本制作のための考え方と技術

試作と本制作の違い

試作の特徴

- アイディアの状態のゲーム内容を、具体化して、面白いかを試す
- 未確定のことは、作りながら考える
- 動きが確認できれば、作り方は問わない。早く動かして確認する
- 面白くなければ、ボツにする
- **1**年でやってきたこと

本制作の特徴

- ゲーム内容は確定している
- 開発する項目はリストアップされて、スケジュールや予算が決まる
- ゲーム内容を考えながら開発することはない
- 大勢で、同時に、効率よく開発できるように設計して、開発を進める
- 余程のことがない限り完成させる必要がある
- **2**年を通して身につけたいこと

大勢で、同時並行で開発するには

- 同時並行して開発できるように、ゲームシステムを分割する
- 機能が実装されている場所が、直感的に分かるようにする
- 変更時の影響が少ないように設計する
- 開発するものを、開発担当者に間違いなく伝えるための資料を作成する
- 要求されたものを、間違いなく理解する読解力を身につける
- テストを用意して、バグがないことを自主確認する

分割の基本：単一責任の原則

- SOLID原則の1つ
- あるクラス（などのモジュール）の変更の理由が、複数あってはならない
- 1つのクラスの役割は、1つに絞る
- 複数の役割があるクラスは、分割を検討する

コード分割のポイント

- 状態ごとに、クラスを用意する
- 状態管理、移動、当たり判定、アニメなど、機能ごとにクラスを分ける
- ゲームシステムとゲーム内で分ける

コードの分割を活かす技術

分割したコードを組み合わせたり、再利用するのに役立つ仕様が、オブジェクト指向言語には用意されている。代表的な機能を紹介する。

デリゲート

メソッドを変数のように渡して、相手に呼び出させる機能。C#では、delegateキーワードを使ってメソッドを型として定義すると、変数と同じようにメソッドを代入できるようになる（参考：https://learn.unity.com/tutorial/delegates）。

Unityでは、デリゲートを手軽に扱えるようにUnityAction型（https://docs.unity3d.com/6000.3/Documentation/ScriptReference/Events.UnityAction.html）が提供されている。これを使うと、delegateによるメソッドの定義を省略できる。利用例を、次に示す。

DelegateUser.cs

次のものは、UnityActionでメソッドを受け取って、それを実行するUseDelegateメソッドを持つクラスの例である。

```
using UnityEngine;
using UnityEngine.Events;

/// <summary>
/// デリゲートの実装例。
/// </summary>

public class DelegateUser : MonoBehaviour
{
    public void UseDelegate(UnityAction action)
    {
        Debug.Log($"基本処理");

        // 処理が終わったら、受け取ったメソッドを呼び出す。
        action?.Invoke();
    }
}
```

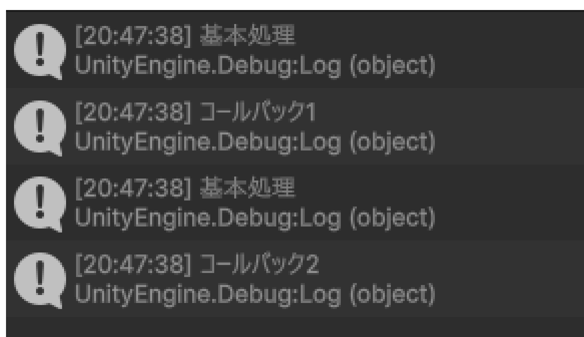
「action?」の「?」は、もしactionがnullだった場合、実行を取りやめるC#の便利機能である。

DelegateCaller.cs

DelegateUserクラスのUseDelegateメソッドを呼び出す例を、次に示す。

```
using UnityEngine;

/// <summary>
/// デリゲートを利用する側
/// </summary>

public class DelegateCaller : MonoBehaviour
{
    [SerializeField]
    DelegateUser delegateUser = default;

    void Start()
    {
        delegateUser.UseDelegate(Callback1);
        delegateUser.UseDelegate(Callback2);
    }

    void Callback1()
    {
        Debug.Log("コールバック1");
    }

    void Callback2()
    {
        Debug.Log("コールバック2");
    }
}
```

InspectorウィンドウのDelegate User欄に、DelegateUserをアタッチしたゲームオブジェクトを設定して実行すると、コンソールに次のように表示される。



基本処理が実行されたあとに、UseDelegateに渡したメソッドが呼び出されていることが分かる。メソッドを呼び出す際に、実行して欲しいメソッドを自由に入れ替えられる。

引数付きデリゲート

ジェネリックを利用することで、引数を持つメソッドのデリゲートも利用できる。メソッドは、次のような定義になる。Invokeの引数に、ジェネリックで指定した型の値を渡せる。

```
public void UseArgDelegate(UnityAction<int> action)
{
    action?.Invoke(0);
}
```

呼び出し側は、次のとおりである。

```
void Start()
{
    delegateUser.UseArgDelegate(CallbackArg);
}

void CallbackArg(int arg)
{
    Debug.Log($"受け取った値={arg}");
}
```

引数は、UnityAction<int, string, float, bool>のように、コンマ区切りで4つまで増やせる。それ以上必要な場合は、delegateを使って、自前で用意する。

デリゲートのメリットとデメリット

デリゲートを使うと、呼び出すメソッドを特定せずに、コードが書ける。これには、2つのメリットがある。1つ目は、呼び出したいメソッドを持つクラスがなくても、開発できることである。仮のメソッドを渡せば、開発を進めることができる。2つ目は、コードを変更せずに、機能を入れ替えられることである。一連の手続きは同じで、呼び出す機能が異なるような場合に、渡すメソッドを変えれば、同じプログラムを使い回すことができる。

デメリットとしては、受け取ったメソッドを呼び出しているだけなので、何を実行しているかの追跡がしにくいことが挙げられる。実行内容が分からなくならないように、シンプルさを保つのが大切である。

イベント

複数のデリゲートを登録して、まとめて呼び出す仕組みがイベントである（参考：https://learn.microsoft.com/ja-jp/dotnet/csharp/language-reference/keywords/event）。

Unityでは、イベントを手軽に扱えるようにUnityEventクラスが提供されている。ButtonのOnClickなどは、UnityEventを利用している。

イベントを外部に公開して、あるタイミングで実行してほしいメソッドを、外部から登録できる仕組みにする。イベントを実行する側は、どのようなメソッドが登録されているかを把握する必要はない。特定のタイミングになったら、そのイベントをInvokeで実行すればよい。

代表的な利用例として、スコアの更新と表示が挙げられる。スコアを管理するクラスに、スコアが変更した時に実行するイベントを用意する。スコア表示を更新する処理を、そのイベントに登録すれば、スコアが変化したら、自動的に表示が更新されるようになる。このように、データを管理するクラスに処理を登録して、必要なタイミングで望みの機能を呼び出させる仕組みを、オブザーバーパターンと呼ぶ。

Score.cs

標準的なスコアの管理用の値オブジェクト。

```
using UnityEngine;
using UnityEngine.Events;

public class Score
{
    /// <summary>
    /// スコアの上限値
    /// </summary>
    public static int ScoreMax => 999999;

    /// <summary>
    /// スコアが変更した時に呼び出したい処理を登録しておくイベント。
    /// </summary>
    public UnityEvent<int> Changed = new();

    /// <summary>
    /// 現在のスコア
    /// </summary>
    int currentScore;

    /// <summary>
    /// スコアを指定の点数にする。
    /// </summary>
    public void Set(int newScore)
    {
        currentScore = Mathf.Clamp(newScore, 0, ScoreMax);
        Changed.Invoke(currentScore);
```

```
    }

    /// <summary>
    /// スコアを加算する。
    /// </summary>
    /// <param name="add">加算値</param>
    public void Add(int add)
    {
        Set(currentScore + add);
    }
}
```

ScoreUser.cs

Scoreの利用例のクラス。

```
using UnityEngine;

public class ScoreUser : MonoBehaviour
{
    Score score = new();

    void Start()
    {
        score.Changed.AddListener(PrintScore);
        score.Set(0);
        score.Add(100);
        score.Add(1000);
        score.Add(10000);
        score.Add(100000);
        score.Add(1000000);
    }

    private void OnDestroy()
    {
        score.Changed.RemoveListener(PrintScore);
    }

    void PrintScore(int score)
    {
        Debug.Log($"Score={score}");
    }
}
```

これを実行すると、コンソールには次のように表示される。

Start()メソッドの最初で、スコアが変化した時に、スコアをログに表示するPrintScoreメソッドを、ScoreクラスのChangedイベントに登録している。それから、0点を設定してから、100点、1000点、10000点、100000点、1000000点と加算している。Start()メソッド内では、PrintScoreメソッドを呼び出していないが、スコアが変化するごとに、コンソールに現在のスコアが表示されている。

スコアの表示を、TextMesh Proに変更するために、Scoreクラスを変更する必要はない。Changedイベントに登録するメソッドを変えれば、表示先はいくらでも変えることができる。これが、イベントを使うメリットである。

オブザーバーパターンとポーリング

この例のように、ある状況で呼び出して欲しい処理をイベントに登録して、必要なタイミングで登録したイベントを呼び出す実装方法を、オブザーバーパターンと呼ぶ。これに対して、外部で**Score**の値を常に確認して、表示を制御する実装方法を、ポーリングと呼ぶ。

値がよく変化する場合はポーリング。値の変化の頻度が低い場合は、オブザーバーパターンを利用するのが効率的である。

イベントの注意点

イベントに登録した処理が不要になったら、RemoveListenerなどで、登録したメソッドを削除する。OnDestroyメソッドで、RemoveListenerを呼び出せば、そのオブジェクトが削除されるタイミングで、イベントからメソッドを削除できる。

同じメソッドが多重で呼び出されたり、解放後のメソッドを呼び出してエラーになったりしないように、メソッドの登録と削除を管理する必要がある。

ポリモーフィズム

オブジェクト指向言語の特徴は、次の3つと言われている。

1. カプセル化
2. 継承
3. ポリモーフィズム

Microsoftは、抽象化を加えた4つを提唱している(
https://learn.microsoft.com/ja-jp/dotnet/csharp/fundamentals/tutorials/oop
)。

カプセル化

外部とやり取りが必要な機能のみ公開して、オブジェクトの内部的な実装や変数を隠すことをカプセル化という。private、protected、publicといった公開レベルで実現する。
オブジェクトは、他のオブジェクトの実装方針を知っている必要はないのが、設計の原則である。内部的な処理やデータへのアクセスを隠すことで、他のオブジェクトからの想定外の操作を防いで、プログラムの安全性を高める。

継承

あるオブジェクトの機能を引き継いだ新しいオブジェクトを定義する機能である。クラスの定義の後ろに「:」を書いて、継承したいクラスを指定する。
Unityでは、MonoBehaviourクラスが、ゲームオブジェクトにアタッチする機能を提供している。新しいC#スクリプトを作成すると、自動的にMonoBehaviourを継承したクラスとして定義される。
DRYの原則(Don't repeat yourself)や、ポリモーフィズムに関連する。

ポリモーフィズム

ポリモーフィズムは、次の2つの機能から成る。

1. あるオブジェクトのインスタンスは、そのオブジェクトの親クラスや、そのオブジェクトが実装するインターフェースの型に代入できる
2. 親クラスやインターフェースの型に代入したインスタンスのメソッドを呼び出すと、代入したインスタンスのメソッドが呼び出される

複数の状態やキャラクターに共通の機能があって、機能の中身が種類によって異なるような場合に、ポリモーフィズムが活用できる。次の手順で、実装を進める。

1. 複数のものを抽象化(アブストラクト)して、共通する機能をリストアップする
2. リストアップした機能を、親クラスやインターフェースに、メソッドとして定義する(デフォルトの機能があれば親クラス。なければインターフェース。なるべく、インターフェースを利用する)
3. 状態やキャラクターを制御する具体的なクラス(コンクリートクラス)を、親クラスを継承したり、インターフェースを実装して、作成する
4. 各状態やキャラクターのインスタンスは、コンクリートクラスではなく、親クラスや、インターフェースに代入して扱う

ポリモーフィズムの例

親クラスを利用する方法と、インターフェースを利用する方法のそれぞれの例を示す。

PolyBase.cs

挨拶を表示するHelloメソッドを定義した親クラスPolyBaseを、次のように定義する。

```
using UnityEngine;

/// <summary>
/// 親クラス
/// </summary>
public class PolyBase
{
    /// <summary>
    /// 挨拶機能
    /// </summary>
    public virtual void Hello()
    {
        Debug.Log("Hello PolyBase!");
    }
}
```

Hello()メソッドには、デフォルトの動作を実装している。継承先のクラスで機能を上書きできるように、virtualキーワードをつけている。

IPolyable.cs

親クラスと違う挨拶を返すHelloInterfaceメソッドを定義したインターフェースを、次のように定義する。

```
/// <summary>
/// インターフェースの例
/// </summary>
public interface IPolyable
{
    /// <summary>
    /// インターフェースに定義した挨拶メソッド
    /// </summary>
    void HelloInterface();
}
```

インターフェースに定義するメソッドは、無条件でpublicになるので、公開修飾子は不要。また、デフォルトの振る舞いは実装できない。メソッド名、戻り値、引数のみを定義する。

PolyA.cs

PolyBaseを継承し、かつ、IPolyableインターフェースを実装したクラスPolyAを、次のように作成する。

```
using UnityEngine;

public class PolyA : PolyBase, IPolyable
{
    public override void Hello()
    {
        Debug.Log("Hello PolyA!");
    }

    public void HelloInterface()
    {
        Debug.Log("Hello Interface PolyA!!");
    }
}
```

Hello()メソッドは、PolyBaseに定義されたものを上書きするので、overrideキーワードをつける。これは、意図せずに親クラスのメソッドを上書きしないための仕組みである。

HelloInterface()メソッドは、IPolyableインターフェースで定義したメソッドである。インターフェースのメソッドは、publicにする。また、デフォルトの実装は持たないので、overrideは不要である。

PolyB.cs

PolyBaseとIPolyableを、別の内容で実装したクラス。

```
using UnityEngine;

public class PolyB : PolyBase, IPolyable
{
    public override void Hello()
    {
        Debug.Log("Hello PolyB!");
    }

    public void HelloInterface()
    {
        Debug.Log("Hello Interface PolyB!!");
    }
}
```

PolySample.cs

ポリモーフィズムを実際に利用するコードを持ったクラス。

```
using UnityEngine;

/// <summary>
/// ポリモーフィズムの利用例
/// </summary>

public class PolySample : MonoBehaviour
{
    void Start()
    {
        // 継承を使ったポリモーフィズム
        var instances = new PolyBase[3];
        instances[0] = new PolyBase();
        instances[1] = new PolyA();
        instances[2] = new PolyB();

        foreach(var instance in instances)
        {
            instance.Hello();
        }

        // インターフェースを使ったポリモーフィズム
        var interfaces = new IPolyable[2];
        interfaces[0] = new PolyA();
        interfaces[1] = new PolyB();
        foreach(var i in interfaces)
        {
            i.HelloInterface();
        }
    }
}
```

以上をゲームオブジェクトにアタッチして実行すると、コンソールに次のように表示される。

```
[14:48:44] Hello PolyBase!
UnityEngine.Debug:Log (object)
[14:48:44] Hello PolyA!
UnityEngine.Debug:Log (object)
[14:48:44] Hello PolyB!
UnityEngine.Debug:Log (object)
[14:48:44] Hello Interface PolyA!!
UnityEngine.Debug:Log (object)
[14:48:44] Hello Interface PolyB!!
UnityEngine.Debug:Log (object)
```

繰り返し処理（foreach）で、配列のHello()やHelloInterface()を呼び出しているだけで、違う処理を呼び出せるのがポリモーフィズムの効果である。

デリゲートやポリモーフィズムを使わずに、繰り返し文を使って同様のことをやろうとすると、ifやswitchによる分岐が必要になる。デリゲートやポリモーフィズムを使うと、ifやswitchといった分岐をなくして、コードをスリムにできる。

PolyBaseは、IPolyableを実装していないので、インターフェースの例にPolyBaseは含められない。

デリゲート、イベント、ポリモーフィズムの効用

これらの技術により、実行する機能を、メソッドやオブジェクトのインスタンスの代入によって入れ替えられる。実行する内容をプログラム内に具体的に書く代わりに、デリゲートやインスタンスのメソッドの呼び出しにしておけば、中身をあとで入れ替えられる。

デリゲートやイベントは、UnityActionやUnityEventといった共通のクラスを参照しているだけで、そこに登録されるメソッドが、どのクラスのなんというメソッドかを知る必要はない。ポリモーフィズムも同様で、扱うのは機能を抽象化した親クラスやインターフェースである。実際に呼び出すクラスを知る必要はない。

あるクラスAから、他のクラスBの機能を呼び出すような実装があると、クラスAは、クラスBがなければ開発できない。このような状況を「クラス**A**は、クラス**B**に依存している」という。クラス間に依存関係が多数あると、用意するクラスが多くなる。また、開発する手順がややこしくなり、チーム開発が難しくなる。

デリゲートやイベント、ポリモーフィズムを使うと、クラスやメソッドを、抽象的に書くことができる。UnityActionやUnityEventは定義済みなので、依存しても開発の妨げにならない。ポリモーフィズムでは、親クラスやインターフェースを用意する必要はあるが、すべてのコンクリートクラスを定義するのに比べれば、数が少ないので手間は大幅に削減できる。この性質が、大勢で、同時並行して開発する際に役に立つのである。

他のオブジェクトの仕組みに手を突っ込まない

1人で開発していると、気にせずにやってしまうアンチパターン。単一責任の原則違反であり、これの予防に役立つのがカプセル化である。

悪い例

プレイヤーが接触したら、プレイヤーに力を加えるバウンドオブジェクトの悪い実装例を次に示す。

BadBouncer.cs

```
using UnityEngine;

public class BadBouncer : MonoBehaviour
{
    [SerializeField]
    Vector3 firstVelocity = 4 * Vector3.right;

    Rigidbody rb;

    void Start()
    {
        rb = GetComponent<Rigidbody>();
        rb.linearVelocity = firstVelocity;
    }

    /// <summary>
    /// 接触の悪い例
    /// </summary>
    /// <param name="other">接触相手</param>
    private void OnTriggerEnter(Collider other)
    {
        if (other.CompareTag("Player"))
        {
            Vector3 dir = other.transform.position - transform.position;
            var player_rb = other.GetComponent<Rigidbody>();
            player_rb.AddForce(
                rb.linearVelocity.magnitude * dir.normalized,
                ForceMode.VelocityChange);
        }
    }
}
```

悪い理由

- プレイヤーの制御方法を、RigidbodyからCharacterControllerに変更しようとすると、プレイヤーの仕様変更なのに、バウンドオブジェクトの変更も必要になる。これは予測できない場所であり、バグの原因となる
- バウンドオブジェクトのクラスを変更する理由が、バウンドオブジェクトだけではなく、プレイヤーの仕様変更も原因になるため、単一責任の原則違反
- プレイヤーに無敵状態があった場合などの状態に応じた制御が難しい

修正方法

- プレイヤーのクラスに、外部から力を受け取るメソッドを用意する
- バウンドするオブジェクトは、プレイヤーが用意したメソッドを呼び出す
- 受け取った力の反映は、プレイヤー自身に任せる

推奨例

プレイヤー用に、次のようなIBounceableインターフェースと、Playerクラスを定義する。

IBounceable.cs

```
using UnityEngine;

public interface IBounceable
{
    /// <summary>
    /// バウンドを設定する。
    /// </summary>
    /// <param name="dir">バウンドベクトル</param>
    void SetBounce(Vector3 bounce);
}
```

Player.cs

```
using UnityEngine;

public class Player : MonoBehaviour, IBounceable
{
    Rigidbody rb;

    void Awake()
    {
        rb = GetComponent<Rigidbody>();
    }

    public void SetBounce(Vector3 bounce)
```

```
    {
        rb.AddForce(bounce, ForceMode.VelocityChange);
    }
}
```

バウンドオブジェクトのクラスは、次のようになる。

GoodBouncer.cs

```
using UnityEngine;

public class GoodBouncer : MonoBehaviour
{
    [SerializeField]
    Vector3 firstVelocity = 4 * Vector3.right;

    Rigidbody rb;

    void Start()
    {
        rb = GetComponent<Rigidbody>();
        rb.linearVelocity = firstVelocity;
    }

    /// <summary>
    /// 接触の推奨例
    /// </summary>
    /// <param name="other">接触相手</param>
    private void OnTriggerEnter(Collider other)
    {
        var bounce = other.GetComponent<IBounceable>();
        if (bounce != null) {
            Vector3 dir = other.transform.position - transform.position;
            bounce.SetBounce(
                rb.linearVelocity.magnitude * dir.normalized);
        }
    }
}
```

推奨例は、インターフェースやプレイヤーへの実装が必要であり、悪い例より手間が増える。試作であれば、手間が少ない悪い例でも問題ない。しかし、本制作では、多少の手間が増えても、推奨例を選びたい。

推奨例であれば、プレイヤーの制御方法の変更が、バウンドオブジェクトに影響を与えることはない。また、バウンドの指示と実装を結びつける**IBounceable**インターフェースを用意したことで、**Player**クラスがなくても、バウンドオブジェクトを開発できる。さらに、プレイヤーだけではなく、IBounceableを実装したすべてのキャラに、力を加えられるようになっている。Playerクラス

というコンクリートクラスを直に参照するのではなく、抽象的なインターフェースを介することで、同時開発が進めやすくなり、かつ、バウンドオブジェクトが影響を与える相手を拡張できる。

インターフェースかクラスか

基本的には、インターフェースは定義した方がよい。ただ、管理が大変になるという面があるので、ちょっとしたものであれば、インターフェースは省略しても構わない。その場合、必要なメソッドを定義しただけのクラスのひな型を用意して、開発者に渡すとよい。

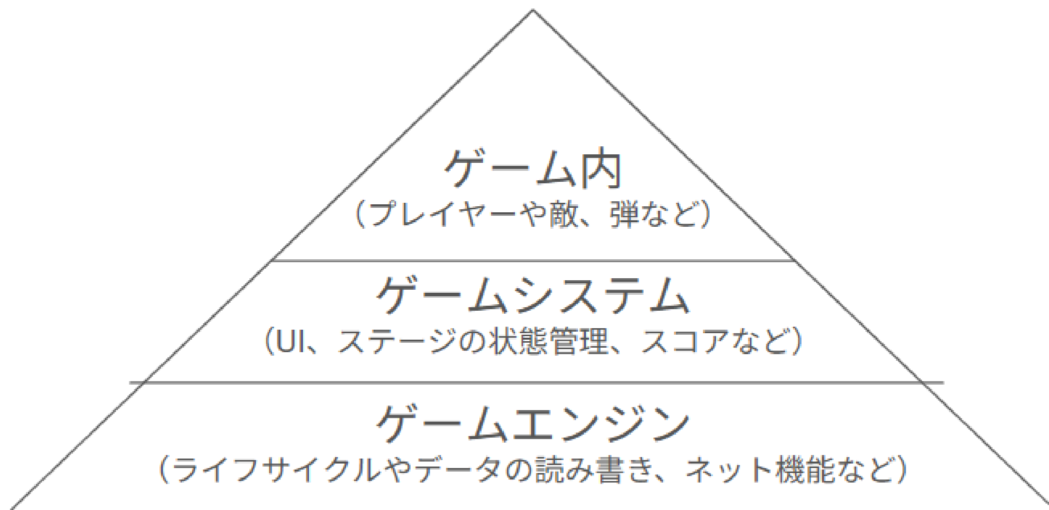
インターフェースを定義するかどうかの明確な判断基準はない。チーム内で合意できていればよい。

インスタンスの受け渡し戦略

あるオブジェクトが、他のオブジェクトとやり取りするには、相手のインスタンスを知る必要がある。オブジェクトを正しく分割しても、インスタンスの受け渡し方法がイマイチだと、システムの柔軟性が落ちてしまう。いくつかの受け渡し方法を紹介する。

オブジェクトがいる階層

ゲームやアプリは、次のようなソフトウェアの階層で構築される。



Unityのゲームオブジェクトは、これらを横断している部分があるので、厳密に分ける必要はない。ただ、インスタンスを受け渡す時に、やりとりするオブジェクトが、どの階層に所属するかを考えると、戦略が立てやすくなる。

依存の向き

オブジェクト指向の設計指針であるSOLID原則の１つに、「依存関係逆転の原則」がある。上記のような階層において、下から上の方向（上下が逆）に、依存させよ、というものである。

プレイヤーや敵を制御するクラスは、ゲームシステムやゲームエンジンの機能への直接の参照は避ける。そのための手段が、デリゲートやイベント、ポリモーフィズムである。イベントやインターフェースを介して、ゲームシステムやゲームエンジンの機能を呼び出せば、直にゲームシステムやゲームエンジンにアクセスする必要がなくなる。

具象から抽象へ依存する

プレイヤーと敵、あるいは、スコアとUIといった、同じ階層のオブジェクト間では、依存関係逆転の原則が適用できない。そのような場合は、具象から抽象へ依存させる方針を採用する。抽象は、親クラスやインターフェースのことである。先に示したソフトウェアの階層の上の方ほど、インターフェースを多様する。

オブジェクトの親子関係

親子関係でまとめられたオブジェクト同士は、GetComponentやInspector渡しを使って、親子内で解決する。検索対象をインターフェースにしたり、Inspector渡しにすると、再利用性が上げられる。

接触するオブジェクト同士

例えば、プレイヤーと、アイテムや仕掛けのように、接触するオブジェクト同士なら、接触したときにインスタンスを交換できる。Unityであれば、OnTriggerEnterやOnCollisionEnterで、Unityから報告される相手のインスタンスから、必要な情報を取り出せばよい。予め、お互いのインスタンスを知っておく必要はない。

スイッチと仕掛けのような組み合わせ

あるスイッチを操作すると、離れた場所にある対応した仕掛けが発動する、という仕組みでは、接触によるインスタンスの受け渡しができない。このような場合は、次の2種類の方法を検討する。

Inspector渡し

ペアとなる相手のインスタンスの変数を[SerializeField]で公開して、Inspectorで設定する方法。ステージをプランナーが作る場合は、この方法が適している。

インスタンスを渡すスクリプトを用意

ScriptableObjectやエクセルなどで、ペアとなるオブジェクトのデータを用意して、それを元に、オブジェクトにインスタンスを渡すスクリプトを用意する。

Inspector渡しの欠点は、正しくデータが設定されているかを確認するのに、個々のオブジェクトのチェックが必要なことである。仕掛けの数が増えると、管理が大変になる。ペアを管理する仕組みを作ることで、管理の手間を減らすことができる。

アンチパターン

ペアとなるオブジェクトを、Findなどで自力で検索して見つけるのは避けたい。これをやると、相手の指定がコード内にハードコーディングされてしまい、再利用しにくいコードになる。また、オブジェクト名に依存すると、タイプミスなどで、設定が外れてしまう恐れがある。このようなミスは見つけにくく、生産性を落とす原因になる。

Inspectorでドラッグ＆ドロップで渡すか、受け渡す仕組みを作るかして、オブジェクトの外部で設定することを検討する。

ゲームシステムとゲーム内のオブジェクト

スコアやゲームの状態管理といったゲームシステムと、プレイヤーやゴールなどのゲーム内のオブジェクトは、接触する機会がないので、何らかの方法でインスタンスを渡す必要がある。

DI（Dependency Injection = 依存関係の注入）

DIとは、あるオブジェクトが必要とするインスタンスを、外部から注入する方法のこと。Webでよく使われている技術で、UnityではvContainerというライブラリが有名である。基本的には、単にインスタンスを渡せばよいので、vContainerがなくてもDIはできる。

依存関係逆転の原則から、ソフトウェアの階層の下から上に対して、インスタンスを渡すのがセオリー。ゲームのシーンを管理するスクリプトなどを用意して、その初期化時に、ゲーム内のキャラが必要とするインスタンスを渡せばよい。

シングルトン

ゲームシステムに1つしか存在しない情報や操作は、シングルトンにして、ゲーム全体から参照できるようにする方法が考えられる。小さいゲームなら、スコアなどのゲームシステムを保持するオブジェクトをシングルトンにすれば、プレイヤーなどから簡単にアクセスできるようになる。

シングルトンは手軽に扱えるが、次のような問題が指摘されている。

- チーム開発時に、事前にシングルトンを用意する必要がある
- シングルトンに強く依存することになる
- テスト時のインスタンスの差し替えが難しい
- 自動的に初期化されないので、初期化漏れによる不具合の恐れ

シングルトンを採用するかは、チーム内で十分に話し合って決める。

DI vs シングルトン

依存を減らしたり、チーム開発のやりやすさは、DIが有利。DIの弱点は、依存先を見つけてインスタンスを送り込む仕組みが必要なことと、処理に時間がかかることである。

シングルトンは、どこからでも直に参照できるので、処理速度は最速であり、コードもシンプルに保てる。弱点は、シングルトンへの強い依存や、初期化の難しさである。

一長一短があるので、プロジェクトの規模や扱う内容に応じて、最適と考えられるものを選択する。

インスタンス渡しのまとめ

インスタンスを渡す方法として、次のようなものが考えられる。

- 親子関係があるオブジェクトは、GetComponentやInspector渡し
- 接触するものは、接触時にやり取りする
- 接触しないオブジェクト同士なら、Inspectorや、インスタンスを渡すためのスクリプトで受け渡す
- ゲームシステムとゲーム内は、ゲームシステムからゲーム内にDIでインスタンスを注入するか、シングルトンにしたクラスをゲーム内から参照する

オブジェクトが、必要なインスタンスを外部に探しに行くと、依存が発生しやすくなる。外部から受け取るDIの構造を検討するとよい。