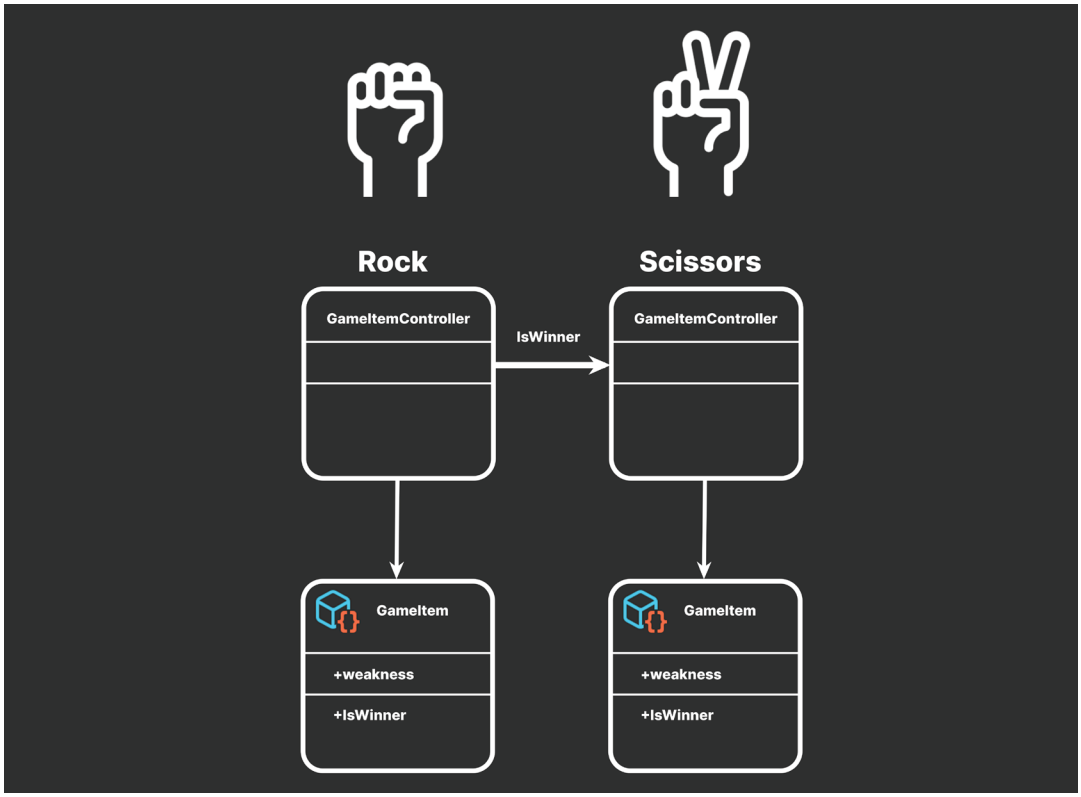
ScriptableObject に、発生しうるインタラクションで勝つ他のアイテムを決定する 弱点 フィールドが追加されました。各 ScriptableObject には、データの保存に加えて、IsWinner の簡単な比較ロジックも含まれています。

その後、ゲームプレイの各アイテムに、特定の ScriptableObject アセットを参照する MonoBehaviour が必要です。この例は、コントローラースクリプトとして機能します。

```
public class GameItemController :MonoBehaviour
{
    // グー、チョキ、パー
    public GameItem gameItem;

    private void OnTriggerEnter(Collider other)
    {
        GameItemController otherController = other.GetComponent<GameItemController>();
        GameItem otherGameItem = otherController.gameItem;

        if (gameItem.IsWinner(otherGameItem))
        {
            Debug.Log(gameItem.name + " beats " + otherGameItem.name);
        }
    }
}
```

ScriptableObject をフィールドとして参照します。OnTriggerEnter で、IsWinner メソッドをチェックして、gameItem が別の gameItem に接触したときにどちらが勝利するかを確認できます。これにより、じゃんけんのようなバトル を発生させる準備が整います。

ScriptableObject は enum とは異なり、簡単に拡張できます。ルックアップテーブルを別に用意したり、新しいデータ配列と関連付ける必要はありません。ロジックを扱うためのフィールドやメソッドを加えるだけです。



比較ロジックを備えた ScriptableObject
ソース:Flaticon

これを従来の enum の維持と比較してください。明示的な番号付けがない enum 値の膨大なリストがある場合、enum の挿入や削除によってそれらの順序が変わる可能性があります。この順序変更により、わずかなバグや意図しない動作が発生する可能性があります。

ScriptableObject ベースの enum にはそのような問題はありません。プロジェクトに追加 (または既存のものを削除) すれば、すべてがうまくいきます。

RPG でアイテムを装備可能にするとします。ScriptableObject にブーリアンフィールドを 1 つ加えれば、それが可能になります。特定のキャラクターが特定のアイテムを持てないようにしたり、一部のアイテムに魔法や特殊能力を付与したい場合も、ScriptableObject ベースの enum でそれを実現できます。

つまり、ゲームデザインを実装する過程でゲームプレイデータを進化させられるのです。初期段階ではフィールドの設定方法を調整する必要がありますが、その後はデザイナーが個別で詳細を記入できます。

パターン：
デリゲートオブジェクト

Unity の ScriptableObject は、単にデータを格納するためだけのものではありません。メソッドを含めることもできます。つまり、処理するもの (ロジック) と使用するもの (データ) の両方を保持できます。

デリゲートとイベントの比較

デリゲートとイベントは C# では密接に関連する概念ですが、それぞれ異なる目的を果たします。デリゲートは、メソッドシグネチャを定義する型です。これにより、メソッドを引数として他のメソッドに渡すことができます。値ではなく、メソッドへの参照を保持する変数と考えてください。

一方、イベントは原則的に特殊なタイプのデリゲートであり、クラス同士が疎結合な方法で通信することを可能にします。イベントについては、オブザーバー (Observer) パターンの章で詳しく説明します。

デリゲートとイベントの違いに関する一般的な情報については、C# のデリゲートとイベントの区別 を参照してください。

特定のタスクを実行する必要がある場合、それらのタスクを実行するアルゴリズムを独自のオブジェクトにカプセル化するという考え方です。オリジナルの ギャングオブフォー (Gang of Four、GoF) は、この一般的なデザインを 戦略パターン と呼んでいます。

迷路を通るルートを計算する経路検索オブジェクトが必要だとしましょう。この場合、オブジェクト自体には経路検索ロジックは含まれません。代わりに、同じ機能を持つ別のオブジェクトへの参照を保持します。

特定の 経路検索技術 (A*、ダイクストラなど) を使用して迷路を解明したい場合は、この別個の “戦略” オブジェクト内に正しいソリューションを実装してください。ランタイムにオブジェクトを交換することで、別のアルゴリズムにスワップできます。

ScriptableObject メソッド

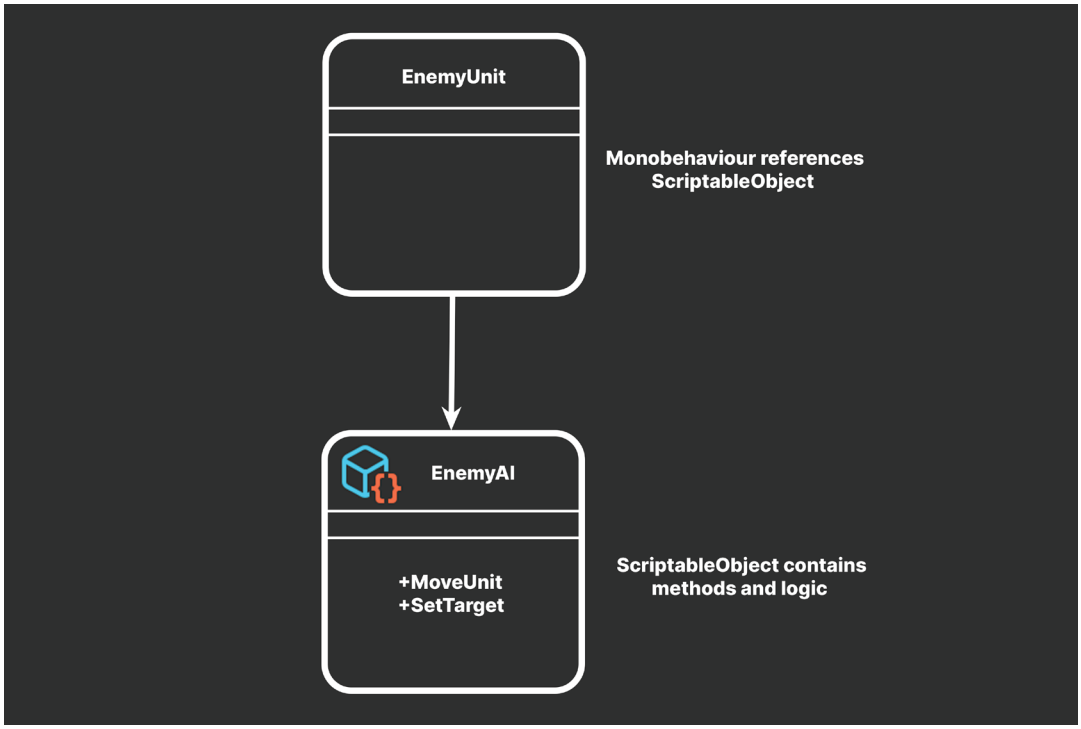
Unity でこのパターンを実装する方法の 1 つは、必要なロジックを含む ScriptableObject を MonoBehaviour に参照させることです。MonoBehaviour は、タスクを実行するときに、自身のメソッドではなく、ScriptableObject の外部メソッドを呼び出します。

ただし、これにはいくつかの制限があります。

— ScriptableObject のメソッドは、Start()、Update()、および OnCollisionEnter() などの MonoBehaviour の PlayerLoop から自動的には呼び出されません。自分で呼び出す必要があります。

— プレハブと同様に、ScriptableObject はシーンオブジェクトを直接参照できません。シーンオブジェクトに対して処理を実行する必要がある場合は、そのオブジェクトをパラメーターとして渡す必要があります。

  ScriptableObject のメソッドを呼び出すとき、MonoBehaviour は多くの場合、自身を引数として渡すか、その他の依存関係を渡すことができます。これにより、ScriptableObject がプロジェクトレベルに存在する場合でも、ロジックやコルーチンなどを柔軟に実行できます。



ScriptableObject には、プラグイン可能な動作やロジックの実装を含めることができます。

例えば、ゲーム内で動作が異なる複数の敵ユニットを定義できます。そのうちのいくつかが、パトロールする、待機する、またはプレイヤーから逃げる必要があるとします。

1 つの EnemyUnit MonoBehaviour が、MoveUnit というメソッドを含む EnemyAI ScriptableObject を参照できます。EnemyUnit スクリプト自体には、移動ロジックや動作ロジックは含まれていません。ScriptableObject の MoveUnit を適切なタイミングでのみ実行します。

メソッドがシーンのデータを必要とする場合、EnemyUnit オブジェクトは自身への参照をパラメーターとして渡すことができます。シーン内の他の必要な依存関係も同様に渡すことができます。

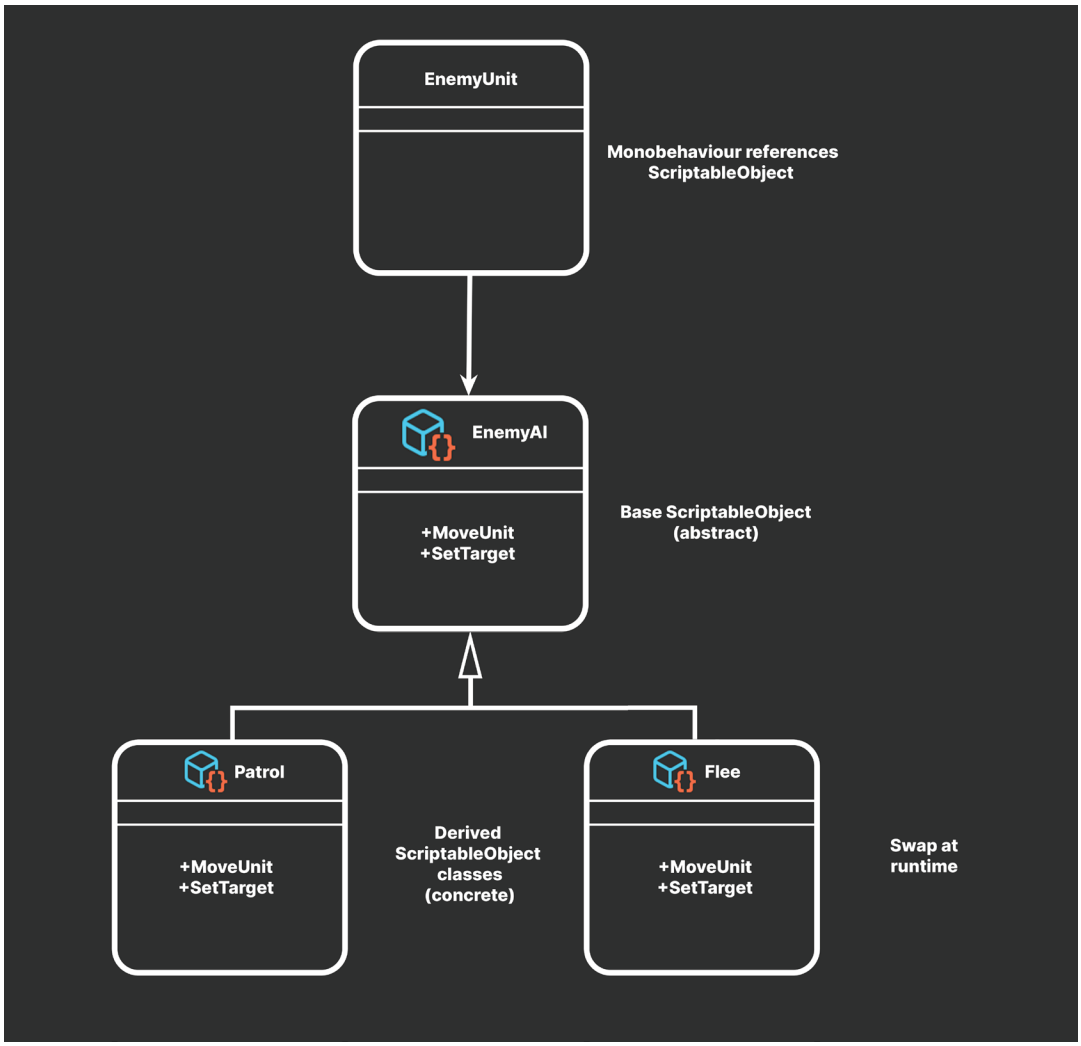
ScriptableObject データの変更

ランタイムに ScriptableObject データを実際に変更できますが、変更する際は注意が必要です。複数の MonoBehaviour が同じ ScriptableObject を共有している場合、同じデータを変更すると問題が発生する可能性があります。

ランタイムに ScriptableObject の インスタンス を作成すると、この問題を回避できます。最初の ScriptableObject は、すべてのロジックとデータを含むテンプレートのように動作します。その後、各 MonoBehaviour は、自由に変更できるその ScriptableObject の独自のインスタンスを作成できます。

プラグ可能動作

このパターンは、EnemyAI ScriptableObject を 抽象 クラスとして定義することで、より便利にすることができます。これにより、EnemyUnit MonoBehaviour と互換性のあるさまざまな ScriptableObject のテンプレートとして機能するため、抽象的な ScriptableObject を複数のアルゴリズムに対応させることができます。



プラグ可能な動作は、ランタイムまたはエディターで変更できます。

したがって、ベースの EnemyAI から派生した Patrol、Idle、Flee などの動作用の具体的な ScriptableObject クラスを設定することができます。すべて同じ MoveUnit メソッドを実装していますが、生成される結果は大きく異なります。

エディターでは、各アセットは交換可能です。選択した ScriptableObject を EnemyAI フィールドにドラッグアンドドロップするだけで済みます。互換性のある ScriptableObject はすべて、この方法で “プラグ可能” となります。

EnemyUnit または別のコンポーネントは、ScriptableObject を切り替えるタイミングやランタイムのスワップ動作を監視する “頭脳” として動作させることができます。これは、EnemyUnit がパトロール状態から逃走状態への移行といったゲームプレイイベントに反応しうる方法の一つです。状態が変更されるたびに EnemyAI ScriptableObject を切り替えるだけで実現できます。

本番環境では、2 人目の開発者またはデザイナーが、ScriptableObject 内に実際の動きや AI ロジックを実装できます。ゲームに動きや動作 (DuckAndCover、Chase など) が追加されても、元の EnemyUnit スクリプトは変更されません。このパターンは、SOLID プログラミングの オープン/クローズド原則 に従って、コードベースをより拡張しやすくするために役立ちます。すべてが既に小さなオブジェクトに分割されているため、チームメンバーを追加したりゲームデザインを変更したりする際にも、結果として得られるプロジェクトはより拡張性が高くなります。

ScriptableObject を使用したゲームプレイ AI

ScriptableObjects を使用した動作の詳細な例については、『Pluggable AI With Scriptable Objects』の動画シリーズを参照してください。これらのライブセッションは、ステート、アクション、ステート間の遷移に ScriptableObject を使用して設定できる有限ステートマシンの AI システムを紹介しています。

例: オーディオデリゲート

ScriptableObject に含まれる動作は、必ずしも複雑である必要はありません。カスタマイズしたサウンドを再生するような基本的なものでも構いません。

オーディオクリップにバリエーションを追加するために役立つ “サウンドデリゲート” ScriptableObject の例を紹介します。AudioDelegateSO は、AudioSource をパラメーターとして受け取る Play メソッドを含む抽象クラスを定義します。

```
using UnityEngine;
using Random = UnityEngine.Random;
using System;

[Serializable]
public struct RangedFloat
{
        public float MinValue;
        public float MaxValue;
}

public abstract class AudioDelegateSO:ScriptableObject
{
        public abstract void Play(AudioSource source);
}
```

この具体的な SimpleAudioDelegate ScriptableObject は、選択肢からランダムなクリップを選択し、再生中に音量とピッチを変更できます。これにより、同じサウンドを繰り返すことによる単調さが軽減されます。

```
[CreateAssetMenu(fileName ="AudioDelegate")]
public class SimpleAudioDelegateSO :AudioDelegateSO
{
        public AudioClip[] Clips;
        public RangedFloat Volume;
        public RangedFloat Pitch;

        public void Play(AudioSource source)
        {
                if (clips.Length == 0 || source == null)
                        return;

                source.clip = clips[Random.Range(0, Clips.Length)];
                source.volume = Random.Range(Volume.minValue, Volume.maxValue);
                source.pitch = Random.Range(Pitch.minValue, Pitch.maxValue);
                source.Play();
        }
}
```

これにより、すべての MonoBehaviour で AudioDelegateSO クラスから派生した ScriptableObject インスタンスを使用できます。また、オーディオエフェクトごとに AudioDelegate のバリエーションを作成することもできます。

ScriptableObject にメソッドを持たせることで、いくつかの可能性が広がります。そのメソッドは、アクションを実行するだけでなく、シーン内の任意のオブジェクトにメッセージを送信できます。

次に、オブザーバーパターンを使用した ScriptableObject ベースのイベントシステムを見てみましょう。

輝かしい ScriptableObject 革命

Unite 2016 での Richard Fine 氏のプレゼンテーション「Overthrowing the MonoBehaviour tyranny in a glorious ScriptableObject revolution」は、この e-book の大部分の基礎となりました。デモの一部 (元のプロジェクトでは AudioEvent) は、この例に合わせて変更されています。
(Youtube翻訳音声・字幕推奨)

実装の詳細と ScriptableObject の使用例については、サンプルプロジェクトを参照してください。

オブザーバー (Observer) パターン

ゲームを開発する場合、データやステートを相互に共有する必要がある複数のゲームオブジェクトがあるのが一般的です。小規模なゲームでは、これらのオブジェクト間で直接参照できますが、これは拡張性に乏しいです。これらの依存関係の管理には多大な労力が必要になる場合があり、多くの場合バグの原因となります。

アプリケーションのサイズが大きくなるにつれて、より優れた解決策が必要になります。

シングルトンの回避

多くの開発者はシングルトンを使用することを選択します。シングルトンとは、シーンの読み込み後も存続するクラスの単一のグローバルインスタンスです。しかし、シングルトンではグローバル状態が発生し、単体テストが難しくなります。

シングルトンを参照するプレハブで作業している場合、分離された関数をテストするためだけにそのすべての依存関係をインポートすることになります。これにより、モジュール性が低下し、コードのデバッグが難しくなります。

代替ソリューションとして、オブジェクト間の通信を支援するために ScriptableObject ベースのイベントを使用できます。

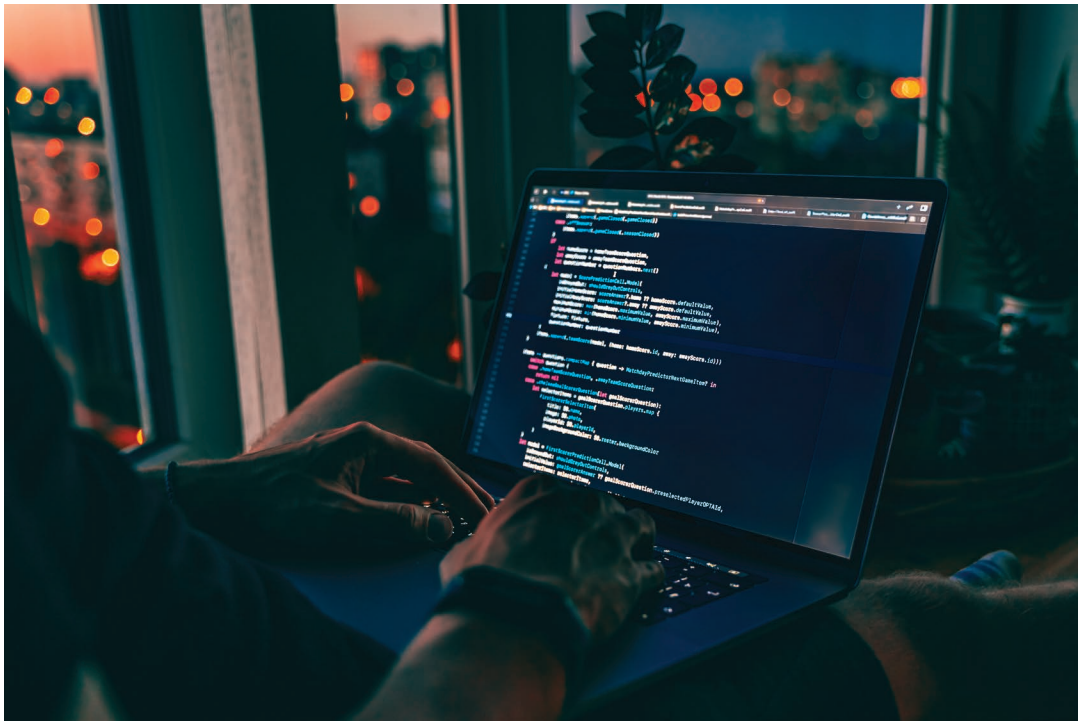
シングルトンの詳細

Unity ゲーム開発におけるシングルトンは、しばしば議論の的になります。シングルトンは、小規模なプロジェクトやプロトタイピングに適したソリューションです。大規模なアプリケーションでは、シングルトンを使用するデメリットがメリットを上回ることがよくあります。この理由から、多くの開発者は、シングルトンをアンチパターンと見なしています。

シングルトンは習得や理解が容易ですが、誤った使用法では問題を引き起こす可能性があります。ここで紹介するパターンのほとんどは、シングルトンへの依存を避けるのに役立ちます。

共有データに簡単にアクセスしたい場合は、ScriptableObject ベースのランタイムセットを検討してください (以下を参照)。オブジェクト間でメッセージを送信する方法が必要な場合は、ScriptableObject ベースのイベントチャンネルをお試しください。シングルトンからアーキテクチャを再構築することで、スケーラビリティとテスト性が向上する可能性があります。

e-book デザインパターンとSOLIDでコードをレベルアップ を読み、シングルトンの長所と短所の詳細を確認してください。

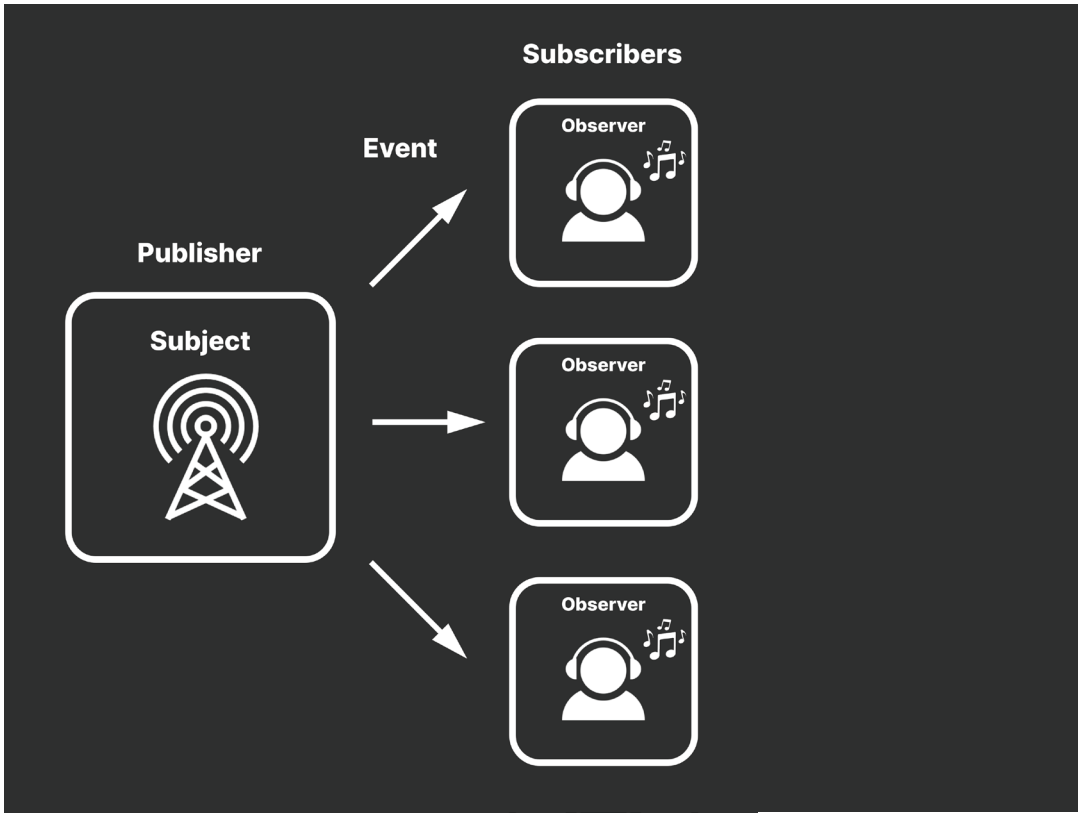


ScriptableObject ベースのイベント

すでにご覧になったように、ScriptableObject はデータ処理だけのために存在するわけではありません。他のスクリプトと同様に、メソッドを含めることができます。これらのメソッドは、オブジェクトが通信を行うための手段として使用できます。

オブザーバーデザインパターン では、サブジェクトが 1 つ以上の疎結合オブザーバーにメッセージをブロードキャストします。各オブザーバーオブジェクトはサブジェクトから独立して反応できますが、他のオブザーバーを認識しません。このサブジェクトは “パブリッシャー” または “ブロードキャスター” とも呼ばれます。オブザーバーは “サブスクライバー” または “リスナー” とも呼ばれます。

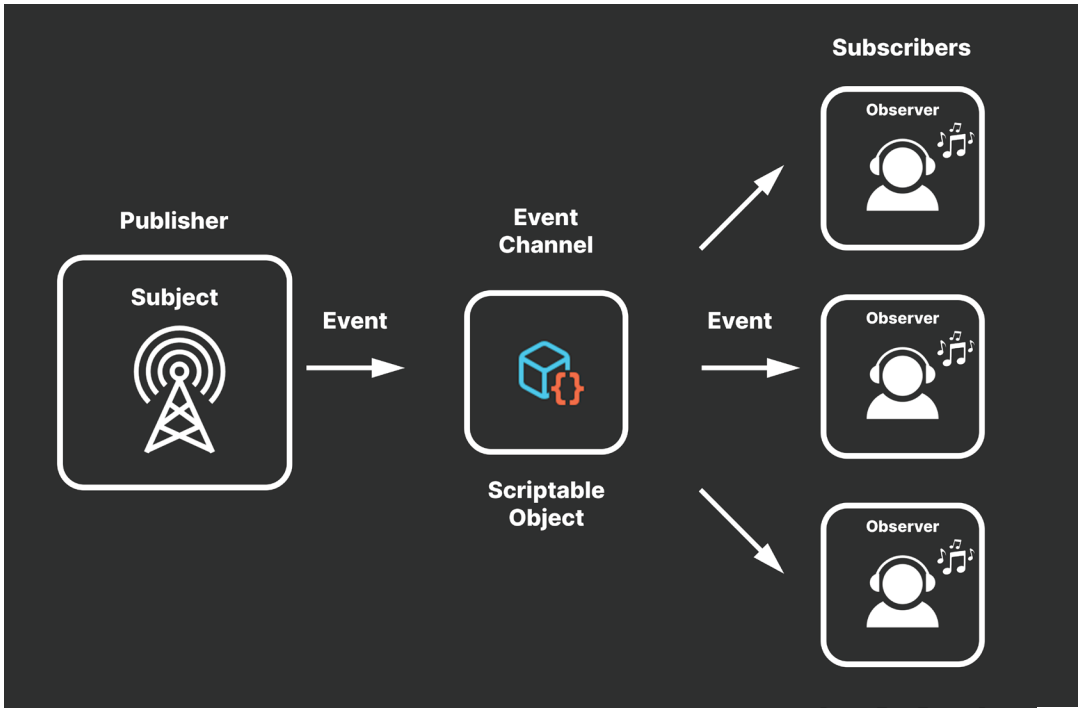
イベントベースのアーキテクチャは、各フレームを実行するのではなく、必要な時にのみ実行されます。このため、多くの場合、MonoBehaviour の更新メソッドにロジックを追加するよりも効果的に最適化されます。



基本的なオブザーバーパターン

オブザーバーパターンは、MonoBehaviour または C# オブジェクト を使用して実装できます。これは Unity 開発ではすでに一般的な手法ですが、スクリプトのみのアプローチでは、デザイナーはゲームプレイ中に必要なすべてのイベントをプログラミングチームに頼ることになります。

別の方法として、ScriptableObject ベースのイベントを作成する方法があります。これはオブザーバーパターンを設定するための、デザイナー向けの方法です。ここで、ScriptableObject はサブジェクトとオブザーバーの間の仲介として機能し、エディターにグラフィカルインターフェースを提供します。



ScriptableObject ベースのイベントチャンネル

この構造体は、一見オブサーバーパターンにオーバーヘッドのレイヤーを追加しただけであるように見えますが、いくつかの利点があります。ScriptableObject はアセットであるため、シーンヒエラルキー内のすべてのオブジェクトにアクセスでき、シーンの読み込み時に消滅することはありません。

そもそも、多くの開発者がシングルトンを使用するのは、特定のリソースに簡単かつ永続的にアクセスできるからです。ScriptableObject は、多くの場合、不要な依存関係をそれほど多く発生させることなく、同じメリットを提供できます。

ScriptableObject ベースのイベントでは、任意のオブジェクトがパブリッシャー (イベントをブロードキャストするもの) として機能し、任意のオブジェクトがサブスクライバー (イベントをリッスンするもの) として機能します。ScriptableObject は中央に配置され、信号をリレーし、2 つの間の中央仲介役のように動作します。

これを考える方法の 1 つが “イベントチャンネル” です。ScriptableObject を、その信号をリッスンするオブジェクトがいくつもある電波塔として捉えてみてください。関心のある MonoBehaviour は、イベントチャンネルに登録して、何かが発生したときに応答できます。

例: イベントチャンネル

以下を含む ScriptableObject は、すべてイベントチャンネルとして機能します。

— **デリゲート (UnityAction または System.Action):**サブスクライバーに通知し、適切なデータをパラメータとして渡します。UnityAction を使用すると、アーティストにとって使いやすい環境になります。そうでない場合は、System.Action デリゲートがここでうまく機能します。

イベントキーワードにより、ScriptableObject クラス (または宣言されている派生クラス) 内からのみ起動できるようにデリゲートが制限されます。

— **イベント発生メソッド:**この public メソッドでデリゲートを呼び出します。

それだけです。

イベントチャンネルはいくつでも設定でき、ゲームプレイのさまざまな側面を決定できます。ScriptableObject はプロジェクトレベルで存在するため、グローバルにアクセス可能なイベントを発生させることができます。これにより、シーン内の無関係なオブジェクトをスケーラブルな方法で接続できます。

System.Action または UnityAction

System.Action は、.NET Framework のシステム名前空間で定義される汎用デリゲートタイプです。カスタムデリゲートを宣言することなく、Unity プロジェクトで使用できます。event キーワードを追加すると、デリゲートタイプは読み取り専用になります。他のオブジェクトはデリゲートの登録されたメソッドをリッスンできますが、それらのメソッドを直接呼び出すことはできません。

UnityAction は、UnityEngine.Events 名前空間内で明確に定義されたデリゲートタイプです。通常は、UnityEvent クラスと一緒に使用します。UnityEvent クラスは、Unity でイベントを作成する代替手段です。UnityEvents と UnityActions は Inspector に表示されるため、多くの場合、オブサーバーパターンを実装するよりユーザーやアーティストにとって使いやすい方法として機能します。

一般的に、特定のニーズに応じて System.Action または UnityAction を使用できます。同じプロジェクトにどちらかまたは両方を展開できます。

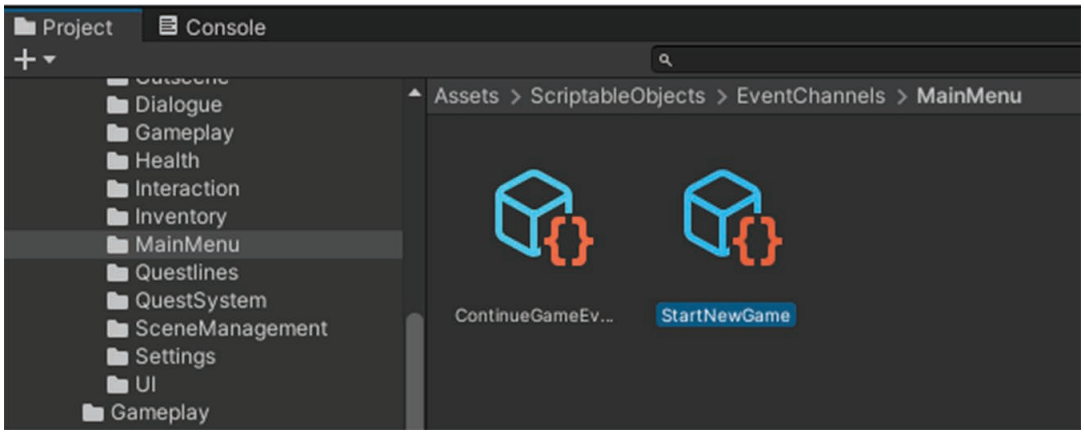
Unity ゲームエンジンに結び付けられていない汎用デリゲートが必要な場合は、System.Action を使用します。UnityEvents 専用のデリゲートが必要な場合は、UnityAction を使用してください。

ここでは、パラメーターを渡さずにイベントを発生させる VoidEventChannelSO を作成できます。これには、OnEventRaised という名前の UnityAction が含まれています。

```
[CreateAssetMenu(menuName = "Events/Void Event Channel")]
public class VoidEventChannelSO :ScriptableObject
{
        public event UnityAction OnEventRaised;
        public void RaiseEvent()
        {
                if (OnEventRaised != null)
                        OnEventRaised.Invoke();
        }
}
```

VoidEventChannelSO 型の ScriptableObject を作成すると、すべての MonoBehaviour が OnEventRaised をリッスンできます。例えば、VoidEventChannelSO 型の StartNewGame ScriptableObject を作成できます。

別のオブジェクトが public RaiseEvent メソッドを呼び出してイベントをトリガーできます。



ScriptableObject ベースのイベント (イベントチャンネルの例)

別の MonoBehaviour は、Inspector でイベントチャンネル ScriptableObject を参照し、OnEventRaised にサブスクライブまたはサブスクライブ解除できます。

イベントチャンネルが OnEventRaised を呼び出すたびに、StartNewGame が応答として呼び出されます。

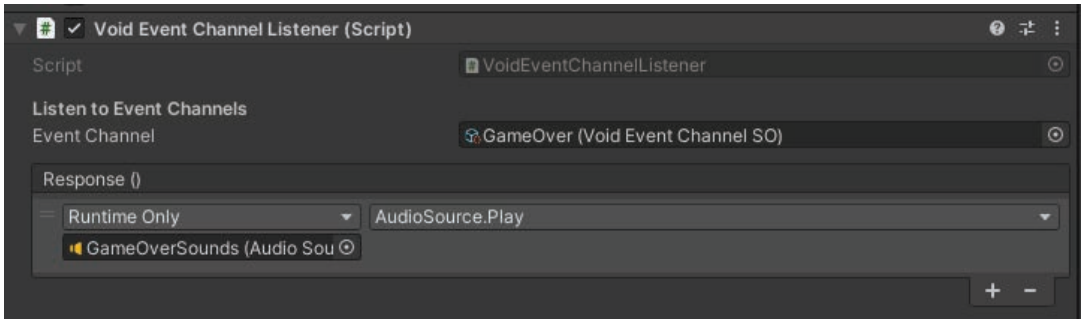
```
public class StartGame :MonoBehaviour
{
	[SerializeField] private VoidEventChannelSO m_onNewGameButton = default;

	private void Start()
	{
		m_onNewGameButton.OnEventRaised += StartNewGame;
	}

	private void OnDestroy()
	{
		m_onNewGameButton.OnEventRaised -= StartNewGame;
	}

	private void StartNewGame()
	{
	// レベルのロジックをここにロード…
	}
}
```

アーティストやデザイナーにとって使いやすいリスニングコンポーネントの場合は、スクリプト設定を必要としない MonoBehaviour を作成することもできます。この VoidEventListener クラスはその他の機能を追加しませんが、Inspector でアクセス可能なフィールドが含まれます。

```
public class VoidEventListener :MonoBehaviour
{
	[SerializeField] private VoidEventChannelSO m_channel = default;

	public UnityEvent OnEventRaised;

	private void OnEnable()
	{
		if (m_channel != null)
			m_channel.OnEventRaised += Respond;
	}
```

```
        private void OnDisable()
        {
                if (m_channel != null)
                        m_channel.OnEventRaised -= Respond;
        }

        private void Respond()
        {
                if (OnEventRaised != null)
                        OnEventRaised.Invoke();
        }
}
```

VoidEventListener をゲームオブジェクトに追加し、イベントチャンネルの ScriptableObject を Inspector の _channel フィールドにドラッグするだけです。OnRaisedEvent で UnityActions を作成し、イベントに応答できるようにします。



イベントリスナーを使用すると、プログラマーでなくてもイベント駆動型のアクションを設定できます。

イベントをリッスンするコンポーネントに関係なく、イベントチャンネルはランタイムにオブジェクト間の通信手段を提供します。プレイヤーがタスクを完了したり、ポイントを獲得しましたか?"ゲームは終了していますか?"など)。イベントは、その情報を必要とするシーン内のすべてのゲームオブジェクトに通知できます。

ScriptableObject ベースのイベントはプロジェクトレベルのアセットであるため、アプリケーションのインフラストラクチャの大部分を動かすことができます。これは、ゲームアーキテクチャを支えるさまざまなシステム間でメッセージを送信する場合に特に便利です。

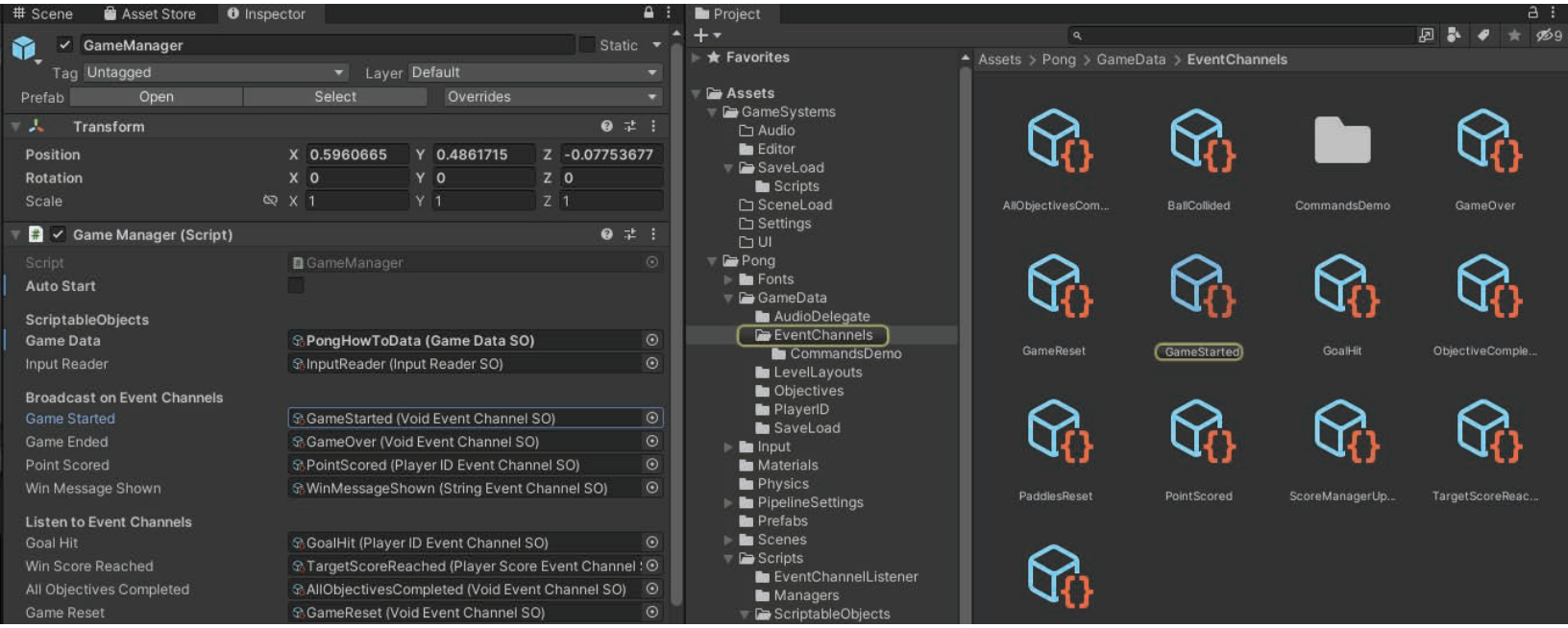
一般的な管理システムには次のようなものがあります。

- **オーディオ管理:**ゲーム内の多くの要素がサウンドを再生するためのトリガーとなります。このシステムは、アプリケーションイベントに応じて AudioClips を再生したり、AudioMixer を調整したりできます。
- **シーン管理**:このシステムは、Unity シーンとゲームレベルのロードとアンロードを処理します。
- **UI 管理:**ゲームプレイ前、ゲームプレイ中、ゲームプレイ後のメニュー画面を管理します。
- **セーブデータ管理:**ゲームデータの保存と読み込み、およびファイルシステムに対する設定を管理します。

これらのシステムはそれぞれ異なるタスクに特化していますが、互いに通信する必要があります。イベントは、イベント同士をつなぐ接着剤となります。

付属の サンプルプロジェクト で、独自の ScriptableObject ベースのイベントを実装する方法の例をもっと見ていきましょう。

サンプルプロジェクトの GameManager はイベントチャンネルを使用します。



異なるイベントペイロードを使用して、イベントごとに異なるタイプのデータを送信できます。例えば、ScriptableObject ベースのイベントには、IntEventChannelSO、Vector2EventChannelSO、VoidEventChannelsSO などがあります。使用されるイベントはコンテキストによって異なります。

ゲームプレイに応じて追加イベントタイプをカスタマイズしましょう。例えば、誰がどの程度の損害を与えたかを示すダメージイベントを渡す必要があるかもしれません。

これらのイベントチャンネルの展開方法は、あなたの創造力次第です。上記の中心的なシステムに加え、相互に作用できるように、イベントが大きく異なるゲーム内システムを結びつける場合が多々あります。

— **カメラ:**揺れや別の透視投影などのドラマチックまたはシネマティックなエフェクトを追加します。

— **クエスト:**プレイヤーがゲームを進めたり、報酬を得たりするために完了しなければならないタスクまたは目標です。クエストには、多くの場合、アイテムを取得する、敵を倒す、パズルを解くなど、さまざまなゲームプレイ要素が含まれています。

— **体力:**多くのゲームにおいて、この重要な要素はプレイヤー、敵、およびプレイヤーにダメージを与える可能性のあるオブジェクトやアクションに関連します。

— **実績:**クエストと同様に、ゲーム内の特定のタスクや目的を果たすことでアンロックされる特別な報酬です。実績は、特定のレベルに到達したり、特定のポイントを貯めたりと、さまざまなゲームプレイ要素に関係します。

これらのゲームプレイ要素は、イベントを通して、オーディオ、UI、セーブデータなどの他の管理システムと相互作用します。このアプローチでは、アーキテクチャの各コンポーネント内のモジュール性と独立性を促進しながら、他のシステムとの通信を可能にします。

イベントチャンネルのデバッグ

カスタムエディターまたはプロパティドロワーで Inspector の “Raise Event” ボタンを作成できます。これは、デバッグのためにイベントを手動で呼び出すために役立ちます。

例として、VoidEventChannelSO のカスタム Inspector ボタンを作成する基本的なエディタースクリプトを紹介します。

```
[CustomEditor(typeof(VoidEventChannelSO))]
public class VoidEventChannelSOEditor :Editor
{
    public override void OnInspectorGUI()
    {
        DrawDefaultInspector();

        VoidEventChannelSO eventChannel = (VoidEventChannelSO)target;

        if (GUILayout.Button("Raise Event"))
        {
            eventChannel.RaiseEvent();
        }
    }
}
```

これにより、イベントを任意に発生させられるボタンが作成され、ランタイムの問題の診断が容易になります。

カスタムエディターボタンは、イベントチャンネルのテストに役立ちます。

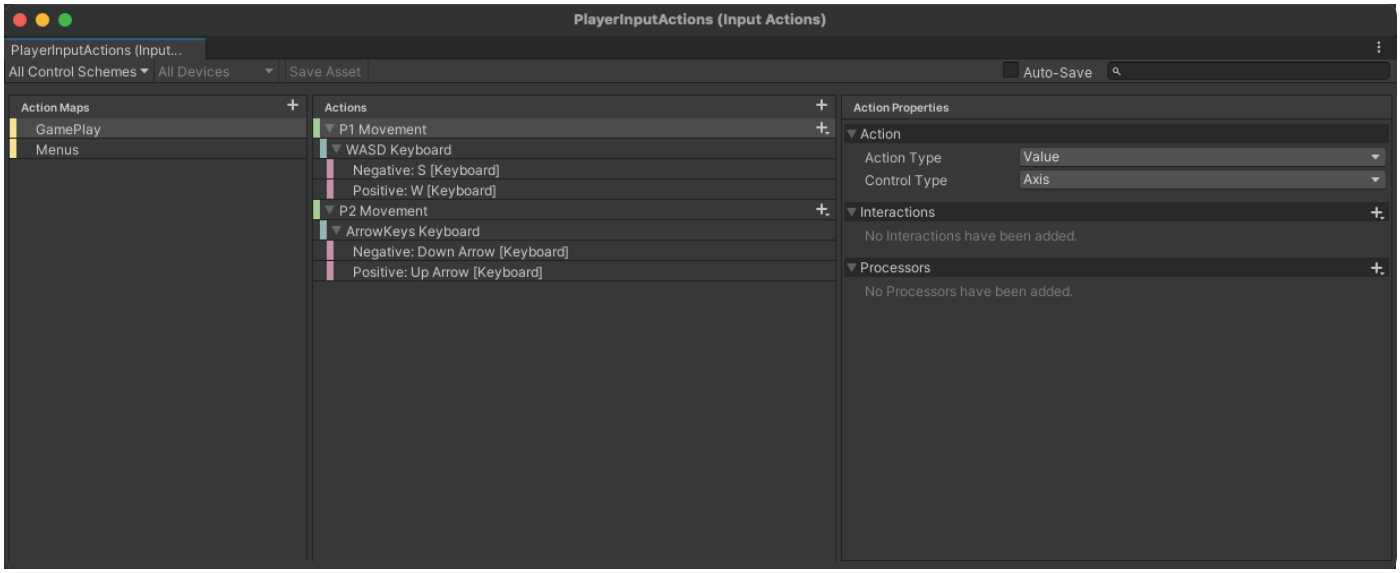
もう少し手間をかけると、データを扱うイベントチャンネルのボタンも作成できます。イベントチャンネル自体にデバッグ値用のフィールドを確保し、それを Editor スクリプトに渡します。これを実装する方法の例は、SOAP または Unity Atoms プロジェクトで確認できます。

オブジェクト分離にイベントチャンネルを使用し続ける場合は、各イベントのすべてのリスナーを記録するなど、デバッグツールの開発を検討してください。イベントチャンネルクラスには、オブジェクトをサブスクライブおよびサブスクライブ解除するメソッドを含めることができ、ランタイム中の特定の動作の原因となっているイベントの特定が容易になります。

例: InputReader

ユーザー入力をリッスンするオブジェクトには、特殊なタイプのイベントチャンネルが必要です。Unity の Input System は、InputActions を使用して Raw 入力データを論理的な概念 (ジャンプ、歩行など) として表現します。

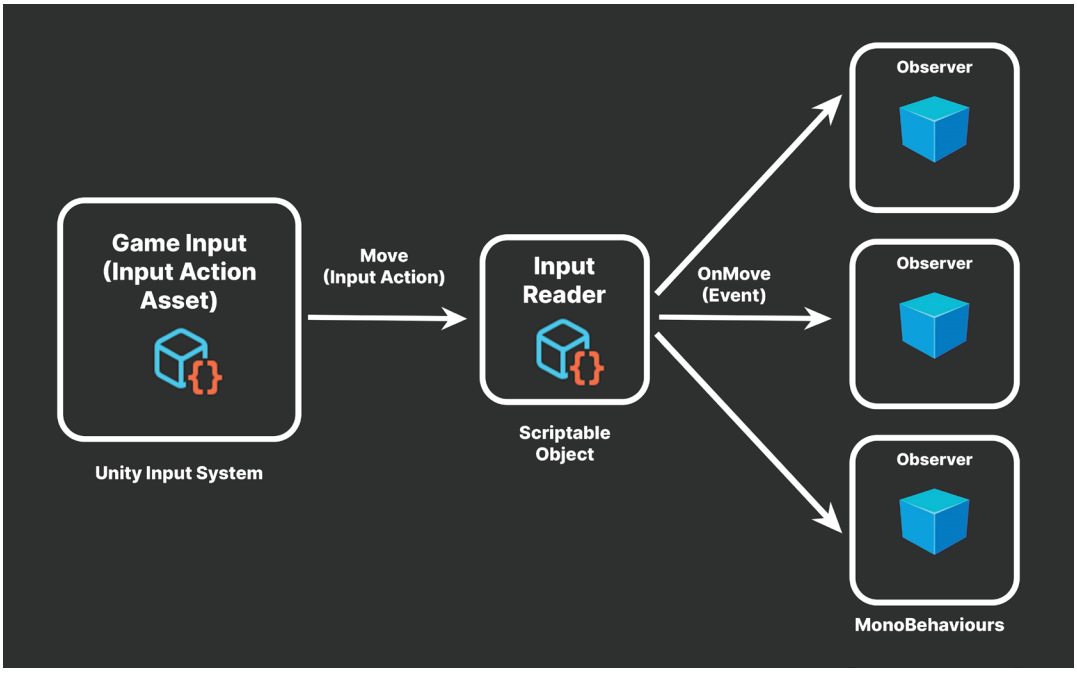
一方、各 InputAction には、それぞれ開始、実行、および キャンセル されたイベントが含まれます。



Input System Editor でのアクションとアクションマップの設定

InputAction バインディングを解読するために、特別な InputReader ScriptableObject を作成できます。ここでも、これはサブジェクトとオブザーバーの間の仲介役として機能します。ただし、この場合、MonoBehaviour は明示的にイベントを発生させません。

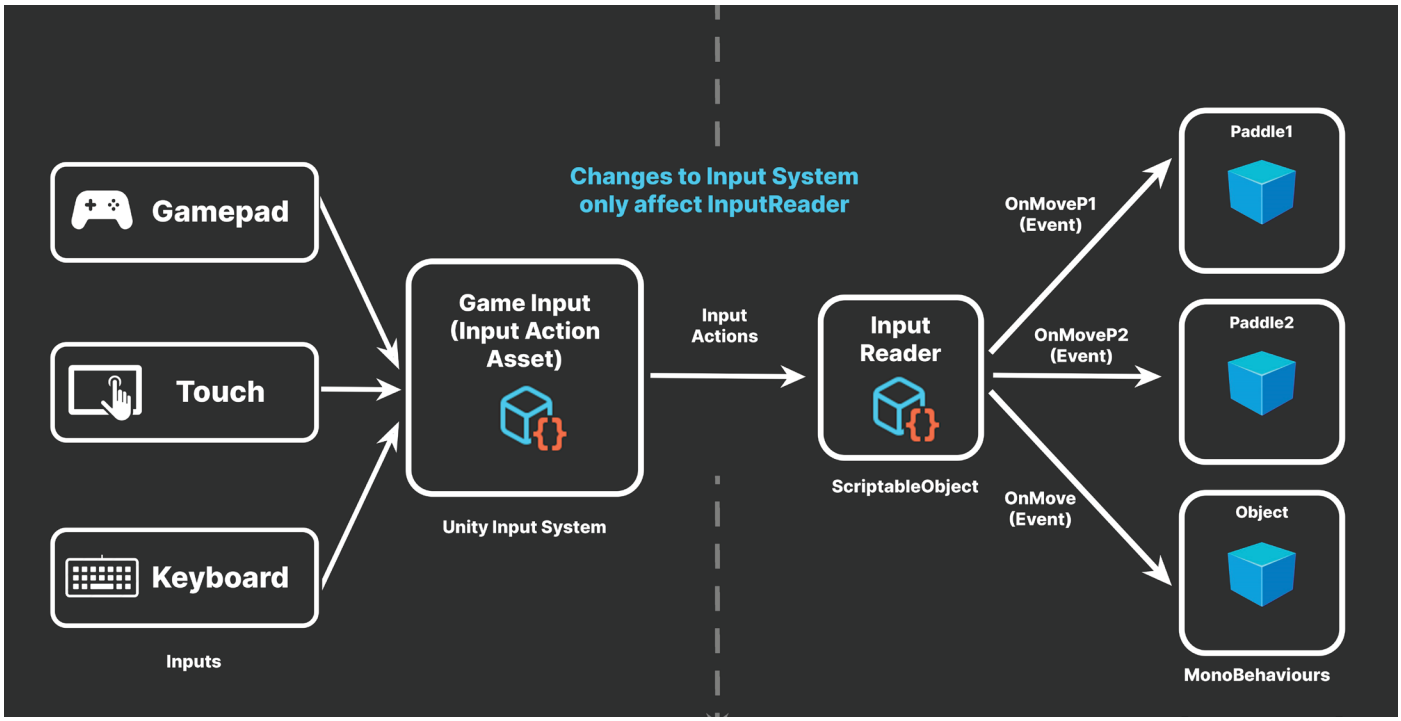
代わりに、サブジェクトまたはブロードキャスターの代わりに Input System が使用されます。



ScriptableObject InputReader は、InputActions のイベントをリレーします。

ここでは、Input System で Actions と ActionMap を設定します。各 InputAction は、別々の入力軸を記述し、キーボード、ゲームパッド、または使用する任意の入力デバイスにバインドします。

パドルコントローラは、InputActions 自体に直接サブスクライブするのではなく、OnMoveP1. performed イベントと OnMoveP2. performed イベントをそれぞれリッスンします。



InputReader は、オブジェクトが入力に直接依存しないようにします。

結果として得られる InputReader は、ゲームオブジェクトがゲームパッドやキーボードアクションを処理する方法を標準化します。入力が必要なゲームオブジェクトは:

— InputReader ScriptableObject への参照を保持します。

— 関連イベントにサブスクライブし、そのイベント処理メソッドを接続します。

このパターンは小規模なゲームでは過剰かもしれませんが、コンセプトをわかりやすくするために紹介しています。

プロジェクトが大きくなり、より多くのコンポーネントが追加されない限り、そのメリットは見えてこないでしょう。入力を消費するゲームオブジェクトから分離することで、柔軟性と再利用性が高まります。

開発中に InputActions を変更する必要がある場合は、InputReader 自体を管理するだけで済みます。イベントを変更する必要がない場合、リッスンオブジェクトは影響を受けません。したがって、入力からオブザーバーへの接続を維持する作業は、特にオブザーバーが多数ある場合に少なくなります。

静的イベントと非静的イベント

静的イベントを使用して、リッスンオブジェクト上の ScriptableObject の場所を特定する負担を軽減できます。

例えば、MonoBehaviour は InputReader の静的 MoveP1Event イベントと MoveP2Event イベントを OnEnable メソッドでサブスクライブできます。

```
InputReader.MoveP1Event += OnMoveP1;

InputReader.MoveP2Event += OnMoveP2;
```

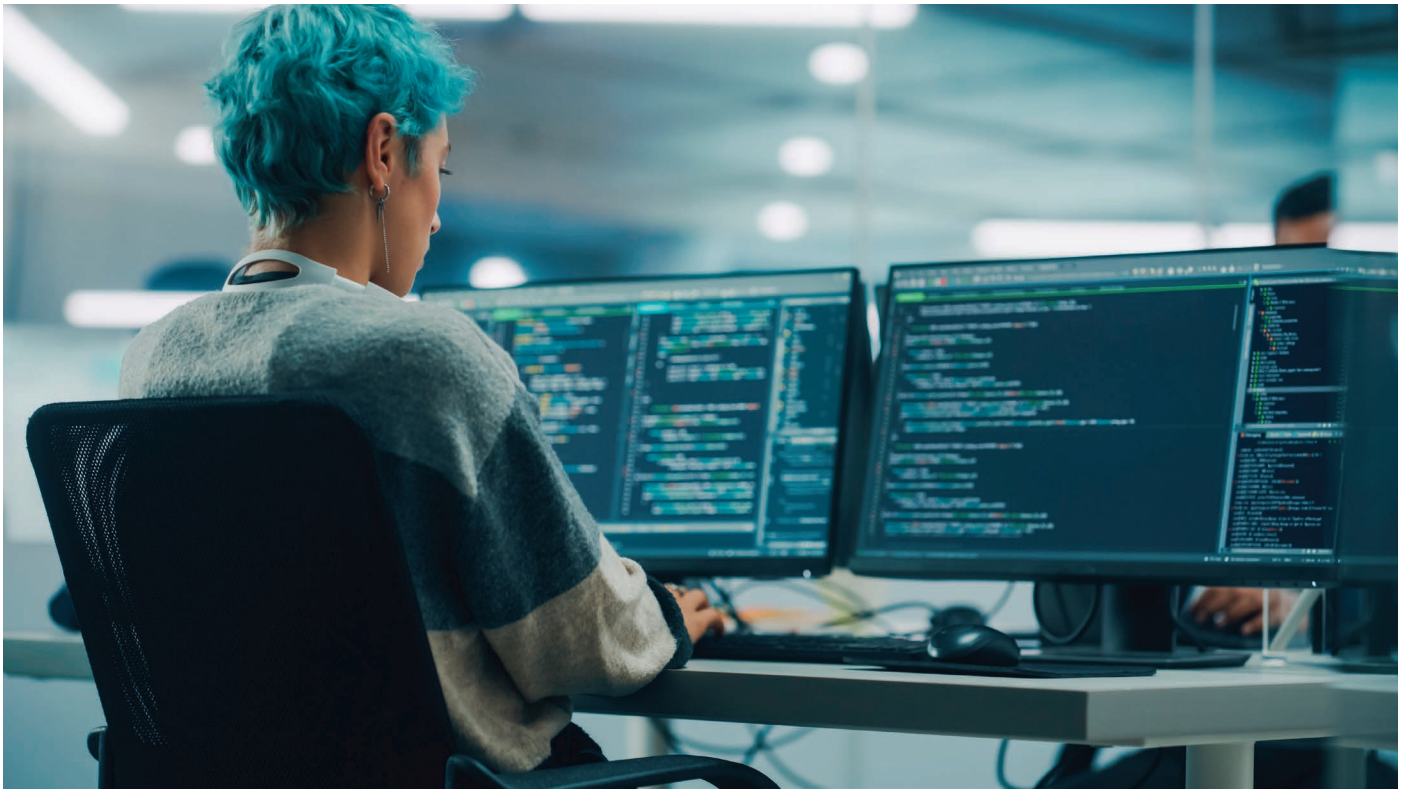
静的イベントを使用する場合は、サブスクリプションの管理を注意深く行ってください。OnDisable でサブスクライブ解除することを忘れないでください。

```
InputReader.MoveP1Event -= OnMoveP1;

InputReader.MoveP2Event -= OnMoveP2;
```

静的イベントは常にアクセス可能で、アクティブなサブスクライバーがある場合はガベージコレクターによって収集されません。サブスクライバーがぶら下がっていると、アプリケーションの継続時間中はクリーンアップができなくなります。

ただし、静的イベントはシリアル化できません。エディターでインタラクティブに作業する場合は、非静的イベントを選択し、Inspector で適切な ScriptableObject を参照してください。

コマンド (Command) パターン



コマンドパターンを使用すると、メソッドを直接呼び出す代わりに、1 つ以上のメソッド呼び出しを "コマンドオブジェクト" としてカプセル化できます。

次に、これらのコマンドオブジェクトをキューやスタックなどのコレクションに格納します。これらは小さなバッファとして機能します。これにより、各コマンドオブジェクトの実行をより柔軟に制御できます。一般的なアプリケーションには、一連のアクションを特定のタイミングで再生したり、それらのアクションを取り消し可能にしたりするものがあります。

お気に入りのゲームジャンルで、この現象に遭遇したことがあるかもしれません。

— リアルタイムストラテジーゲームでは、コマンドパターンを使用してユニットや建物のアクションをキューに入れることができます。その後、ゲームはリソースが利用可能になったとき、またはユニットの一連の移動アクションとして、各建物のコマンドを実行します。

— ターン制ストラテジーゲームでは、プレイヤーはユニットを選択し、その動きやアクションをキューなどのコレクションに格納することができます。ターンの終わりに、ゲームはプレイヤーのキューにあるすべてのコマンドを実行することができます。

— パズルゲームでは、コマンドパターンによってプレイヤーがアクションを元に戻したりやり直したりできます。

— 格闘ゲームでは、特定のコマンドリストでボタンの押し方やゲームパッドのモーションを読み取ると、コンボや必殺技を繰り出せます。

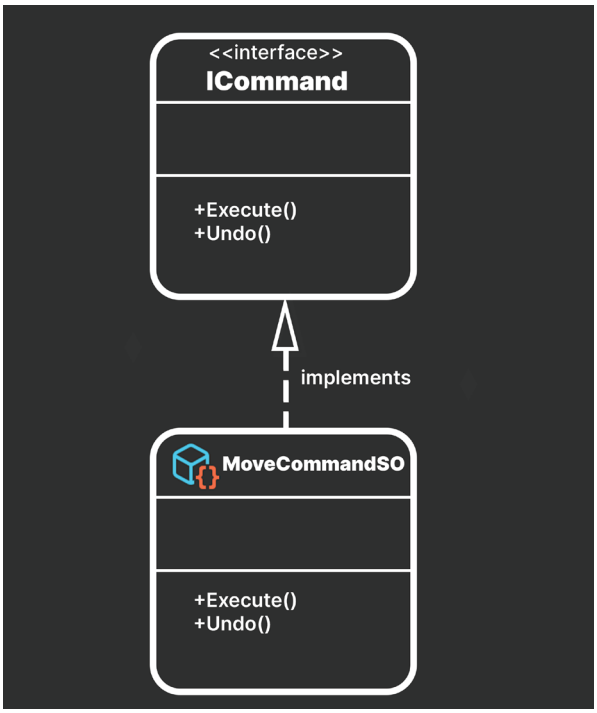
このようなコマンドパターンは、ScriptableObject を使用して実装できます。

例えば、Transform の動きを定義するコマンドを作成できます。各アクションを別のオブジェクト内にラップすると、より細かい制御が可能になります。

インターフェース ICommand、Execute メソッド、および 元に戻す メソッドを定義します (抽象クラスを使用することもできます)。

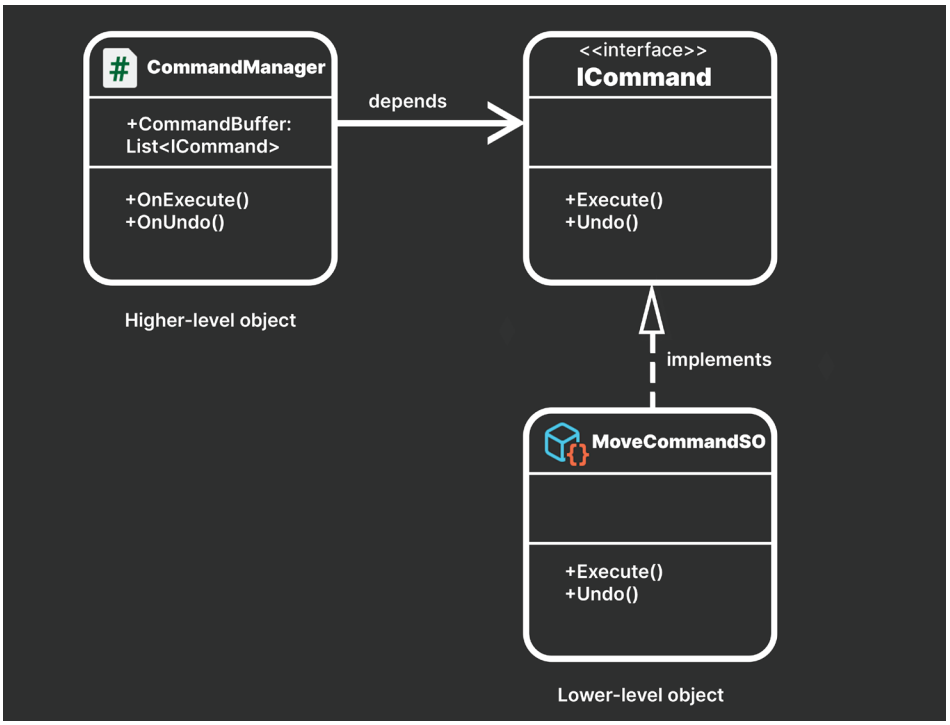
```
public interface ICommand
{
    public abstract void Execute();
    public abstract void Undo();
}
```

次に、ScriptableObject に ICommand インターフェースを実装させます。各コマンドオブジェクトは、独自の実装の詳細で独自の実行メソッドと元に戻すメソッドを記入します。



ScriptableObject を使用したコマンドパターンの実装

その後、MonoBehaviour または ScriptableObject は、コマンドオブジェクトを含むコマンドバッファを定義できます。これは、リスト、スタック、配列、キューなどのコレクションです。



CommandManager は ICommand オブジェクトのコレクションを保持します。

このシンプルな構造体により、コマンドを順番に実行できます。ゲームオブジェクトが所定のアクションまたはアニメーションのセット内を移動するチュートリアルまたはカットシーンを想像してみてください。コマンドパターンは、これに適しています。

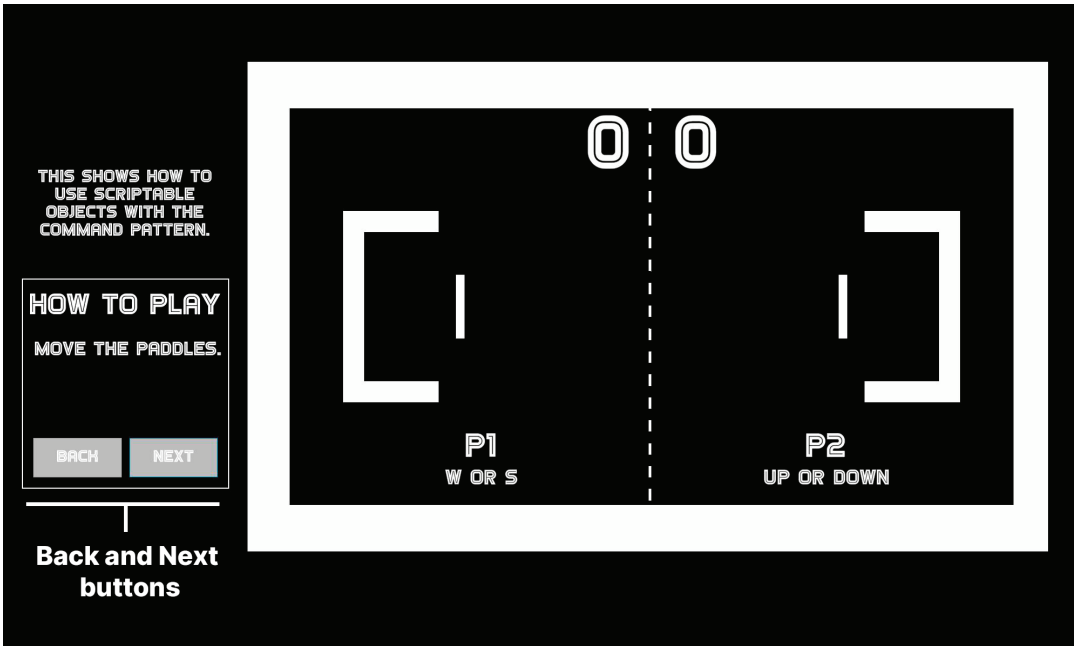
各 コマンド は個別のオブジェクトであるため、並べ替えは簡単です。CommandBuffer をどのように維持するかを決めるだけです。

— スタックとして作成する場合は、実行時にコマンドをスタックにプッシュします。アクションを取り消すときは、そのアクションを切り離して別のやり直しスタックを保持できます。

— リストまたは配列を使用している場合は、現在のコマンドのインデックスを追跡し、コマンドの元に戻すまたはやり直しの必要に応じてインデックスをインクリメントまたはデクリメントします。

取り消し可能な移動の例については、サンプルプロジェクトの MoveCommandSO クラスと CommandManager クラスを参照してください。この基本的なチュートリアルシーンでは、ゲームボードのパーツにラベルを付けて、遊び方を説明しています。

次へボタンをクリックして、説明文を進めましょう。同様に、戻るボタンをクリックして説明文を巻き戻すことができます。



コマンドパターンの簡単な実装

コマンドパターンの詳細については、e-book デザインパターンとSOLIDでコードをレベルアップ を参照してください。また、このコミュニティ投稿「Command pattern with ScriptableObjects」では、ScriptableObject を使用したこのパターンを紹介しています。

ScriptableObject か C# クラスか

プロジェクトに適したコードアーキテクチャを決定する際には、チームのスキルや好み、ゲームのパフォーマンス要件を考慮することが重要です。デザイナーの中には、Unity エディターをインタラクティブに使用することを好む人もいれば、すべての作業を C# コードで行うことを好む人もいます。

これらを考慮し、チームの全員が使いやすいコードベースを作成してください。もちろん、デザインパターンは万能のソリューションではありません。実装する前に、それぞれの長所と短所を慎重に評価する必要があります。

“正しい” コードアーキテクチャは、チームとプロジェクトにとって最適なアーキテクチャであるということに過ぎないことを覚えておいてください。

ランタイムセットのパターン

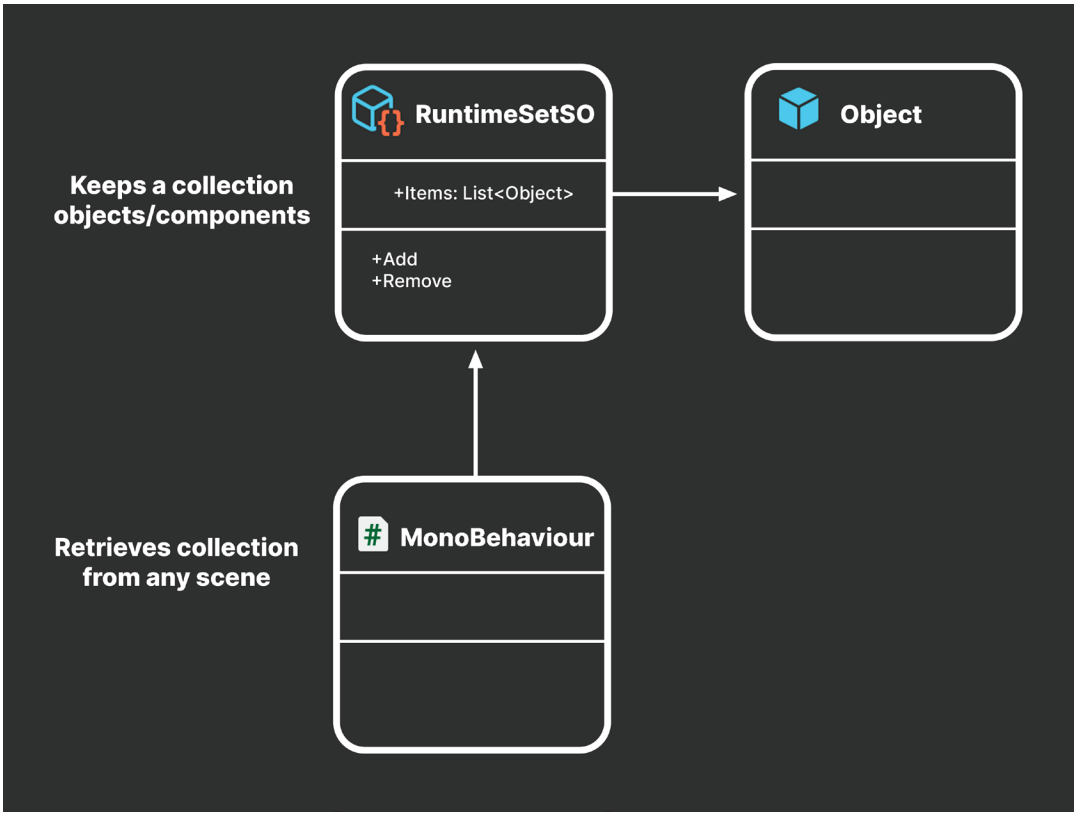
ランタイムでは、多くの場合、シーン内のゲームオブジェクトまたはコンポーネントのリストを追跡する必要があります。例えば、敵や NPC のリストを維持する必要がある場合があります。

ScriptableObject インスタンスはプロジェクトレベルに表示されるため、あらゆるシーンのあらゆるオブジェクトで利用できるデータを保存できます。繰り返しになりますが、こうすることでシングルトンによる簡単なグローバルアクセスの大部分を再現しつつ、パターンの既知の欠点を避けることができます。

また、ScriptableObject から直接データを読み取ることは、Object.FindObjectOfType や GameObject.FindWithTag などの検索操作でシーン階層を検索するよりも最適です。ユースケースや階層のサイズにもよりますが、これらは比較的負荷の高いメソッドであり、フレームごとの更新では非効率的です。

基本ランタイムセット

代わりに、ScriptableObject にデータを “ランタイムセット” として保存することを検討してください。これは特殊なデータコンテナで、要素のパブリックコレクションを維持しますが、コレクションに追加および削除するための基本的なメソッドも提供します。



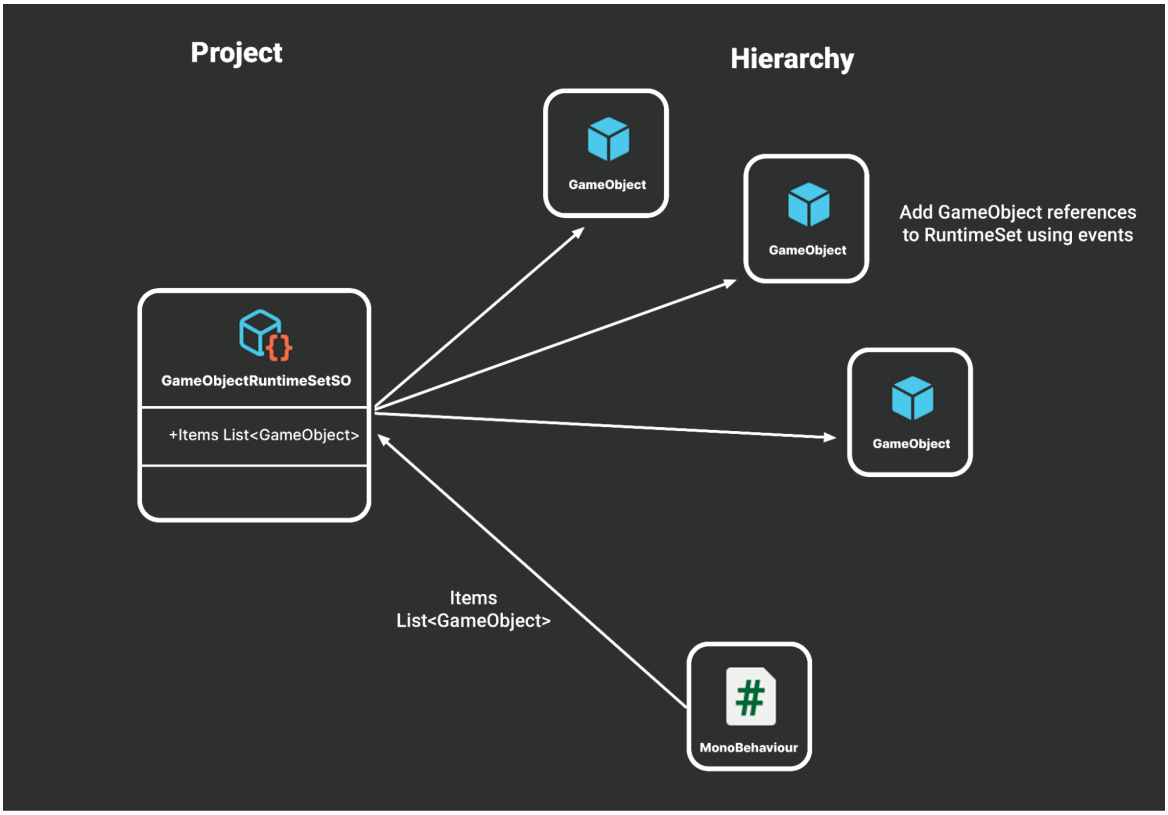
ランタイムセットは、データのコレクションへのグローバルアクセスを提供します。

ここでは、ゲームオブジェクトのリストを追跡する基本的なランタイムセットを紹介します。

```
using System.Collections.Generic;
using UnityEngine;

[CreateAssetMenu(menuName = "GameObject Runtime Set", fileName = “GORuntimeSet”)]
public class GameObjectRuntimeSetSO :ScriptableObject
{
   private List<GameObject> items = new List<GameObject>();
   public List<GameObject> Items ⇒ items;

   public void Add(GameObject thingToAdd)
   {
      if (!items.Contains(thingToAdd))
         items.Add(thingToAdd);
   }
```

```
    public void Remove(GameObject thingToRemove)
    {
        if (items.Contains(thingToRemove))
            items.Remove(thingToRemove);
    }
}
```

ランタイムに、すべての MonoBehaviour が public の Items プロパティを参照して、完全なリストを取得できます。別のスクリプトまたはコンポーネントで、このリストへのゲームオブジェクトの追加または削除方法を管理する必要があります。



ゲームオブジェクトランタイムセット

MonoBehaviour でランタイムセットを参照します。次に、OnEnable イベント関数と OnDisable イベント関数で、ランタイムセットの Items リストからオブジェクトを追加または削除します。または、イベントチャンネルを使用して、ゲームオブジェクトをそのペイロードとして送信します (例: GameObjectEventChannel)。

汎用バージョン

ランタイムセットは、特定の種類の MonoBehaviour と組み合わせて使用することもできます。例えば、これにより、シーン内のあらゆるものにアクセス可能な敵アイテムやピックアップアイテムのリストを保持できます。この場合、タイプごとに特定のランタイムセットを作成できます(EnemyRuntimeSet、PickupRuntimeSetなど)。

追加のランタイムセットの作成を効率化する 1 つの方法は、汎用の抽象クラスを使用することです。

```
public abstract class RuntimeSetSO<T> :ScriptableObject
{
    [HideInInspector]
    public List<T> Items = new List<T>();

    public void Add(T thing)
    {
        if (!Items.Contains(thing))
            Items.Add(thing);
    }

    public void Remove(T thing)
    {
        if (Items.Contains(thing))
            Items.Remove(thing);
    }
}
```

これは元の GameObjectRuntimeSet と似ていますが、柔軟性も向上しています。カスタム Foo コンポーネントのランタイムセットを作成する場合は、次のように具象的な FooRuntimeSetSO を作成します。

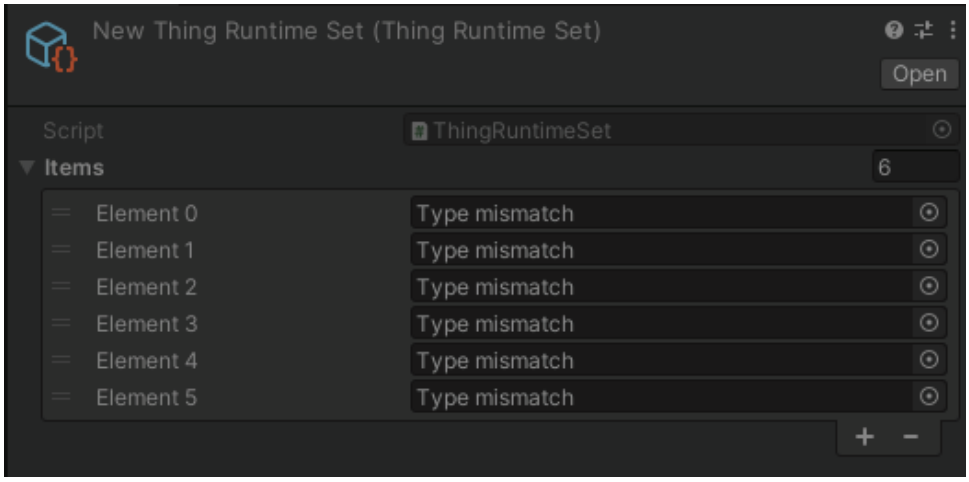
```
[CreateAssetMenu(menuName = "Foo Runtime Set", fileName = "FooRuntimeSet")]
public class FooRuntimeSetSO :RuntimeSet<Foo>
{
}
```

ゲームプレイに必要な数の具象クラスをビルドします (例: 敵、NPC、インベントリアイテム、クエストなどに独自のランタイムセットを設けることができます)。必要なのは、適切な型で新しい空のクラスを宣言することだけです。

イベントを使用する代わりに、各 Foo コンポーネントで、その OnEnable メソッドまたは OnDisable メソッドを使用して自分自身を追加または削除できます。次に、Inspector で FooRuntimeSet フィールドを設定すると、Foo コンポーネントがランタイムセットに自動的に表示されます。これは、プレハブで Foo コンポーネントを使用している場合に特に便利です。

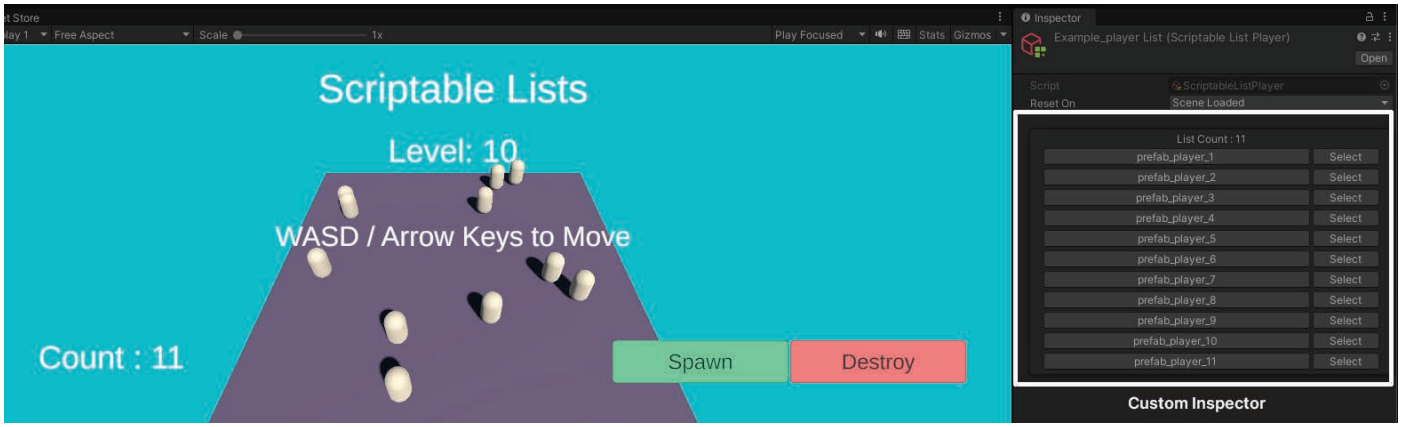
```
public class Foo :MonoBehaviour
{
    public FooRuntimeSetSO RuntimeSet;

    private void OnEnable()
    {
        RuntimeSet.Add(this);
    }

    private void OnDisable()
    {
        RuntimeSet.Remove(this);
    }
}
```

注意:この手法の制限として、ランタイムに ScriptableObject を検査すると、Inspectorで ランタイムセットリストの内容が表示されないことがあります。Inspector でリストを公開しようとすると、次のようになります。



ランタイムセットは Inspector にシーンオブジェクトやコンポーネントを表示しません。

デフォルトでは、ScriptableObject はシーンオブジェクトをシリアル化できないため、各要素のフィールドに "Type mismatch" (型の不一致) と表示されます。リストは正常に機能しますが、データは正しく表示されません。Inspector にリストが表示されないようにして混乱を防ぐには、public プロパティまたは HideInInspector 属性を使用しましょう。この問題は、カスタム Editor スクリプトと Inspector を使用して修正することもできます。これについては、アセットストアの SOAP (ScriptableObject Architectural Pattern) を参照してください。



SOAP のカスタム Editor は、ランタイムセットの内容を表示します。

foo と bar に関する興味深い情報

foo と bar という用語は、プログラミングでは一般的なプレースホルダー名です。これらの用語はおそらく短くて覚えやすく、また特徴的な響きがあるという理由で選ばれたのでしょう。

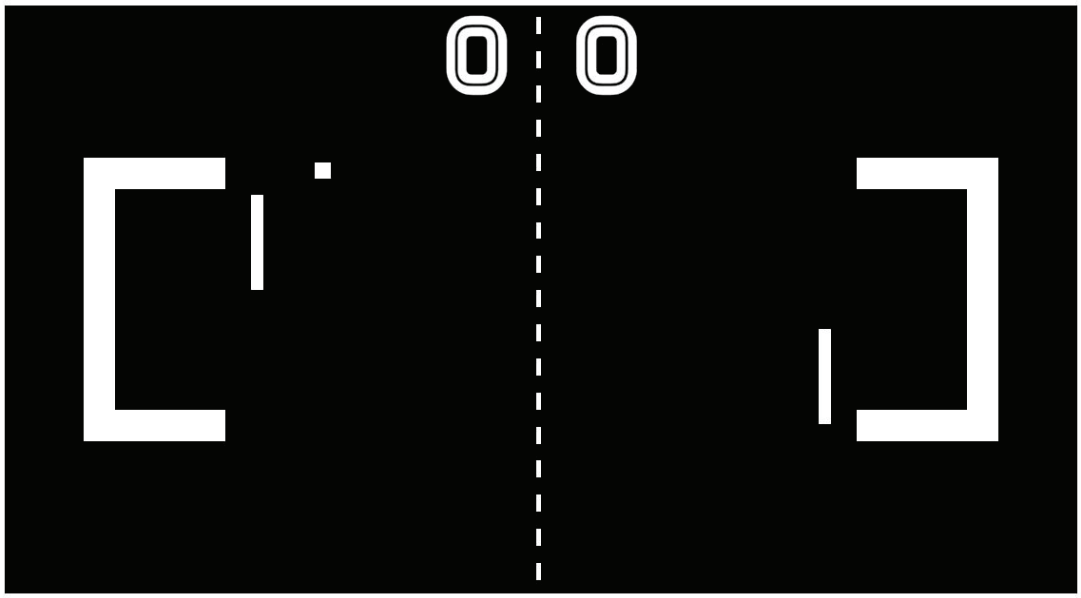
その正確な起源は不明ですが、第二次世界大戦のレーダー操作員に由来するという説もあります。“foo” というばかげた造語は、1930 年代のコミックストリップのキャッチフレーズにもなりました。

プログラミングの文脈では、それらの使用は一般的に 1960 年代の MIT の Tech Model Railway Club に由来するとされています。MIT のトレインルームのドア脇には “foo” と “bar” というラベルの付いた 2 つの汎用ボタンがありました。MIT のハッカーは、しばしばこれらの名前を自分たちのアイデアに転用したため、変数の一般的な名前として foo や bar が採用されるようになりました。現在、変数のダミー名として foo と bar を使用することは、プログラミングコミュニティで広く普及した慣習となっています。

サンプルプロジェクトを探る

このガイドには、ScriptableObject の操作と学習に役立つ サンプルプロジェクト が含まれています。古典的なボールとパドルのゲームメカニクスに着想を得たこのプロジェクトは、ScriptableObjects が Unity プロジェクトのアーキテクチャをいかに改善できるかを理解する絶好の機会を提供します。

SOLID の原則に従い、大規模なプロジェクトでよく採用される手法を使用することで、ScriptableObject がコードの再構築にどのように役立つかを理解できます。



サンプルプロジェクト

ここでは、ScriptableObject の適用例を確認し、プロジェクトの効率と編成の改善にどのように活用できるかをより深く理解できます。

サンプルには、このガイドで説明したパターンの多くが含まれています。

- **データコンテナ:**共有設定データは ScriptableObject として抽出されます。その後、ゲームプレイの設定や一般的なステートがアセットとしてプロジェクトに格納されます。また、ScriptableObject を入れ替えることで、初期レベルのレイアウトを変更できます。
- **拡張可能な enum:**空の ScriptableObject は、さまざまなプレイヤーを分類するための enum として機能します。
- **デリゲートオブジェクト:**簡単なオーディオデリゲートを通して、ScriptableObject を入れ替えるだけでボールの衝突音をランダム化する方法を理解できます。目標も、勝ち負け条件を決定するためにゲーム管理システムに接続する ScriptableObject となっています。
- **イベントチャンネル:**オブザーバーパターンは、UI、サウンド、スコアリングのゲームイベントを設定するために役立ちます。異なるゲームオブジェクトは、特定のラジオブロードキャストを調整するように、異なる “イベントチャンネル” に登録できます。
- **デュアルシリアル化:**一部のゲームデータは、エディターで使いやすいようにレベルレイアウトのように ScriptableObject に格納されますが、JSON ファイルとして保存することも可能です。外部で変更された JSON データは、元の設定スクリプトと連携する ScriptableObject を再構築できます。

もちろん、ゲーム自体はこのサンプルの主役ではありません。パドルとボールのアーケードゲームは、これよりもはるかに少ないコード行で構築できます。この サンプルプロジェクト は、ScriptableObject がテスト可能でスケーラブル、かつデザイナーにとって使いやすいコンポーネントの作成にいかに役立つか示すことを目的としています。

まとめ

ScriptableObject は、ツールセットに追加すると非常に汎用性の高い存在となります。これらのデザインパターンは、Unity 開発を整理するための新たな可能性として考えてください。

このガイドで紹介するテクニックの多くは ScriptableObject ではなく C# クラスを使用することで実現できます。状況によっては C# クラスの使用を好む開発者もいるでしょう。しかし、Unity エディターでは、ScriptableObject をより簡単に表示および編集できる利便性が提供されています。これにより、アーティストチームやデザインチームがプロジェクトと連携しやすくなり、すべてをコード経由で制御する必要がなくなります。

デザイナーが、ソフトウェアチームからの定期的なサポートなしでゲームプレイデータを設定したいと考えているなら、プロジェクトで ScriptableObject を採用する余地があるかもしれません。

もちろん、パターンを使用しないことも同様に重要です。あるアプリケーションには自然に適合するものでも、別のアプリケーションには適さない場合があります。パターンを展開する前に、そのパターンの長所と短所を評価しましょう。

Unity プロジェクトの構造に絶対的なルールはありません。チームのスキルセットと個人の好みをコードアーキテクチャと調和させましょう。

そうすれば、プレイヤーにとって魅力的なゲーム体験を提供する大切な仕事に注力できます。

その他のリソース

ドキュメント

— ScriptableObject スクリプティング API

— ScriptableObject マニュアルページ

Unity のテクニカル e-book

Unity のデザインパターンとコーディングに関連

— *デザインパターン と SOLID でコードをレベルアップ* は、Unity での SOLID 開発とデザインパターンの入門編です。

— *C# スタイルガイドの作成（Unity 6 版）* では、プロジェクトを設定する際のベストプラクティスをいくつか説明しています。

その他のベストプラクティスガイド

— Unityドキュメントの 上級者向けベストプラクティスガイド

— Unity ベストプラクティスハブ

Unite の参考資料

— Overthrowing the MonoBehaviour tyranny in a glorious ScriptableObject revolution (ScriptableObject 革命で MonoBehaviour の専制を打倒せよ) (Youtube翻訳音声・字幕推奨)

— ScriptableObject を使用したゲームアーキテクチャ (Youtube翻訳音声・字幕推奨)

コードアーキテクチャに関する一般情報については、以下もおすすめです。

— From *Pong* to 15-person project (Pong から 15 人のプロジェクトへ)
(Youtube翻訳音声・字幕推奨)

その他のプロジェクト例

次のコミュニティプロジェクトの例では、アーキテクチャで ScriptableObject が幅広く使用されています。

— Unite のTanks デモのプレゼンテーション (Richard Fine氏)では、ScriptableObject を使用してオーディオ、プラグ可能な AI、破壊可能な建物をカスタマイズしています。

— Meta のシニアフルスタックゲームエンジニアである Ryan Hipple 氏の GitHub プロジェクト では、Unite Austin でのプレゼンテーションの大部分を説明しています。氏による、該当する ブログ記事 も参照してください。

— アセットストアで入手できる Soap (Scriptable Object Architectural Pattern) は、この e-book で説明されているパターンの多くを実装し、ScriptableObject を扱う際の利便性を大幅に向上させる機能を追加します。

— Unity Atoms プロジェクトは、ScriptableObject を幅広く使用するオープンソースライブラリです。このページ を読み、使用を開始してください。

— Reactive Menu System は、ScriptableObject を使用して、UI の状態の変更とメニュー管理を行う UGUI システムを構築します。

— Unity Open Project (Chop Chop) は、ScriptableObject に大きく依存しており、イベントチャンネルを使用してゲームプレイを管理します。

ゲームデザイナー向け

Christo Nobbs 氏は、システムゲームデザインと Unity (C#) を専門とするシニアテクニカルゲームデザイナーで、*The Unity Game Designer Playbook* の寄稿者であり、Unity におけるゲームシステム設計に関する短いブログ記事シリーズの主な執筆者でもあります。

— 生態系を作るシステム: 創発的ゲームデザイン

— 予測できない面白さ: ゲームデザインにおけるランダム化の価値

— 究極のデザインレバー、アニメーションカーブ

プロフェッショナルトレーニング

Unity プロフェッショナルトレーニングは、オンラインと対面の両方でトレーニングを提供し、あなたとあなたのチームが Unity でより生産的に作業し、効率的にコラボレーションするためのさらなるスキルと知識を習得できるよう支援します。Unity プロフェッショナルトレーニングの詳細については、こちら を参照してください。


Unity®