Unity®

UNITY 6 プレビュー版 → E ブック

デザインパターンと SOLID で コードをレベルアップ

目次

デザインパターン入門

Unity を使用して作業する際に、すべてをゼロから開発する必要はありません。必要なものは、誰かがすでに作成している可能性が高いです。

ソフトウェアデザインに関して皆さんを翻弄しているさまざまな問題についても、これまで多くの開発者がいつかどこかで既に経験してきたものです。先人に直接アドバイスを求めることはできないかもしれませんが、デザインパターンを通じて先人がどのような判断を下してきたかを学ぶことができます。

デザインパターンとは、ソフトウェアエンジニアリングにおける一般的な問題に対する汎用的なソリューションのことです。コピーして、そのまま自分のコードに貼り付けてしまえるような完成したソリューションではありませんが、デザインパターンは、手元の道具箱に入れておくプラスアルファの道具だと考えることができます。デザインパターンは、わかりやすい自明なものから、そうでないものまで様々です。

このガイドでは、Unity 開発においてよく知られているデザインパターンを集めました。このガイドで紹介する例は、理解しやすいよう簡略化しており技術的な専門用語も減らしていますが、実際に作業する際には、C# の基礎的な知識を習得しておく必要があります。

重要事項:この第 2 版には、Unity コミュニティのメンバーからリクエストのあった新しいパターンをいくつか追加しています。さらに、このガイドに掲載しているコード例とプロジェクトは、Unity 6 で動作するようアップグレードされています。Unity 6 は今年後半にリリースされます。このガイドのコード例やデモプロジェクトに従って作業する場合は、Unity 6 プレビュー版 をダウンロードしてください。

デザインパターン初心者の方や、簡単に復習しておきたい方向けに、そのデザインパターンを適用できる一般的なゲーム開発シナリオも紹介しています。別のオブジェクト指向言語 (Java、C++ など) から C# に移行する方のために、これらのサンプルでは Unity に特化したパターン適応方法を説明しています。

デザインパターンとは、端的に言えばアイデアのことです。すべての状況に適合するわけではありません。しかし正しく使用することで、スケールする大規模なアプリケーションの構築に役立ちます。プロジェクトに組み込むことで、コードの可読性を高め、コードベースを整理された状態に保ちます。パターンの使用経験を重ねることで、どのタイミングで使用すれば開発プロセスのスピードアップにつながるかわかります。

すでにあるものを作り直すのではなく、新しいものに取り組みましょう。

寄稿者

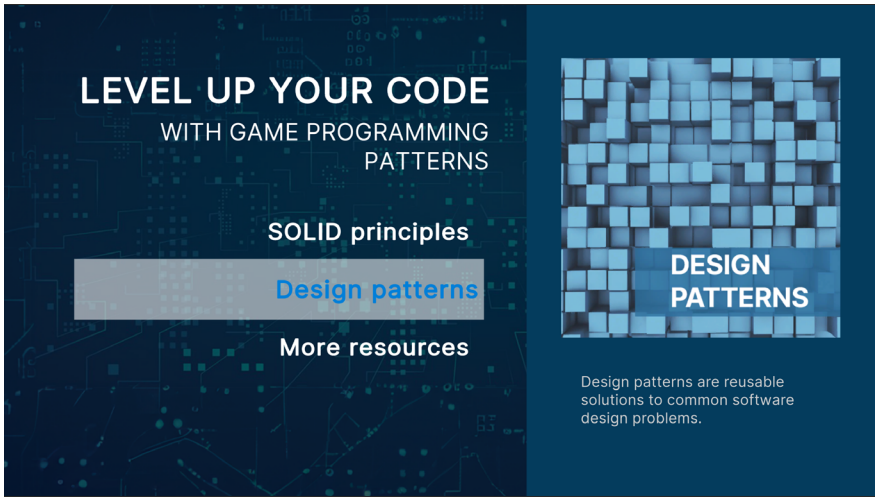
このガイドは、ビジュアルエフェクトアーティストとして映画業界やテレビ業界にて 3D と VFX の制作に 15 年以上にわたって取り組んできた経験を持ち、現在は独立系ゲーム開発者兼教育者である、Wilmer Lin 氏によって制作されました。制作には、シニアテクニカルコンテンツマーケティングマネージャー Thomas Krogh-Jacobsen のほか、Unity のシニアエンジニア Peter Andreasen と Scott Bilas も大きく貢献しています。

このガイドの使い方

このガイドの目的は、コードについて考え、整理する新たな方法を提示することです。さまざまなソフトウェアデザインパターンを、Unity 開発に特化して紹介しています。

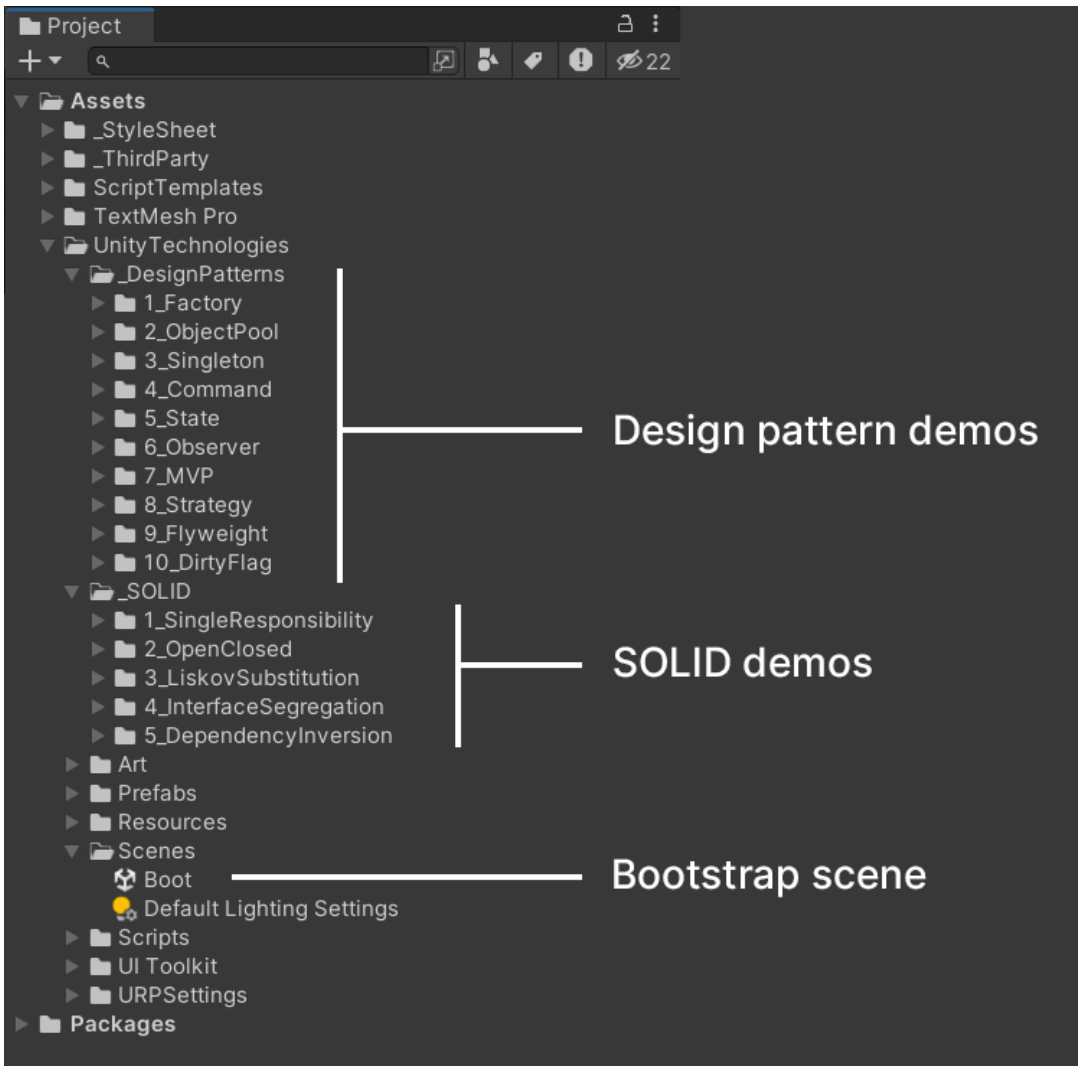
サンプルプロジェクト

この e ブックには、コンテキストに沿ったコードの一部を示す サンプルプロジェクト が付属しています。Asset Store からプロジェクトをダウンロードし、対応するシーンを利用して、デザインパターンとそれを支える原則を確認してください。



メニューから SOLID とデザインパターンのサンプルに移動する。

Boot シーンからプロジェクトを開始します。これは、デモを構成し、メインメニューへのアクセスを提供するブートストラップシーン (後述の囲み記事参照) です。メニューから目的のサンプルに移動できます。それぞれのシーンでは、異なる SOLID コンセプトあるいはデザインパターンが示されます。



サンプルプロジェクトの構成。

サンプルプロジェクトと本ガイドのコードサンプルでは、若干の違いが生じる場合があることにご注意ください。わかりやすさと読みやすさを優先し、一部の例では簡略化されたコード (public フィールドなど) を使用しています。

チームによっては、本ガイドやサンプルプロジェクトで使用されている表記法とは異なるコーディングスタイルを好む場合もあるでしょう。具体的なニーズに特化した C# スタイルガイドを作成し、チーム全体で一貫してそれに従うことをお勧めします。

詳細なガイダンスは、e ブック *C# スタイルガイドを作成する:拡張性のある、よりクリーンなコードを書く* を参照してください。コードスタイルのガイドラインに適応、作成、実装する方法のヒントが得られます。

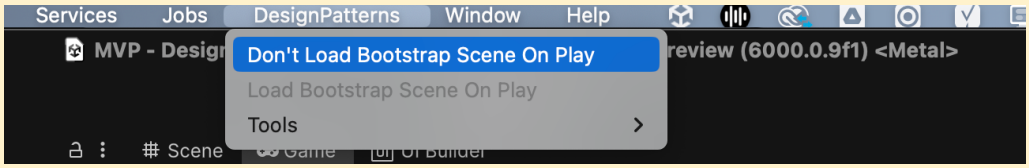
SceneBootstrapper

このプロジェクトには、プロジェクトを複数の Unity シーンに分割するときの開発プロセスを効率化するために用意された SceneBootstrapper クラスがあります。

ブートストラップロジックは、再生モードに入るたびに、指定した **Boot** シーンを最初に自動的にロードします。そのため、Boot シーンは、**File > Build Settings** で最初にリストされる必要があります。

このアプローチにより、ゲームの出発点に一貫性を持たせることができます。現在 Boot シーンを開いていない場合でも、再生モードにするとそこからプロジェクトが強制的にロードされます。

また、SceneBootstrapper は再生モードが開始される前にエディターで最後にアクティブだったシーンを追跡し、その情報を EditorPrefs に保存します。再生モードを終了すると、最後にアクティブだったシーンに戻るため、中断した場所から再開しやすくなります。

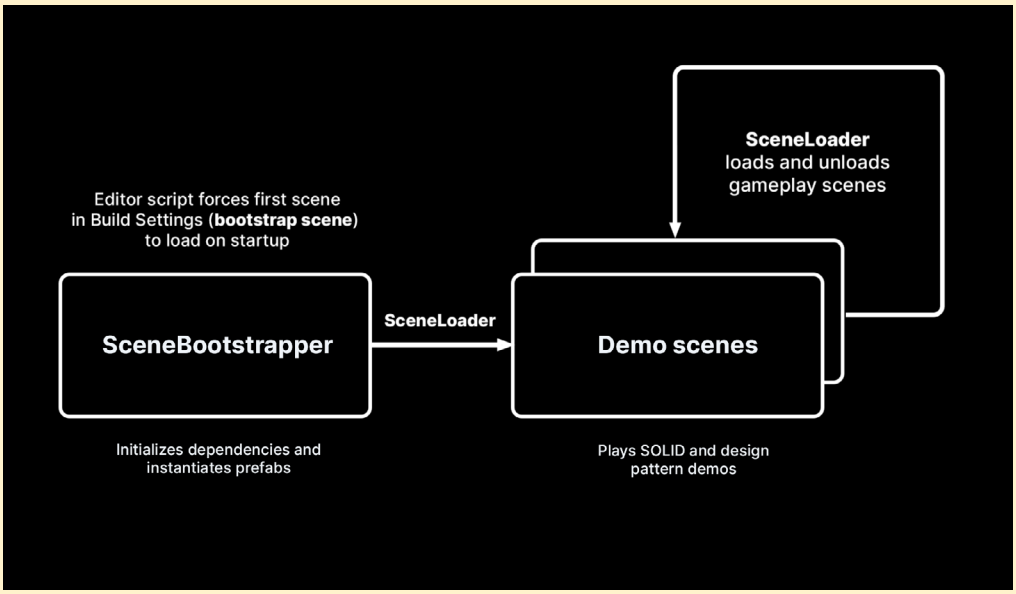


メニューを使用して SceneBootstrapper を切り替える。

Boot シーンを毎回経由せずに、シーンを個別に確認したい場合は、Design Patterns メニューから bootstrapper を無効にします (**Design Patterns > Don't Load Bootstrap on Play**)。同じメニューから bootstrapper を再び有効にできます。

アプリケーションを正しく機能させるには、すべてのシーンが Build Settings にリストされ、bootstrapper シーンがインデックス 0 に配置されている必要があります。これを行わない場合、実装例の `IsSceneInBuildSettings` メソッドはエラーになります。

Bootstrapper の仕組みの詳細については、付録のセクションを参照してください。この記事では、QuizU プロジェクトからの類似の bootstrapper を紹介しています。



Bootstrapper は Boot シーンをロードする。

KISS の原則

サンプルを参考にする際、問題にアプローチする "正しい方法" は 1 つだけではないことに留意してください。サンプルコードは、数あるソリューションのうちの 1 つに過ぎません。

疑問が生じた際には、本ガイドのすべてを、KISS の原則 (keep it simple, stupid;できるだけ単純にしておく) のフィルターを通して見るようにしてください。必要な場合のみ、複雑さを追加するようにしましょう。

すべてのデザインパターンにはトレードオフが伴います。維持すべき構造の追加や、初期設定の追加が必要になります。実装する前に、デザインパターンがもたらすメリットが追加作業に見合うものであるかどうかを判断してください。

あるパターンが今抱えている具体的な問題に当てはまるかどうか定かでない場合は、そのパターンがより自然に適合するように感じる状況になるまで待つことをお勧めします。新しいから、斬新であるから、という理由でパターンを使用するのではなく、必要なときに使用してください。

そうすることで、より優れたソフトウェアの開発をサポートするという本来の目的で、デザインパターンを活用できるようになります。

それでは、始めていきましょう。

SOLID 原則

SOLID は、ソフトウェアデザインにおける核となる 5 つの原則の 略語 です。オブジェクト指向 のデザインを、理解しやすく、柔軟で、保守しやすく するためにコーディング時に留意すべき 5 つの基本ルールと考えてください。

デザインパターンの説明の前に、デザインパターンに影響するいくつかのデザイン原則を見てみましょう。

5 つの基本原則は以下のとおりです。

— 単一責任 (Single responsibility)

— オープン/クローズド (Open-closed)

— リスコフの置換 (Liskov substitution)

— インターフェース分離 (Interface segregation)

— 依存性逆転 (Dependency inversion)

これらの原則のコンセプトを確認し、コードをよりわかりやすく、柔軟で、メンテナンスしやすくするのにどう役立つかを見ていきましょう。

単一責任の原則

クラスを変更する理由は 1 つであるべきで、1 つのクラスは 1 つの責任のみを持つべきです。

SOLID における最初のかつ最も重要な原則は、単一責任の原則 (SRP) です。これは、各モジュール、クラス、関数は 1 つの責任を持ち、ロジックのその部分のみをカプセル化するという原則です。

つまり、1 つの巨大なクラスを作成するのではなく、多数の小規模なクラスを作成するべきだということです。短いクラスやメソッドのほうが説明が簡単で、理解しやすく、実装も容易です。

Unity の使用経験があれば、このコンセプトにすでに慣れていることでしょう。ゲームオブジェクトを作成すると、その中には小さな各種コンポーネントが保持されます。例えば、以下のようなものです。

3D モデルへの参照を格納する MeshFilter

— モデルのサーフェスが画面上にどのように表示されるかを管理する Renderer

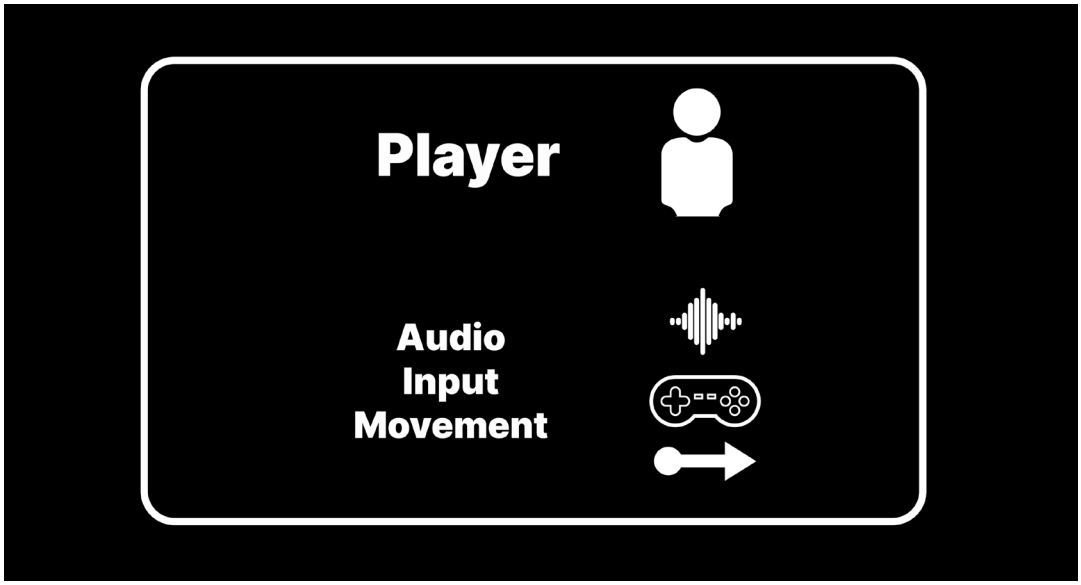
— スケール (scale)、回転 (rotation)、位置 (position) が格納される Transform コンポーネント

— 物理シミュレーションと相互作用する必要がある場合は Rigidbody

各コンポーネントは 1 つのことを担い、十分な役割を果たします。シーン全体は、複数のゲームオブジェクトで構成されます。それらのコンポーネント間の相互作用で、ゲームは成り立っています。

スクリプト化されたコンポーネントも同様に構築します。それぞれが明確に理解できるようデザインします。そして、それらを協調させて複雑な動作を実現します。

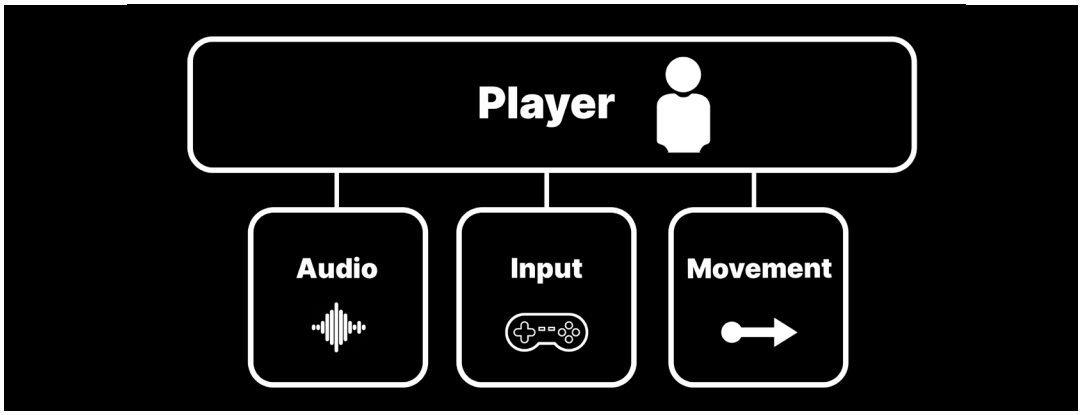
単一責任の原則を無視すると、以下のようなカスタムコンポーネントになってしまうおそれがあります。



複数の責任を持つ Player スクリプト

```
public class UnrefactoredPlayer :MonoBehaviour
{
    [SerializeField] private string inputAxisName;
    [SerializeField] private float positionMultiplier;
    private float yPosition;
    private AudioSource bounceSfx;

    private void Start()
    {
        bounceSfx = GetComponent<AudioSource>();
    }

    private void Update()
    {
        float delta = Input.GetAxis(inputAxisName) * Time.deltaTime;

        yPosition = Mathf.Clamp(yPosition + delta, -1, 1);
        transform.position = new Vector3(transform.position.x, yPosition *
        positionMultiplier, transform.position.z);
    }

    private void OnTriggerEnter(Collider other)
    {
        bounceSfx.Play();
    }
}
```

この UnrefactoredPlayer クラスにはいくつかの役割が混在しています。プレイヤーが何かに衝突すると音を再生し、入力を管理し、移動の処理も担います。現時点では比較的短いクラスであっても、プロジェクトの規模が大きくなるにつれて、メンテナンスは難しくなります。Player クラスを小さなクラスに分割することを検討してください。



単一の責任を持つ複数のクラスにリファクタリングされた Player

```
[RequireComponent(typeof(PlayerAudio), typeof(PlayerInput),
typeof(PlayerMovement))]
public class Player :MonoBehaviour
{
    [SerializeField] private PlayerAudio playerAudio;
    [SerializeField] private PlayerInput playerInput;
    [SerializeField] private PlayerMovement playerMovement;

    private void Start()
    {
        playerAudio = GetComponent<PlayerAudio>();
        playerInput = GetComponent<PlayerInput>();
        playerMovement = GetComponent<PlayerMovement>();
    }
}

public class PlayerAudio :MonoBehaviour
{
    ...
}

public class PlayerInput :MonoBehaviour
{
    ...
}

public class PlayerMovement :MonoBehaviour
{
    ...
}
```

Player スクリプトがスクリプト化された他のコンポーネントを管理することは変わりませんが、各クラスは 1 つのことのみを担うようになっています。このデザインにより、特にプロジェクトの要件が時間とともに変化する場合に、コードの修正が容易になります。

一方で、単一責任の原則を適用する際には、常識の範囲内でバランスを取る必要があります。1 つのメソッドしか持たないクラスばかり作成するといった極端な簡略化は避けてください。

単一責任の原則に従って作業する場合は、以下のことを念頭に置いてください。

- **可読性:**クラスは短いほうが読みやすいです。明確なルールはありませんが、多くの開発者は 200 行から 300 行を上限としています。どの程度を "短い" と見なすかを、自分自身やチームで決めるようにしてください。この上限値を超えるときは、より小さなパーツにリファクタリングできないか検討してください。
- **拡張性:**小さなクラスのほうがより簡単に継承できます。意図せず機能を壊してしまうことを心配せずに、変更を加えたり置き換えたりできます。
- **再利用性:**クラスを小さく、かつモジュール化するようデザインすることで、ゲームのその他の部分で再利用できるようになります。

リファクタリングする際には、コードを再編成することが自分や他のチームメンバーの QOL (生活の質) をどれほど向上させるかを考慮に入れてください。最初にちょっとした手間をかけることで、後のトラブル回避につながることが往々にしてあります。

"シンプル" とは "簡単" のことではない

シンプルさはソフトウェアデザインにおいてよく話題に上がるテーマであり、信頼性を高めるうえで欠かせないものです。そのソフトウェアデザインは制作中に変更を処理できますか?長期にわたってアプリケーションを拡張してメンテナンスを行うことができますか?

このガイドで紹介するデザインパターンや原則の多くは、物事をシンプルにするのに役立ちます。そのことにより、コードの柔軟性が高まり、よりスケーラブルかつ読みやすくなります。ただし、そのためには追加作業と計画が必要です。"シンプル" と "簡単" は同義ではありません。

パターンを使用せずに同じ機能を作成することは (多くの場合、より短時間で) 可能ですが、時間をかけずに簡単に作成したものがシンプルなものに仕上がるとは限りません。シンプルなものを作ることは、焦点を絞るということです。1 つのことだけをさせるようデザインし、他のタスクで複雑にし過ぎないでください。

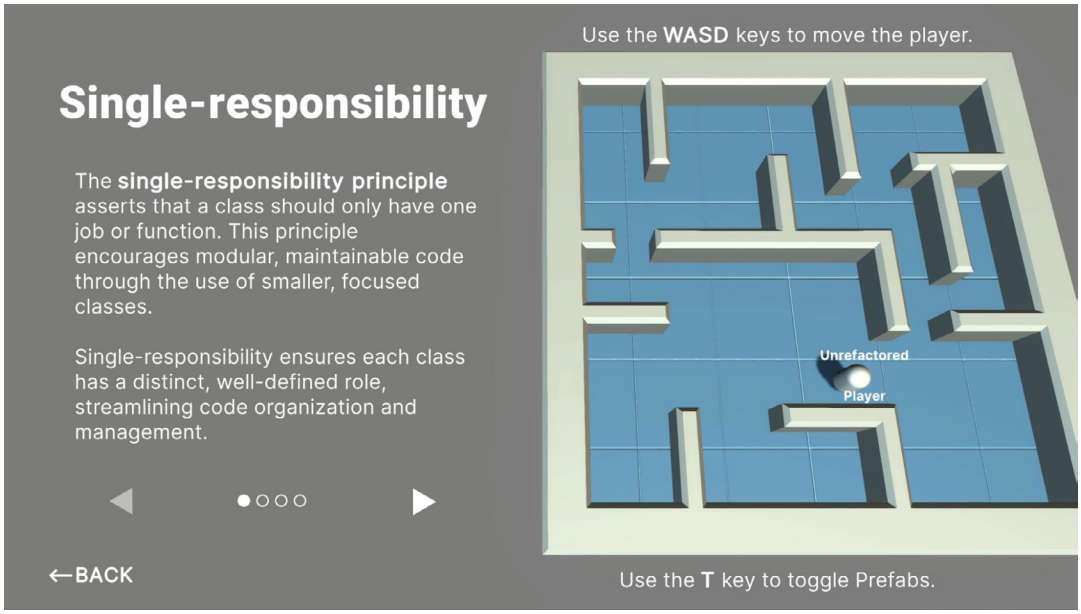
Rich Hickey 氏による講演 Simple Made Easy を視聴し、より優れたソフトウェア構築にシンプルさがどのように寄与するかについて理解を深めてください。

例：サンプルプロジェクト

サンプルプロジェクトには、プレイヤーキャラクターに単一責任の原則を適用する簡単なデモがあります。`Player` クラスは、プレイヤーの特定の動作それぞれを処理する他のスクリプトを参照します。

- `PlayerInput` は、キーボードからの入力をキャプチャして処理し、方向ベクトルに変換します。
- `PlayerMovement` は `PlayerInput` クラスからの入力ベクトルに基づいてプレイヤーの動きを制御します。
- `PlayerAudio` は、プレイヤーが障害物に衝突したときにサウンドエフェクトを再生します。
- `PlayerFX` は、プレイヤーのパーティクルシステムを処理します。

単一責任により、コードベースがよりモジュール化され、読みやすくなります。また、他のコンポーネントに影響を与えずに各コンポーネントを更新したり拡張したりすることが簡単になります。



単一責任のデモでは、Player が小さなコンポーネントに分割されている。

オープン/クローズドの原則

SOLID デザインにおける オープン/クローズドの原則 (OCP) とは、クラスは拡張に対して開かれており、修正に対して閉じていなければならないという原則のことです。昔からの例として、ある形状の面積計算があります。元のコードを修正することなく新しい動作を作成できるようクラスを構成します。

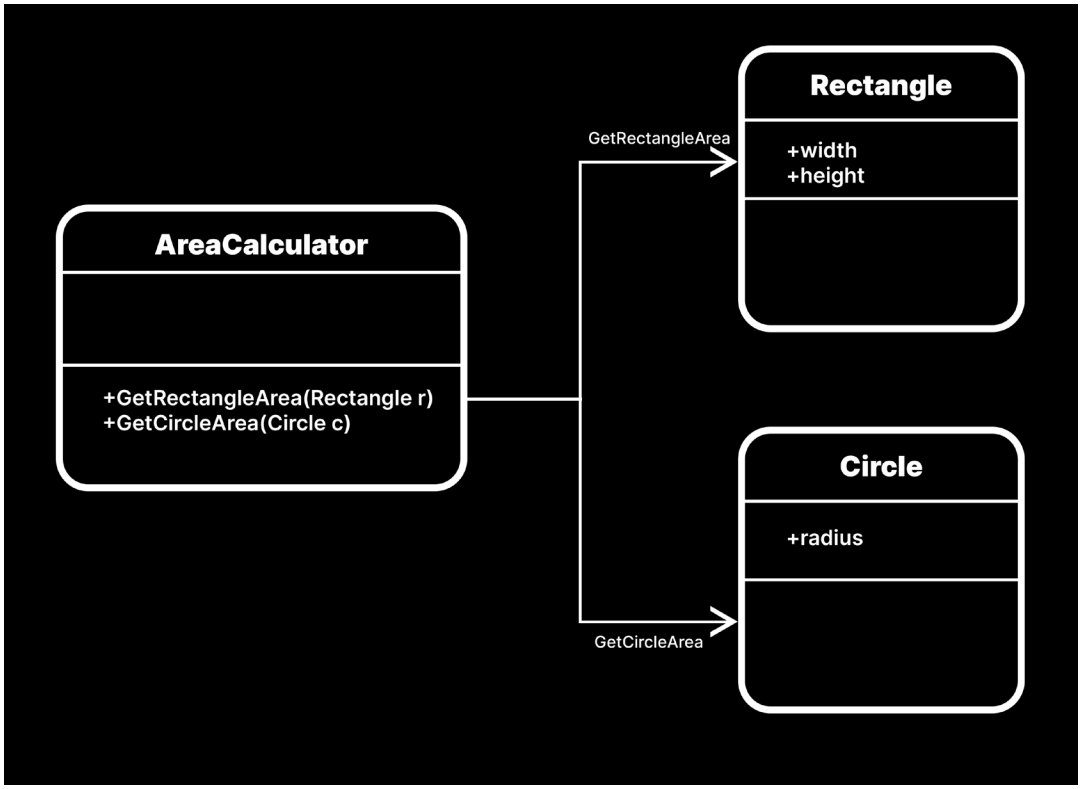
この例では、AreaCalculator クラスに、矩形の面積を返すメソッドと円の面積を返すメソッドがあります。面積を計算するために、Rectangle クラスには Width と Height があります。Circle に必要なのは Radius と円周率の値だけです。

```
public class AreaCalculator
{
    public float GetRectangleArea(Rectangle rectangle)
    {
        return rectangle.width * rectangle.height;
    }
    public float GetCircleArea(Circle circle)
    {
        return circle.radius * circle.radius * Mathf.PI;
    }
}

public class Rectangle
{
     public float width;
     public float height;
}

public class Circle
{
    public float radius;
}
```

これで十分に機能しますが、AreaCalculator にさらに形状を追加する必要がある場合は、新しい形状ごとに新しいメソッドを作成しなければなりません。後で五角形や六角形を渡す必要がある場合はどうでしょう?さらに 20 個の形状が必要な場合は?この AreaCalculator クラスはすぐに膨れ上がり、手に負えなくなってしまいます。

Shape という基本クラスを作り、その形状を処理する 1 つのメソッドを作成する方法もあります。ただし、そうすると各種形状を処理するために、ロジック内に複数の if ステートメントが必要になります。そうなるとうまくスケールできません。

元のコード (AreaCalculator の内部) を修正することなく、拡張 (新しい形状を使用する機能) に対してプログラムを開く必要があります。現状の AreaCalculator は機能的には問題ありませんが、オープン/クローズドの原則に違反しています。



新しい形状を受け取るように AreaCalculator をデザインするにはどうすればよいか?

代わりに、抽象クラス Shape を定義することを検討してください。

```
public abstract class Shape
{
    public abstract float CalculateArea();
}
```

これには、CalculateArea という抽象メソッドが含まれています。Rectangle と Circle が Shape を継承されるように設定すると、各形状で自身の面積を計算できるようになり、結果が返されます。

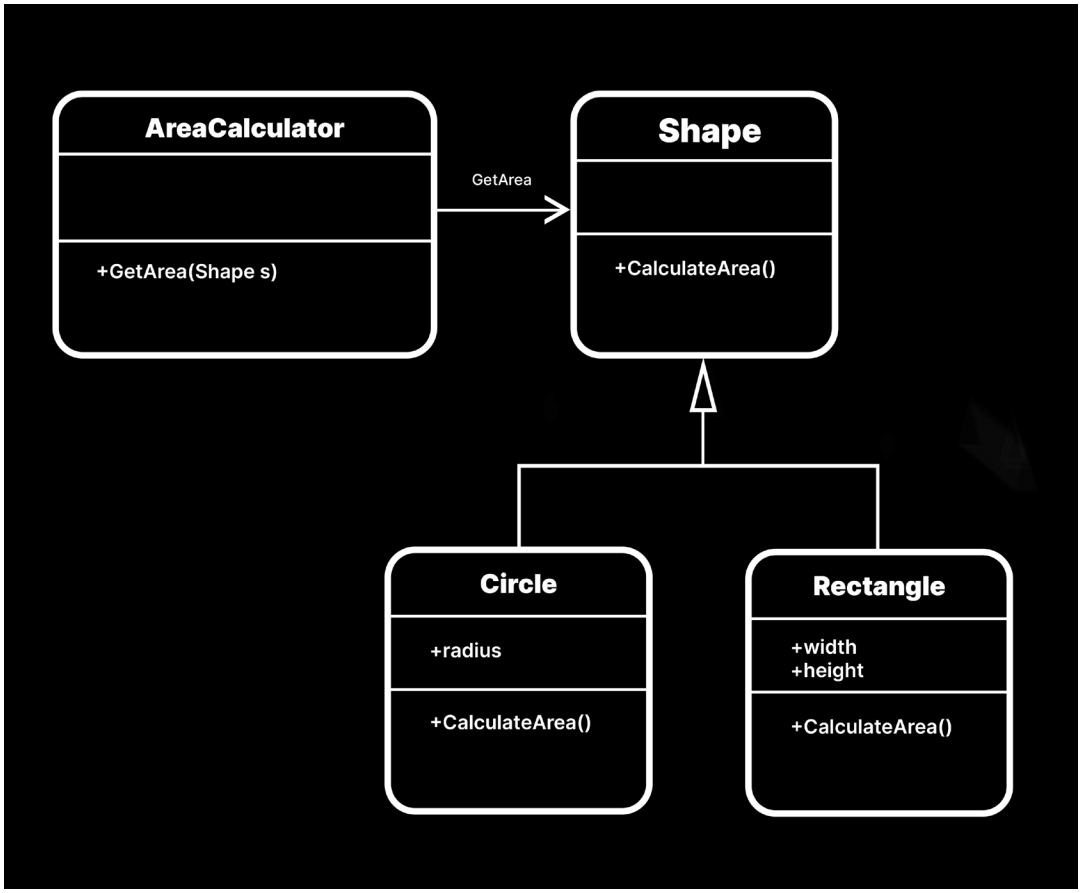
```
public class Rectangle :Shape
{
    public float width;
    public float height;
    public override float CalculateArea()
    {
        return width * height;
    }
}

public class Circle :Shape
{
    public float radius;
    public override float CalculateArea()
    {
        return radius * radius * Mathf.PI;
    }
}
```

AreaCalculator は、以下のようにシンプルにできます。

```
public class AreaCalculator
{
    public float GetArea(Shape shape)
    {
        return shape.CalculateArea();
    }
}
public class AreaCalculator :MonoBehaviour
```

```
{
    private void Start()
    {
        Debug.Log(GetArea(new RectAngle { width = 2, height = 3 }));
        Debug.Log(GetArea(new Circle { radius = 3 }));
    }
    public float GetArea(Shape shape)
    {
        return shape.CalculateArea();
    }
}
```

修正された AreaCalculator クラスでは、抽象クラス Shape を適切に実装している、あらゆる形状の面積を取得できます。これで、元のソースは一切変更せずに、AreaCalculator の機能を拡張できます。



オープン/クローズドの原則に対応するようクラスを修正

新しいポリゴンが必要になるたびに、Shape を継承する新しいクラスを定義するだけです。これによって、サブクラス化された各形状が CalculateArea メソッドをオーバーライドし、正しい面積を返します。

この新しいデザインにより、デバッグが簡単になります。新しい形状でエラーが発生した場合でも、AreaCalculator を見直す必要はありません。古いコードは変えていないので、新しいコードに問題のあるロジックがないか確認するだけで済みます。

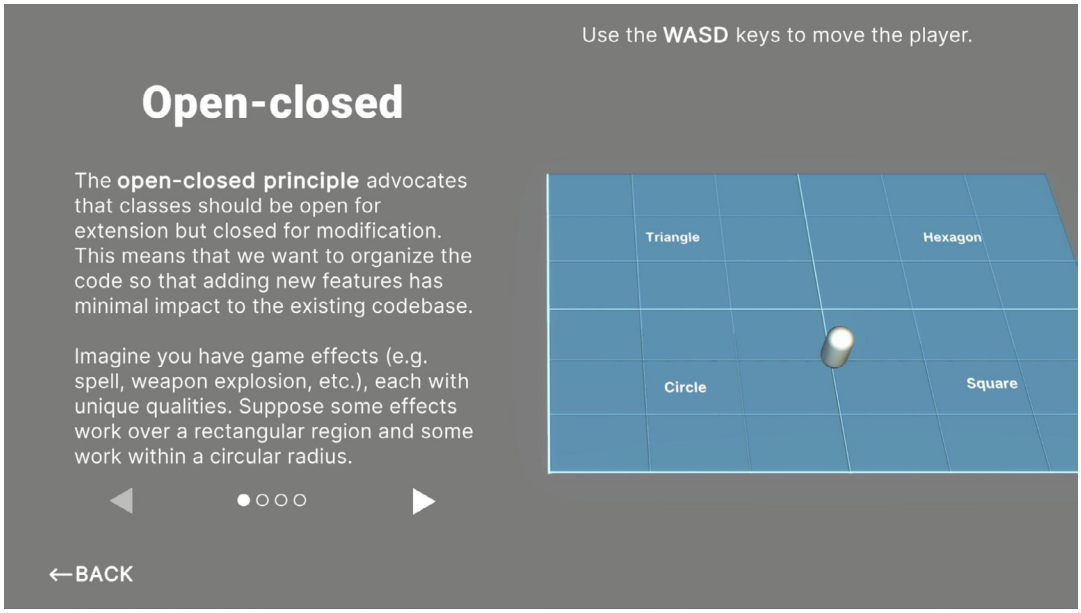
Unity で新しいクラスを作成するときは、インターフェースや抽象化を利用してください。これにより、後々の拡張を難しくする扱いにくい switch ステートメントや if ステートメントをロジック内に作成せずに済みます。OCP に配慮してクラスを設定することに慣れれば、新しいコードの追加が長期的には単純になります。

例：サンプルプロジェクト

サンプルプロジェクトでは、オープン/クローズドの原則を適用する同様の例を簡単なデモで紹介しています。ここでは、抽象基本クラス AreaOfEffect に CalculateArea という抽象メソッドが導入されています。

派生クラス (CircleEffect、HexagonalEffect、RectangleEffect、 TriangularEffect) は、エフェクトの面積を計算したり、ビジュアルエフェクトを再生したりするための独自の数式を実装できます。それぞれが CalculateArea 内で独自のロジックを定義するだけです。新しいエリアエフェクトタイプを追加しても、既存のコードは変わりません。

プレイヤーが EffectTrigger コンポーネントと衝突すると、各エフェクトの具体的な詳細を知らなくても、AreaOfEffect と相互作用します。これにより、新しいエフェクトを追加する柔軟性と拡張性が高まります。



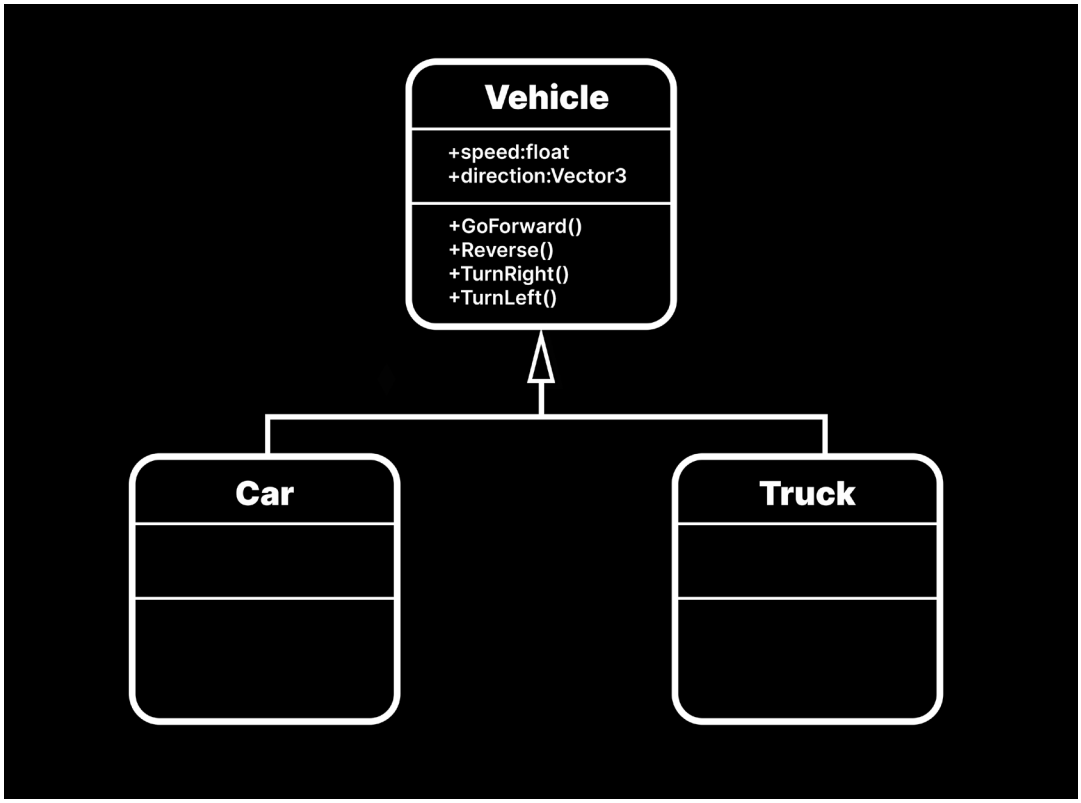
オープン/クローズドの原則により、コードの拡張性が高まる。

リスコフの置換原則

リスコフの置換原則 (LSP) とは、派生クラスはその基本クラスと置換可能でなければならないという原則のことです。オブジェクト指向プログラミングでは、継承を行うことで、サブクラスを通じて機能を追加することができます。しかし、これは注意しないと不要な複雑さにつながるおそれがあります。

SOLID の 3 本目の柱であるリスコフの置換原則は、サブクラスをより堅牢かつ柔軟にするために、継承をどのように適用するかを示します。

ゲームに `Vehicle` というクラスが必要だとします。これがアプリケーションのために作成する車両サブクラスの基本クラスになります。例えば、自動車やトラックが必要になることがあります。



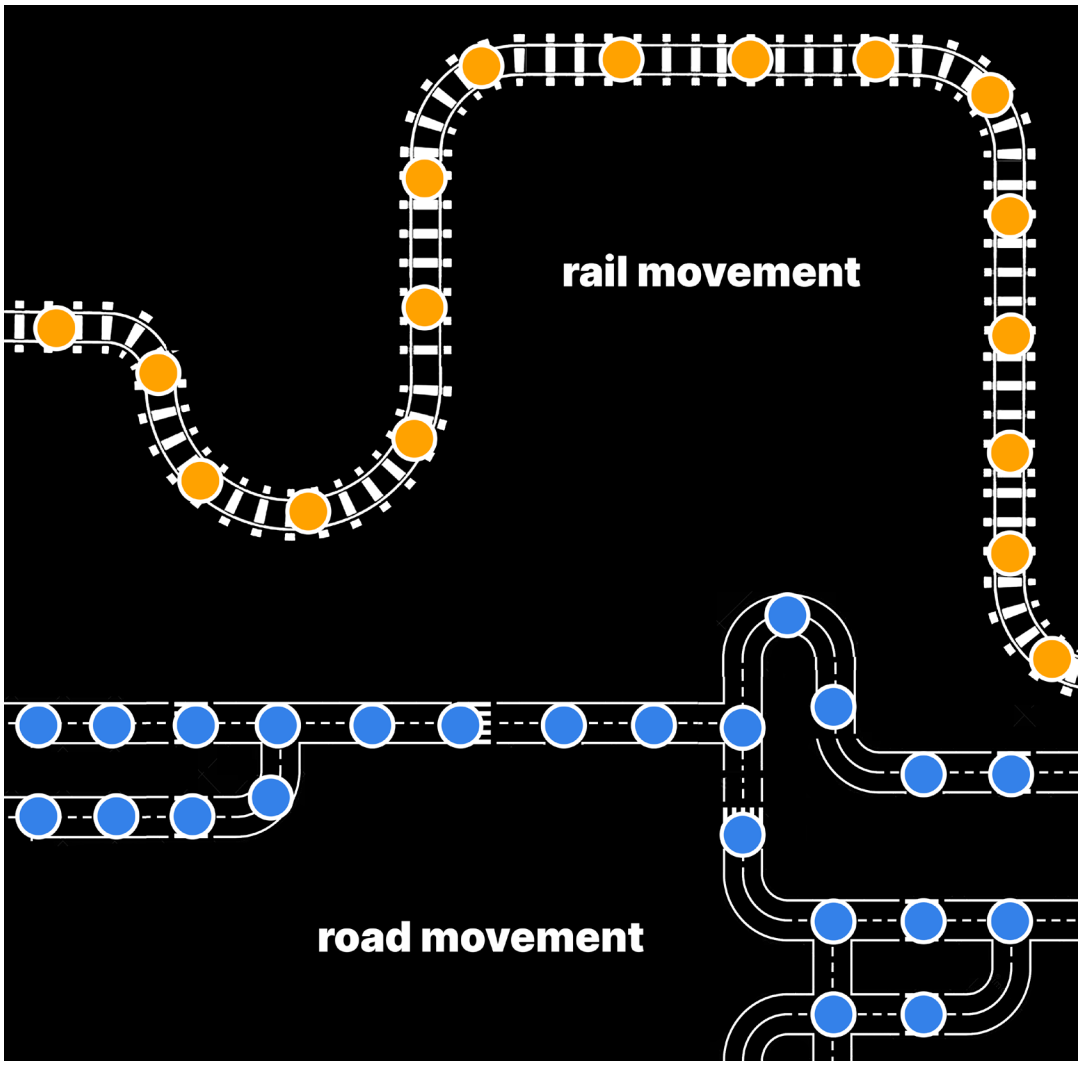
すべてが Vehicle を継承する。

基本クラス (`Vehicle`) が使用できるところではどこでも、`Car` や `Truck` などのサブクラスを、アプリケーションを壊すことなく使用できるようにする必要があります。

Vehicle クラスは、例えば以下のようになります。

```
public class Vehicle
{
    public float speed = 100;
    public Vector3 direction;

    public void GoForward()
    {
            ...
    }

    public void Reverse()
    {
            ...
    }

    public void TurnRight()
    {
            ...
    }

    public void TurnLeft()
    {
            ...
    }
}
```

ボード上で車両を動かすターン制ゲームを制作しているとします。

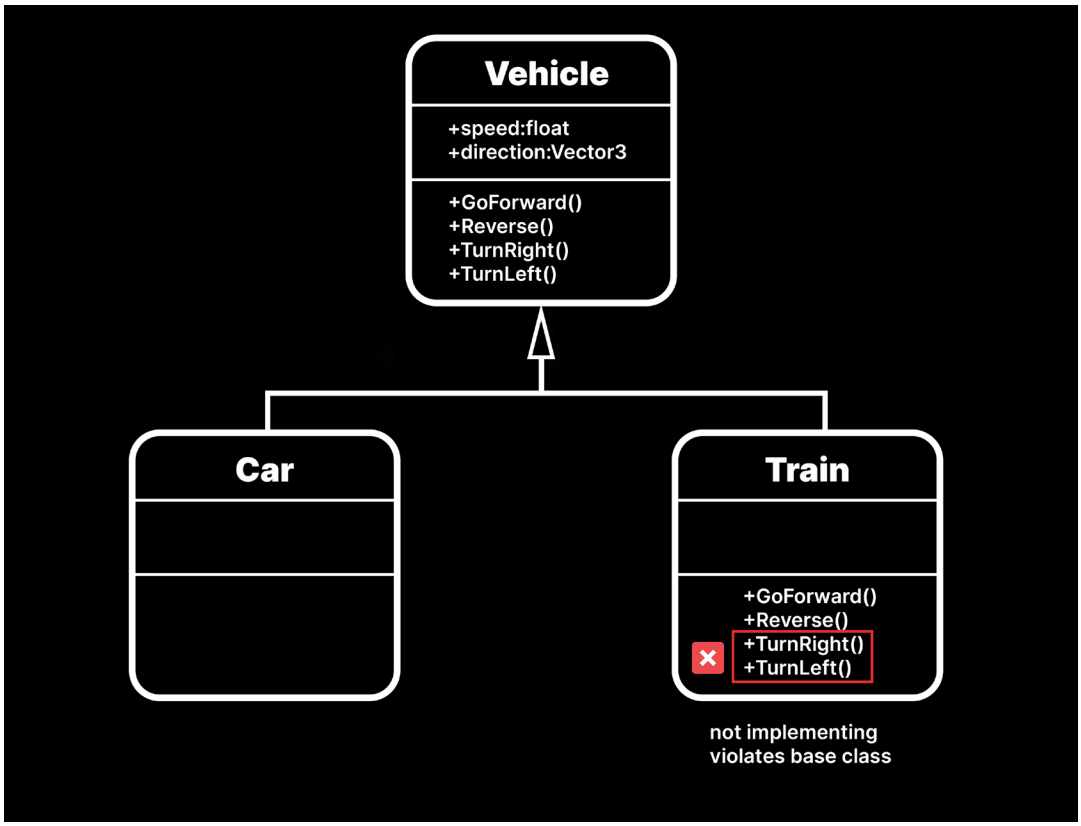


自動車と電車のゲームの例

`Navigator` という別のクラスを作成して、定められた経路に沿って車両を操縦することができます。

```
public class Navigator
{
    public void Move(Vehicle vehicle)
    {
        vehicle.GoForward();
        vehicle.TurnLeft();
        vehicle.GoForward();
        vehicle.TurnRight();
        vehicle.GoForward();
    }
}
```

このクラスにより、あらゆる車両を Navigator の `Move` メソッドに渡せるようになることが期待できます。これは自動車とトラックでは問題なく機能します。しかし、`Train` というクラスを実装する必要がある場合はどうなるでしょうか?



Train は基本クラスに違反する。

列車は線路から外れることができないため、`TurnLeft` メソッドと `TurnRight` メソッドは `Train` クラスでは機能しません。Navigator の `Move` メソッドに列車を渡すと、その行に到達したときに未実装の `Exception` がスローされます (または何も行われません)。サブタイプを型に置換できない場合は、リスコフの置換原則に違反します。

`Train` は `Vehicle` のサブタイプであるため、`Vehicle` クラスを使用可能なところであれば、どこでも使用できると予測できます。そうでないと、コードが予測せぬ動作をする可能性があります。

リスコフの置換原則をより忠実に守るために、以下のヒントを考慮に入れてください。

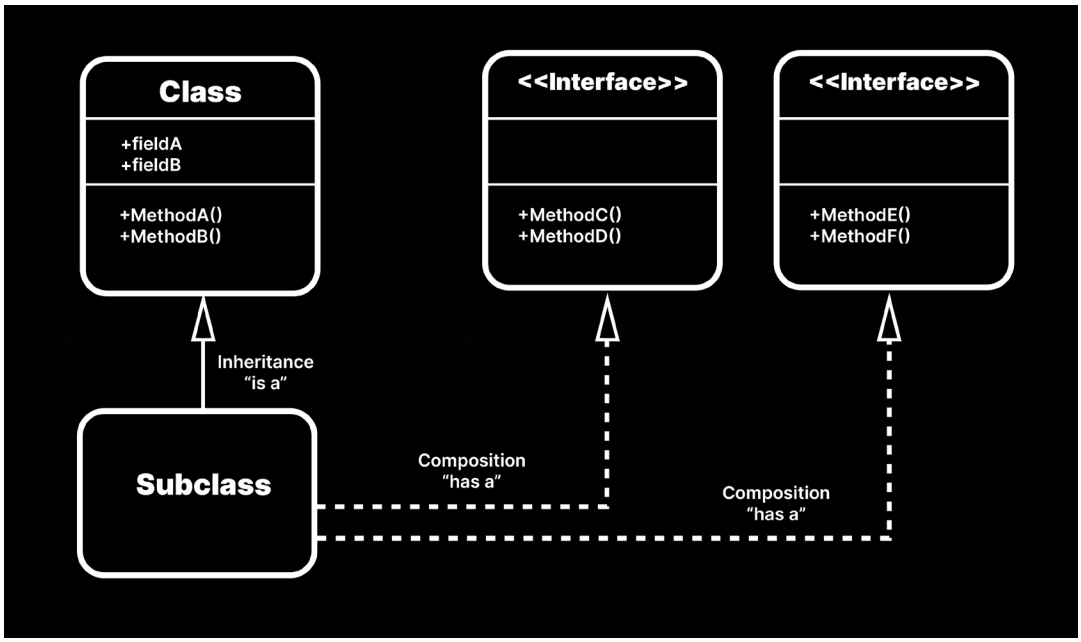
— **サブクラス化するときに機能を削除すると、リスコフの置換原則に違反する可能性が高まる:** `NotImplementedException` はこの原則に違反した決定的な証拠です。メソッドを空のままにする場合も同様です。サブクラスがその基本クラスのように動作しない場合は、たとえ明示的なエラーや例外が発生していない場合でも、LSP に従っていません。

— **抽象化はシンプルに保つ:**基本クラスにロジックを追加すればするほど、LSP に違反する可能性が高まります。基本クラスで表現するのは、その派生サブクラスの一般的な機能のみとする必要があります。

— **サブクラスには基本クラスと同じ public メンバーが必要:**これらのメンバーを呼び出す際には、同じシグネチャと動作も必要です。

— **クラス階層を確立する前にクラス API を検討する:**すべてを車両と見なしていますが `Car` と `Train` は別個の親クラスを継承するほうが理にかなっているかもしれません。実際の分類がクラス階層と常に一致するとは限りません。

— **継承よりも合成を優先する:**継承を通じて機能を渡そうとする代わりに、インターフェースまたは別のクラスを作成して、特定の動作をカプセル化します。そして、いろいろと組み合わせて別の機能の "合成" を作りあげます。

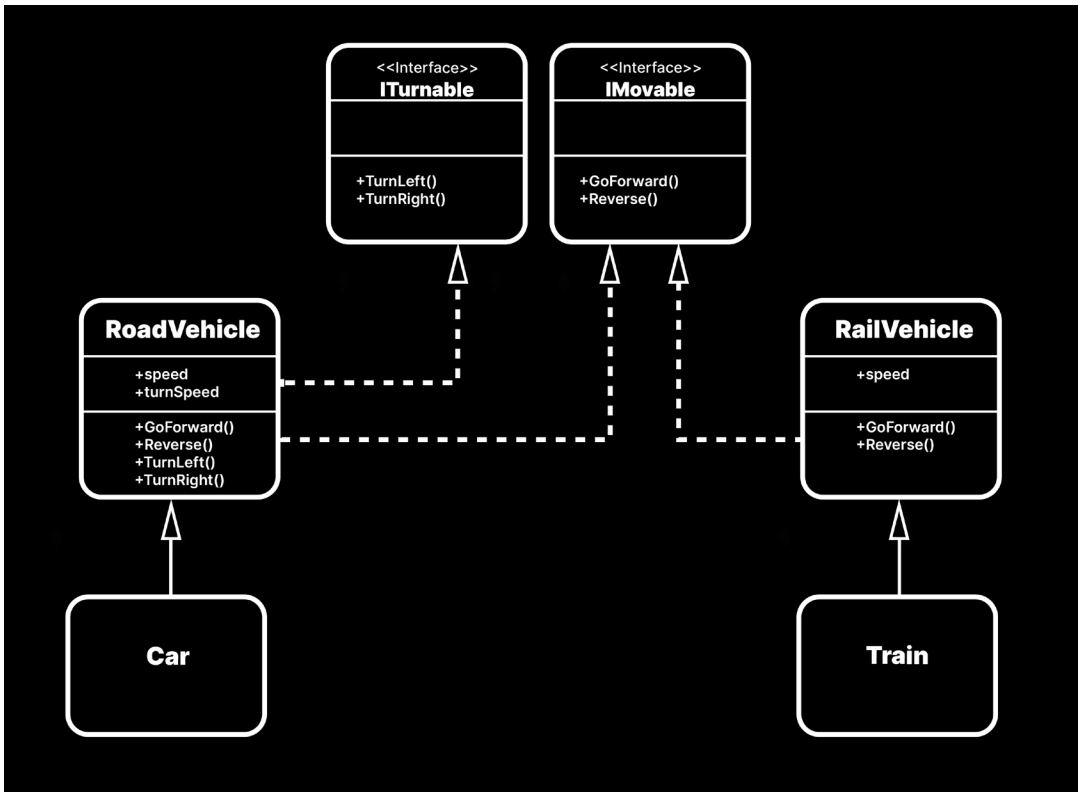


継承よりも合成を優先する

このデザインを修正するには、元の `Vehicle` 型を破棄して、その機能の大部分をインターフェースに移行させます。

```
public interface ITurnable
{
    public void TurnRight();
    public void TurnLeft();
}

public interface IMovable
{
    public void GoForward();
    public void Reverse();
}
```

RoadVehicle 型と RailVehicle 型を作成することで、LSP の原則に、より厳密に従います。これにより、Car と Train がそれぞれに対応する基本クラスを継承するようになります。



リスコフの置換を考慮したリファクタリング

```
public class RoadVehicle :IMovable, ITurnable
{
    public float speed = 100f;
    public float turnSpeed = 5f;
    public virtual void GoForward()
    {
        ...
    }

    public virtual void Reverse()
    {
        ...
    }

    public virtual void TurnLeft()
    {
        ...
    }

    public virtual void TurnRight()
    {
        ...
    }
}

public class RailVehicle :IMovable
{
    public float speed = 100;
    public virtual void GoForward()
    {
        ...
    }

    public virtual void Reverse()
    {
       ...
    }
}

public class Car :RoadVehicle
{
    ...
}
public class Train :RailVehicle
{
    ...
}
```

この方法により、継承ではなくインターフェースを通じて機能するようになります。`Car` と `Train` の基本クラスは同じでなくなるため、LSP を満たすようになりました。`RoadVehicle` と `RailVehicle` は同じ基本クラスから派生するように指定できますが、このケースではその必要はそれほどありません。

この考え方は、直感的には理解しにくいかもしれません。現実世界に関してある種の思い込みを持っているからです。ソフトウェア開発においては、これは 円 - 楕円問題と呼ばれます。実際の "is-a" の関係がすべて継承にあてはまるわけではありません。ソフトウェアデザインに必要なのは、現実に関する事前知識ではなく、クラス階層を確立することであることを忘れないでください。

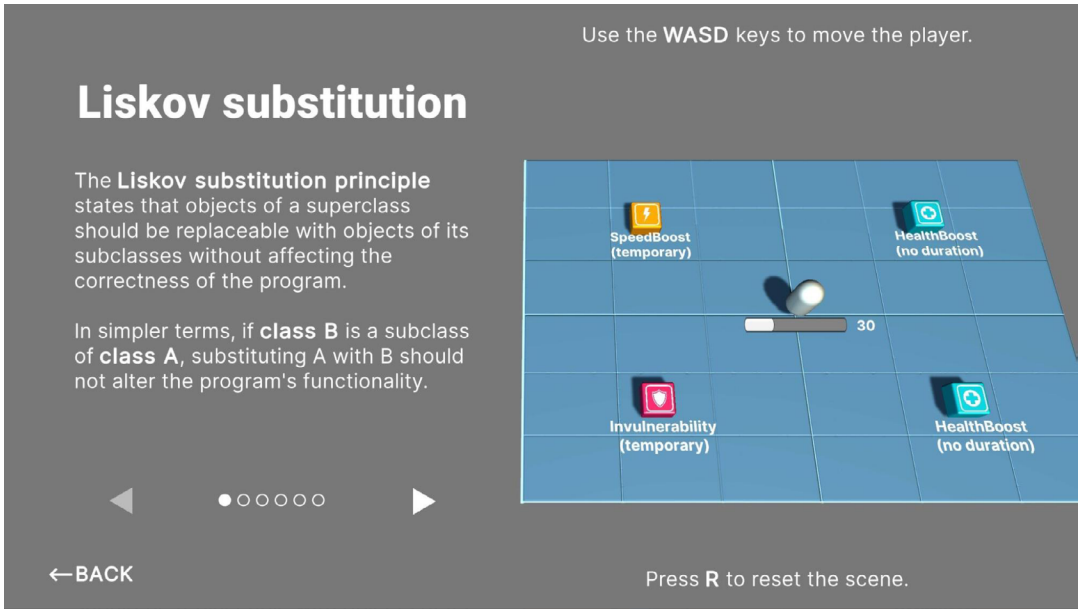
リスコフの置換原則に従うことで、継承の使い方に制限を設け、コードベースの拡張性と柔軟性を高く保つようにしてください。

例：サンプルプロジェクト

このサンプルプロジェクトでは、一連のパワーアップを通じてリスコフの置換原則を説明しています。`PowerUp` 抽象クラスは、プレイヤーバフ (`InvulnerabilityPowerUp`、`HealthBoost`、`SpeedBoost` など) の基本クラスとして機能します。各サブクラスは、特定のロジックを実装するために `ApplyEffect` メソッドをオーバーライドします。

リスコフの置換により、`PowerUp` のインスタンスはすべて、そのサブクラスのインスタンスに置き換えることができます。これにより、遭遇するパワーアップの種類に関係なく、ゲームが正しく動作することが保証されます。

その結果、コードの再利用性と保守性が向上します。オープン/クローズドの原則を強化し、将来的に新しいタイプのパワーアップを追加しても、既存のコードを変更する必要はありません。



リスコフの置換では、サブクラスのオブジェクトは基本クラスのオブジェクトと常に置き換えることができる。

インターフェース分離の原則

インターフェース分離の原則 (ISP) とは、クライアントは、使用しないメソッドへの依存を強制されてはならないという原則のことです。

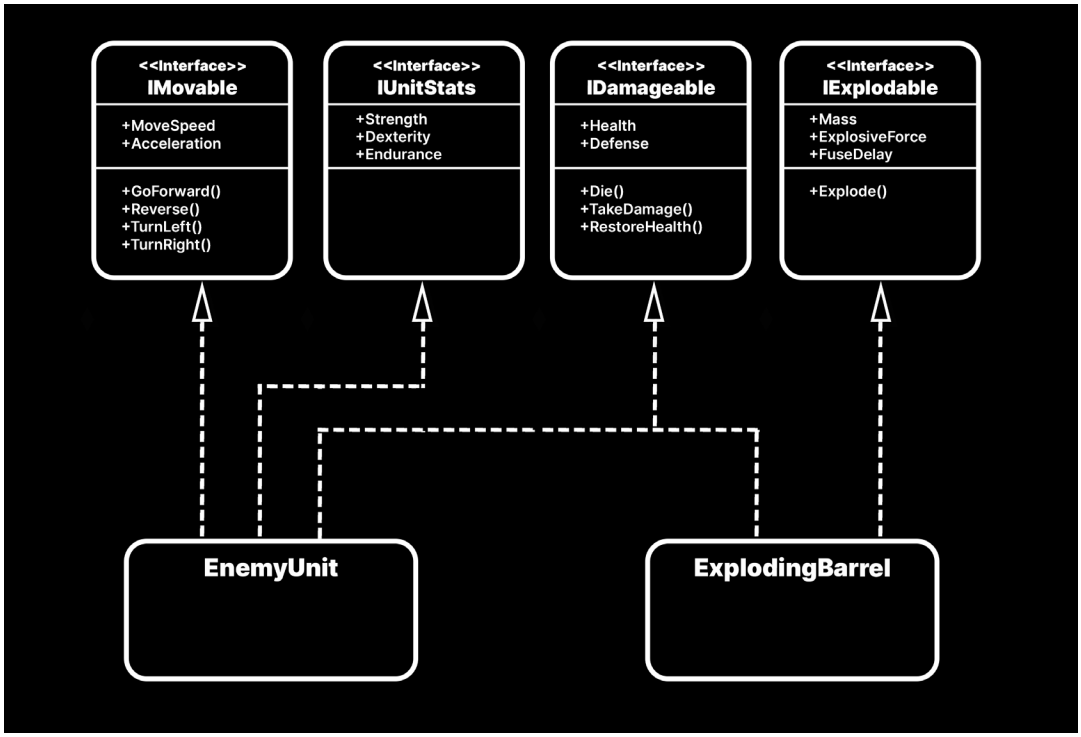
言い換えると、巨大なインターフェースを避けることです。クラスやメソッドは短くせよ、という単一責任の原則と同じ考え方に従います。これにより、柔軟性が最大限に高まり、インターフェースがコンパクトかつ焦点を絞った状態に保たれます。

さまざまなプレイヤーユニットが登場する戦略ゲームを制作しているとします。ユニットごとに HP やスピードなどのステータスが異なります。すべてのユニットに類似の機能が実装されていることを保証するインターフェースを作成するとよいでしょう。

```
public interface IUnitStats
{
    public float Health { get; set; }
    public int Defense { get; set; }

    public void Die();
    public void TakeDamage();
    public void RestoreHealth();

    public float MoveSpeed { get; set; }
    public float Acceleration { get; set; }

    public void GoForward();
    public void Reverse();
    public void TurnLeft();
    public void TurnRight();

    public int Strength { get; set; }
    public int Dexterity { get; set; }
    public int Endurance { get; set; }
}
```

壊せる樽や木箱などの破壊可能な小道具を作成する必要があるとします。この小道具は動きませんが、HP の概念も必要です。また、樽や木箱には、ゲーム内の他のユニットに関連付けられている多くのアビリティも設定されません。

破壊可能な小道具に多数のメソッドを設定するインターフェースを 1 つ作成するのではなく、複数の小さなインターフェースに分割します。それらを実装するクラスは、必要とするものを組み合わせるだけで済みます。



インターフェースを小さなものに分割する。

```
public interface IMovable
{
    public float MoveSpeed { get; set; }
    public float Acceleration { get; set; }
    public void GoForward();
    public void Reverse();
    public void TurnLeft();
    public void TurnRight();
}

public interface IDamageable
{
    public float Health { get; set; }
    public int Defense { get; set; }
    public void Die();
    public void TakeDamage();
    public void RestoreHealth();
}

public interface IUnitStats
{
    public int Strength { get; set; }
    public int Dexterity { get; set; }
    public int Endurance { get; set; }
}
```

爆発する樽には、以下のように IExplodable インターフェースを追加することもできます。

```
public interface IExplodable
{
    public float Mass { get; set; }
    public float ExplosiveForce { get; set; }
    public float FuseDelay { get; set; }

    public void Explode();
}
```

1 つのクラスに複数のインターフェースを実装できるため、IDamageable、IMoveable、IUnitStats で敵ユニットを構成できます。

爆発する樽は、他のインターフェースの不要なオーバーヘッドなしに、IDamageable と IExplodable を使用できます。

```
public class ExplodingBarrel :MonoBehaviour, IDamageable, IExplodable
{
    ...
}

public class EnemyUnit :MonoBehaviour, IDamageable, IMovable, IUnitStats
{
    ...
}
```

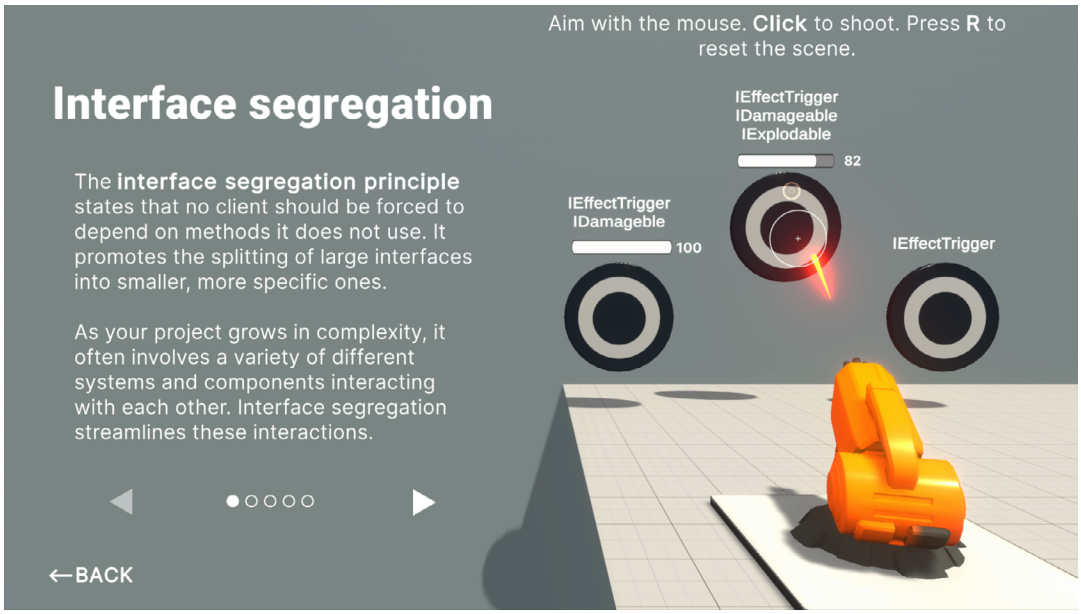
例：サンプルプロジェクト

このサンプルプロジェクトでは、一連のターゲットオブジェクトを通じてインターフェースを分離する方法を紹介します。マウスで銃の照準を合わせ、マウスの左ボタンで撃ちます。

各ターゲットは必要なメソッドだけを実装しています。各クラスは、IEffectTrigger、IExplodable、IDamageable など、より小さく、焦点を絞ったインターフェースを定義することで、関係する機能だけを実装しています。これにより、クラスとインターフェース間の不要な依存関係が軽減されます。

- IEffectTrigger を使用すると、オブジェクトは特定の場所でビジュアルエフェクトやオーディオエフェクトをトリガーすることができます。このケースでは、弾丸は衝突時に小さなヒットエフェクトを示します。
- IDamageable を使用すると、オブジェクトにダメージを与えることができます。HealthBar を付けられたターゲットは、HP がなくなると消滅します。
- IExplodable は、ターゲットが消滅する際に爆発プレハブをインスタンス化します。

こうしたインターフェースの分離により、ゲーム環境におけるオブジェクトの相互作用の柔軟性が高まります。例えば、`Projectile` クラスは、各ターゲットの具体的な実装について直接知らなくても、他のオブジェクトに影響を与えられます。



インターフェースの分離とは、クライアントが使用していないメソッドに依存してはいけないということです。

インターフェースのシリアル化

SerializeField 属性をインターフェース型のフィールドに適用しても、public にしても、そのフィールドは Inspector に表示されません。Unity のシリアル化システムは具象クラス、特に `MonoBehaviour` や `ScriptableObject` から継承されたクラスを扱うように設計されています。

インターフェースは本質的に抽象的なものであり、それ自体は具体的なデータを保持しないため、シリアル化メカニズムの直接の対象になりません。この制限に対処するには、以下のようにします。

— インターフェースをシリアル化しようとする代わりに、インターフェースを実装する具象オブジェクト (例えば、MonoBehaviour や ScriptableObject) への参照をシリアル化します。

— ランタイムに `is` キーワードを使用し、シリアル化されたオブジェクトをチェックしてキャストします。そして、必要なインターフェースを実装しているかどうかを確認できます。

例を挙げてみましょう。

```
 // インターフェースを定義する
public interface IInteractable
{
    void Interact();
}
// インターフェースを実装している具象クラス
public class DoorController :MonoBehaviour, IInteractable
{
    public void Interact()
    {
        // ここにドアロジックを配置
        Debug.Log(“Door opened”);
    }
}
public class GameManager :MonoBehaviour
{
    [SerializeField]
    private MonoBehaviour interactableObject;
    private void Start()
    {
        // ランタイムにチェックしてキャスト
        if (interactableObject is IInteractable interactable)
        {
            interactable.Interact();
        }
    }
}
```

繰り返しになりますが、リスコフの置換の例と同様に、継承より合成 が優先されます。インターフェース分離の原則により、システムを切り離し、変更や拡張が容易になります。

依存性逆転の原則

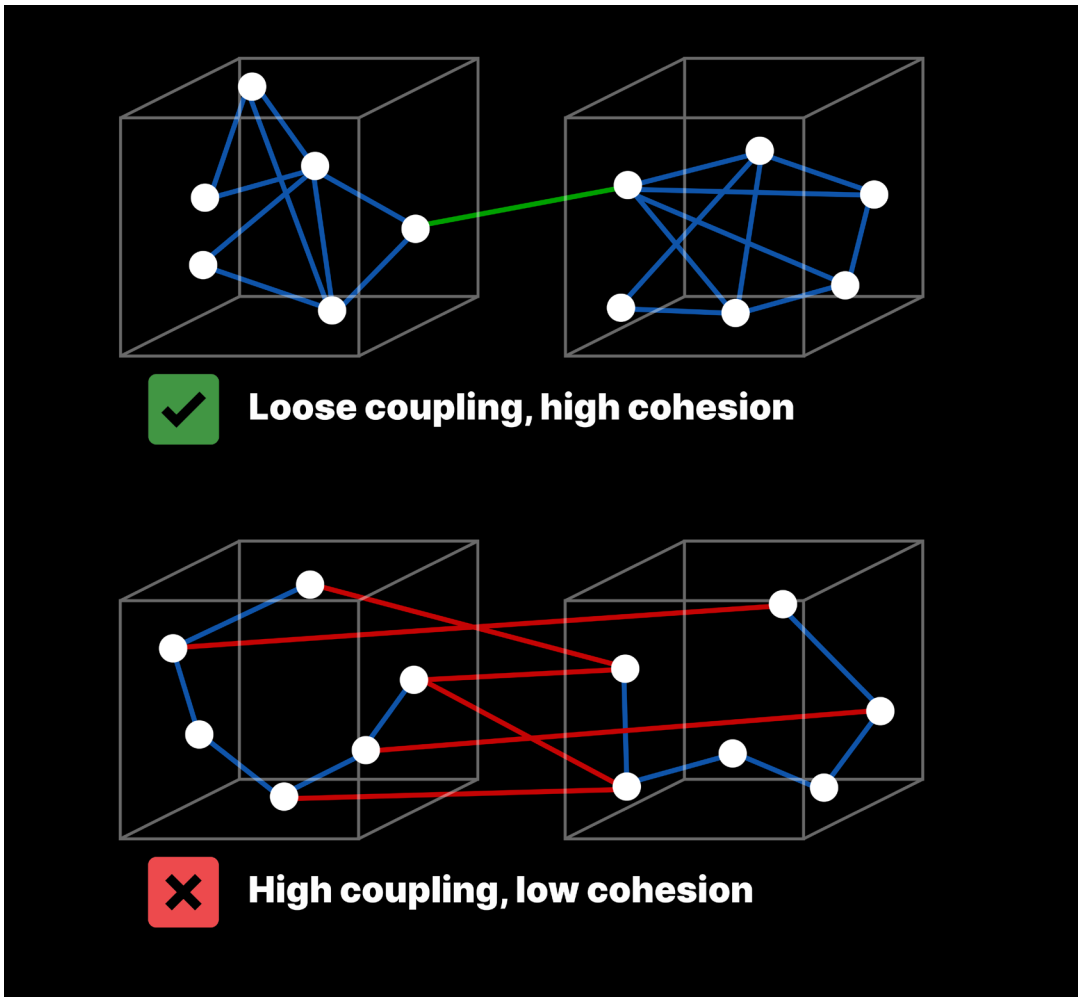
依存性逆転の原則 (DIP) とは、上位のモジュールは下位のモジュールから直接何かを持ち込んではならないという原則のことです。両方とも抽象化に依存すべきです。

それぞれが持つ意味を詳しく説明していきます。あるクラスに他のクラスとの関係がある場合、それは 依存関係または結合 です。ソフトウェアデザインにおける各依存関係には、一定のリスクがあります。

あるクラスに別のクラスの仕組みに関する情報が多すぎると、最初のクラスに変更を加えることで 2 つ目のクラスが破損する (または 2 つ目のクラスの変更により最初のクラスが破損する) 可能性があります。過度な結合は、クリーンでないコーディングによくある例と見なされています。アプリケーションの一部分のエラーが、雪だるま式に増える可能性があります。

クラス間の依存関係をできるだけ少なくすることを目指してください。また、各クラスは、外部との接続に依存するのではなく、そのクラス内部のパーツを連携させる必要があります。internal または private ロジックで機能するオブジェクトは、凝集度が高いと見なされます。

ベストシナリオとして、結合を疎にし、凝集度を高めることを目指してください。

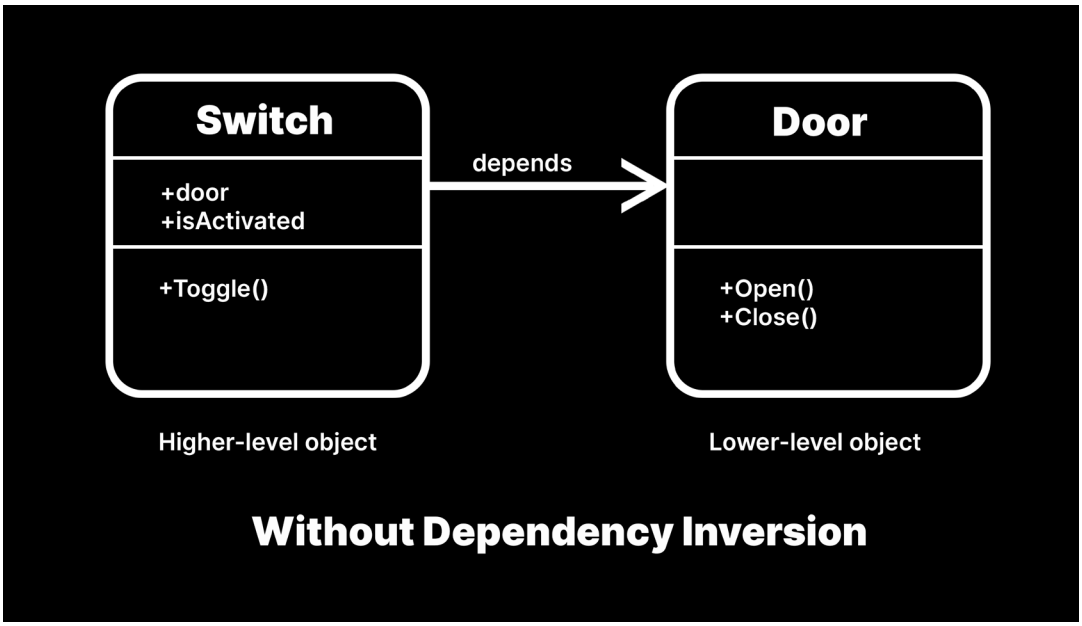


凝集性の高い疎結合を目指す。

制作したゲームアプリケーションは変更と拡張ができるようにする必要があります。アプリケーションが壊れやすく、変更するのが難しい場合は、現在どのような構成になっているか調べてください。

依存性逆転の原則は、クラス間の密な結合を緩めるのに役立ちます。アプリケーションにクラスやシステムを構築する際に、自然と "上位" のものと "下位" のものがあります。上位のクラスでは、何かを実行するために、下位のクラスに依存します。SOLID では、これを切り替えるようにすべき、ということです。

キャラクターがステージを探索し、ドアを作動させて開くゲームを制作しているとします。この場合、`Switch` というクラスと、`Door` というもう 1 つのクラスを作成するとよいでしょう。



`Switch` (上位) は `Door` (下位) クラスに直接依存する。

上位では、キャラクターを特定の場所に動かし、何かを発生させる必要があります。`Switch` がその役割を担います。

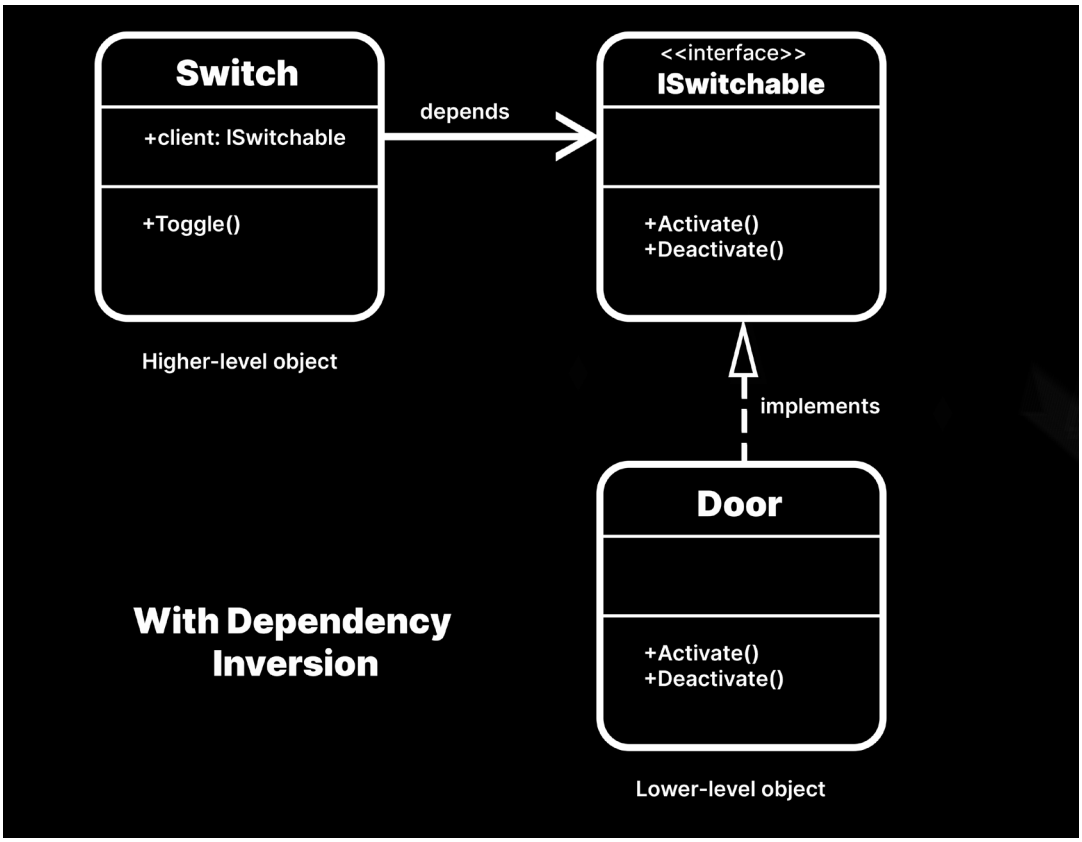
下位にはもう 1 つのクラス `Door` があり、ドアのジオメトリを開く方法が実装されています。シンプルにするために、ドア開閉のロジックを表す `Debug.Log` ステートメントが追加されています。

```
public class Switch :MonoBehaviour
{
    public Door door;
    public bool isActivated;

    public void Toggle()
    {
        if (isActivated)
        {
            isActivated = false;
            door.Close();
        }
        else
        {
            isActivated = true;
            door.Open();
        }
    }
}

public class Door :MonoBehaviour
{
    public void Open()
    {
        Debug.Log(“The door is open.”);
    }
    public void Close()
    {
        Debug.Log(“The door is closed.”);
    }
}
```

Switch は、ドアを開いたり閉じたりするために、Toggle メソッドを呼び出すことができます。機能はしていますが、問題は Door から Switch への依存関係が直接固定されてしまっていることです。Switch のロジックを、例えばライトや巨大なロボットをアクティベートするなど、Door 以外でも機能させる必要がある場合はどうすればいいでしょうか?

別のメソッドを Switch クラスに追加することはできますが、オープン/クローズドの原則に違反することになります。機能を拡張する必要性が出てくるたびに、元のコードを変更しなくてはなりません。

ここでも抽象化が役立ちます。クラスとクラスの間に、ISwitchable というインターフェースを挟むことができます。



2 つのクラス間にある 1 つの ISwitchable インターフェース

ISwitchable に必要なのは、それがアクティブであるかどうかを把握するための 1 の public プロパティと、それを Activate および Deactivate するいくつかのメソッドのみです。

```
public interface ISwitchable
{
    public bool IsActive { get; }
    public void Activate();
    public void Deactivate();
}
```

これにより、Switch は、ドアと直接ではなく、ISwitchable client に依存し、以下のようになります。

```
public class Switch :MonoBehaviour
{
    public ISwitchable client;
    public void Toggle()
    {
        if (client.IsActive)
        {
            client.Deactivate();
        }
        else
        {
            client.Activate();
        }
    }
}
```

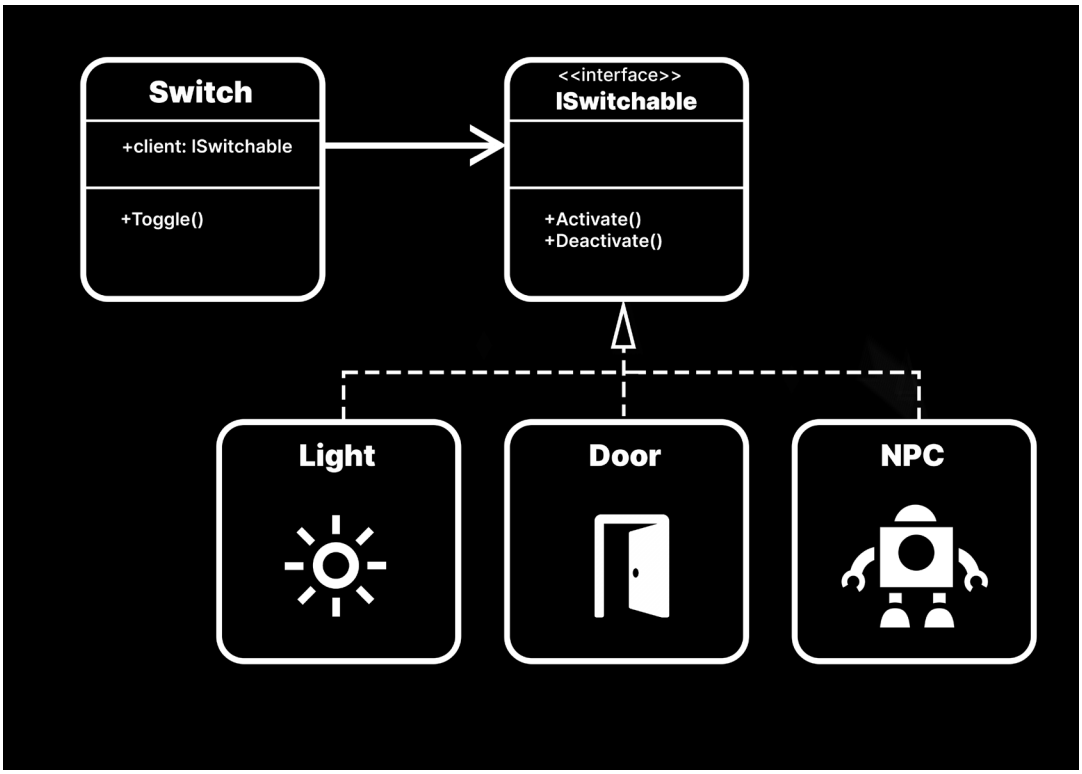
一方、Door を以下のように手直しして、ISwitchable を実装する必要があります。

```
public class Door :MonoBehaviour, ISwitchable
{
    private bool isActive;
    public bool IsActive => isActive;
    public void Activate()
    {
        isActive = true;
        Debug.Log(“The door is open.”);
    }

    public void Deactivate()
    {
        isActive = false;
        Debug.Log(“The door is closed.”);
    }
}
```

これで依存関係を逆転しました。スイッチをドア専用に固定するのではなく、インターフェースがそれらの間に抽象化の層を作ります。Switch とドア固有のメソッド (Open と Close) の間に直接的な依存関係がなくなります。代わりに、ISwitchable の Activate と Deactivate が使用されます。

このような、ちょっとした変更が重要であり、再利用性を高めることにつながります。一方で、以前は Switch は Door のみで機能していましたが、今では ISwitchable を実装すればどこでも機能します。

これにより、Switch によってアクティブにできるクラスをさらに作成できます。上位の Switch は、落とし戸であろうと、レーザービームであろうと機能します。必要なのは、互換性のある client と、そこに実装される ISwitchable だけです。



Switch により任意の ISwitchable オブジェクトがアクティベートされるようになった。

SOLID の他の原則と同じように、依存性逆転の原則では、クラス間の関係を通常、どのように設定するかを検討するかが求められます。疎の結合を使用して、プロジェクトを適宜スケールしましょう。

例：サンプルプロジェクト

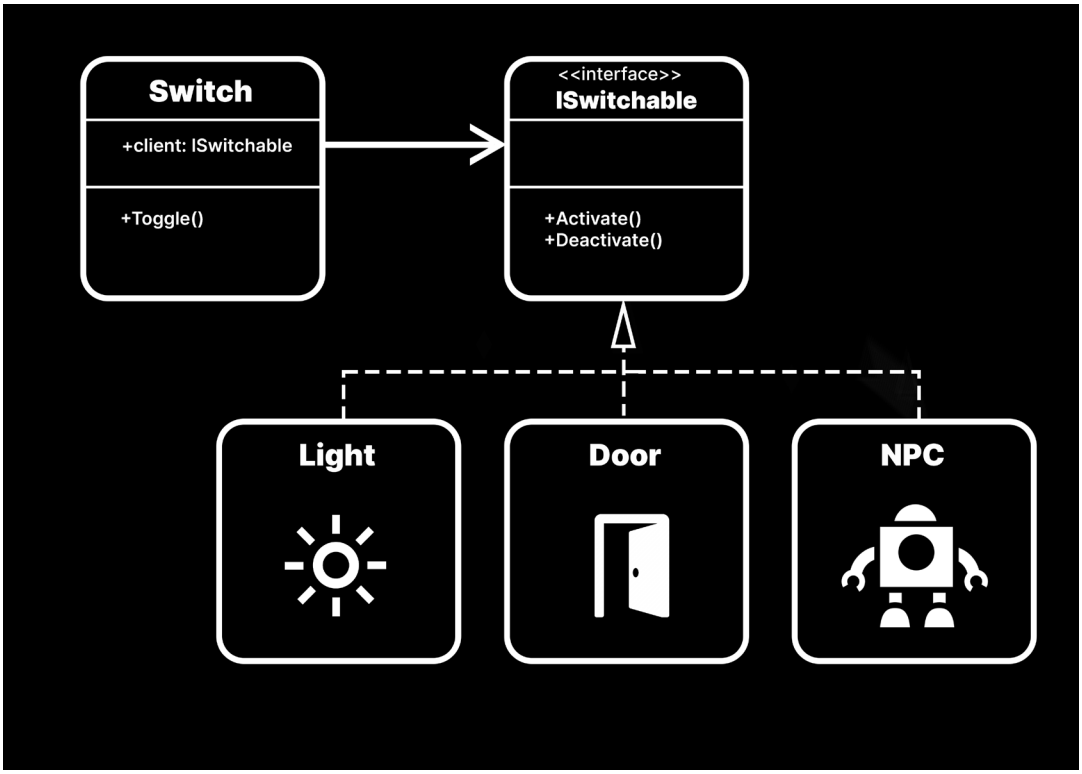
サンプルプロジェクトでは、依存性逆転について、ドアとトラップの実装例を紹介しています。該当するデバイスをアクティブにするには、それぞれのスイッチをクリックします。上位のモジュール (スイッチなど) は下位のモジュール (ドアやトラップなど) に依存すべきではないことを覚えておいてください。

代わりに、ISwitchable インターフェースがそれらの間の抽象レイヤーとして機能します。オブジェクトの具体的な実装に関係なく、オブジェクトをアクティブ化または非アクティブ化するための契約を定義します。

Door クラスと Trap クラスは、ISwitchable インターフェースを実装します。これにより、具体的な動作を直接知らなくても、システムの他の部分から制御できるようになります。

そのため、Door は開閉機構を管理し、Trap はアクティブ化と非アクティブ化のロジックをすべて同じインターフェースで処理できます。

具体的な実装ではなく抽象化に依存することで、切り替え可能な新しいタイプのオブジェクトでシステムを簡単に拡張できます。



依存性逆転では、上位モジュールは下位モジュールに依存すべきでない。どちらも抽象化に依存する。

インターフェースと抽象クラスの比較

このガイドの多くの例は、"継承よりも合成" を優先する理念に従って、インターフェースが使用されています。ただし、抽象クラスでも、デザイン原則やパターンの多くに従うことができます。

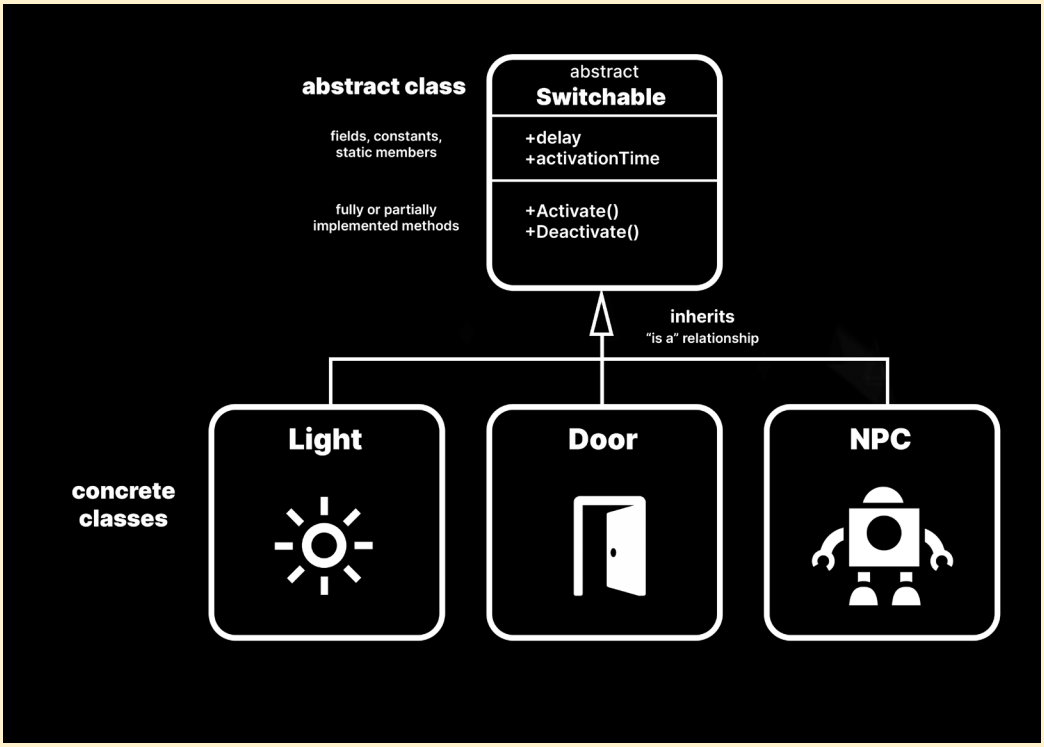
両方とも C# で抽象化を実現する有効な方法です。どちらを使用するかは、そのときのニーズによって変わります。

抽象クラス

`abstract` キーワードを使用すると、基本クラスを定義できるため、継承を通じて一般的な機能 (メソッド、フィールド、定数など) をサブクラスに渡すことができます。

抽象クラスは直接的にはインスタンス化できません。代わりに、具象クラスを派生させる必要があります。

前述の例では、抽象クラスで同じ依存性逆転を実現することができます。ただし、アプローチが異なります。そのため、インターフェースを使用する代わりに、具象クラス (例:`Light`、`Door`) を `Switchable` という抽象クラスから派生します。

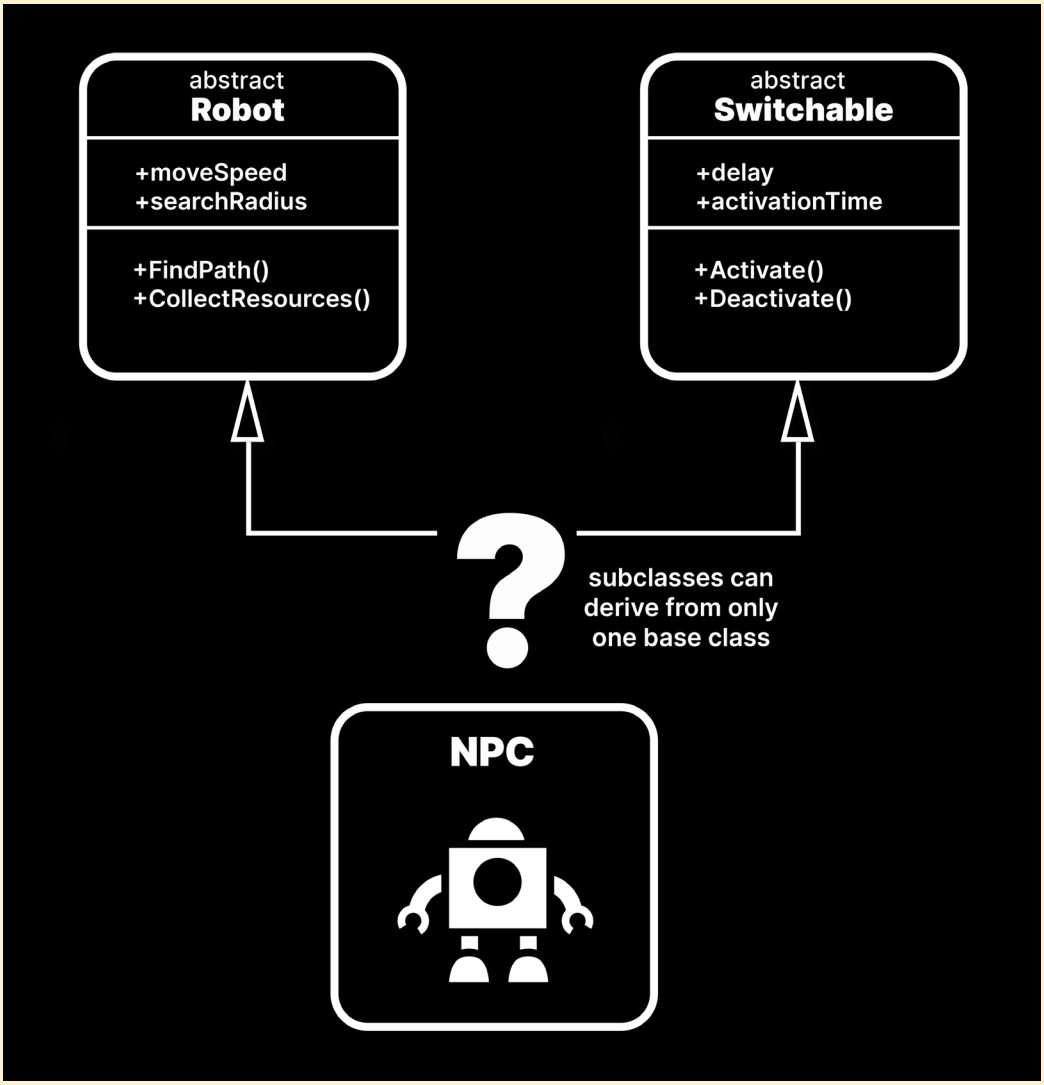


抽象クラスの使用

継承によって "is-a" 関係が定義されます。上図に示すのは、すべてオンとオフの "切り替え可能" なものです。

抽象クラスの利点は、静的メンバーだけでなく、フィールドと定数も指定できることです。また、より制約のあるアクセス修飾子 (protected、private など) を適用できます。インターフェースとは異なり、抽象クラスを使用すると、具象クラス間でコア機能を共有可能にするロジックを実装できます。

継承は、2 つの異なる基本クラスの特性を備えた派生クラスを作成する必要が出てくるまでは問題なく機能します。C# では、複数の基本クラスを継承することはできません。



どちらの基本クラスを使用するか

ゲームに登場するすべてのロボット用に別の抽象クラスがあった場合、何から派生させるかを決めるのが難しくなります。基本クラス `Robot` と `Switchable` のどちらを使用しますか?

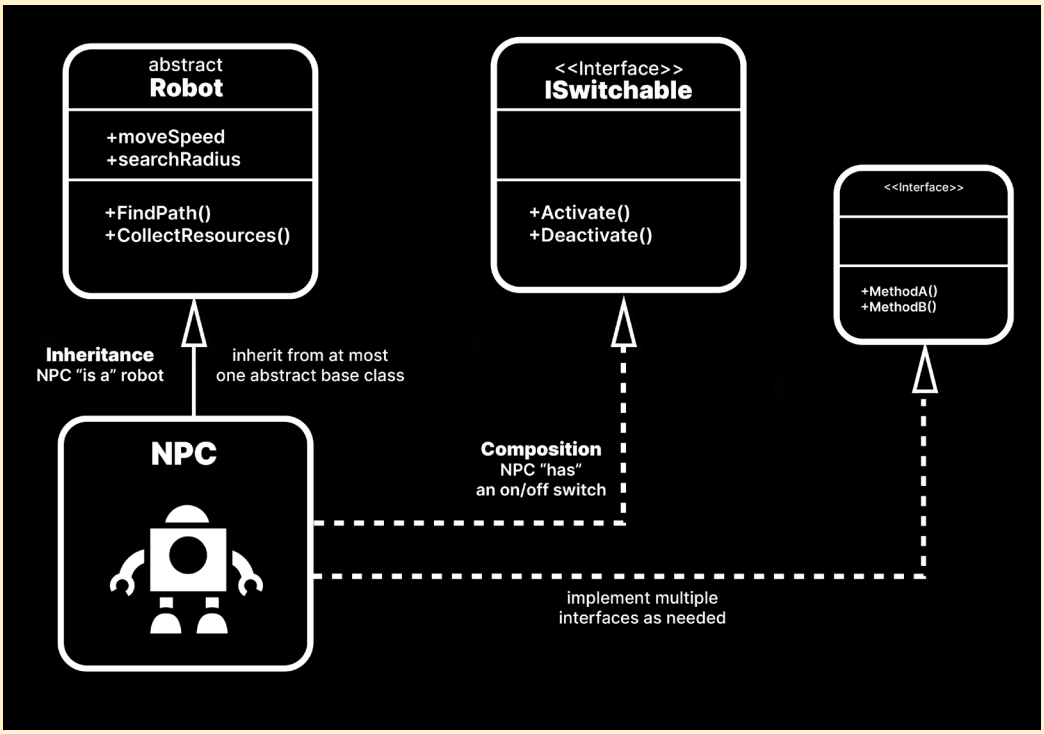
インターフェース

インターフェース分離の原則で見られるように、継承の枠組みにうまく収まらないときに、インターフェースはより高い柔軟性をもたらします。"has-a" 関係を使用して、より簡単に選択することができます。

ただし、インターフェースにはメンバーの宣言しかありません。そのインターフェースを実際に実装するクラスが、特定のロジックを詳細化する役割を担います。

このように、常に二者択一の選択とは限りません。コードを共有する基本機能を定義するには、抽象クラスを使用します。高い柔軟性が求められる周辺機能を定義するには、インターフェースを使用します。

この例では、基本クラス `Robot` から NPC を派生してそのコア機能を継承できますが、その後にインターフェース `ISwitchable` を使用して NPC のオンとオフを切り替える機能を追加します。



両方を使用する NPC のロボット

抽象クラスとインターフェースには以下の違いがあることに注意してください。

抽象クラス	インターフェース
完全または部分的にメソッドを実装する	メソッドを宣言するが実装できない
変数とフィールドを宣言/使用する	メソッドとプロパティのみを宣言する (フィールドは宣言しない)
静的メンバーがある	静的メンバーを宣言/使用できない
コンストラクターを使用する	コンストラクターを使用できない
すべてのアクセス修飾子 (protected、private など) を使用できる	アクセス修飾子は使用できない (すべてのメンバーが暗黙的に public)

忘れないでください。1 つのクラスが継承できる抽象クラスは最大 1 つですが、複数のインターフェースを実装できます。

SOLID について理解を深める

SOLID の原則には、日頃のさまざまな場面で触れることになります。コーディング中に常に頭に入れておく 5 つの基本原則であると考えてください。簡単にまとめてみました。

- **単一責任:**クラスは 1 つのことのみを担い、変更する理由は 1 つだけにします。
- **オープン/クローズド:**クラスにすでに備わっている機能を変えることなく、機能を拡張できるようにすべきです。
- **リスコフの置換:**サブクラスは基本クラスと置換可能であるべきです。
- **インターフェース分離:**メソッド数を少なくして、インターフェースを短く保ちます。クライアントには必要なものだけ実装します。
- **依存性逆転:**抽象化に依存するようにします。ある具象クラスから別の具象クラスに直接的には依存しないようにします。

SOLID の原則は、よりクリーンなコードを記述し、メンテナンスや拡張をより効率的に行えるよう手助けするガイドラインです。SOLID の原則は、拡張が必要な大規模なアプリケーションに最適なため、エンタープライズレベルのソフトウェアデザインを 20 年近く主導してきました。

場合によっては、SOLID の原則を忠実に守ることで、事前に追加作業が発生することがあります。一部の機能を抽象化やインターフェースにリファクタリングする必要が生じる場合もあります。ただし、長期的には節約になるという見返りがあることも多いです。

この原則にどの程度厳密に従うかは、プロジェクトごとの自身の判断となります。絶対的なものではありません。ここでは取り上げない微妙な差異やさまざまな実装方法があります。重要なのは、具体的な構文ではなく、その原則の背景にある考え方であることに留意してください。

使い方についてわからないことがある場合は、KISS の原則に立ち返ってください。シンプルに保つようにしましょう。しかし、原則を適用するという、ただそれだけのためにスクリプトに当てはめることはやめましょう。必要な場面で、原則が自然に効果を発揮できるようにしてください。

詳細については、Unite Austin の Unity による SOLID に関するプレゼンテーション を視聴してください。

ゲーム開発のための デザインパターン

SOLID の原則を理解したら、デザインパターンに深く踏み込んでいきましょう。

デザインパターンにより、日々見つかるソフトウェアの問題に対して既存のソリューションを再利用することができます。ただし、パターンは、棚から取り出してすぐに使えるライブラリやフレームワークではありません。アルゴリズムのような、ある成果を達成するための具体的な一連の手順でもありません。

そうではなく、デザインパターンとは青写真のようなものであると考えてください。汎用的なプランであり、実際の構成は各自に委ねられます。同じパターンに従っている 2 つのプログラムでも、コードはまったく異なることがあります。

何の前提もなしに同じ問題に直面した場合、多くの開発者は必然的に同じようなソリューションを見いだします。そういったソリューションが幾度も形になるまで繰り返されるうち、誰かが 1 つのパターンとして "発見" し、そこでようやく正式に名前を付けられた存在となるのです。

Gang of Four

今日のソフトウェアデザインパターンの多くは、Erich Gamma 氏、Richard Helm 氏、Ralph Johnson 氏、John Vlissides 氏による共著 *Design Patterns:Elements of Reusable Object-Oriented Software (オブジェクト指向における再利用のためのデザインパターン)* に端を発しています。同書では、毎日使用されるさまざまなアプリケーションで特定された 23 個のパターンを紹介しています。

原著者は "Gang of Four (ギャング・オブ・フォー)" (GoF) と呼ばれることが多く、オリジナルのパターンは GoF パターンと称されます。引用されている例はほとんど C++ (および Smalltalk) ですが、その考え方は C# などのオブジェクト指向言語全般に適用できます。

GoF が *Design Patterns* の初版を発行したのは 1994 年のことで、それ以降多くの開発者がさまざまな分野でいくつものオブジェクト指向パターンを発見しています。多くのエンジニアリング専門分野には、確立されたパターンがあります。ゲーム開発もその一つです。

デザインパターンを学ぶ

デザインパターンについて勉強しなくてもゲームプログラマーとして働くことはできますが、学習しておけば、より優れた開発者になれるでしょう。デザインパターンは、よく知られた問題に対する一般的なソリューションであるため、パターンとして分類されるのです。

ソフトウェアエンジニアであれば、普段の開発過程で繰り返し目にしているでしょう。そういったパターンのいくつかを無意識のうちにすでに実装しているかもしれません。

自分で見つけられるようになりましょう。以下を実践することが役立ちます。

- **オブジェクト指向プログラミングを学習する:**デザインパターンは、Stack Overflow の難解な投稿に埋もれた秘密ではありません。日々の開発における困難を乗り越えるための一般的な方法です。その同じ問題に対して、どれほど多くの開発者が取り組んできたかを示しています。たとえ自分がパターンを使用していなくても、他の誰かはパターンを使用していることを忘れないでください。
- **他の開発者に相談する:**パターンは、チームとしてコミュニケーションをとろうとするときに、省略表現のようなものとして機能することがあります。"コマンドパターン" や "オブジェクトプール" と言えば、経験豊富な Unity の開発者であれば何をやりたいのかを正確に汲み取ってくれます。
- **新しいフレームワークを探し出す:**Asset Store からビルトインのパッケージなどをインポートする際には、必然的にここで紹介するパターンのいくつかに出くわすことになります。デザインパターンを認識しておくと、新しいフレームワークの仕組みやその作成に関わる思考プロセスの理解に役立ちます。

もちろん、すべてのデザインパターンがどのゲームアプリケーションにも当てはまるわけではありません。マズローのハンマーを手にして探さないでください。そのように探しても、見つかるのは釘だけでしょう。

他のツールと同様に、デザインパターンの実用性はコンテキストに依存します。それぞれが一定の状況下でメリットをもたらし、同時に欠点ももたらします。ソフトウェア開発におけるあらゆる判断には、妥協が伴います。

大量のゲームオブジェクトを一気にまとめて生成していますか?そのことでパフォーマンスに影響は出ていますか?コードを再構築することで解決できますか?

デザインパターンを頭に入れておき、適切なタイミングでゲーム開発のヒントを引き出して、目の前の問題を解決しましょう。

参考資料

GoF による *Design Patterns:Elements of Reusable Object-Oriented Software* (オブジェクト指向における再利用のためのデザインパターン) に加えて、注目すべきは Robert Nystrom 氏による *Game Programming Patterns* (Game Programming Patterns ソフトウェア開発の問題解決メニュー) です。さまざまなソフトウェアパターンをわかりやすく解説しています。Web 版は、gameprogrammingpatterns.com で無料で閲覧できます。

Unity におけるパターン

Unity にはゲーム開発において確立されたパターンがすでにいくつか実装されており、自分で記述する手間を省けます。これには以下のものが含まれます。

— **ゲームループと更新**:ゲームアプリケーションを動かすハードウェアは多岐にわたるため、すべてのゲームにおいて核となるのは、クロック周波数とは無関係に機能する必要がある無限ループです。スピードが異なるさまざまなコンピューターに対応するために、ゲーム開発者は多くの場合、固定時間ステップ (設定した 1 秒あたりのフレーム数) と、前のフレームからの経過時間をエンジンが測定する変動時間ステップとを使用する必要があります。

これは Unity がデフォルトでサポートしているので、自分で実装する必要はありません。やらなければならないことは、Update、LateUpdate、FixedUpdate などの MonoBehaviour メソッドを使用してゲームプレイを管理することのみです。そして、ゲームクロックのフレームごとにゲームオブジェクトとコンポーネントを変更できます。

— **プロトタイプ**:元のオブジェクトに影響を及ぼすことなくオブジェクトをコピーする必要があるケースは多いです。この作成パターンは、他のオブジェクトを自身に似たものにするために、オブジェクトを複製してクローンを作成する際の問題を解決します。この方法により、ゲーム内にすべての種類のオブジェクトをスポーンする別のクラスを定義せずに済みます。

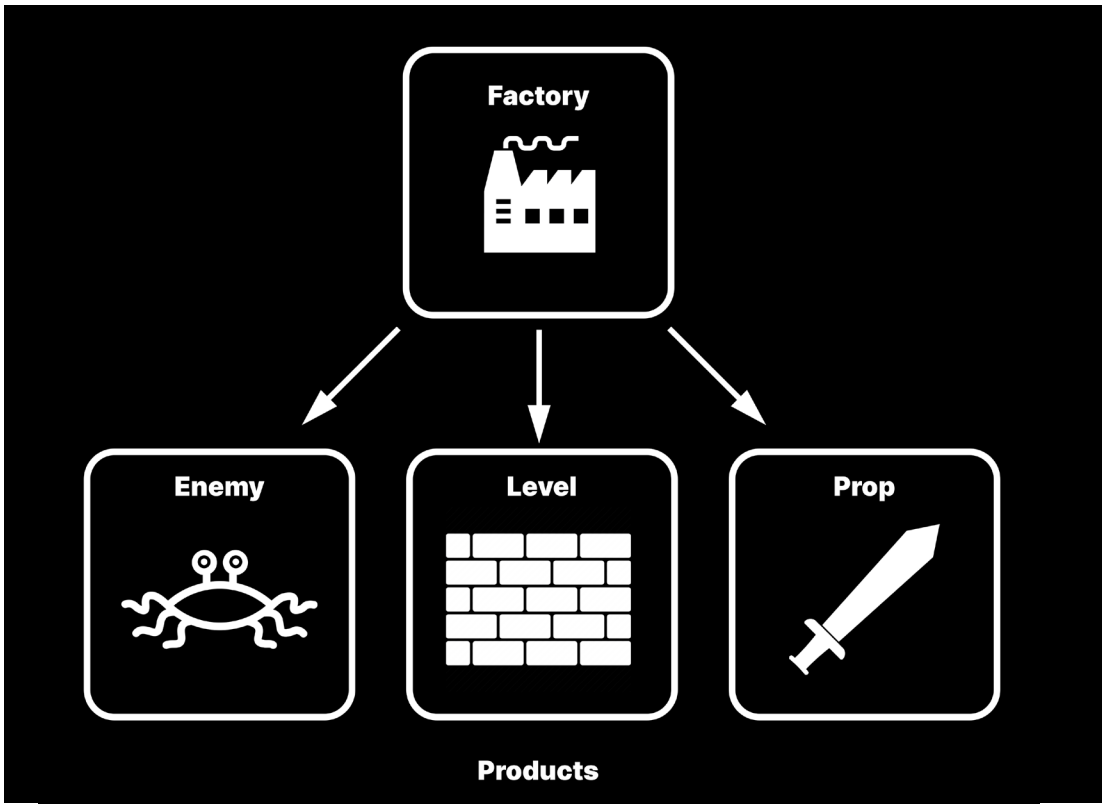
Unity の プレハブシステムは、ゲームオブジェクト用のプロトタイピングの一形態を実装しています。これにより、コンポーネントがすべて設定されたテンプレートオブジェクトを複製できます。特定のプロパティをオーバーライドして、プレハブバリアント を作成したり、他のプレハブ内に ネストプレハブ を作成して、階層を作ります。特別な プレハブ編集モード を使用して、単独またはコンテキスト内でプレハブを編集します。

— **コンポーネント**:Unity で作業しているほとんどの人は、このパターンを知っています。複数の役割を持つ大きなクラスを作成する代わりに、それぞれ 1 つのことを担う小さなコンポーネントを構築します。

合成を使用してコンポーネントを選択する場合は、それらを組み合わせて複雑な動作を実現します。物理演算の場合は、Rigidbody コンポーネントと Collider コンポーネントを追加します。3D ジオメトリの場合は、MeshFilter と MeshRenderer を追加します。各ゲームオブジェクトの機能と独自性は、そのコンポーネントのコレクションによって変わります。

もちろん、Unity がすべてをやってくれるわけではありません。ビルトインでないその他のパターンも必要になります。次の章から、そのうちのいくつかを見てみましょう。

ファクトリー (Factory) パターン



1 つのファクトリー (Factory) では 1 つ以上の製品をスポーンできる。

ときには、他のオブジェクトを作成する特別なオブジェクトがあると便利です。多くのゲームでは、ゲームプレイ中にさまざまなものがスポーンされますが、ランタイムに何が必要であるかは、実際に必要な局面になるまでわからないことが往々にしてあります。

ファクトリーパターンでは、想像どおり、ファクトリーという特別なオブジェクトを指定します。あるレベルで見れば、その "製品" (products) のスポーンに関わる多くの詳細がカプセル化されています。すぐに得られるメリットは、コードが整理されることです。

しかし、各製品が 1 つの共通のインターフェースや基本クラスに従っているならば、さらに一歩進めて、各製品が独自の構築ロジックを持つようにして、ファクトリーの外に置くこともできます。このように、新しいオブジェクトの作成はさらに拡張性が高くなります。

また、ファクトリーをサブクラス化して、特定の製品専用の複数のファクトリーを作成することもできます。こうすることで、敵キャラクターや障害物などをランタイムに生成できるようになります。

例：シンプルなファクトリー

あるゲームステージ向けに、アイテムをインスタンス化するファクトリーパターンを作成する必要があるとします。プレハブを使用してゲームオブジェクトを作成することはできますが、各インスタンスの作成中に、いくつかのカスタム動作を実行するとよいでしょう。

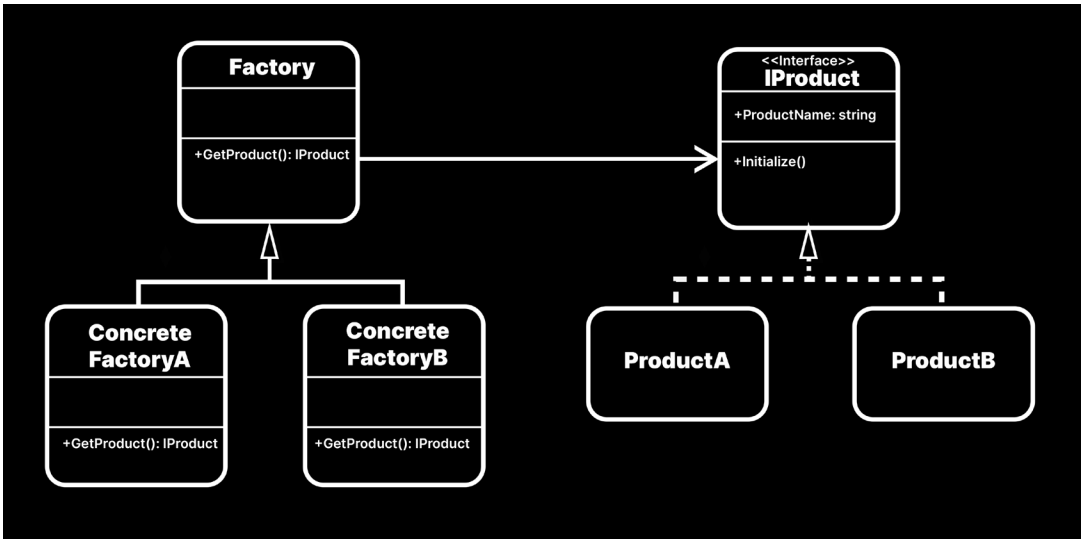
if ステートメントや `switch` を使用してこのロジックを保つよりも、`IProduct` というインターフェースと `Factory` という抽象クラスを作成します。

```
public interface IProduct
{
    public string ProductName { get; set; }

    public void Initialize();
}

public abstract class Factory :MonoBehaviour
{
    public abstract IProduct GetProduct(Vector3 position);

    // すべてのファクトリーでの共有メソッド
    ...
}
```

製品 (products) はメソッドのために特定のテンプレートに従う必要はありますが、それ以外に機能を共有することはありません。この理由から、IProduct インターフェースを定義することになります。

ファクトリーには共有された状態の共通機能が必要になることがあるため、このサンプルでは抽象クラスを使用します。サブクラスを使用する際には、SOLID の原則の 1 つであるリスコフの置換原則に注意してください。

下図のような構造になる場合があります。



インターフェースを使用して製品間で共有されたプロパティとロジックを定義する

`IProduct` インターフェースでは、製品間で共通の要素を定義します。このケースでは、1 つの `ProductName` プロパティと、その製品が `Initialize` で実行する任意のロジックがあるだけです。

その後、`IProduct` インターフェースに従っている限り、必要な数の製品を定義できます (`ProductA`、`ProductB` など)。

基本クラス `Factory` には、`IProduct` を返す `GetProduct` メソッドがあります。これは抽象クラスであるため、`Factory` のインスタンスを直接作成することはできません。いくつかの具象サブクラス (`ConcreteFactoryA` と `ConcreteFactoryB`) を派生させ、異なる製品をそれぞれ実際に取得します。

この例の `GetProduct` には、特定の場所でプレハブゲームオブジェクトをより簡単にインスタンス化できるように、Vector3 の位置情報があります。具体的なファクトリーそれぞれのフィールドには、対応するテンプレートプレハブも格納されます。

ProductA と ConcreteFactoryA のサンプルを紹介します。

```
public class ProductA :MonoBehaviour, IProduct
{
    [SerializeField] private string productName = “ProductA”;
    public string ProductName { get => productName; set => productName
    = value ; }

    private ParticleSystem particleSystem;

    public void Initialize()
    {
        // この製品で独自のロジック
        gameObject.name = productName;
        particleSystem = GetComponentInChildren<ParticleSystem>();
        particleSystem?.Stop();
        particleSystem?.Play();
    }
}

public class ConcreteFactoryA :Factory
{
    [SerializeField] private ProductA productPrefab;

    public override IProduct GetProduct(Vector3 position)
    {
        // プレハブインスタンスを作成して製品コンポーネントを取得する
        GameObject instance = Instantiate(productPrefab.gameObject,
        position, Quaternion.identity);
        ProductA newProduct = instance.GetComponent<ProductA>();

        // 各製品が独自のロジックを持つ
        newProduct.Initialize();

        return newProduct;
    }
}
```

ここでは、IProduct を実装する製品クラスの MonoBehaviour にファクトリーのプレハブを利用させました。

製品ごとに独自のバージョンの `Initialize` を設定する方法に注目してください。この ProductA プレハブの例には、ParticleSystem が含まれており、ConcreteFactoryA でコピーをインスタンス化されると再生されます。ファクトリー自体には、パーティクルをトリガーする具体的なロジックはありません。すべての製品に共通の `Initialize` メソッドを呼び出すだけです。

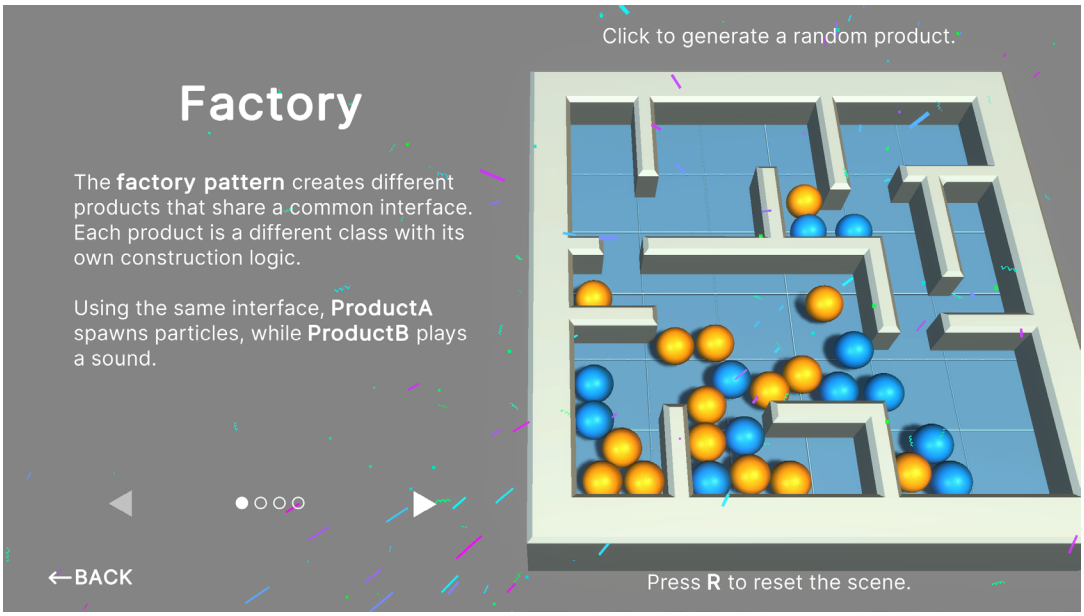
サンプルプロジェクトを調べ、`ClickToCreate` コンポーネントがファクトリー間で、どのように切り替わり、動作が異なる ProductA と ProductB を作成するか確認してください。Product B がスポーンされると音が鳴り、Product A はパーティクルエフェクトを発生させます。

長所と短所

ファクトリーパターンで一番恩恵を受けるのは、多数の製品を設定するときです。アプリケーションで新しい種類の製品を定義する場合に、既存の製品を変更することも、以前のコードを変更する必要もありません。

各製品の内部ロジックを独自のクラスに分離することで、ファクトリーコードを比較的短く保つことができます。各ファクトリーで認識されているのは各製品の `Initialize` を呼び出すことのみで、その下にある詳細については関知しません。

欠点は、パターンを実装するために、多数のクラスやサブクラスを作成する必要があることです。他のパターンと同じように、製品の種類がそれほど多くない場合に不要なオーバーヘッドが少し発生してしまいます。



ある製品はサウンドを再生し、別の製品はパーティクルを再生する。どちらも同じインターフェースを使用している。

改善点

ファクトリーパターンの実装は、ここで紹介したものと大きく異なったものになる場合があります。独自のファクトリーパターンを構築する際には、以下の調整を検討してください。

— **ディクショナリを使用して製品を検索する:**製品をキーと値のペアとしてディクショナリに格納するとよいでしょう。一意の文字列識別子 (名前、ID など) をキー、型を値として使用します。これにより、製品や対応するファクトリーの取得がさらに便利になることがあります。

— **ファクトリー (またはファクトリーマネージャー) を静的 (static) にする:**これにより使い勝手は良くなりますが、追加の設定が必要です。静的クラスは Inspector に表示されないため、製品のコレクションも静的にする必要があります。

— **ゲームオブジェクト以外、MonoBehaviour 以外に適用する:**プレハブやその他 Unity 固有のコンポーネントに制限しないようにしてください。ファクトリーパターンはあらゆる C# オブジェクトで機能します。

— **オブジェクトプールパターンと組み合わせる:**ファクトリーでは、必ずしもオブジェクトをインスタンス化することや、新しいオブジェクトを作成することを必要としません。また、階層内の既存のオブジェクトを取得することもできます。一度に多数のオブジェクト (武器から発射される弾など) をインスタンス化する場合は、オブジェクトプールパターンを使用して、メモリ管理をさらに最適化することができます。

ファクトリーを使用すれば、任意のゲームプレイ要素を必要に応じてスポーンできます。ただし、多くの場合、製品を作成することが唯一の目的ではないことに注意してください。ファクトリーパターンを別の大きなタスクの一部 (例えば、ゲームステージの一部であるダイアログボックスの UI 要素の設定) として使用することがあります。

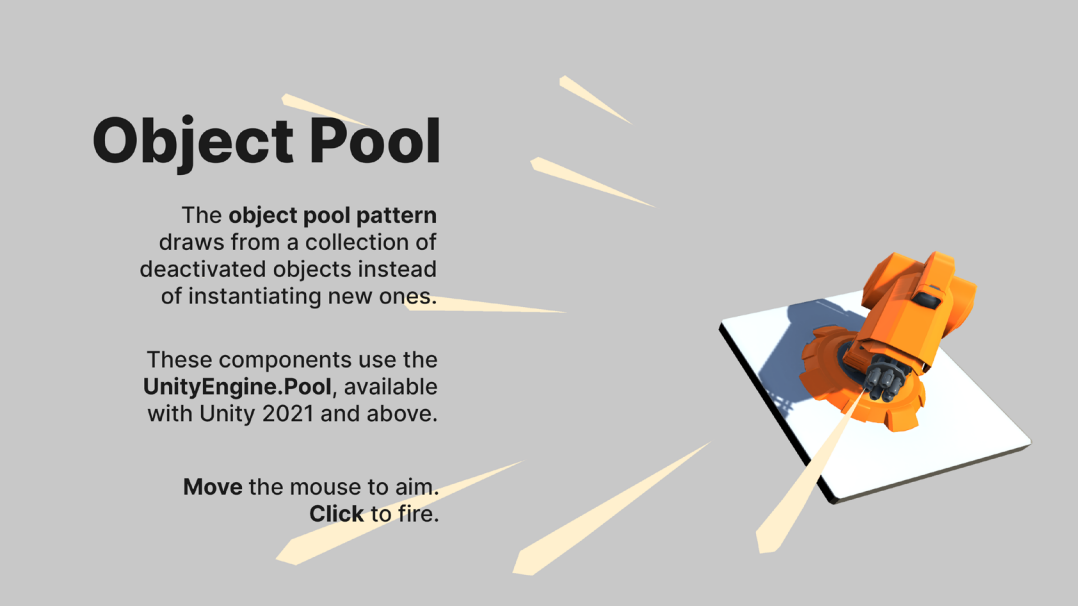
オブジェクトプール (Object pool)

ゲームシーン内の多数のオブジェクトのライフサイクル管理は、最適なパフォーマンスを実現するための鍵となります。C# の自動メモリ管理システムにはガベージコレクション機能があり便利ですが、オブジェクトが頻繁に作成されたり破棄されたりすると、目立ったスタッターやスパイクが発生することがあります。

これを軽減するために、**オブジェクトプール** (Object pool) パターンの使用を検討してください。この手法では、ゲームオブジェクトを再利用することでパフォーマンスを最適化します。オブジェクトを随時作成して破棄するのではなく、初期化され非アクティブ化されたオブジェクトの "プール" を保持します。オブジェクトが必要になったとき、アプリケーションがオブジェクトをインスタンス化するのではありません。代わりに、プールのゲームオブジェクトをリクエストし、有効にします。

オブジェクトは使用後に非アクティブ化され、プールに戻されるので、破棄によるオーバーヘッドを回避できます。理想的には、スタッターを防ぐために、あまり目立たないタイミング (ローディング画面中など) でオブジェクトプールを初期化してください。この最適化手法は、多数のゲームオブジェクトを作成、破棄したいときに役立ちます。

Unity の ParticleSystem を使用したことがあれば、オブジェクトプールを直に体験したことがあります。ParticleSystem コンポーネントでは、パーティクルの最大数を設定できます。これによって使用可能なパーティクルがリサイクルされるため、エフェクトが最大数を超えるのを防ぎます。オブジェクトプールは、選択した任意のゲームオブジェクトで、同じように機能します。



オブジェクトプールは、ゲームプレイにスタッターをもたらすことなく弾丸を撃つのに役立ちます。

例：シンプルなプールシステム

Unity には、UnityEngine.Pool 名前空間からアクセス可能なビルトインのオブジェクトプーリング機能が含まれています。Unity 2021 LTS 以降で利用可能なこの名前空間は、オブジェクトプールの管理を容易にし、オブジェクトのライフサイクルやプールサイズのコントロールなどの作業を自動化します。

独自のオブジェクトプールを作成してみると、このパターンがどのように機能するのか、その基本原理を理解するのに役立ちます。それでは、シンプルなオブジェクトプールを構築し、その仕組みを見ていきましょう。

2 つの MonoBehaviour が定義された、以下のようなシンプルなプーリングシステムを考えてみましょう。

— 描画元のゲームオブジェクトのコレクションを保持する `ObjectPool`

— クローンされた各アイテムがプールへの参照を保持できるよう、プレハブに追加された `PooledObject` コンポーネント

ObjectPool で、プールサイズを記述したフィールド、格納する必要がある PooledObject プレハブ、プール自体を形成するコレクション (この例では stack) を設定します。

```
public class ObjectPool :MonoBehaviour
{
    [SerializeField] private int initPoolSize;
    [SerializeField] private PooledObject objectToPool;

    // コレクション内のプールされたオブジェクトを格納する
    private Stack<PooledObject> stack;

    private void Start()
    {
        SetupPool();
    }

    // プールを作成する（ラグが目立たなくなったら呼び出す）
    private void SetupPool()
    {
        stack = new Stack<PooledObject>();
        PooledObject instance = null;

        for (int i = 0; i < initPoolSize; i++)
        {
            instance = Instantiate(objectToPool);
            instance.Pool = this;
            instance.gameObject.SetActive(false);
            stack.Push(instance);
        }
    }
```

SetupPool メソッドは、オブジェクトプールにデータを設定します。PooledObject の新しいスタックを作成してから、objectToPool のコピーをインスタンス化して、initPoolSize 要素で満たします。Start で SetupPool を呼び出し、ゲームプレイ中に 1 回だけ実行されるようにします。

また、プールされたアイテムを取得するメソッド (GetPooledObject) と、そのアイテムをプールに返すメソッド (ReturnToPool) も必要です。

```
    // プールから最初のアクティブなゲームオブジェクトを返す
    public PooledObject GetPooledObject()
    {

        // プールサイズが十分でない場合は、新しい PooledObject をインスタンス化する
        if (stack.Count == 0)
        {
            PooledObject newInstance = Instantiate(object
            ToPool);
            newInstance.Pool = this;
            return newInstance;
        }

        // それ以外の場合は、リストから以下のものを取得する
        PooledObject nextInstance = stack.Pop();
        nextInstance.gameObject.SetActive(true);
        return nextInstance;
    }

    public void ReturnToPool(PooledObject pooledObject)
    {
        stack.Push(pooledObject);
        pooledObject.gameObject.SetActive(false);
    }
}
```

プールが空の場合にのみ、GetPooledObject が、新しい PooledObject を作成します。それ以外の場合は、単に次の有効な要素を返します。プールサイズが十分であれば、ほとんどの場合、既存のゲームオブジェクトへの参照のみが返されます。

GetPooledObject を呼び出しているクライアントは、プールされたオブジェクトを所定の位置に移動/回転する必要があります。

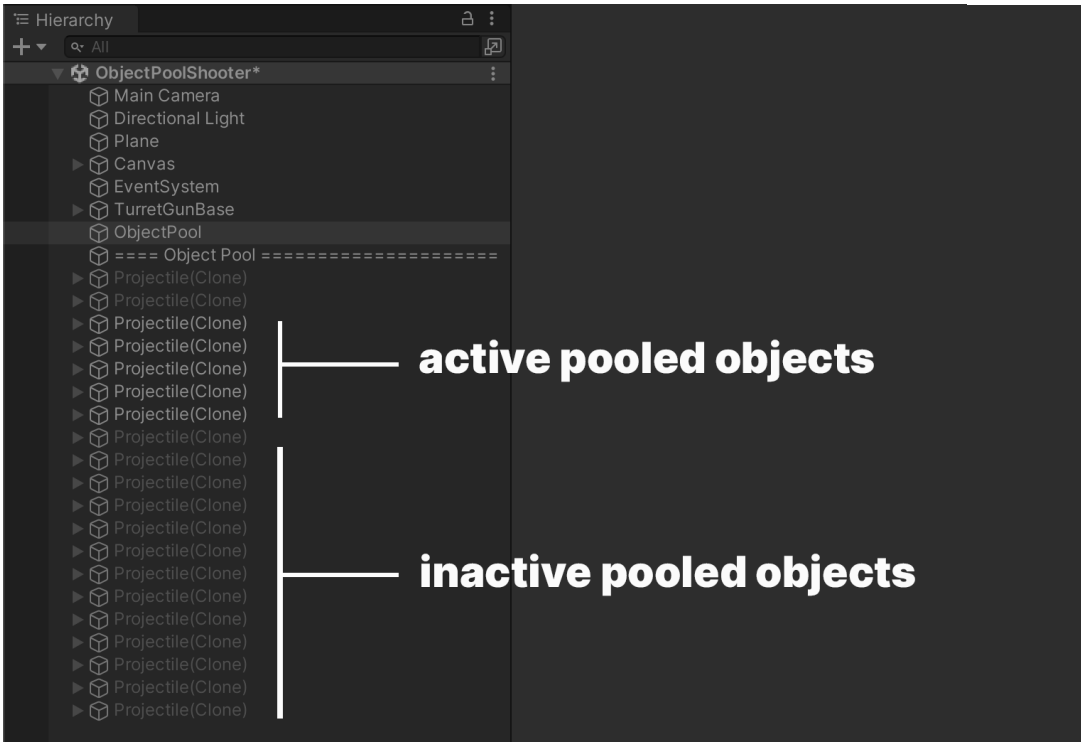
プールされた各要素には、ObjectPool を参照する役割のみを担う小さな PooledObject コンポーネントが存在します。

```
public class PooledObject :MonoBehaviour
{
    private ObjectPool pool;
    public ObjectPool Pool { get => pool; set => pool = value; }

    public void Release()
    {
        pool.ReturnToPool(this);
    }
}
```

Release を呼び出すと、そのゲームオブジェクトが無効になり、プールのキューへと返されます。

この付属のプロジェクトには、ゲームオブジェクトにアタッチされた ExampleGun スクリプトが含まれています。そこにオブジェクトプールへの参照が格納されます。ユーザーが弾を撃つと、武器のスクリプトは、Object.Instantiate を呼び出す代わりに、そのスクリプトの GetPooledObject メソッドを呼び出します。

発射される弾自体には ExampleProjectile スクリプトと PooledObject スクリプトがあります。ExampleProjectile には、発射されたそれぞれの弾のゲームオブジェクトを数秒後に無効にし、使用可能なプールに返すことを担う、Deactivate メソッドがあります。



プールされたオブジェクトを無効にして再利用する

この方法により、何百発もの弾を画面外で発射しているように見せて、実際にはそれらを無効にしてリサイクルすることができます。念のため、プールサイズが、同時にアクティブなオブジェクトを表示するのに十分であることを確認してください。

プールサイズを超えてしまう場合は、そのプールで追加のオブジェクトをインスタンス化できます。ただし、ほとんどの場合は、既存の非アクティブなオブジェクトから持ってきます。

オブジェクトプールをゼロから作成する実装については、サンプルプロジェクトの **ManualExample** フォルダーを参照してください。

UnityEngine.Pool

Unity には、UnityEngine.Pool 名前空間 (Unity 2021 LTS 以降で利用可能) を介してオブジェクトプールシステムが組み込まれているため、前の例のように独自の `PooledObject` クラスや `ObjectPool` クラスを作成する必要はありません。

これにより、スタックベースの `ObjectPool` を利用して、オブジェクトプールパターンを使用してオブジェクトを追跡できます。必要に応じて、`CollectionPool` (List、HashSet、Dictionary など) を使用することもできます。

サンプルプロジェクトでは、`UnityEngine.Pool` にビルトインされた `ObjectPool` を使用して、手動で作成した弾丸のプールを再構築する方法を示しています。

```
using UnityEngine.Pool;

public class RevisedGun :MonoBehaviour
{
    ...

    // Unity 2021 以降で利用可能なスタックベースの ObjectPool
    private IObjectPool<RevisedProjectile> objectPool;

    // すでにプール内にある既存のアイテムを返そうとすると例外がスローされる
    [SerializeField] private bool collectionCheck = true;

    // プールの容量と最大サイズを制御する追加のオプション
    [SerializeField] private int defaultCapacity = 20;
    [SerializeField] private int maxSize = 100;
```

```
    private void Awake()
    {
        objectPool = new ObjectPool<RevisedProjectile>
        (CreateProjectile,OnGetFromPool, OnReleaseToPool,
        OnDestroyPooledObject, collectionCheck, defaultCapacity, maxSize);
    }

    // オブジェクトプールに読み込むアイテムの作成時に呼び出される
    private RevisedProjectile CreateProjectile()
    {
        RevisedProjectile projectileInstance = Instantiate(projectilePrefab);
        projectileInstance.ObjectPool = objectPool;
        return projectileInstance;
    }

    // オブジェクトプールにアイテムを返すときに呼び出される
    private void OnReleaseToPool(RevisedProjectile pooledObject)
    {
        pooledObject.gameObject.SetActive(false);
    }

    // オブジェクトプールから次のアイテムを取得するときに呼び出される
    private void OnGetFromPool(RevisedProjectile pooledObject)
    {
        pooledObject.gameObject.SetActive(true);
    }
    // プールされたアイテムの最大数を超える（プールされたオブジェクトを破棄する）と呼び出される
    private void OnDestroyPooledObject(RevisedProjectile pooledObject)
    {
        Destroy(pooledObject.gameObject);
    }
    private void FixedUpdate()
    {
        ...
    }
}
```

このスクリプトの大部分は、元の ExampleGun スクリプトで機能します。ただし、以下のタイミングでいくつかのロジックを設定するための便利な機能が、ObjectPool コンストラクターに追加されています。

— プールに入れるアイテムを最初に作成するとき

— プールからアイテムを取得するとき

— プールにアイテムを戻すとき

— プールされたオブジェクトを破棄するとき (上限に達した場合など)

次に、コンストラクターに渡す対応メソッドをいくつか定義する必要があります。

ビルトインの `ObjectPool` に、デフォルトのプールサイズと最大プールサイズのオプションがどのように含まれているかにも注目してください。アイテム数が最大プールサイズを超えていると、自己破壊するアクションがトリガーされ、メモリ使用量が管理されます。

発射される弾のスクリプトに少し修正が加えられ、`ObjectPool` への参照が保たれます。これにより、オブジェクトを解放してプールへ戻すのが少しやりやすくなります。

```
public class RevisedProjectile :MonoBehaviour
{
    ...

    private IObjectPool<RevisedProjectile> objectPool;

    // 発射される弾に対して、その ObjectPool への参照を与える public プロパティ
    public IObjectPool<RevisedProjectile> ObjectPool { set => object
    Pool = value; }

   ...
}
```

UnityEngine.Pool API を使用すると、オブジェクトプールをより短時間で設定できるようになり、ゼロからパターンを再構築する必要がなくなります。これで、すでにあるものをゼロから作成する手間が 1 つ減ることになります。

長所と短所

オブジェクトプールはパフォーマンスを最適化するための強力なツールですが、すべてのデザインパターンに関連する考慮事項があることに注意してください。

オブジェクトプールには以下の利点があります。

— **ガベージコレクションのオーバーヘッドの削減:**オブジェクトを作成して破棄する代わりに再利用することで、ガベージコレクションの必要性を減らします。これにより、ランタイムのパフォーマンスのスパイクやスタッターを防止できます。

— **パフォーマンスの向上:**オブジェクトを事前に初期化し、必要に応じて再アクティブ化することで、テンポの速いゲーム (シューティングゲームなど) でスムーズなパフォーマンスを実現できます。

— **初期化の最適化:**オブジェクトの作成をあまり影響のないタイミングに分散させることで、リソースとアプリケーションの起動時間を最適化します。

ただし、以下の欠点があることを覚えておいてください。

— **複雑さの増大:**オブジェクトプールの管理には手間がかかります。オブジェクトの初期化と解放を適切に行ってください。そうしないと、エラーやバグが発生する可能性があります。

— **メモリ使用量:**オブジェクトプールはガベージコレクションを減らす一方で、静的メモリ使用量の増加につながる可能性があります。オブジェクトプールは、未使用のオブジェクトであっても、あらかじめ設定された数のオブジェクトをメモリに格納します。プロジェクトのニーズに応じてプールサイズを調整してください。

— **さらなる管理:**オブジェクトプールの最適なサイズの決定が難しい場合があります。プールが小さすぎるとメモリ割り当てが頻繁になり、プールが大きすぎると割り当てられたメモリが十分に活用されない場合があります。

改善点

上記の例はシンプルなものです。実際のプロジェクトにオブジェクトプールを導入する際には、以下のアップグレードを検討してください。

— **静的またはシングルトンにする:**さまざまなソースからプールされたオブジェクトを生成する必要がある場合は、オブジェクトプールを静的で再利用可能にすることを検討してください。これにより、アプリケーション内のどこからでもアクセスできるようになりますが、Inspector は使用できなくなります。または、オブジェクトプールパターンをシングルトンパターンと組み合わせて、グローバルにアクセス可能にすることで、使い勝手が良くなります。

— **ディクショナリを使用して複数のプールを管理する:**プールするプレハブの種類が多い場合は、それらを別個のプールに格納し、キーと値のペアを格納することで、クエリ対象のプールを把握しやすくなります (プレハブの InstanceID を一意のキーとして使用できます)。

— **使用されていないゲームオブジェクトをうまく削除する:**オブジェクトプールを効果的に利用する方法の 1 つは、使用されていないオブジェクトを非表示にしてプールに戻すことです。あらゆる機会を利用して、プールされたオブジェクトを非アクティブにします (画面外、爆発で隠すなど)。

— **エラーがないか確認する:**すでにプール内にあるオブジェクトを解放することは避けてください。**ObjectPool<T>** のインスタンスを作成するときに、`collectionCheck` パラメーターを true に設定できます。これにより、すでにあるプールにオブジェクトを返そうとした場合に、エディターで例外がスローされます。

— **最大サイズ/上限を追加する:**プールされたオブジェクトが大量にあると、メモリの消費につながります。ObjectPool コンストラクターの `maxSize` パラメーターを使用してプールサイズに上限を設定します。

オブジェクトプールをどう利用するかは、アプリケーションによって異なります。このパターンは一般的に、弾幕系シューティングなど、銃や武器から複数の弾を発射する必要があるときに利用されます。

大量のオブジェクトをインスタンス化するたびに、ガベージコレクションのスパイクにより短時間の中断が発生するおそれがあります。オブジェクトプールはこのような問題を軽くし、ゲームプレイをスムーズに保ちます。

シングルトン (Singleton) パターン

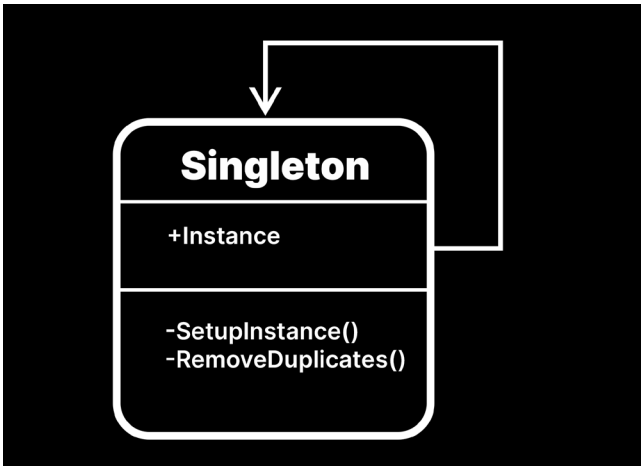
シングルトン (Singleton) は評判がよくありません。Unity 開発が初めての方にとって、おそらく最初に認識するパターンの 1 つがシングルトンでしょう。また、最も批判を受けるデザインパターンの 1 つでもあります。

オリジナルの Gang of Four によると、シングルトンパターンは以下のとおりです。

— クラスがクラス自体の 1 つのインスタンスしかインスタンス化できないことを保証する

— その単一のインスタンスへのグローバルアクセスを提供する

これは、シーン全体でアクションを調整するただ 1 つのオブジェクトが必要な場合には便利です。例えば、メインのゲームループを導くゲームマネージャーは、シーン内に 1 つだけにするのがよいでしょう。また、一度に 1 つのファイルマネージャーだけがファイルシステムに書き込めるようにするのがよいでしょう。このような中心的な役割を担うマネージャーレベルのオブジェクトは、シングルトンパターンを採用する有力な候補となる傾向にあります。



SimpleSingleton は最初のインスタンスより後のすべてのインスタンスを破棄する。

Game Programming Patterns において、シングルトンはプラスよりもむしろマイナスに作用すると言われており、アンチパターンとして挙げられています。悪評の理由は、このパターンの使い勝手の良さが乱用につながるおそれがあるためです。開発者はあまりふさわしくない状況でシングルトンを適用する傾向にあり、不要なグローバル状態や依存関係が生み出されてしまいます。

Unity でシングルトンを構築する方法を確認し、その長所と短所を比較検討してみましょう。それを踏まえて、自分のアプリケーションに組み込む価値があるかどうかを判断してください。

例：シンプルなシングルトン

最もシンプルなシングルトンの 1 例です。

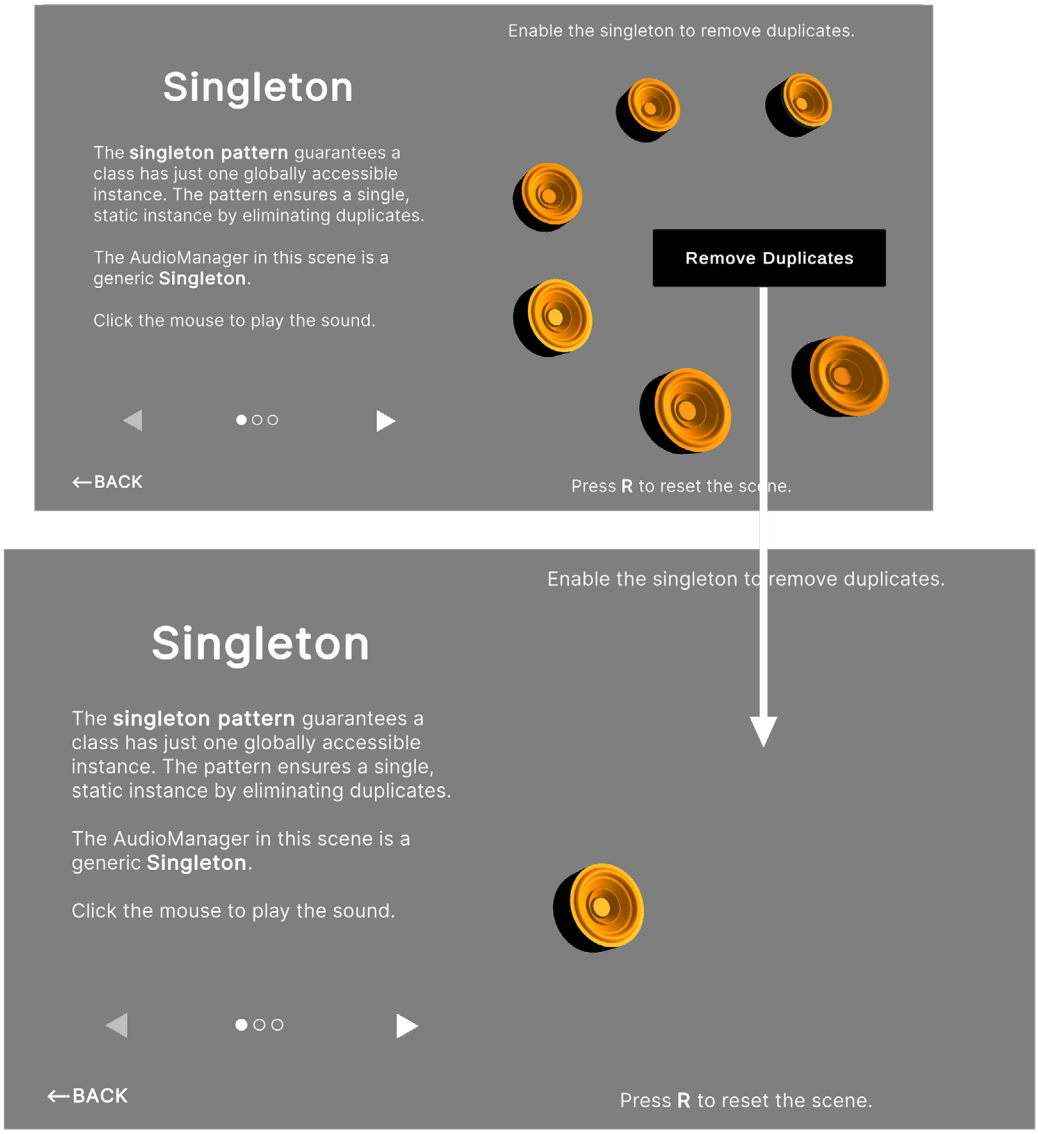
```
using UnityEngine;

public class SimpleSingleton :MonoBehaviour
{
    public static SimpleSingleton Instance;

    private void Awake()
    {
        if (Instance == null)
        {
            Instance = this;
        }
            else
        {
            Destroy(gameObject);
        }
    }
}
```

public static `Instance` にシーン内の `Singleton` の 1 つのインスタンスが保持されます。

`Awake` メソッドで、それがすでに設定されているかチェックします。`Instance` が現在 null の場合、Instance がこの特定のオブジェクトに設定されます。これはシーン内の最初のシングルトンであるはずです。

それ以外の場合、このインスタンスは必ず複製です。`Destroy(gameObject)` を呼び出して、シングルトンにはシーン内にそのようなコンポーネントが 1 つだけあることを保証します。

ランタイムにスクリプトを階層内の複数のゲームオブジェクトにアタッチすると、Awake のロジックは最初のオブジェクトを残し、それ以外を破棄します。



シングルトンパターンでは 1 つのインスタンスのみを許可する。

`Instance` フィールドは public かつ static です。どのコンポーネントも、シーン内のどこからでも唯一のシングルトンにグローバルにアクセスできます。

永続性と遅延インスタンス化

`SimpleSingleton` は記述されているとおりに機能します。ただし、以下の 2 つの問題に苦しめられることになります。

新しいシーンをロードすると、ゲームオブジェクトが破壊されます。

— シングルトンを使用する前に、そのシングルトンを階層に設定する必要があります。

— シングルトンはどこにでも存在するマネージャースクリプトとして機能することが多いため、`DontDestroyOnLoad` を使用して永続化するメリットはあります。

さらに、遅延インスタンス化 を使用して、最初に必要になったときにシングルトンを自動的に構築できます。必要なのは、ゲームオブジェクトを作成して適切な Singleton コンポーネントを追加するいくつかのロジックだけです。

改良されたシングルトンは以下のようになります。

```
public class Singleton :MonoBehaviour
{
      private static Singleton instance;
      public static Singleton Instance
      {
          get
          {
              if (instance == null)
              {
                  SetupInstance();
              }
              return instance;
          }
      }

      private void Awake()
      {
          if (instance == null)
          {
              instance = this;
              DontDestroyOnLoad(this.gameObject);
          }
          else
          {
              Destroy(gameObject);
          }
      }

      private static void SetupInstance()
      {
          instance = FindObjectOfType<Singleton>();
          if (instance == null)
          {
              GameObject gameObj = new GameObject();
              gameObj.name = “Singleton”;
              instance = gameObj.AddComponent<Singleton>();
              DontDestroyOnLoad(gameObj);
          }
      }
```

これで Instance は、private instance バッキングフィールドを参照している public プロパティになりました。初めてシングルトンを参照するときに、取得しているものの中に Instance が存在するかどうかをチェックします。存在しない場合は、SetupInstance メソッドが、適切なコンポーネントが備わったゲームオブジェクトを作成します。

DontDestroyOnLoad(gameObject) は、シーンのロード時に階層からシングルトンを消去しないようにします。これでシングルトンインスタンスが永続化されたため、ゲーム内のシーンを変更してもアクティブなままになります。

ジェネリックの使用

どちらのバージョンのスクリプトも、同じシーン内に異なるシングルトンを作成する方法には対応していません。例えば、AudioManager として機能するシングルトンと、GameManager として機能する別のシングルトンが必要な場合は、現時点では共存させることはできません。関連するコードを複製し、そのロジックを各クラスに貼り付ける必要があります。

代わりに、以下のようにジェネリックバージョンのスクリプトを作成します。

```
public class Singleton<T> :MonoBehaviour where T :コンポーネント
{
    private static T instance;
    public static T Instance
    {
        get
        {
            if (instance == null)
            {
                instance = (T)FindObjectOfType(typeof(T));
                if (instance == null)
                {
                    SetupInstance();
                }
            }
             return instance;
        }
    }

    public virtual void Awake()
    {
        RemoveDuplicates();
    }

    private static void SetupInstance()
    {
        instance = (T)FindObjectOfType(typeof(T));

        if (instance == null)
        {
```

```
            GameObject gameObj = new GameObject();
            gameObj.name = typeof(T).Name;
            instance = gameObj.AddComponent<T>();
            DontDestroyOnLoad(gameObj);
        }
    }

    private void RemoveDuplicates()
    {
        if (instance == null)
        {
            instance = this as T;
            DontDestroyOnLoad(gameObject);
        }
        else
        {
            Destroy(gameObject);
        }
    }
}
```

これにより、任意のクラスをシングルトンに変えることができます。クラスを宣言するときに、ジェネリックシングルトンを継承するだけです。例えば、以下のように宣言することで、GameManager という MonoBehaviour をシングルトンにできます。

```
public class GameManager:Singleton<GameManager>
{
    // ...
}
```

これで、必要なときにいつでも public かつ static な `GameManager.Instance` を参照できます。

長所と短所

シングルトンは、このガイドで紹介するその他のパターンとは異なり、いくつかの点で SOLID の原則を破ります。以下のようなさまざまな理由で、多くの開発者から嫌われています。

— **シングルトンにはグローバルアクセスが必要:**シングルトンをグローバルインスタンスとして使用するため、数多くの依存関係が隠されてしまい、バグのトラブルシューティングが非常に難しくなるおそれがあります。

— **シングルトンではテストが難しくなる:**単体テストは互いに独立している必要があります。シングルトンによってシーン全体にわたって数多くのゲームオブジェクトの状態が変わる可能性があるため、テストに干渉するおそれがあります。

— **シングルトンにより密な結合が助長される:**このガイドで紹介するほとんどのパターンでは、依存関係を切り離そうとします。シングルトンではその逆を行います。密な結合により、リファクタリングが難しくなります。1 つのコンポーネントを変更すると、接続されているすべてのコンポーネントに影響が及び、コードがクリーンでなくなるおそれがあります。

シングルトンに対する拒否反応はかなりのものです。エンタープライズレベルのゲームを制作している方であれば、今後何年間にもわたるメンテナンスが待ち受けているため、シングルトンには関わらないことをお勧めします。

しかし、多くのゲームはエンタープライズレベルのアプリケーションではありません。ビジネスソフトウェアほど継続的に拡張する必要はありません。

実際、拡張性を必要としない小さなゲームを制作している場合には、シングルトンに備わっている以下のようなメリットを魅力的に感じるかもしれません。

— **シングルトンは比較的短時間で習得できる:**そのコアパターン自体はそれほど難しいものではありません。

— **シングルトンは使い勝手が良い:**別のコンポーネントからシングルトンを使用するには、public かつ static なインスタンスを参照するだけで済みます。シングルトンインスタンスは、シーン内のあらゆるオブジェクトから必要に応じて常に使用できます。

— **シングルトンは高性能:**静的なシングルトンインスタンスに常にグローバルにアクセスできるため、低速になりがちな `GetComponent` 操作や Find 操作の結果がキャッシュされることがなくなります。

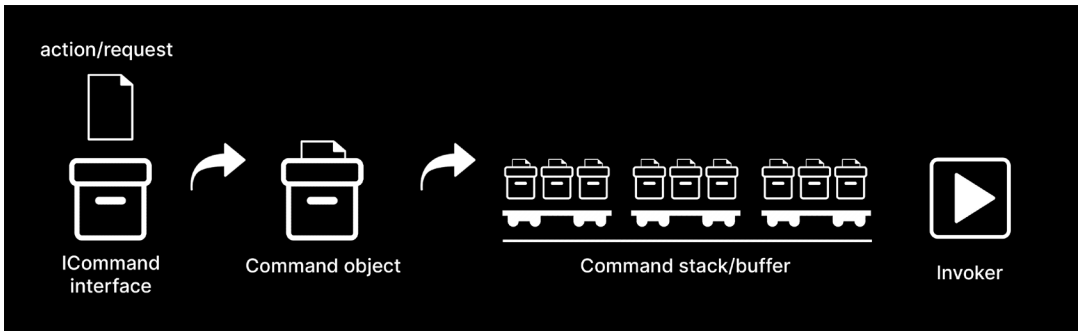
この方法により、シーン内の他のすべてのゲームオブジェクトから常にアクセス可能なマネージャーオブジェクト（ゲームフローマネージャーやオーディオマネージャーなど）を作成できます。また、オブジェクトプールを実装している場合は、プーリングシステムをシングルトンとしてデザインすることで、プールされたオブジェクトを簡単に取得できるようになります。

プロジェクトでシングルトンを使用することにした場合は、最小限に留めるようにしてください。乱用しないでください。グローバルアクセスのメリットを享受できる一握りのスクリプトのためにシングルトンを取っておいてください。

コマンド (Command) パターン

オリジナルの Gang of Four パターンの 1 つであるコマンド (Command) は、特定の一連のアクションを追跡する必要があるときに便利です。取り消し (undo)/やり直し (redo) 機能を使用したり、ユーザーの入力履歴をリストに残したりするゲームをプレイしたことがある方であれば、コマンドパターンが動いているのを見たことがあるでしょう。ユーザーが実行に移す前に複数のターンにわたる計画を立てられるストラテジーゲームを想像してみてください。それがコマンドパターンです。

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。



コマンドパターンを使用したアクションの格納

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することで、それらの実行タイミングを制御できます。これは小さなバッファとして機能します。その後、一連のアクションを遅らせて後で再生するか、取り消すことができます。

コマンドパターンを実装するには、アクションを含む一般的なオブジェクトが必要です。このコマンドオブジェクトに、実行するロジックとそれを取り消す方法が保持されます。

コマンドオブジェクトとコマンド呼び出し元

実装する方法はいくつかありますが、インターフェースを使用するバージョンを紹介します。

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

このケースでは、ゲームプレイのすべてのアクションに `ICommand` インターフェースが適用されます (これを抽象クラスで実装することもできます)。

各コマンドオブジェクトは、それぞれ独自の `Execute` メソッドと `Undo` メソッドを担います。したがって、ゲームにさらにコマンドを追加しても、既存のコマンドには影響を及ぼしません。

コマンドを実行および取り消す別のクラスが必要になります。`CommandInvoker` クラスを作成します。`ExecuteCommand` メソッドと `UndoCommand` メソッドに加えて、コマンドオブジェクトのシーケンスが保持される取り消しのスタックがあります。

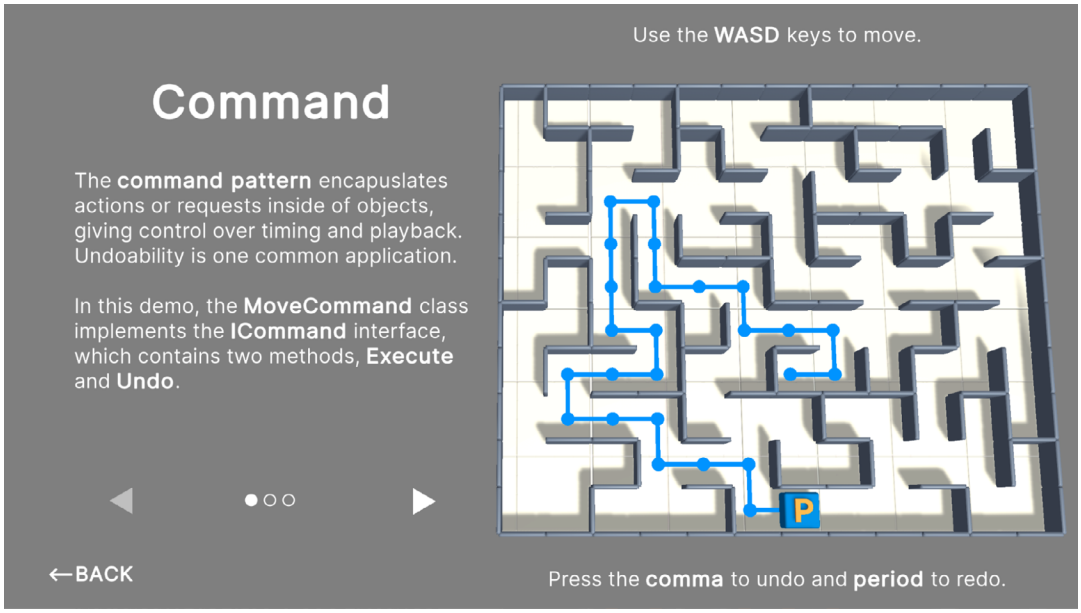
```
public class CommandInvoker
{
    private static Stack<ICommand> undoStack = new Stack<ICommand>();

    public static void ExecuteCommand(ICommand command)
    {
        command.Execute();
        undoStack.Push(command);
    }

    public static void UndoCommand()
    {
        if (undoStack.Count > 0)
        {
            ICommand activeCommand = undoStack.Pop();
            activeCommand.Undo();
        }
    }
}
```

例：取り消し可能な動作

アプリケーションで、プレイヤーに迷路の中を動き回ってもらいたいとします。これには、プレイヤーの位置を動かすことを担う `PlayerMover` を作成する方法があります。

```
public class PlayerMover :MonoBehaviour
{
    [SerializeField] private LayerMask obstacleLayer;
    private const float boardSpacing = 1f;

    public void Move(Vector3 movement)
    {
        transform.position = transform.position + movement;
    }

    public bool IsValidMove(Vector3 movement)
    {
        return !Physics.Raycast(transform.position, movement, board
        Spacing, obstacleLayer);
    }
}
```

Move メソッドに Vector3 を渡し、4 つのコンパス方向に沿ってプレイヤーをガイドします。また、レイキャストを使用して、該当する LayerMask の壁を検出することもできます。もちろん、コマンドパターンに適用したいものを実装することは、パターン自体とは別です。



コマンドパターンはアクションを取り消し可能にできる。

コマンドパターンに従うには、PlayerMover の Move メソッドをオブジェクトとしてキャプチャします。Move を直接呼び出す代わりに、新しいクラス MoveCommand を作成します。このクラスには、ICommand インターフェースを実装します。

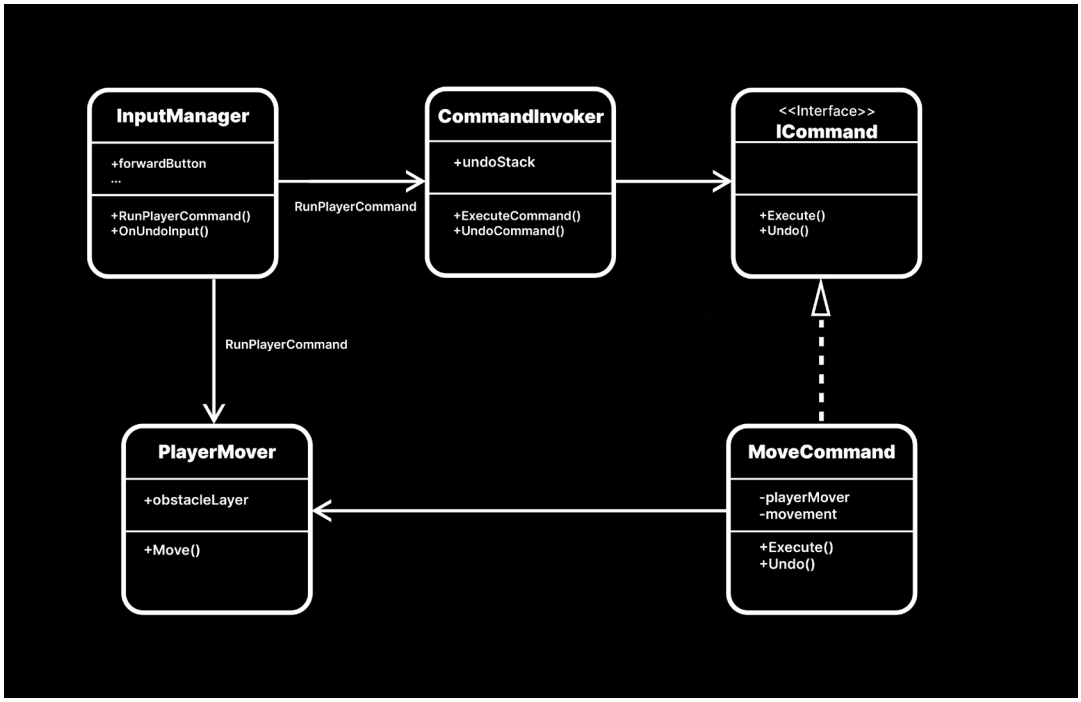
```
public class MoveCommand :ICommand
{
    PlayerMover playerMover;
    Vector3 movement;
    public MoveCommand(PlayerMover player, Vector3 moveVector)
    {
        this.playerMover = player;
        this.movement = moveVector;
    }

    public void Execute()
    {
        playerMover.Move(movement);
    }

    public void Undo()
    {
        playerMover.Move(-movement);
    }
}
```

やろうとしていることを格納するには、ICommand に Execute メソッドが必要になります。実行するロジックが何であってもここに入るため、movement ベクトルを指定して Move を呼び出します。

シーンを前の状態に復元するためには、ICommand に Undo メソッドも必要です。このケースでは、Undo ロジックによって movemenet ベクトルが減算され、必然的にプレイヤーを反対方向に移動させます。

MoveCommand には、実行に必要なすべてのパラメーターが保存されます。これらをコンストラクターで設定します。このケースでは、該当する PlayerMover コンポーネントとその movement ベクトルを保存します。

コマンドオブジェクトを作成して、必要とされるパラメーターを保存したら、CommandInvoker の静的な ExecuteCommand メソッドと UndoCommand メソッドを使用して、MoveCommand を渡します。これにより、MoveCommand の Execute または Undo が実行され、取り消しのスタックでコマンドオブジェクトが追跡されます。



CommandInvoker、ICommand、MoveCommand

InputManager では、PlayerMover の Move メソッドを直接呼び出すことはありません。代わりに、RunMoveCommand メソッドを追加して新しい MoveCommand を作成し、それを CommandInvoker に送信します。

```
private void RunPlayerCommand(PlayerMover playerMover, Vector3 movement)
{
    if (playerMover == null)
    {
        return;
    }

    if (playerMover.IsValidMove(movement))
    {
        ICommand command = new MoveCommand(playerMover, movement);
        CommandInvoker.ExecuteCommand(command);
    }
}
```

その後、UI ボタンのさまざまな onClick イベントを設定し、4 つの movement ベクトルを使用して RunPlayerCommand を呼び出します。

InputManager の実装の詳細についてサンプルプロジェクトを確認するか、キーボードまたはゲームパッドを使用して独自の入力を設定してください。これで、プレイヤーが迷路の中を動き回れるようになりました。Undo (取り消し) ボタンをクリックすると、スタート地点に戻っていくことができます。

長所と短所

やり直し機能や取り消し機能を実装することは、コマンドオブジェクトのコレクションを生成することと同じくらい簡単です。また、コマンドバッファを使用して、特定のコントロールでアクションを順番に再生することもできます。

例えば、一連の特定のボタンクリックにより、コンボムーブやコンボアタックがトリガーされる格闘ゲームがあるとします。コマンドパターンを使用してプレイヤーのアクションを格納することで、そのようなコンボをとても簡単に設定できます。

その一方で、他のデザインパターンと同じように、コマンドパターンを使用すると構造が複雑になってしまいます。アプリケーションにコマンドオブジェクトを展開するにあたって、そうした追加のクラスやインターフェースが十分なメリットをもたらす場面であるか判断する必要があります。

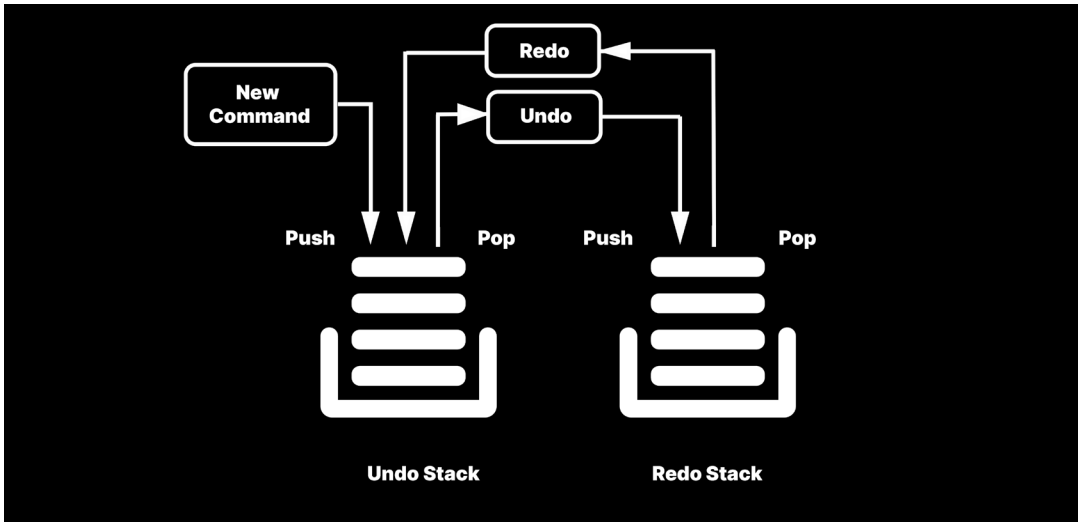
改善点

基本を覚えれば、コマンドのタイミングを調整し、コンテキストに応じて連続して再生したり、逆再生したりできます。

コマンドパターンを組み込む際には、以下のことを考慮してください。

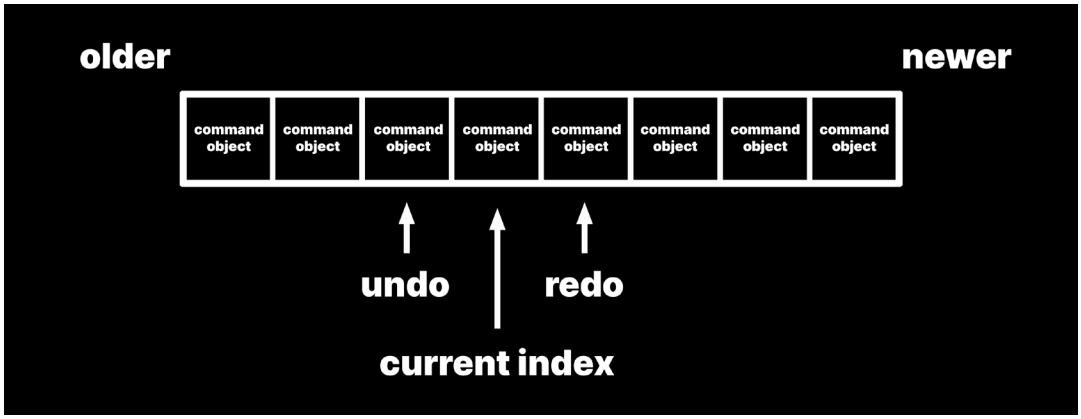
— **より多くのコマンドを作成する:**サンプルプロジェクトに含まれているコマンドオブジェクトは 1 種類 (`MoveCommand`) のみです。`ICommand` を実装するコマンドオブジェクトをいくつでも作成し、`CommandInvoker` を使用してそれらを追跡できます。

— **やり直し機能を追加することは、別のスタックを追加すること:**コマンドオブジェクトを取り消すときは、やり直し操作を追跡するスタックにそれをプッシュします。この方法により、取り消し履歴を素早く繰り返したり、それらの操作をやり直したりできます。ユーザーがまったく新しい動作を呼び出すときには、やり直しスタックを空にします (付属のサンプルプロジェクトで実装を確認できます)。



取り消し (Undo) スタックとやり直し (Redo) スタック

— **コマンドオブジェクトのバッファに別のコレクションを使用する:**先入れ先出し (FIFO) 動作が必要な場合は、キューのほうが便利である可能性があります。リストを使用する場合は、現在アクティブなインデックスを追跡します。アクティブなインデックスより前のコマンドは元に戻すことができます。インデックスより後のコマンドはやり直し可能です。



リストやその他のコレクションがコマンドバッファとして機能する。

— **スタックのサイズを制限する:**取り消し操作ややり直し操作は、すぐにサイズが大きくなり、手に負えなくなってしまうおそれがあります。スタックを直近のコマンド数に制限してください。

— **必要なすべてのパラメーターをコンストラクターに渡す:**これにより、MoveCommand の例で見られるように、ロジックがカプセル化されます。

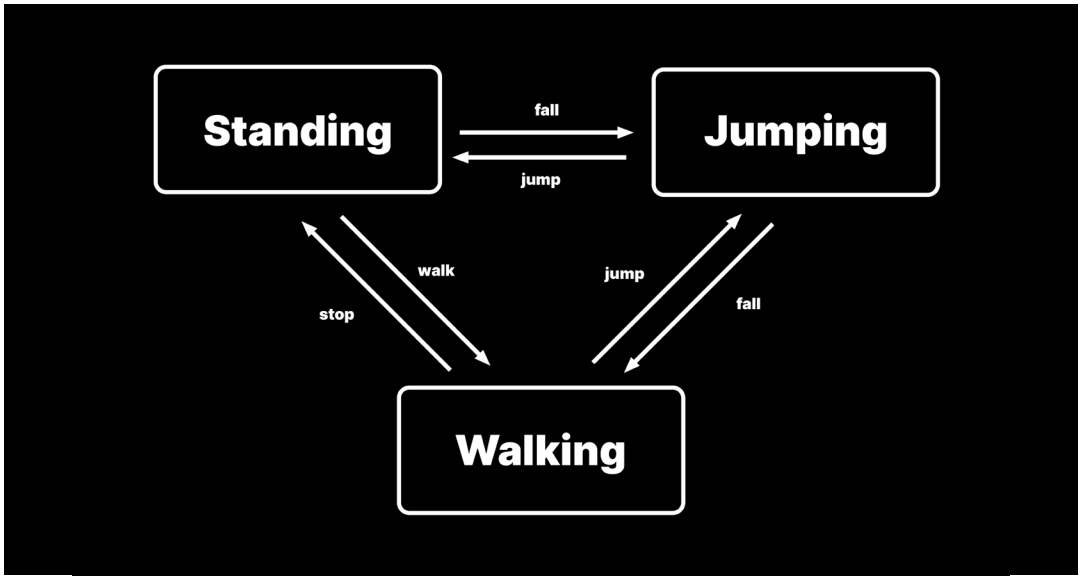
`CommandInvoker` で行われるのは、他の外部オブジェクトと同じように、Execute または Undo を呼び出すことのみで、コマンドオブジェクトの内部構造は見ません。コンストラクターを呼び出すときに、処理する必要があるすべてのデータをコマンドオブジェクトに渡します。

ステート (State) パターン

あるプレイアブルキャラクターを構築するとします。ある瞬間、キャラクターが地面に立っていることがあります。コントローラーを動かすと、歩いたり走ったりします。ジャンプボタンを押すと、キャラクターが空中に跳び上がります。数フレーム後、着地し、静止した立ち位置に戻ります。

ステートとステートマシン

ゲームはインタラクティブであり、ランタイムに変化する数多くのシステムを追跡する必要があります。キャラクターのさまざまなステートを表す ダイアグラム を描くと、次のようになるかもしれません。



シンプルなステートダイアグラム

これは、有限ステートマシン (FSM) と呼ばれるもので、フローチャートと似ていますが、いくつかの違いがあります。

— ダイアグラムはいくつかのステート (静止している/立っている、歩いている、走っている、ジャンプしているなど) で構成されており、ある時点では、1 つのステートだけがアクティブになります。

— 各ステートは、ランタイムの条件に基づいて、別の 1 つのステートへの遷移をトリガーできます。

— 遷移が起こると、遷移先のステートが新しいアクティブなステートになります。

ゲーム開発における FSM の典型的な用途の 1 つは、ゲームのアクターまたは小道具の内部ステートを追跡することです。

基本的なステートマシンをコードで記述するには、`enum` と `switch` ステートメントを使用した素朴なアプローチを使用するとよいでしょう。

```
public enum PlayerControllerState
{
    Idle,
    Walk,
    Jump
}

public class UnrefactoredPlayerController :MonoBehaviour
{
    private PlayerControllerState state;

    private void Update()
    {
        GetInput();
        switch (state)
        {
            case PlayerControllerState.Idle:
                Idle();
                break;
            case PlayerControllerState.Walk:
                Walk();
                break;
            case PlayerControllerState.Jump:
                Jump();
                break;
        }
    }
    private void GetInput()
    {
```

```
            // 歩くコントロールとジャンプのコントロールを処理する
        }
        private void Walk()
        {
            // 歩くロジック
        }
        private void Idle()
        {
            // 静止のロジック
        }
        private void Jump()
        {
            // ジャンプのロジック
        }
}
```

これでも機能しますが、PlayerController スクリプトはすぐに収拾がつかなくなってしまいます。ステート数と複雑さを追加すると、クラスが膨れあがることになります。また、変更を加えるたびに、PlayerController スクリプトの内部を見直す必要もあります。

SOLID の原則に従い、クラスをより短く、焦点をより絞ったものにしましょう。変更に対しては閉じ、拡張に対しては開くことで、スケーラビリティと管理性が向上します。

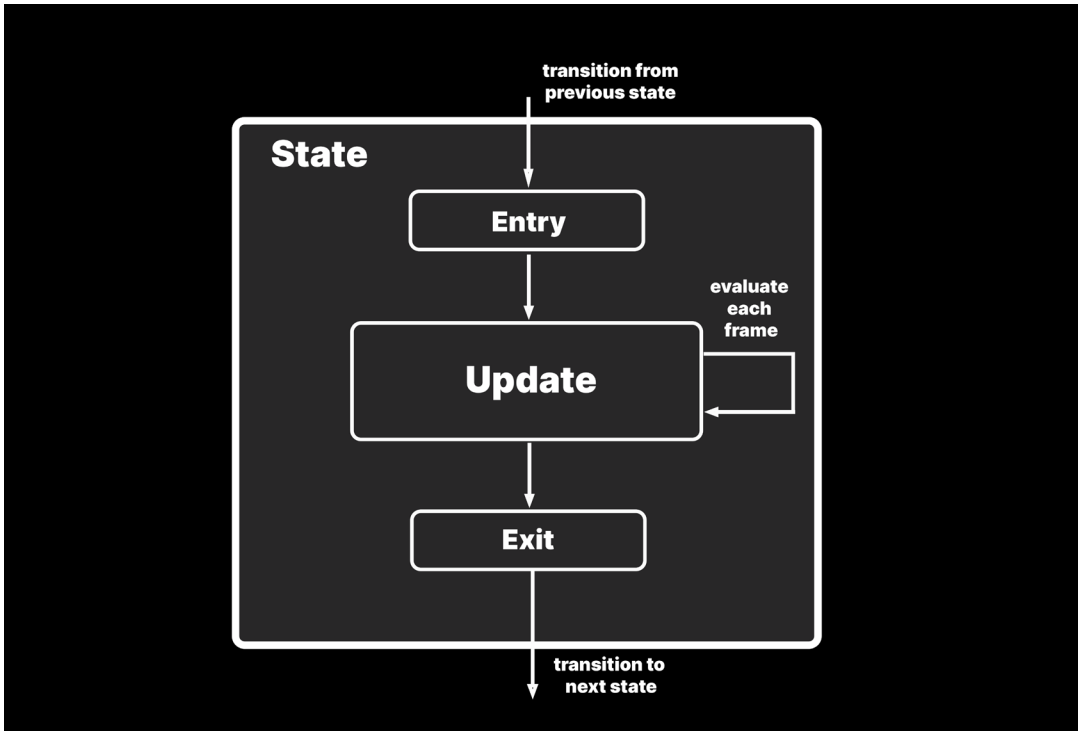
例：シンプルなステートパターン

幸いなことに、ステートパターン はロジックの再整理に役立ちます。オリジナルの Gang of Four によると、ステートパターンは次の 2 つの問題を解決します。

- オブジェクトは、内部ステートの変化に伴い動作を変更する必要があります。
- ステート固有の動作は独立して定義されます。新しいステートを追加しても、既存のステートの動作には影響しません。

上記の例の `UnrefactoredPlayerController` クラスはステートの変化を追跡できますが、2 つ目の問題は解消されません。新しいステートを追加するときに、既存のステートへの影響を最小限に抑える必要があります。1 つのステートを 1 つのオブジェクトとしてカプセル化することで解決できます。

各ステートを次のように構成するとします。



カプセル化されたステートに、Entry、Exit、Execute の機能が備わっている

ここでは、ステートに入り、条件によってコントロールフローが終了するまで各フレームをループします。このパターンを実装するために、インターフェース `IState` を作成します。

```
public interface IState
{

    public void Enter()
    {
        // そのステートに最初に入ると実行されるコード
    }

    public void Execute()
    {
        // フレームごとのロジック、新しいステートに遷移する条件が含まれる
    }

    public void Exit()
    {
        // そのステートを出ると実行されるコード
    }
}
```

ゲーム内の具体的なステートにより、IState インターフェースが実装されます。

— **Entry (1 回):**このロジックはそのステートに最初に入るときに実行されます。

— **Execute:**このロジックは毎フレーム実行されます (Tick または Update と呼ばれることがあります)。MonoBehaviour と同じように、Update、FixedUpdate、LateUpdate などで Execute メソッドをさらにセグメント化することもできます。

Execute 内の機能は、各フレームでステートの変更をトリガーする条件が検出されるまで実行されます。

— **Exit (1 回):**このコードは、ステートを離れ、新しいステートに遷移するときに実行されます。

IState を実装する各ステートに 1 つのクラスを作成する必要があります。このサンプルプロジェクトには、WalkState、IdleState、JumpState に別個のクラスが設定されています。

その後、別のクラス (StateMachine) によって、コントロールフローがそのステートに入る方法と出る方法が管理されます。この 3 つのサンプルステートにより、StateMachine　は次のようになります。

```
 [Serializable]
public class StateMachine
{
    public IState CurrentState { get; private set; }

    public WalkState walkState;
    public JumpState jumpState;
    public IdleState idleState;

    public void Initialize(IState startingState)
    {
        CurrentState = startingState;
        startingState.Enter();
    }

    public void TransitionTo(IState nextState)
    {
        CurrentState.Exit();
        CurrentState = nextState;
        nextState.Enter();
    }

    public void Execute()
    {
        if (CurrentState != null)
        {
            CurrentState.Execute();
        }
    }
```

パターンに従うために、StateMachine ではその管理下にある各ステート (このケースでは `walkState`、`jumpState`、`idleState`) の public オブジェクトを参照します。StateMachine は MonoBehaviour を継承しないため、各インスタンスを設定するためにコンストラクターを使用します。

```
public StateMachine(PlayerController player)
{
    this.walkState = new WalkState(player);
    this.jumpState = new JumpState(player);
    this.idleState = new IdleState(player);
}
```

必要とされるパラメーターをコンストラクターに渡すことができます。このサンプルプロジェクトでは、各ステートで PlayerController が参照されています。その後、それを使用して各ステートをフレームごとに更新します (後述の IdleState の例を参照)。

StateMachine については、以下の点に注意してください。

— Serializable 属性を使用すると、StateMachine (とその public フィールド) を Inspector に表示できます。これにより、別の MonoBehaviour (PlayerController や EnemyController など) でその StateMachine をフィールドとして使用できます。

— `CurrentState` プロパティは読み取り専用です。StateMachine 自体はこのフィールドを明示的に設定しません。これにより、PlayerController のような外部オブジェクトで `Initialize` メソッドを呼び出してデフォルトの State を設定できます。

— 各 State オブジェクトは、現在のアクティブなステートを変更するために `TransitionTo` メソッドを呼び出す条件を自ら決定します。StateMachine インスタンスの設定中に、各ステートに必要な依存関係 (StateMachine 自体を含む) を渡すことができます。

  このサンプルプロジェクトでは、PlayerController に StateMachine への参照がすでに含まれているため、1 つの `player` パラメーターのみを渡します。

内部ロジックは各ステートオブジェクトでそれぞれ管理され、ゲームオブジェクトやコンポーネントを記述するのに必要な数だけステートを作成することができます。各ステートが、`IState` を実装する独自のクラスを持ちます。SOLID の原則に沿い、ステートをさらに追加しても、以前作成したステートに及ぼす影響が最小限に抑えられます。

こちらが IdleState の例です。

```
public class IdleState :IState
{
    private PlayerController player;

    public IdleState(PlayerController player)
    {
        this.player = player;
    }

    public void Enter()
    {
        // そのステートに最初に入ると実行されるコード
    }

    public void Execute()
    {
        // ここでは別のステートに遷移する条件が存在するかどうかを
        // 検出するロジックを追加する
         ...
    }

    public void Exit()
    {
        // そのステートを出ると実行されるコード
    }
}
```

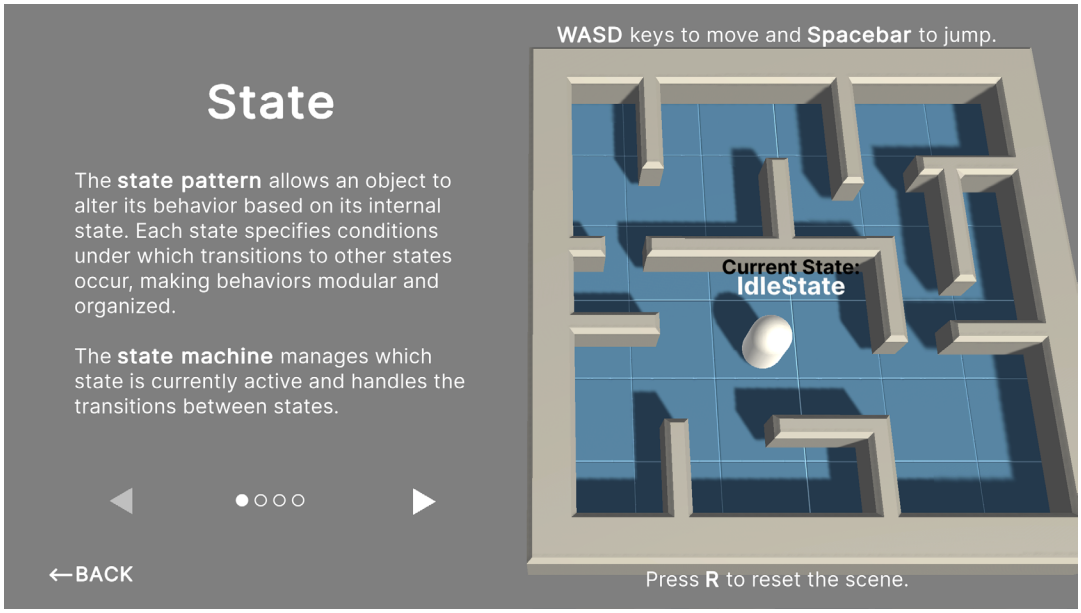
繰り返しになりますが、コンストラクターを使用して PlayerController オブジェクトを渡します。この例では、この `player` に StateMachine への参照と、Update ロジックに必要なその他すべてのものが含まれています。`idleState` によって、Character Controller の速度やジャンプステートが監視され、その後適宜 `StateMachine` の `TransitionTo` メソッドが呼び出されます。

`WalkState` と `JumpState` の実装についても、サンプルプロジェクトで確認してください。動作を切り替える 1 つの大きなクラスがあるのではなく、各ステートに独自の Update ロジックがあります。こうすることで、ステートが独立して機能するようになります。

長所と短所

ステートパターンは、オブジェクトに内部ロジックを設定する際に、SOLID の原則を守るのに役立ちます。各ステートは比較的小さく、別のステートへの遷移条件のみを追跡します。オープン/クローズドの原則に従って、手間のかかる `switch` ステートメントや `if` ステートメントを使用せず、既存のステートに影響を及ぼすことなくステートを追加できます。

一方、追跡するステートが少ない場合は、余分な構造が過剰になるおそれがあります。このパターンが適しているのは、ステートがある程度複雑になることが予想される場合に限られるかもしれません。



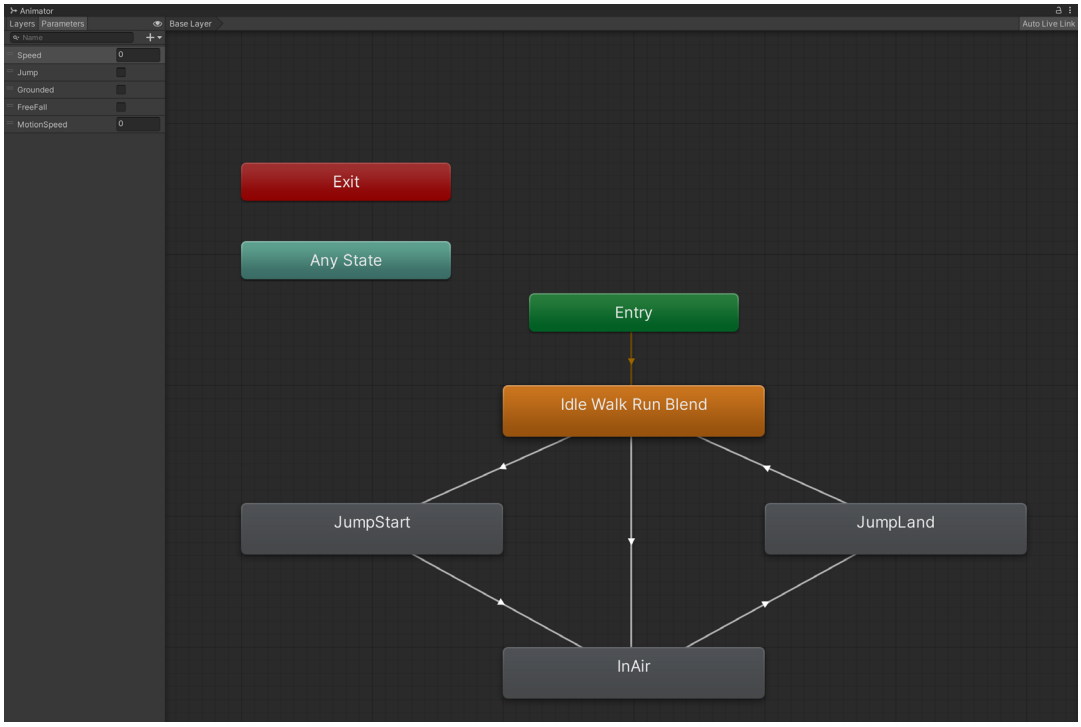
ステートパターンは、オブジェクトの内部ステートを追跡する。

改善点

このサンプルプロジェクトのカプセルは色が変化し、プレイヤーの内部ステートによって UI が更新されます。実際の例では、ステートの変化に伴うさらに複雑なエフェクトを作成することもできます。

— **ステートパターンをアニメーションと組み合わせる:**ステートパターンを適用する一般的な場面の 1 つとして、アニメーションが挙げられます。プレイヤーや敵キャラクターは多くの場合、マクロレベルでプリミティブ (カプセル) として表現されます。その後、内部ステートの変化に反応するアニメーション化されたジオメトリを用意することで、ゲームアクターが走ったり、ジャンプしたり、泳いだり、登ったりしているように見せることができます。

Unity の Animator ウィンドウを使用したことがある方であれば、そのワークフローがステートパターンに適していることに気付くと思います。各アニメーションクリップで 1 つのステートが使用され、一度に 1 つのステートのみがアクティブになります。



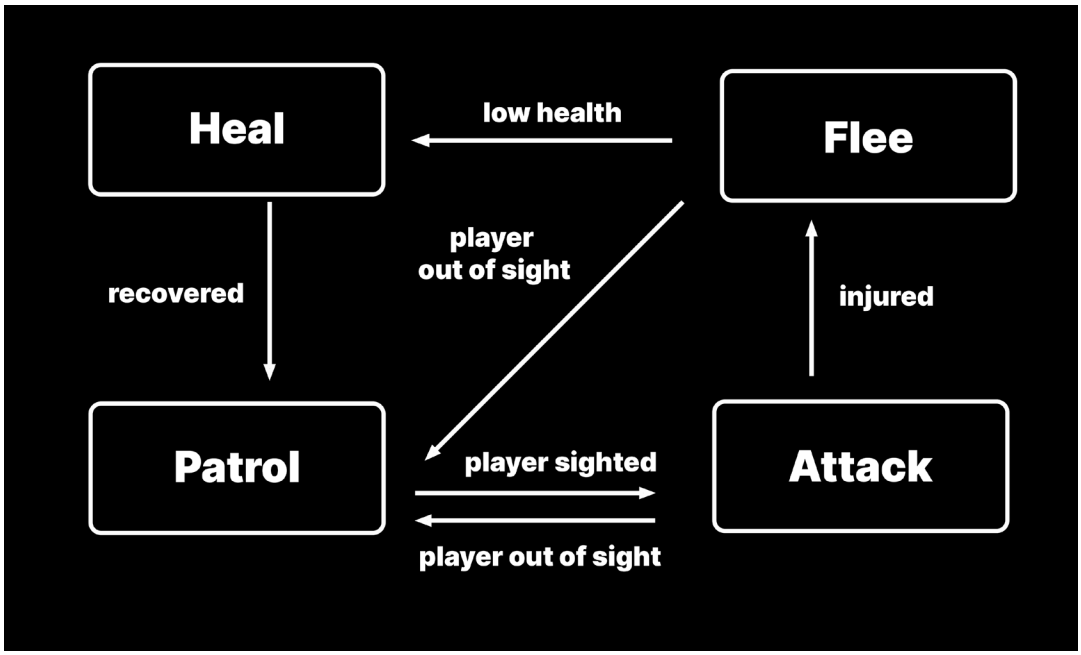
Animator のステートグラフの例。その構造を StateMachine と比較してみよう。

— **イベントを追加する:**ステートの変化を外部のオブジェクトに伝えるには、イベントを追加するとよいでしょう (オブザーバー (Observer) パターンを参照)。ステートに入る、または出るためのイベントを用意することで、関連するリスナーに通知し、ランタイムに応答させることができます。

— **階層を追加する:**ステートパターンを使用してより複雑なエンティティの記述を始めるにあたっては、階層化されたステートマシンを実装すると良いでしょう。一部のステートは似たものになります。例えば、プレイヤーやゲームアクターが地面に立っている場合は、WalkingState でも RunningState でも屈んだりジャンプしたりできます。

  SuperState を実装すれば、共通の動作を 1 つにまとめることができます。そうすることで、継承を使用して、具体的な何かをサブステート内でオーバーライドできます。例えば、最初に GroundedState を宣言するとします。その後、そこから RunningState または WalkingState を継承できます。

— **シンプルな AI を実装する:**有限ステートマシンは、敵の基本的な AI の生成にも便利です。NPC の知能を構築する FSM アプローチは、次のようになります。



ステートパターンをベースとするシンプルな AI

こちらは、まったく別のコンテキストで機能するもう 1 つのステートパターンです。各ステートが、攻撃 (Attack)、逃走 (Flee)、偵察 (Patrol) などのアクションを表します。一度に 1 つのステートのみがアクティブになり、各ステートにより次のステートへの遷移が決まります。

例:ゲームの状態

前の例では、プレイヤーが移動したり、ジャンプしたり、静止したりすると、キャラクターのマテリアルの色と UI ラベルが更新されます。オブジェクトの内部状態を追跡する必要があるところに、ステートパターンを適用します。キャラクターアニメーションはその代表的な例であり、Unity の AnimatorController に ビルトインステートマシン が含まれているほどです。

サンプルプロジェクトには、ステートパターンのもう 1 つの実用的な適用例として、ゲームステートを維持する、より高度なステートマシンが含まれています。デモ自体は、このステートマシンを使用してランタイムの動作を管理します。

Scripts/StateMachine フォルダー内には、より洗練されたステートマシンを構築およびカスタマイズするためのコンポーネントがいくつかあります。

— `StateMachine` はオブジェクトの現在の状態を追跡し、異なるステート間の遷移を処理します。各ステートのライフサイクルメソッドを実行し、ループで状態の変化を監視します。

— `IState` インターフェースは、各ステートオブジェクト (`Enter`、`Execute`、`Exit` などのライフサイクルメソッド、および他のステートへの遷移) の標準化された機能を定義します。

— `AbstractState` は `IState` インターフェースを実装し、すべてのステートの基本クラスとして機能します。

ステートパターンを設定するには、次のように具体的なステートを定義します。

— 汎用の `State` クラスは、ステートに入ったときおよび実行時に、事前定義されたアクションを実行できます。

— `DelayState` には、次のステートに遷移するまでの待機期間が導入されており、進捗バーやロード画面に便利です。

— `LoadSceneState` と `UnloadLastSceneState` は、シーンの遷移を管理するためのステートクラスです。これらのステートではシーンを順次ロードまたはアンロードできるため、プロジェクトのコンテンツを個々の Unity シーンに分割できます。

他のステートに遷移するには、特定の条件やイベントに応答するロジックを実装します。これにより、ゲームイベントやユーザー入力によるステートの変更が可能になります。

— `ILink` インターフェースは、ステート間の遷移を定義します。

— 特定の C# イベントに基づいて遷移をトリガーする `EventLink` を実装します。

— 特定の ScriptableObject ベースのイベントに基づいて遷移をトリガーする `EventSOLink` を実装します。

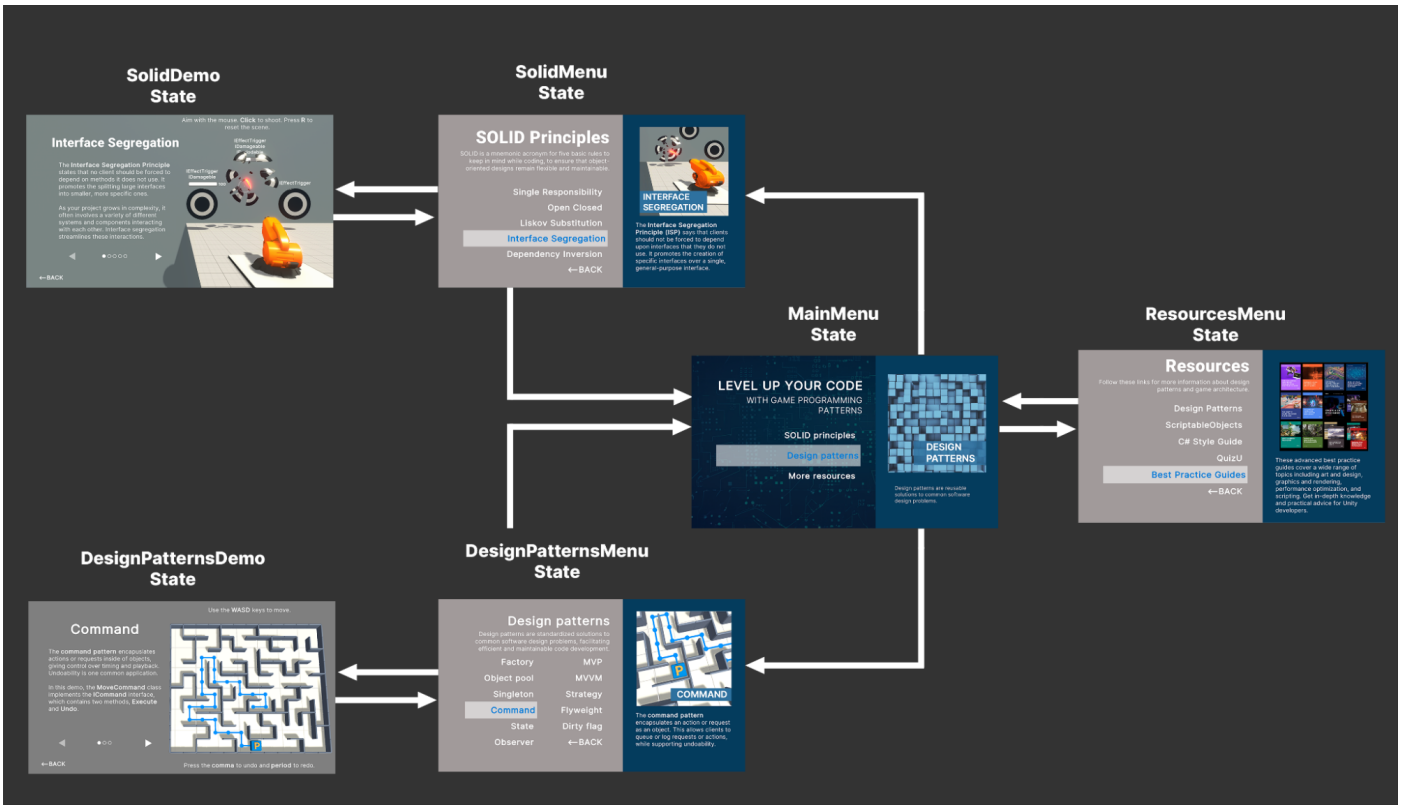
— 特定のシーン読み込みイベントに基づいて遷移をトリガーする `SceneEventSOLink` を実装します。

ステートマシンは複数のイベントチャンネル (カスタム C# イベントと ScriptableObject ベースのイベントの両方を使用) により、アプリケーション内の他のシステムと通信します。

これらすべてを組み合わせれば、さまざまな種類のアプリケーションで動作するステートマシンを構築できます。プロジェクトの必要に応じて、追加のステートや遷移を作成するだけです。

サンプルプロジェクトでは、`GameManager` はこのステートマシンを使用してアプリケーションの一般的なフローを操作します。システムは ScriptableObject ベースのイベントを使用して、メニュー UI からデモコンテンツに遷移します。

ユーザーインタラクション (ボタンのクリックなど) は、GameManager に内部ステートを変更するよう通知します。すると、このステートダイアグラムに従って UI が更新されます。



GameManager のステートダイアグラム。

この例では UI の更新に焦点を当てていますが、GameManager のステートはアプリケーションの特定のニーズに合わせてカスタマイズできます。

ここでステートパターンを使用すると、アプリケーションを小さなパーツから簡単に組み立てることができます。各メニューボタンはイベントを発生させます。そのイベントは、次に、新しいステートへの遷移をトリガーし、対応するデモコンテンツをロードします。

新機能の導入は、新しいステートを追加し、必要な遷移を設定するのと同じくらい簡単です。SOLID に沿って、アプリケーションに新しいパーツを構築しても、既存のプロジェクトには影響しません。

QuizU プロジェクトを探る

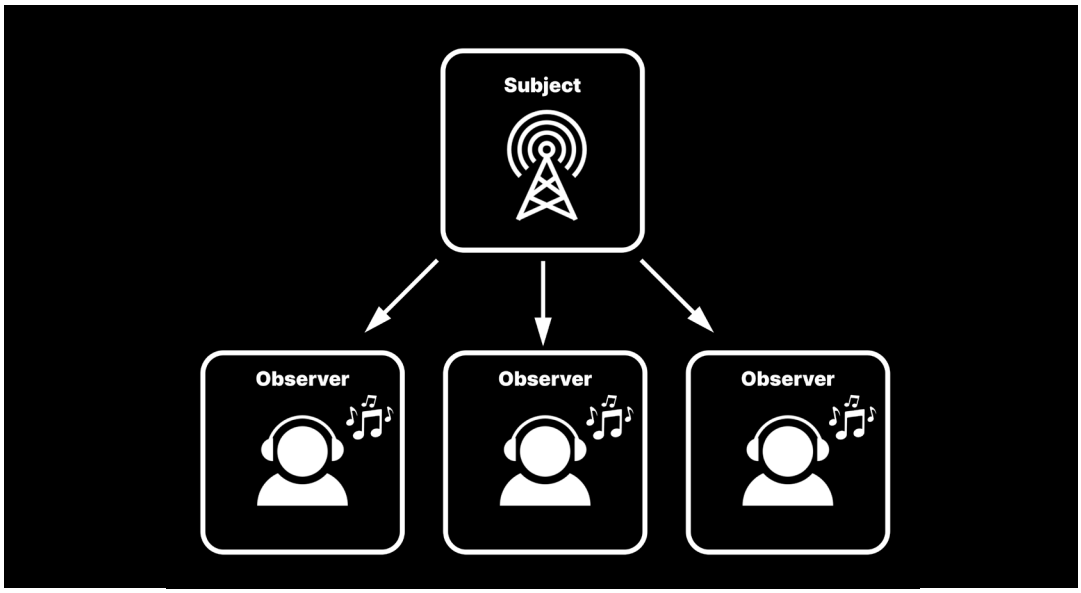
実際のデザインパターンをもっと見たいですか?QuizU サンプルプロジェクトでは、UI Toolkit を使用して構築されたメインメニューでの MVP とステートパターンの使用例も紹介しています。このプロジェクトでは、メインのゲームループにもこのステートマシンのバリエーションを使用しています。このプロジェクトは、Unity Discussions の関連シリーズ記事 と一緒に読むことで詳しく見ることができます。

QuizU プロジェクトでは、ゲームの状態の管理にステートパターンを使用している。

オブザーバー (Observer) パターン

ランタイムには、ゲーム内でさまざまなことが発生します。敵を倒したら何が起こるでしょうか?パワーアップを集めたり、目標をクリアしたらどうなるでしょうか?多くの場合、不要な依存関係が作成されることがないように、オブジェクトが他のオブジェクトを直接参照することなく通知できるようにするメカニズムが必要です。

オブザーバー (Observer) パターンは、このような問題を解決する一般的なソリューションです。"1 対多" の依存関係を使用して疎の結合を保ちながら、オブジェクト間で情報のやりとりができるようになります。あるオブジェクトのステートが変化すると、それに依存するすべてのオブジェクトに自動的に通知されます。これは、多数のリスナーに向けて電波を発信する電波塔に似ています。



オブザーバーパターンは電波塔のように機能する。サブジェクトからオブザーバーに発信される。

発信しているオブジェクトのことを、サブジェクト (主体) と呼び、それを聞いているその他のオブジェクトを、オブザーバー (観察者) と呼びます。

オブザーバーパターンは、サブジェクトをゆるく分離します。サブジェクトにはオブザーバーに関する情報がなく、シグナルを受け取った後にそこで何が行われるかについては関知しません。オブザーバーにはサブジェクトに対する依存関係がある一方で、オブザーバー同士は互いのことを知りません。

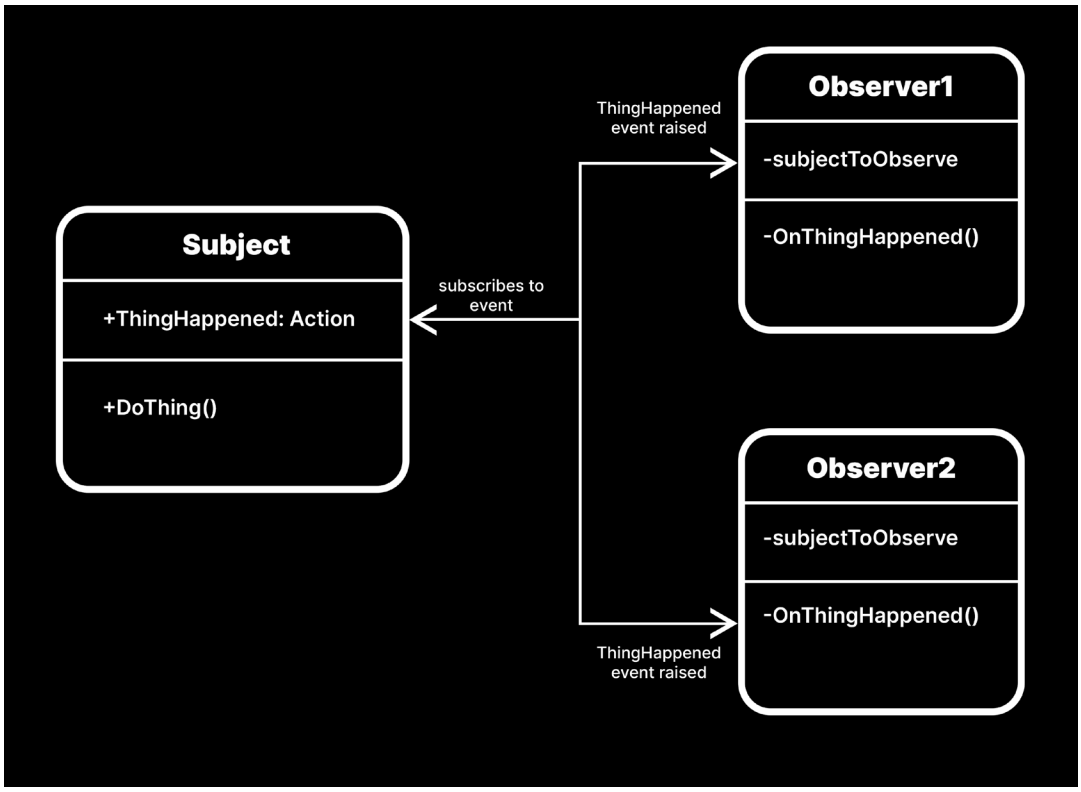
イベント

オブザーバーパターンは広く普及しており、C# 言語にも組み込まれています。サブジェクト - オブザーバークラスを独自にデザインすることもできますが、通常は不要です。すでにあるものをゼロから作成しないようにしましょう。C# には、イベントを使用したパターンがすでに実装されています。

イベントは、単に何かが発生したことを示す通知です。これにはいくつかのパーツが関わっています。

- パブリッシャー (サブジェクト) が、デリゲートに基づいてイベントを作成し、特定の関数シグネチャを確立します。イベントは、サブジェクトでランタイム時に実行されるアクション (ダメージを受ける、ボタンをクリックするなど) にすぎません。
- その後、サブスクライバー (オブザーバー) はそれぞれイベントハンドラーと呼ばれるメソッドを作成します。これはデリゲートのシグネチャと一致する必要があります。
- 各オブザーバーのイベントハンドラーが、パブリッシャーのイベントをサブスクライブします。必要に応じて、任意の数のオブザーバーをサブスクリプションに結合させることができます。それらのオブザーバーはすべて、イベントがトリガーされるのを待ちます。
- ランタイムにパブリッシャーからイベントが起こったシグナルが送信されることを、イベント発生と言います。これにより、サブスクライバーのイベントハンドラーが呼び出され、それに応じて独自の内部ロジックが実行されます。

このようにして、多くのコンポーネントをサブジェクトからの 1 つのイベントに反応するよう設定します。ボタンがクリックされたことがサブジェクトにより示されると、オブザーバーは、アニメーションやサウンドの再生、カットシーンのトリガー、ファイルの保存を行うことができます。応答は何であってもよく、そのことが、オブジェクト間でメッセージを送信するためにオブザーバーパターンがよく使用される理由です。



サブジェクトがイベントを発生させて、オブザーバーに通知する。

例：シンプルなサブジェクトとオブザーバー

例えば、次のように基本的なサブジェクト/パブリッシャーを定義できます。

```
using System;

public class Subject:MonoBehaviour
{
    public event Action ThingHappened;

    public void DoThing()
    {
        ThingHappened?.Invoke();
    }
}
```

ここでは、ゲームオブジェクトに簡単にアタッチするために MonoBehaviour を継承しますが、それは必須ではありません。

独自のカスタムデリゲートを自由に定義できますが、ほとんどのケースで System.Action が有効です。イベントを使用してパラメーターを送信する必要がある場合は、Action<T> デリゲートを使用して、山かっこで囲んで List<T> として渡します (最大 16 個のパラメーター)。

ThingHappened が実際のイベントで、サブジェクトが DoThing メソッドで呼び出します。

イベントをリッスンするために、サンプルの Observer クラスを構築できます。ここでは便宜上 MonoBehaviour を継承しますが、必須ではありません。

```
public class Observer :MonoBehaviour
{
    [SerializeField] private Subject subjectToObserve;
    private void OnThingHappened()
    {
        // イベントに応答する任意のロジックがここに入る
        Debug.Log(“Observer responds”);
    }

    private void OnEnable()
    {
        if (subjectToObserve != null)
        {
            subjectToObserve.ThingHappened += OnThingHappened;
        }
    }

    private void OnDisable()
    {
        if (subjectToObserve != null)
        {
            subjectToObserve.ThingHappened -= OnThingHappened;
        }
    }
}
```

このコンポーネントをゲームオブジェクトにアタッチし、subjectToObserver を Inspector で参照して、ThingHappened イベントをリッスンします。

OnThingHappened メソッドは、イベントに応答してオブザーバーが実行する任意のロジックを持つことができます。多くの場合、開発者はイベントハンドラーであることを示すために "On" というプレフィックスを付けます (お使いのスタイルガイドの命名規則に従ってください)。

Awake または Start では、+= 演算子を使用してそのイベントをサブスクライブできます。これにより、オブザーバーの OnThingHappened メソッドとサブジェクトの ThingHappened が結合されます。

何かによってサブジェクトの DoThing メソッドを実行されると、イベントが発生します。これにより、オブザーバーの OnThingHappened イベントハンドラーが自動的に呼び出され、デバッグステートメントが出力されます。

注：ThingHappened をサブスクライブした状態で、オブザーバーをランタイムに削除または解除した場合、そのイベントを呼び出すことによりエラーが発生する可能性があります。そのため、MonoBehaviour の OnDestroy メソッドで -= 演算子を使用してイベントのサブスクライブを解除することが重要です。

MonoBehaviour の OnDestroy メソッドで -= 演算子を使用してイベントのサブスクライブを解除することは非常に重要です。Unity オブジェクトのライフサイクル全体を通してイベントのサブスクリプションを管理することで、メモリリークや null 参照を回避し、コードをクリーンな状態に保ちます。

命名規則

オブザーバーパターンの各パーツに統一された命名規則はありません。ご自身のスタイルガイドで、これらのパーツの命名方法を明確にしてください。

- **イベント:**実際のアクションまたはシグナルです。この例では、イベントを ThingHappened と呼んでいます。
- **イベントハンドラー:**イベントに応答して発生するロジックです。この例では、イベントハンドラーに "On" というプレフィックスを付けています。イベントハンドラー OnThingHappened は、ThingHappened イベントに応答して実行されます。
- **イベント発生メソッド:**イベントを呼び出すメソッドです。この例では、DoThing がイベント発生メソッドです。

イベントには、アクションや発生を示す説明的な動詞が付けられることが多いです（DoorOpened、DamageReceived など）。イベント、イベントのトリガー、イベントへの応答に命名規則を使用すると、それらの関係をより明確にすることができます。

チーム独自の C# スタイルガイドの作成の詳細については、e ブック *Create a C# style guide:Write cleaner code that scales (C# スタイルガイドの作成: スケーラブルでクリーンなコードの記述)* を参照してください。

オブザーバーパターンは、ゲームプレイ中に発生するほぼすべてのことに適用できます。例えば、ゲームでプレイヤーが敵を倒したりアイテムを拾ったりするたびにイベントを発生させることができます。スコアや達成度を追跡する統計システムが必要な場合、オブザーバーパターンを使用すれば、元のゲームプレイのコードに影響を与えずに作成できます。

Unity のアプリケーションの多くは、以下のことに対してイベントを適用します。

— 中短期的な目標や最終目標

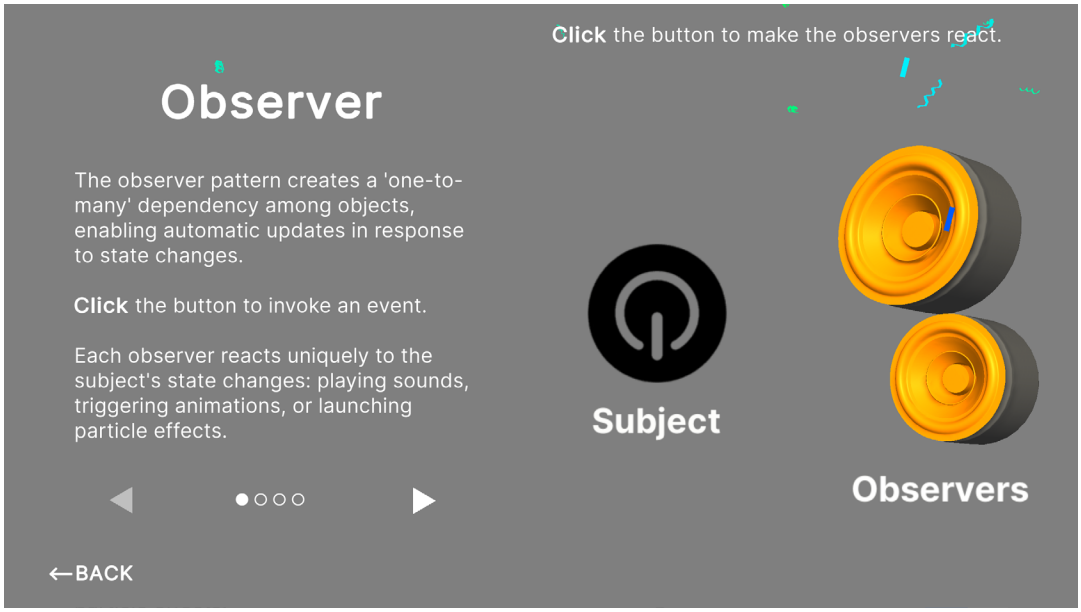
— 勝敗条件

— PlayerDeath、EnemyDeath、Damage

— アイテムの収集

— ユーザーインターフェース

サブジェクトに必要なのは適切なタイミングでイベントを発生させることだけで、サブスクライブできるオブザーバーの数に制限はありません。



オブザーバーのサンプルシーン

このサンプルプロジェクトでは、`ButtonSubject` によりユーザーがマウスのボタンを使用して `Clicked` イベントを呼び出すことができます。`AudioObserver` コンポーネントと `ParticleSystemObserver` コンポーネントを持つその他の複数のゲームオブジェクトが、独自の方法でイベントに応答できるようになります。

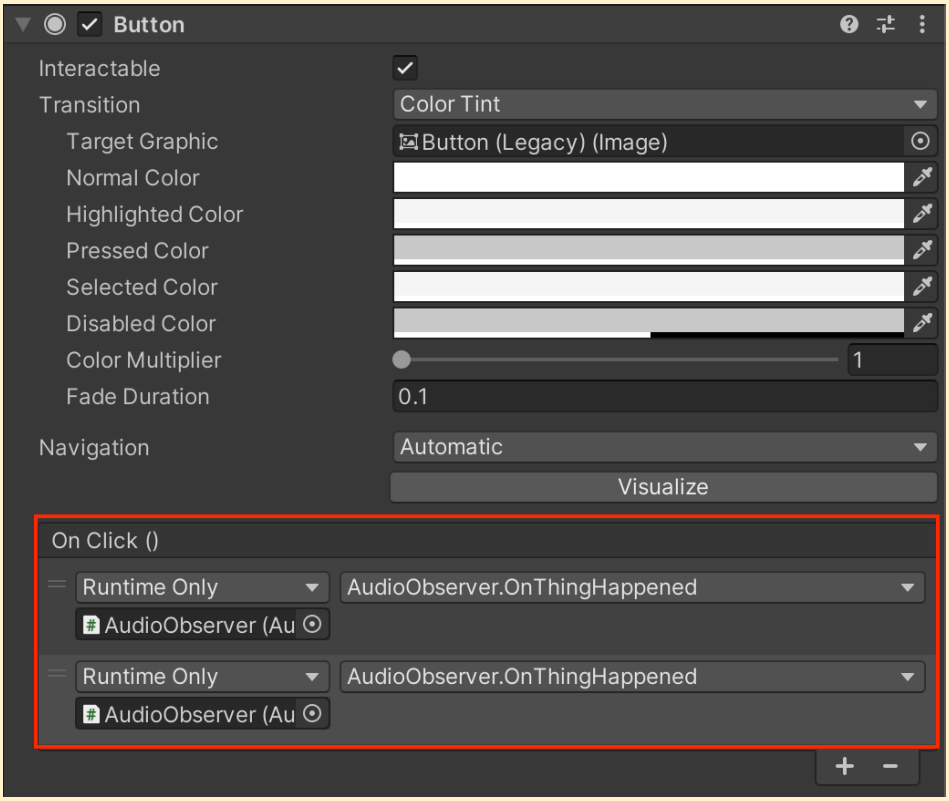
どのオブジェクトが "サブジェクト" で、どれが "オブザーバー" であるかは、用途によってのみ決まります。イベントを発生させるオブジェクトはすべてサブジェクトとして機能し、イベントに応答するオブジェクトはすべてオブザーバーになります。同じゲームオブジェクトの異なるコンポーネントがサブジェクトになることもオブザーバーになることもあります。同じコンポーネントでも、あるコンテキストではサブジェクトになり、別のコンテキストではオブザーバーになることがあります。

例えば、この例にある `AnimObserver` によって、クリックされたときにボタンに少しの動きが追加されます。ButtonSubject ゲームオブジェクトの一部でありながら、オブザーバーとして機能します。

UnityEvents と UnityActions

Unity には、**UnityEvents** という別個のシステムがあり、**UnityEngine.Events** API から **UnityAction** デリゲートを使用します。

UnityEvents には、オブザーバーパターン用のグラフィカルインターフェースが備わっています。これらは、追加のコードを必要とせずに、迅速なプロトタイピングや、インタラクションの設定を行うことができる、アーティストにとって使いやすいアプローチです。Unity の UI システム (UI Button の OnClick イベントの作成など) を使用したことがある方であれば、すでに経験があることになります。



UnityEvents には、ユーザーが設定するためのグラフィカルコンポーネントが備わっている。

この例では、ボタンの OnClick イベントにより AudioObserver の 2 つの OnThingHappened メソッドからの応答が呼び出されてトリガーされます。このようにして、サブジェクトのイベントをコードなしで設定できます。

UnityEvents は、デザイナーや、プログラマーではないユーザーが、ゲームプレイのイベントを作成できるようになる便利な機能です。ただし、System 名前空間からの同等のイベントやアクションよりも低速になるおそれがあることに注意してください。

UnityEvents や UnityActions を使用する際には、パフォーマンスと使用量を比較検討してください。Unity Learn の Create a Simple Messaging System with Events (イベントを使用して簡単なメッセージングシステムを作成する) モジュールで例を確認してください。

長所と短所

イベントを実装することには追加の作業が伴いますが、以下のような利点があります。

- **オブザーバーパターンは、オブジェクト同士を切り離すのに役立つ:**イベントパブリッシャーに、そのイベントサブスクライバー自体に関する情報は一切不要です。クラス間で直接的な依存関係を作成する代わりに、ある程度切り離された状態を保ちつつ、サブジェクトとオブザーバー間でコミュニケーションが取られます。
- **構築する必要がない:**C# には確立されたイベントシステムが備わっており、独自のデリゲートを定義する代わりに System.Action デリゲートを使用できます。また、Unity には UnityEvents と UnityActions も備わっています。
- **各オブザーバーに独自のイベントハンドリングロジックが実装される:**この方法により、オブザーバー側の各オブジェクトに、応答に必要とされるロジックが保持されます。これにより、デバッグと単体テストが簡単になります。
- **ユーザーインターフェースに適している:**ゲームプレイの主要なコードと UI ロジックを切り離しておくことができます。UI 要素が具体的なゲームイベントや条件をリッスンし、適切に応答できるようになります。MVP パターンや MVC パターンでは、この目的のためにオブザーバーパターンを使用します。

オブザーバーパターンでは、次の点に注意してください。

- **複雑さが増す:**他のパターンと同様に、イベント駆動型アーキテクチャを作成するには、事前により多くのことを設定する必要があります。また、サブジェクトやオブザーバーを削除する際にも注意してください。必ず OnDestroy でオブザーバーの登録を解除してください。
- **オブザーバーにはイベントを定義するクラスへの参照が必要:**オブザーバーには引き続き、イベントを公開するクラスに対する依存関係が存在します。すべてのイベントを処理する静的な EventManager (後述) を使用すると、オブジェクト間の関連を整理するのに役立つ場合があります。
- **パフォーマンスが犠牲になるおそれがある:**イベント駆動型アーキテクチャを使用すると、余計なオーバーヘッドが発生します。大規模シーンや多数のゲームオブジェクトが、パフォーマンスを低下させるおそれがあります。

改善点

ここでは基本的なオブザーバーパターンのみを紹介していますが、ゲームアプリケーションのあらゆるニーズに対応するよう拡張できます。

オブザーバーパターンを設定する際には、以下の提案を考慮に入れてみてください。

- **ObservableCollection クラスを使用する:**C# には、具体的な変更を追跡することを担う、動的な ObservableCollection が用意されています。アイテムが追加または削除されたとき、あるいはリストが更新されたときにオブザーバーに通知できます。
- **一意のインスタンス ID を引数として渡す:**階層内の各ゲームオブジェクトには一意の インスタンス ID があります。複数のオブザーバーに適用される可能性のあるイベントをトリガーする場合は、そのイベントに一意の ID を (Action<int> 型を使用) 渡します。その後、ゲームオブジェクトが一意の ID に一致する場合に、イベントハンドラーでそのロジックのみを実行します。

— **静的な EventManager を作成する:**ゲームプレイのかなりの部分がイベント駆動型であるため、Unity アプリケーションの多くでは静的またはシングルトンの EventManager を使用します。この方法により、オブザーバーでゲームイベントの中心的なソースをサブジェクトとして参照できるようになるため、設定がより簡単になります。

FPS Microgame では、静的な EventManager がうまく実装されています。カスタム GameEvents が実装されており、リスナーを追加または削除することを担う静的なヘルパーメソッドが含まれています。

また、Unity Open Project にて、スクリプタブルオブジェクトを使ったゲームアーキテクチャ を紹介しています。オーディオの再生や新しいシーンのロードにイベントを使用します。

— **イベントキューを作成する:**シーン内に大量のオブジェクトがある場合は、一度にすべてのイベントを発生させない方がよいでしょう。1 つのイベントを呼び出すと何千ものオブジェクトから音が鳴る。それがどれほど耳障りであるか想像してみてください。

オブザーバーパターンをコマンドパターンと組み合わせることで、複数のイベントを 1 つのイベントキューにカプセル化できます。これにより、コマンドバッファを使用してイベントを一度に 1 つずつ再生するか、必要に応じて選択的に無視できます (一斉に音を発することのできるオブジェクトの最大数を設定する場合など)。

オブザーバーパターンは、モデルビュープレゼンター (MVP) アーキテクチャパターンに深く関わっています。MVP パターンについては次の章で詳しく説明します。

モデルビュープレゼンター (MVP)

モデルビューコントローラー（MVC）は、ユーザーインターフェースの開発時によく使用されるデザインパターンのファミリーです。

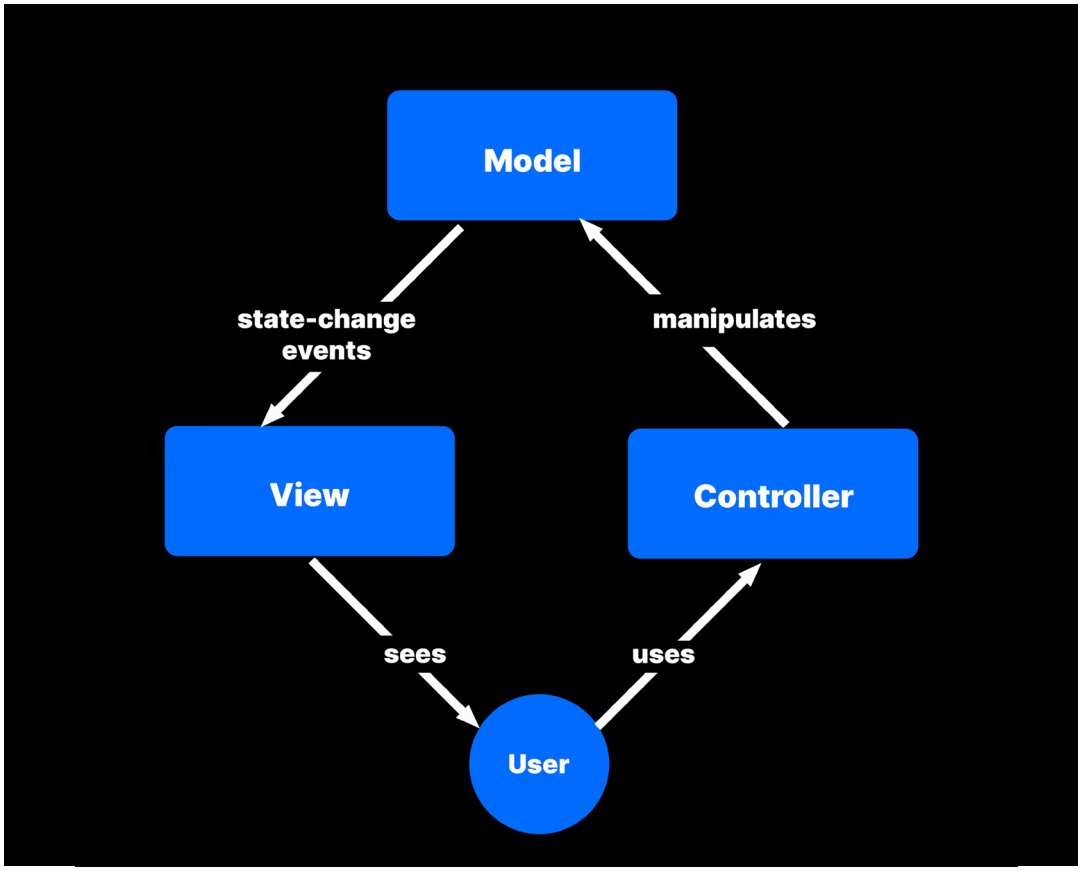
MVC の背後にある概念は、ソフトウェアのロジック部分をデータおよびプレゼンテーションから切り離すことです。ゲームは他のアプリケーションと同様、ユーザーインターフェースを通して、プレイヤーとプログラムの基盤であるデータとを接続します。UI とデータコンポーネントは多くの場合、アプリケーションのさまざまな部分にまたがっていることが多く、直接結合すると問題が発生するおそれがあります。

コンポーネントが緊密に結合されていると、不必要な依存関係が生じ、コードベースの複雑さを増大させ、バグの影響を受けやすくなるおそれがあります。MVC パターンは、アプリケーションの各パーツ間のモジュール性を促進し、結合をゆるやかにします。これにより、不要な依存関係が減り、スパゲッティコード を削減できる可能性があります。

モデルビューコントローラー (MVC) デザインパターン

その名前のとおり、MVC パターンによってアプリケーションが以下の 3 つのレイヤーに分割されます。

- **モデルにデータが格納される:**モデルとは、厳密に言えば値が保持されるデータコンテナのことです。ゲームプレイのロジックを実行することも、計算を行うこともありません。
- **ビューはインターフェース:**ビューは、データを画面上にグラフィカルに表現するためにフォーマットし、レンダリングします。
- **コントローラーがロジックを処理する:**頭脳のようなものと考えてください。ゲームデータを処理し、ランタイムに値がどのように変化するかを計算します。



モデル、ビュー、コントローラー

このように関心を分離することにより、これら 3 つのパーツの相互作用も具体的に定義されます。モデルはアプリケーションデータを管理し、ビューはそのデータをユーザーに表示します。コントローラーは入力を処理し、ゲームデータに関わるあらゆる決定や計算を行います。その後、モデルに結果を返します。

このため、コントローラー自体にはゲームデータは一切含まれていません。ビューも同様です。MVC デザインにより、各レイヤーの役割が制限されます。あるパーツがデータを保持し、別のパーツがデータを処理し、最後のパーツがそのデータをユーザーに表示します。

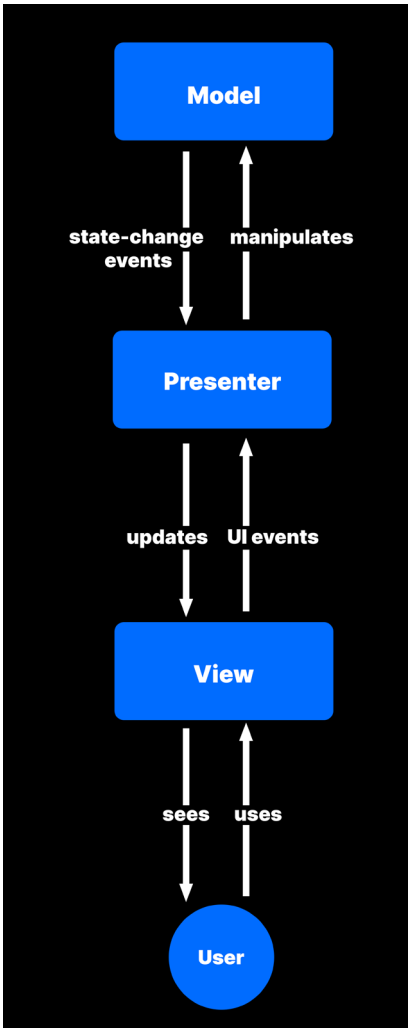
表面上は、単一責任の原則を拡張したものと見なすことができます。各パーツが 1 つのことを担います。これが MVC アーキテクチャの利点の 1 つです。

モデルビュープレゼンター (MVP) と Unity

MVC を使用して Unity プロジェクトを開発する際に、既存の UI フレームワーク (UI Toolkit または Unity UI) は必然的にビューとして機能します。エンジンにはユーザーインターフェースの完全な実装が備わっているため、個々の UI コンポーネントをゼロから開発する必要はありません。

ただし、従来型の MVC パターンに従うには、モデルのデータの変化をランタイムにリッスンする、ビュー固有のコードが必要です。

これは有効なアプローチではありますが、多くの Unity 開発者は、コントローラーが仲介役として機能する MVC のバリエーションを使用することを選択します。そこでは、ビューはモデルを直接は観察しません。代わりに、以下のようなことを行います。



MVP:MVC のバリエーション

この MVC のバリエーションのことを、モデルビュープレゼンターデザイン (MVP) と呼びます。MVP でも引き続き関心の分離が保たれており、3 つの別個のアプリケーションレイヤーがあります。ただし、各パーツの役割が少し変わります。

MVP では、(MVC のコントローラーに該当する) プレゼンターがその他のレイヤー間の仲介役となります。モデルからデータを取得し、そのデータをビューで表示するためにフォーマットします。MVP では、入力を処理するレイヤーが MVC とは異なります。コントローラーではなく、ビューがユーザー入力の処理を担います。

このアーキテクチャがイベントとオブザーバーパターンをどのように利用しているかに注目してください。ビューがボタン、トグル、スライダーなどの UI 要素を通じてユーザー入力をキャプチャし、それらの入力はイベントを介してプレゼンターに渡されます。そして、プレゼンターがこれらのインタラクションに基づいてモデルを更新します。データが更新されると、別のイベントがプレゼンターに通知します。その後、変更したデータを使用して UI が更新されます。

例:HP インターフェース

MVP の例を具体化するために、キャラクターやアイテムの HP を表示する簡単なシステムを頭に思い浮かべてみてください。1 つのクラスにデータや UI などすべてを詰め込むこともできますが、スケールするには不都合です。機能追加が多いほど、拡張が必要になったときに煩雑になります。

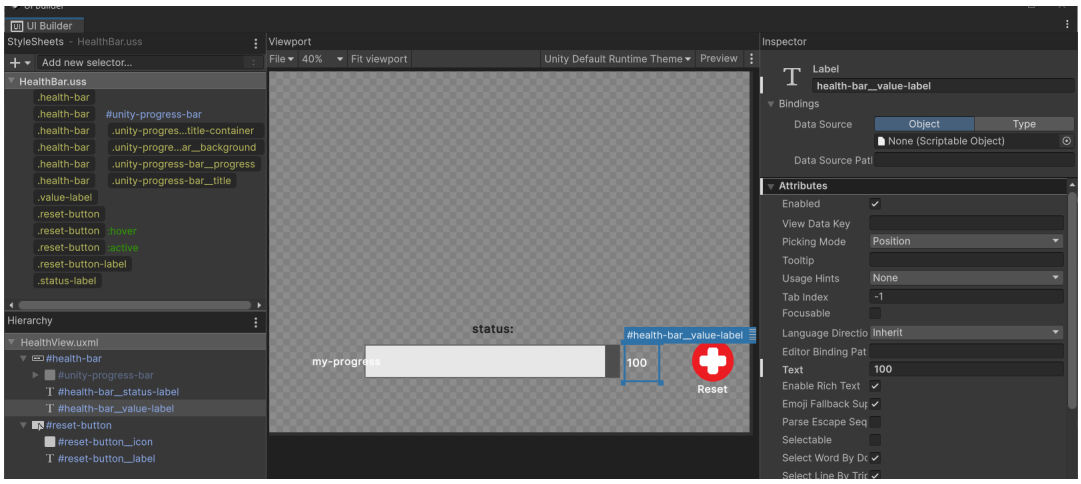
代わりに、MVP をより重視したやり方で HP コンポーネントを書き換えることができます。スクリプトを `HealthModel` と `HealthPresenter` に分割します。

MVP では、どのようなオブジェクトでも HP データを保持できますが、ScriptableObject を使用すると動作がデータ自体から切り離されるため、うまく機能します。サンプルの `HealthModel ScriptableObject` は次のようになります。

```
[CreateAssetMenu(fileName = “HealthData”, menuName = “DesignPatterns/MVP/HealthModel”)]
public class HealthModel :ScriptableObject
{
    public event Action HealthChanged;

    public int CurrentHealth;
    public string LabelName;

    ...
    public void Increment(int amount) { ... }
    public void Decrement(int amount) { ... }
    public void Restore() { ... }
}
```

`HealthModel` は、現在の HP 値 `CurrentHealth` のみを格納し、その値が変更されるたびにイベント `HealthChanged` を呼び出します。`HealthModel` にはゲームプレイロジックがなく、データをインクリメントおよびデクリメントするメソッドのみが含まれます。また、`LabelName` の文字列フィールドも含まれます。

サンプルでは UI Toolkit を使用しているため、ビューは UXML で定義されています。このインターフェースには、HP バー、ステータスラベル、値ラベルが含まれます。視覚的な表現は USS ファイルを使用してスタイル設定されます。これらのアセットは、UI Builder で管理することも、テキストとして直接管理することもできます。



UI Builder の UXML

`HealthPresenter` は、モデルのデータレイヤーとビューのユーザーインターフェースの間の仲介役として機能します。HealthModel の変更に応じて UI を更新し、ユーザー入力を処理して、HP データを変更します。

シリアライズされたフィールドは、UI Document (ビュー) と `m_HealthModelAsset` ScriptableObject (モデル) を参照します。

```
public class HealthPresenter :MonoBehaviour
{

    [SerializeField] private UIDocument m_Document;
    [SerializeField] private HealthModel m_HealthModelAsset;
    private VisualElement m_Root;
    private ProgressBar m_HealthBar;
    private Label m_StatusLabel;
    private Label m_ValueLabel;

    private void OnEnable()
    {
        NullRefChecker.Validate(this);
        m_Root = m_Document.rootVisualElement;

        ...

        if (m_HealthModelAsset != null)
        {
            m_HealthModelAsset.HealthChanged += OnHealthChanged;
            UpdateUI();
        }
    }

    private void OnHealthChanged() => UpdateUI();

    private void OnDisable()
    {
        if (m_HealthModelAsset != null)
            m_HealthModelAsset.HealthChanged -= OnHealthChanged;
    }

    private void UpdateUI()
    {
        ...
        // HP モデルデータに基づいて UI 要素を更新するロジック
    }
```

```
    private void RegisterElements()
    {
        var resetButton = m_Root.Q<Button>(“reset-button”);
        if (resetButton != null)
            resetButton.clicked += RestoreHealth;
    }

    public void RestoreHealth() => m_HealthModelAsset.Restore();
    public void ApplyDamage(int damage) => m_HealthModelAsset
    .Decrement(damage);
}
```

その他のゲームオブジェクトでは、ApplyDamage メソッドと RestoreHealth メソッドを使用して HP 値を変更するために、HealthPresenter を使用する必要があります。

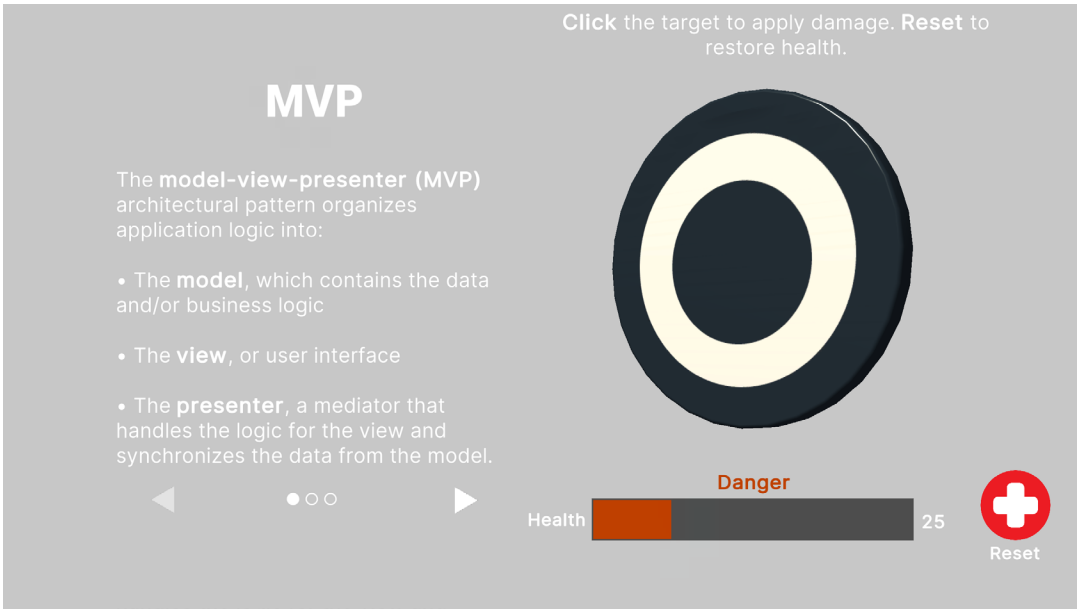
重要なのは、HealthPresenter には、ビューをモデルデータと同期させる役割の UpdateUI メソッドが含まれていることです。このメソッドは、HealthModel の HP データが変更されるたびに発生する OnHealthChanged イベントハンドラーで呼び出されます。

```
private void UpdateUI()
{
    float healthPercentage = (float)m_HealthModelAsset.CurrentHealth /
    m_HealthModelAsset.MaxHealth;

    // 現在の HP を反映するようにプログレスバーを更新する
    m_HealthBar.value = healthPercentage * 100;

    // HP パーセントに基づいて HP バーの色を変更する
    m_HealthBar.Q<VisualElement>(“progress”).style.backgroundColor =
        new StyleColor(Color.Lerp(Color.red, Color.green,
        healthPercentage));

    // HP のパーセンテージに基づいてステータスラベルを更新する
    m_StatusLabel.text = healthPercentage switch {
        < 0.33f => “Danger”,
        < 0.66f => “Neutral”,
        _ => “Good”
    };

    // 数値ラベルを更新する
    m_ValueLabel.text = m_HealthModelAsset.CurrentHealth.ToString();
}
```

`UpdateUI` は、ProgressBar の値を計算し、塗り潰し部分の背景色を変更して、両方の Label を更新します。このロジックは、各要素について必要に応じて、整数値を文字列メッセージまたは色に変換します。

`CurrentHealth` 値が変更されたときに `UpdateUI` メソッドをトリガーするために、`HealthPresenter` が `HealthModel` からイベントにサブスクライブする必要があることが重要な注意点です。

`UpdateUI` メソッド外のデータ (`LabelName` など) は開始時に 1 回だけ初期化され、`CurrentHealth` が変化しても自動的には更新されません。



MVP を使用した HP インターフェースのサンプル

サンプルプロジェクトでは、ターゲットをクリックして HP バーにダメージを与えたり、ボタンで HP をリセットしたりできます。これらの UI 要素は、`Health` を直接変更するのではなく、(`ApplyDamage` または `ResetHealth` を呼び出す) `HealthPresenter` に通知します。

Unity UI での MVP

Unity UI を使用している場合は、UGUI に対応した古いバージョンのサンプルシーンもあります。**7_MVP** ディレクトリ内で **MVP** シーンを探してみてください。Unity シーンにアクセスする前に、必ず SceneBootstrapper を無効にしてください。

長所と短所

MVP (と MVC) は、アプリケーションの規模が大きくなるほど真価を発揮します。ゲームの開発にかなりの規模のチームが必要とされ、ローンチしてから長期にわたってメンテナンスすることが想定される場合、以下のメリットが得られる可能性があります。

— **分割作業が円滑になる:**プレゼンターからビューを切り離しているため、ユーザーインターフェースの開発と更新を残りのコードベースからほぼ独立して行うことができるようになります。

これにより、専門の開発者間で作業を分担することができます。チームにフロントエンド開発のエキスパートがいれば、彼らにビューを任せましょう。他のメンバーとは別に作業を進めることができます。

— **MVP と MVC により単体テストが簡単になる:**これらのデザインパターンでは、ユーザーインターフェースからゲームプレイのロジックを切り離します。そのため、エディターで再生モードに入らなくても、記述したコードによるオブジェクトの動作をシミュレートできます。これは大幅な時間の節約につながります。

— **コードを読みやすく保つことができる:**このデザインパターンでは小さなクラスを作成する傾向があるため、読みやすくなります。依存関係が少ないということは、一般的に、ソフトウェアが壊れる場所が少なくなり、バグが潜んでいるおそれのある場所が少なくなることを意味します。

MVC と MVP はウェブ開発やエンタープライズソフトウェアで幅広く採用されていますが、多くの場合、アプリケーションのサイズと複雑さがある程度大きくならないと、そのメリットははっきりとはわかりません。Unity プロジェクトにいずれかのパターンを実装する前に、以下のことを考慮する必要があります。

— **事前に計画を立てる必要がある:**MVC と MVP は、このガイドで紹介する他のパターンに比べて大きなアーキテクチャパターンです。使用するにはクラスを役割別に分割する必要があるため、ある程度の組織と、多くの事前作業が求められます。

— **Unity プロジェクト内のすべてがうまく当てはまるわけではない:**MVC または MVP の "純粋な" 実装では、画面にレンダリングされるあらゆるものがビューの一部です。Unity のすべてのコンポーネントをデータ、ロジック、インターフェースに簡単に分割できるとは限りません (MeshRenderer など)。また、シンプルなスクリプトでは MVC や MVP の多くのメリットを享受できない可能性があります。

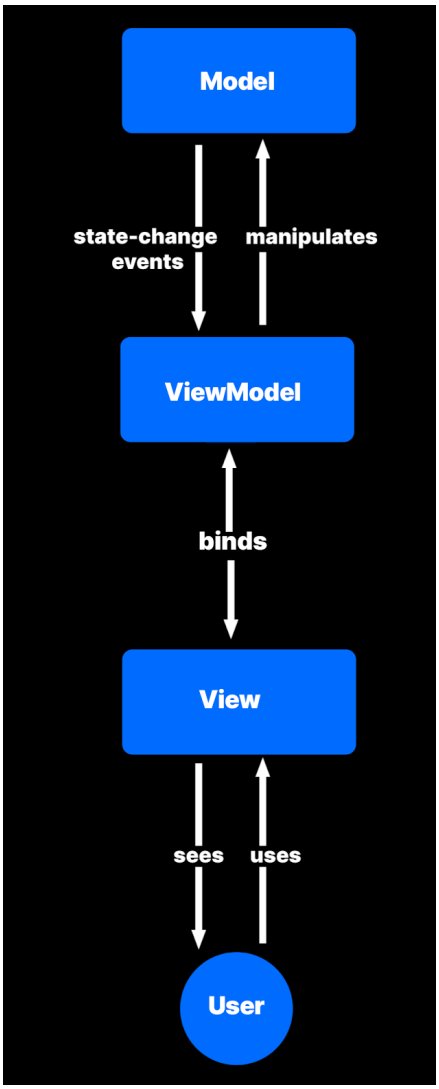
そのパターンから最もメリットを享受できる場面がどこであるか判断する必要があります。一般的には、単体テストが指針となります。MVC や MVP によってテストを円滑に進めることができるのであれば、アプリケーションのその部分で採用することを検討してください。そうでない場合は、プロジェクトにそのパターンを無理に当てはめることはしないでください。

モデルビュービューモデル (Model-View-ViewModel)

MVP は関心の分離をプロジェクトに適用しますが、プレゼンターが行うことの多くは、モデルとビューの間で単純にデータをやり取りするだけです。これは、データ処理のための多くの定型コードに相当します。

HP などのプレイヤーの統計情報がある、前述のシナリオを考えてみましょう。この値を UI で表現する方法はいくつもあります。例えば、HP 値を色 (緑は HP がフル、赤は HP が 0 に近い) で表示したり、HP が低下したときに警告メッセージを表示したりすることがあります。

MVP では、プレゼンターはインターフェースに問い合わせ、必要に応じて各 UI 要素を更新するロジックを設定する必要があります。ほとんどのケースでは、プレゼンテーションレイヤーは、既存のデータをビュー用にフォーマットし、前処理して提供するだけです。これを自動化することで、ワークフローを簡素化できます。



MVVM パターン。

Unity 6 の MVVM

そのため、Unity 6 には、MVP パターンを モデルビュービューモデル (MVVM) パターンにアップグレードする ランタイムデータバインディングシステム が備わっています。MVP と同様に、MVVM も 3 つの主要な部分で構成されます。

- **モデル:**アプリケーションのデータとビジネスロジックを表します。これは任意のオブジェクトで、多くの場合は ScriptableObject または MonoBehaviour の形式を取ります。
- **ビュー:**データを表示してユーザーと対話するユーザーインターフェースです。UI Toolkit では通常、UXML ファイルと USS スタイルシートで構成されます。
- **ビューモデル**:MVP のプレゼンターと同様に、ビューモデルはモデルとビューの間の仲介役として機能します。これは通常、MonoBehaviour として実装されます。

見覚えがありますか?そのはずです。MVVM は MVC デザインパターンの仲間です。主な違いは、MVVM では **データバインディング** (下記囲み参照) が追加されていることです。データバインディングにより、モデルのプロパティが変更された際、ビューの更新がより自動的になります。これにより、基礎となるデータとユーザーインターフェースを同期するための反復コードの多くが簡素化され、削減されます。

データバインディング

データバインディングにより、非 UI オブジェクトのプロパティ (MonoBehaviour の文字列プロパティなど) と UI 要素のプロパティ (TextField の値プロパティなど) の同期が保証されます。バインディングとは本来、非 UI プロパティとそれを変更する UI 要素の間のリンクです。

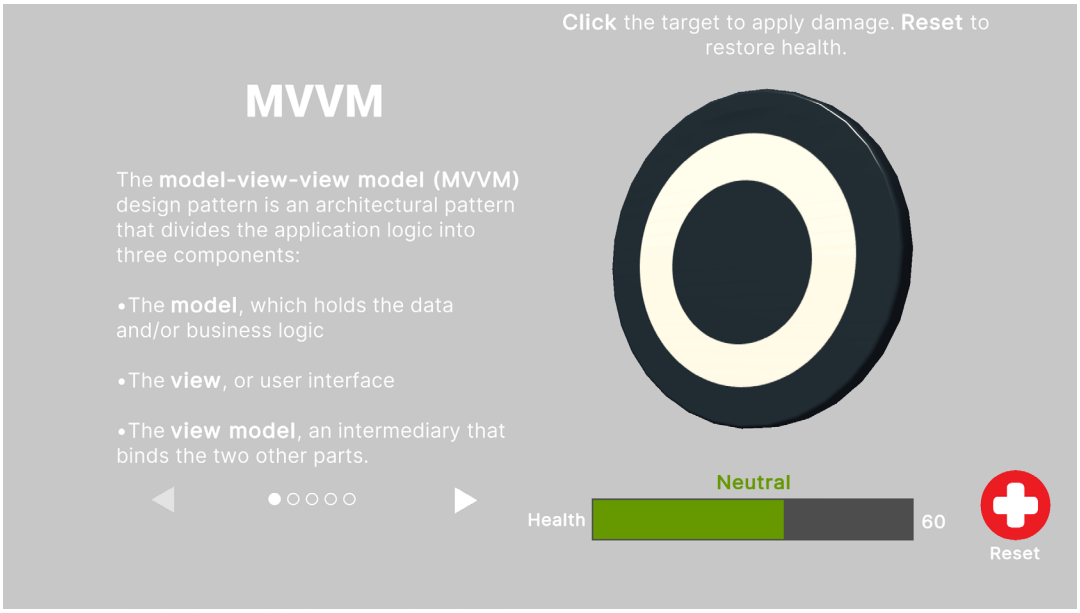
バインディングを設定すると、基礎となるデータと対応するビジュアル要素との間で変更が自動的に同期されます。これにより、UI の更新ごとにイベントハンドラーを手動で記述する必要がなくなります。

Unity 6 の UI Toolkit で、ランタイムデータバインディング がサポートされるようになりました。この機能により、ランタイムの UI 操作中に C# オブジェクトのプロパティを UI コントロールのプロパティにバインドできます。シリアライズされたデータでない限り、エディターの UI でも使用できます。

例：更新されたサンプルプロジェクト

このデモシーンは、MVP サンプルシーンから HP バーの例を取り、MVVM と UI Toolkit のランタイムデータバインディングを使用して再構築しています。

MVP の例と同様に、シーンにはターゲットの HP バーを更新するインタラクティブな要素が含まれています。コライダーをクリックするとターゲットにダメージを与え、右下のボタンをクリックすると HP バーがリセットされます。



MVVM サンプルシーン。

MVP サンプルシーンの同じ HP バーの例と比べると、デザインパターンの違いがわかります。

この場合も、`HealthModel` は ScriptableObject で、`CurrentHealth` のフィールドと、その値を加算、減算、およびリセットするいくつかの基本メソッドがあります。また、データバインディングに使用されるデータコンバーターもいくつか追加されていますが、それ以外は同じです。

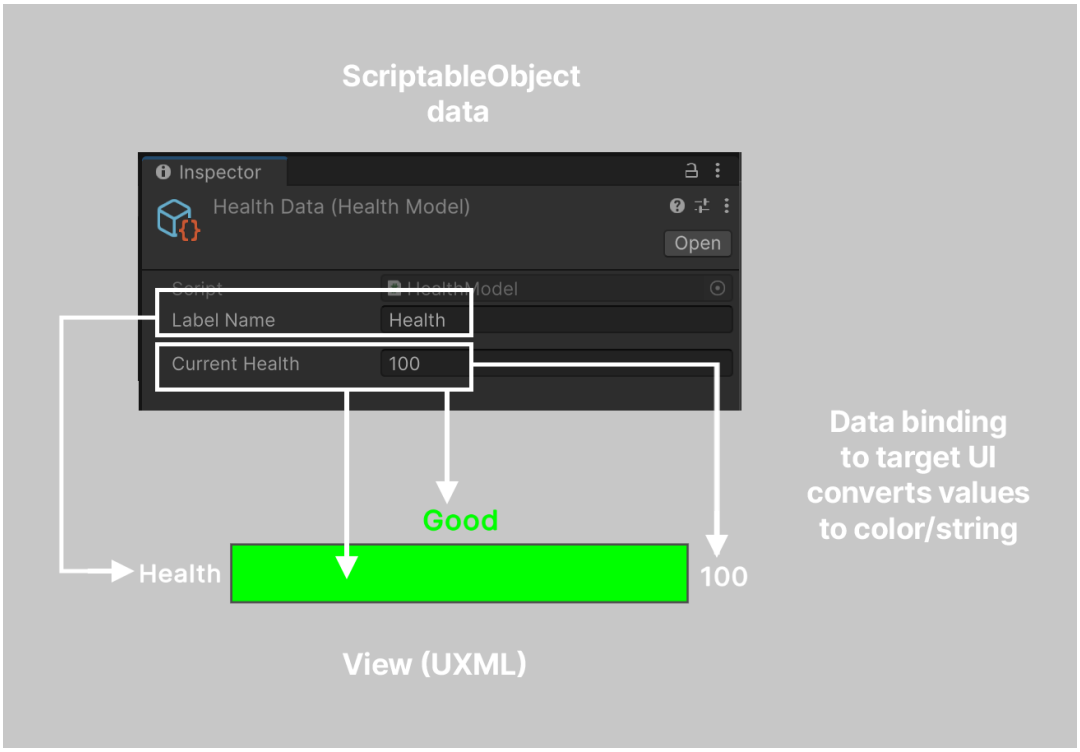
`HealthView` はほぼ変わらず、UXML と `HealthBar` のスタイルシートは同じです。

`HealthViewModel` は、ここでもモデルとビューの間の仲介役として機能します。ただし、UI の更新に使用されるロジックの多くは、UI Toolkit のランタイムデータバインディングにオフロードされています。スクリプトコンポーネントは、ボタンのインタラクティブ性を再度設定し、C# でデータバインディングを再現する方法を示します。

UI と HealthModel ScriptableObject の間のデータバインディングがどのように更新されるかに注目してください。

- **Label Name** は、左側の UI ラベルにバインドされます。フィールドを変更すると、画面上で自動的に更新されます。
- **Current Health** フィールドの値が、右側のラベルに表示されます。値が変化すると、テキストは自動的に更新されます。
- ステータスラベルのテキストプロパティは、`CurrentHealth` 値に基づいて "Good"、"Neutral"、"Danger" を示します。ラベルの色は、それに応じて緑から赤へと変化します。
- プログレスバーの色がそれに合わせて更新されます。これは、`HealthViewModel` スクリプトを使用してデータバインディングを設定する方法を示しています。

データバインディングを活用することで、MVVM パターンはモデルとビューの同期を簡素化します。



UI は HealthModel ScriptableObject の値にバインドされる。

データバインディング:UI Builder

UI Builder を使用したデータバインディングの基本的な設定例を見てみましょう。UI でデータを ScriptableObject から直接変換する必要がある場合、このバインディングは多くの場合、別のスクリプトを必要とせずに実行できます。

`HealthModel` を準備するために、`InitializeOnLoadMethod` 属性を持つ静的メソッドを追加できます。`RegisterConverters` は、HP を表す整数値を色 (緑から赤) または文字列表現 ("Danger"、"Neutral"、"Good") に変換できる ConverterGroup (この例では "Int to HealthBar") を追加します。これらはビジュアルまたは文字によるフィードバックを提供できます。

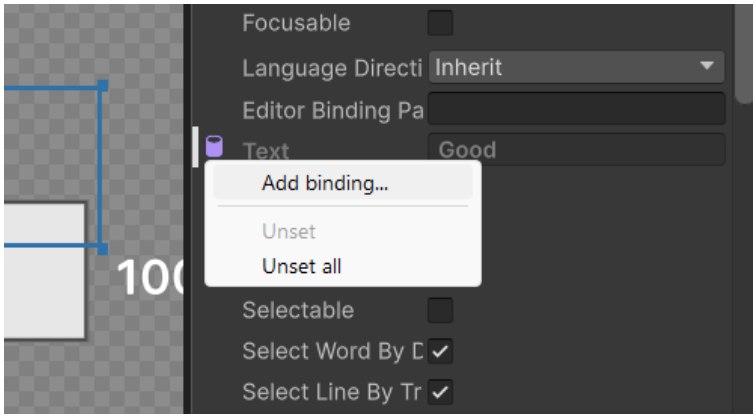
これを実装する方法を以下に示します。

```
[InitializeOnLoadMethod]
public static void RegisterConverters()
{

    var converter = new ConverterGroup(“Int to HealthBar”);

    converter.AddConverter((ref int value) =>
        new StyleColor(Color.Lerp(Color.red, Color.green, value /
        (float)k_MaxHealth)));

    converter.AddConverter((ref int value) =>
    {
        float healthRatio = (float)value / (float)k_MaxHealth;
        return healthRatio switch
        {
            >= 0 and < 1.0f / 3.0f => “Danger”,
            >= 1.0f / 3.0f and < 2.0f / 3.0f => “Neutral”,
            _ => “Good”
        };
    });

    ConverterGroups.RegisterConverterGroup(converter);
}
```

その後、UI Builder で UXML を開き、データバインディングをインタラクティブに適用できます。

— プロパティにバインドする UI 要素を見つけます。サンプルプロジェクトでは、CurrentHealth をステータスラベルと値ラベルにバインドしています。

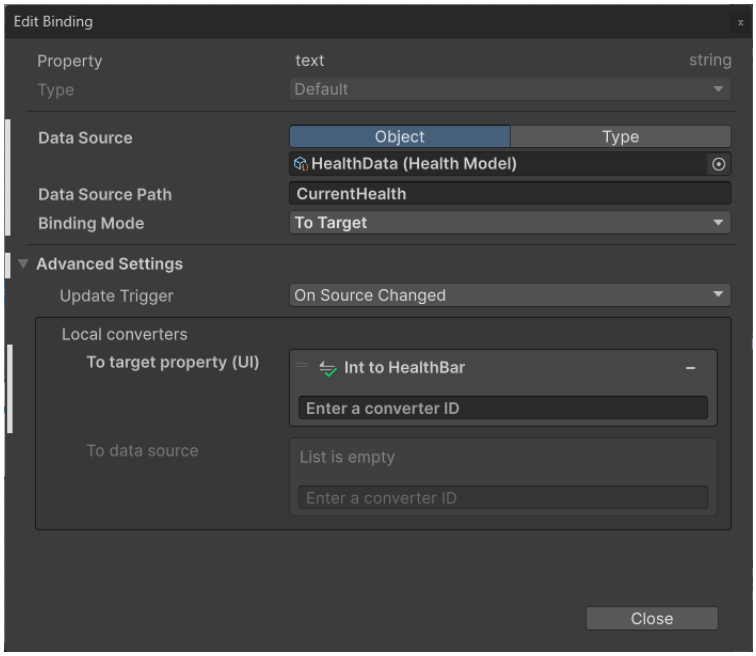
— 右クリックして、コンテキストメニューから **Add binding…** を選択します (すでにバインドが存在する場合は **Edit binding…** を選択します)。



Inspector でプロパティにデータバインディングを追加する

— 次に、Add Binding ウィンドウで、**Data Source**、**Data Source Path**、**Binding Mode** を選択します。

例えば、ステータスラベルでは、Data Source は **HealthData** アセットです。Data Source Path は **CurrentHealth** プロパティです。BindingMode は **To Target** 設定を使用します。これは、データはソースから UI に一方向にのみバインドされるということです。つまり、UI はデータを反映するように変更され、逆方向にはバインドされません。



Edit Binding ウィンドウ。

— ScriptableObject から特定のコンバーターを選択する場合は、**Advanced Settings** を開きます。

ここで、ローカルコンバーターは HealthModel ScriptableObject で作成された **"Int to HealthBar"** ConverterGroup を使用します。

設定が完了すると、UI Builder の Inspector にデータバインディングアイコンが表示されます。

データバインディングが Inspector に表示される。

データバインディングが設定されると、ユーザーインターフェースは追加コードなしで機能します。この簡素化されたワークフローを、MVP サンプルシーンの `HealthPresenter` と比べてみてください。

ターゲットをクリックして、`CurrentHealth` にダメージを与えます。プログレスバーとラベルがすぐに更新され、新しい値が反映されます。

UXML をテキストエディターで開き、シーンの背後で起こっていることを見てみます。データバインディングで設定された各要素には、UI Builder で設定されたすべての情報を含む Binding ブロックがあります。

```
<engine:UXML xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xmlns:engine="UnityEngine.UIElements" xmlns:editor="UnityEditor.UIElements" noNamespaceSchemaLocation="..
  UIElementsSchema/UIElements.xsd" editor-extension-mode="False">
    <Style src="project://database/Assets/UnityTechnologies/_DesignPatterns/7_MVVM/UI/HealthBar.uss?
      fileID=7433441132597879392&amp;guid=06a90fe64cdfac041af01e5206e789e6&amp;type=3#HealthBar" />
    <engine:ProgressBar value="46.9" title="my-progress" name="health-bar" class="health-bar" style="background-color: rgba(224, 130, 130, 0);">
        <engine:Label text="status: " name="health-bar__status-label" class="health-bar__status-label status-label">
            <Bindings>
                <engine:DataBinding property="text" data-source-path="CurrentHealth" data-source="project://database/Assets/MVVM/Data/HealthData.asset?
                  fileID=11400000&amp;guid=55ce2fc17afc2f847889c6ee80a12fe4&amp;type=2#HealthData" binding-mode="ToTarget" source-to-ui-converters="Int to HealthBar" />
                <engine:DataBinding property="style.color" data-source-path="CurrentHealth" data-source="project://database/Assets/MVVM/Data/HealthData.asset?
                  fileID=11400000&amp;guid=55ce2fc17afc2f847889c6ee80a12fe4&amp;type=2#HealthData" binding-mode="ToTarget" source-to-ui-converters="Int to HealthBar" />
            </Bindings>
        </engine:Label>
        <engine:Label text="100" name="health-bar__value-label" class="value-label">
            <Bindings>
                <engine:DataBinding property="text" data-source-path="CurrentHealth" data-source="project://database/Assets/MVVM/Data/HealthData.asset?
                  fileID=11400000&amp;guid=55ce2fc17afc2f847889c6ee80a12fe4&amp;type=2#HealthData" binding-mode="ToTarget" />
            </Bindings>
        </engine:Label>
        <Bindings>
            <engine:DataBinding property="value" data-source-path="CurrentHealth" data-source="project://database/Assets/MVVM/TargetHealth.asset?
              fileID=11400000&amp;guid=55ce2fc17afc2f847889c6ee80a12fe4&amp;type=2#TargetHealth" binding-mode="ToTarget" />
            <engine:DataBinding property="title" data-source-path="LabelName" data-source="project://database/Assets/MVVM/TargetHealth.asset?
              fileID=11400000&amp;guid=55ce2fc17afc2f847889c6ee80a12fe4&amp;type=2#TargetHealth" binding-mode="ToTarget" />
        </Bindings>
    </engine:ProgressBar>
    <engine:Button text="&#10;" name="reset-button" data-source-type="DesignPatterns.MVVM.HealthViewModel, Assembly-CSharp" class="reset-button">
        <engine:VisualElement name="reset-button__icon" style="flex-grow: 1; position: absolute; width: 70%; height: 70%; bottom: 10%; background-image: url(&quot;project:
          UnityTechnologies/_DesignPatterns/7_MVP/Textures/HealthIcon.png?fileID=21300000&amp;guid=4325c0feb824443e1a6b094bbaf98353&amp;type=3#HealthIcon&quot;);" />
        <engine:Label text="Reset" name="reset-button__label" class="reset-button-label" />
    </engine:Button>
</engine:UXML>
```

データバインディングは、UXML 内にコードブロックとして表示される。

例えば、HP バーの上にある `health-bar__status-label` という名前のラベルは、`CurrentHealth` を適切な文字列と色に変換します。その後、これらの値は "text" プロパティと "style.color" プロパティにバインドされます。

データバインディング:スクリプティング

ケースによっては、UI Builder の代わりに C# を使用してデータバインディングを設定する必要があります。これは、特定の UI 要素に内部パーツやサブ要素が含まれている場合に便利です。例えば、ProgressBar の背景と塗り潰し部分は、UI Builder の Inspector で直接選択することはできません。

このスニペットは、`HealthViewModel` スクリプトでデータバインディングを設定する方法を示しています。

```
private void SetDataBindings()
{
    var healthBar = m_Root.Q<ProgressBar>(“health-bar”);
    var healthBarProgress = healthBar?.Q<VisualElement>(className:
    “unity-progress-bar__progress”);
    if (healthBarProgress != null)
    {
        healthBarProgress.dataSource = m_HealthModelAsset;

        var binding = new DataBinding
        {
             dataSourcePath = new PropertyPath(nameof(HealthModel
             .CurrentHealth)),
             bindingMode = BindingMode.ToTarget,
         };
         binding.sourceToUiConverters.AddConverter((ref int value) =>
             new StyleColor(Color.Lerp(Color.red, Color.green,
                 (float)value / m_HealthModelAsset.MaxHealth)));

         healthBarProgress.SetBinding(“style.backgroundColor”,
         binding);
    }
}
```

SetDataBindings メソッドは VisualElement 階層に対してクエリを実行し、"health-bar" という名前の ProgressBar を探します。その後、UI Builder と同じようにデータバインディングを設定します。

- データソースを設定します。このケースでは、データソースはプロジェクト内の HealthData ScriptableObject アセットです。
- dataSourcePath とバインディングモードを含むデータバインディングオブジェクトを作成します。
- 必要に応じてデータコンバーターを定義します。上記の例は、整数値を ProgressBar の塗り潰し部分で StyleColor に変換する方法を示しています。

次に、UI 要素で `SetBinding` を呼び出し、プロパティ (ProgressBar の塗り潰し部分の色 "style.backgroundColor" など) と バインドしている オブジェクトを渡します。データバインディングが必要な要素プロパティごとに、このプロセスを繰り返します。

これは、ランタイムに ScriptableObject インスタンスを作成する必要があるなど、ScriptableObject アセットに直接バインドできない場合に特に重要になります。HealthViewModel を持つゲームオブジェクトが個々の HP オブジェクトを参照する必要がある場合は、UI Builder ではなくスクリプティングを介してデータバインディングを設定します。

データバインディングを使用した場合と、使用しない前述の MVP の例を比較します。両方で同じ UXML と USS を使用します。

データバインディングなし (MVP)	データバインディングあり (MVVM)
`HealthPresenter` は、更新するために `HealthModel` のイベントをリッスンします。 `HealthPresenter` は、UI 要素 (`UpdateUI`) に表示する値を変換します。 `HealthModel` には、データと基本的なビジネスロジック (`Increment`、`Decrement`) が格納されます。	`HealthModel` は `ConverterGroup` を登録し、モデルデータを UI で使用される形式に変換します。 `HealthViewModel` は、HealthModel から UI へのバインディング (`SetDataBindings`) を作成します。 `HealthModel` には、データと基本的なビジネスロジック (`Increment`、`Decrement`) が格納されます。

MVP (Presenter)

```
public class HealthPresenter : MonoBehaviour
{
...

    private void OnEnable()
    {
...
        if (m_HealthModelAsset != null)
        {
            m_HealthBar.title = m_HealthModelAsset.LabelName;
            m_HealthModelAsset.HealthChanged += OnHealthChanged;
        }
        UpdateUI();
    }

    private void OnDisable()
    {
        if (m_HealthModelAsset != null)
        {
            m_HealthModelAsset.HealthChanged -= OnHealthChanged;
        }
    }

    private void OnHealthChanged()
    {
        UpdateUI();
    }

    private void RegisterElements(){...}

    private void UpdateUI()
    {
        float healthRatio = (float)m_HealthModelAsset.CurrentHealth / m_HealthModelAsset.MaxHealth;
        Color healthColor = Color.Lerp(a: Color.red, b: Color.green, healthRatio);
        m_HealthBar.value = healthRatio * 100f;
        var healthBarProgress = m_HealthBar?.Q<VisualElement>(className: "unity-progress-bar__progress");

        if (healthBarProgress != null)
        {
            healthBarProgress.style.backgroundColor = new StyleColor(healthColor);
        }

        m_StatusLabel.text = healthRatio switch
        {
            >= 0 and < 1.0f / 3.0f => "Danger",
            >= 1.0f / 3.0f and < 2.0f / 3.0f => "Neutral",
            _ => "Good"
        };

        m_StatusLabel.style.color = new StyleColor(healthColor);
        m_ValueLabel.text = m_HealthModelAsset.CurrentHealth.ToString();
    }


    public void RestoreHealth(){...}

    public void ApplyDamage(int damage){...}
}
```

Model notifies Presenter through event

Presenter transforms data for use with View

MVVM (ViewModel)

```
public class HealthModel : ScriptableObject
{
    public static HealthModel CreateInstance(HealthModel original){...}

    [InitializeOnLoadMethod]
    public static void RegisterConverters()
    {
        float HealthRatio(int health) => health / (float)k_MaxHealth;

        var converter = new ConverterGroup(id: "Int to HealthBar");

        converter.AddConverter((ref int value) =>
            new StyleColor(v: Color.Lerp(a: Color.red, b: Color.green, t: HealthRatio(value))));

        ConverterGroups.RegisterConverterGroup(converter);
    }
```

Register Converters in Model

```
public class HealthViewModel : MonoBehaviour
{

    private void OnEnable(){...}

    private void RegisterElements(){...}

    private void SetDataBindings()
    {
        var healthBar = m_Root.Q<ProgressBar>(name: "health-bar");
        var healthBarProgress = healthBar?.Q<VisualElement>(className: "unity-progress-bar__progress");

        if (healthBarProgress != null)
        {
            healthBarProgress.dataSource = m_HealthModelAsset;

            var binding = new DataBinding
            {
                dataSourcePath = new PropertyPath(nameof(HealthModel.CurrentHealth)),

                bindingMode = BindingMode.ToTarget,
            };

            binding.sourceToUiConverters.AddConverter((ref int value) =>
                new StyleColor(v: Color.Lerp(a: Color.red, b: Color.green,
                    t: (float)value / (float)m_HealthModelAsset.MaxHealth)));

            healthBarProgress.SetBinding(bindingId: "style.backgroundColor", binding);
        }
    }

    public void RestoreHealth(){...}

    public void ApplyDamage(int damage){...}
}
```

Data binding propagates Model updates to the View

MVP では、プレゼンターはモデルからのイベントをサブスクライブして、状態の変更を検出します。変更が通知されると、プレゼンターはビューに適した形式にデータを処理し、それに応じてビューを更新します。

MVVM では、まず必要なコンバーターをモデルに登録します。次にビューモデルでデータバインディングを確立します。この設定により、既存のバインディングを通じて、モデルの変更でビューが自動的に更新することが可能になります。

ランタイムデータバインディングの詳細については、ドキュメント を参照してください。

長所と短所

MVVM は、テストが容易になる、懸念事項が分離できるなど、MVP のメリットの多くを共通して持っています。データバインディングにより、UI と基礎となるデータとの同期を維持するために必要な定型コードの量を削減し、モデルから発生するイベントの数を減らすことができます。これにより、コードがより簡潔になり、読みやすく、保守が簡単になります。また、データバインディングにより、古いデータや誤ったデータが表示されるリスクが減るため、UI の一貫性も向上します。

ただし、各データバインディングの設定による追加のオーバーヘッドを考慮する必要があります。データソース、データソースパス、バインディングモード、コンバーターなどの事前設定に少し手間がかかります。このパターンは、複雑さのコストを上回るメリットが得られる大規模なユーザーインターフェースにのみ適しているかもしれません。

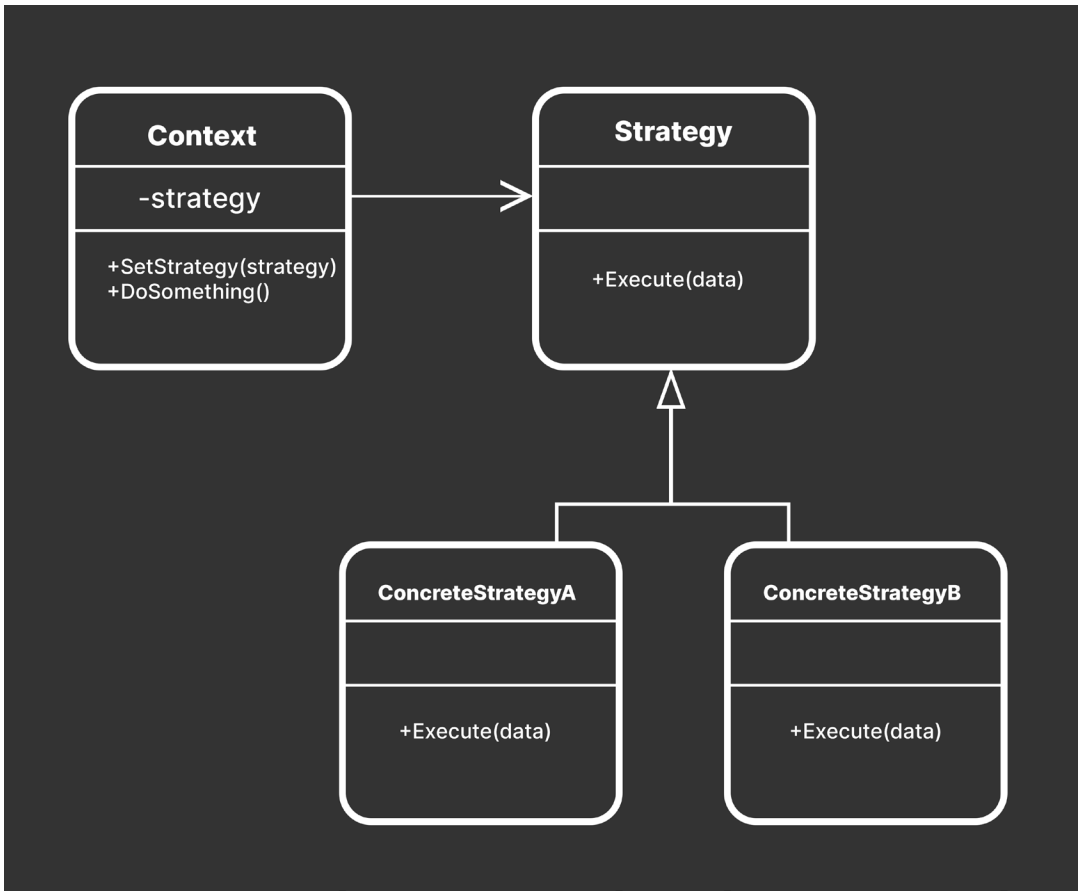
ストラテジー (Strategy) パターン

ゲームプレイが静止していることはめったにありません。ランタイムでは、ゲームオブジェクトは条件の変化に適応し、それに応じて自身を更新する必要があります。

例えば、ステルスゲームを想像してください。プレイヤーの移動スタイルは、警備員をかわしながら忍び寄る動きと、発見された後で逃げる動きを切り替える必要があります。あるいは、近接、遠距離、魔法など、さまざまな攻撃モードをキャラクターが発動できる戦闘システムを考えてみてください。

これらの動的な動作をクリーンでメンテナンス性の高い方法で実装することは、ゲームが大きくなるにつれて難しくなるおそれがあります。ステートパターンと同様に、switch　ステートメントを使用すると、クラスが大規模に膨張するおそれがあります。

ストラテジー (Strategy) パターンは、アルゴリズムや動作をオブジェクト内に包み込み、それらを交換可能にすることで、この問題に対する解決策を提供します。各ストラテジーオブジェクトは、動的に実行可能な個別の動作をカプセル化します。そのためクライアントオブジェクトは、自身のクラス構造を変更することなく、さまざまなストラテジーオブジェクトを参照してランタイムに動作を切り替えることができます。



ストラテジーパターンはランタイムに動作を交換可能にする。

例：アビリティシステム

ゲームが進むにつれて新しいアビリティ (能力) を獲得していくゲームを開発しているところを想像してみてください。これらのアビリティは、対戦型 FPS やアクション RPG で優れたパフォーマンスを発揮した場合の報酬や "特典" として使用できます。プレイヤーが新しいアビリティを獲得できるようになると、対応する UI ボタンが画面に表示され、そのアビリティが使用可能であることが示されることがあります。

リファクタリングの前に

まず、すべての特別なアビリティを処理する単一のスクリプトを作成することもできるでしょう。このアプローチは有効ですが、新しいアビリティを追加したり既存のアビリティを変更したりする必要があると、メンテナンスが難しくなります。

これらのアビリティを定義するための初期設定は、例えば次のようになります。

```
public class AbilityRunner :MonoBehaviour
{
    public enum Ability
    {
        RadarPulse,
        AirSupport,
        FirstAid
    }

    public Ability currentAbility;

    void Update()
    {
        if (Input.GetKeyDown(KeyCode.Space))
        {
            ActivateAbility(currentAbility);
        }
    }

    void ActivateAbility(Ability ability)
    {
        switch (ability)
        {
            case Ability.RadarPulse:
                // Radar Pulse ロジック
                Debug.Log(“Activating Radar Pulse”);
                break;

            case Ability.AirSupport:

                // Air Support ロジック
                Debug.Log(“Calling in Air Support”);
                break;

            case Ability.FirstAid:

                // ファーストエイド/ヒーリングロジック
                Debug.Log(“Using First Aid”);
                break;
        }
    }
}
```

ゲームの発展に伴い、このスクリプトはどんどん複雑になり、管理が難しくなります。新しいアビリティごとに既存のコードを変更する必要があり、オープン/クローズドの原則に反します。私たちが目指すのは、ソフトウェアが拡張に対してはオープンで、変更に対してはクローズな状態を維持することです。

ストラテジーパターンの実装

ストラテジーパターンを使用してアビリティシステムを再検討しましょう。まず、抽象クラス Ability またはインターフェースを作成します。これにより、Use というメソッドが定義されます。このメソッドは、具体的なアビリティすべてに実装される必要があります。この例では、ScriptableObject を拡張します (ただし、ここでは任意のオブジェクトを使用できます)。

```
public abstract class Ability :ScriptableObject
{
    public string abilityName;
    public abstract void Use(GameObject gameObject);
}
```

次に、各特定のアビリティに対応する Ability クラスの具体的な実装を作成します。これらのクラスは、Use メソッド内に実際のロジックを実装し、独自のアクションを実行します。

```
[CreateAssetMenu(fileName = “RadarPulseAbility”, menuName = “Abilities/
RadarPulse”)]
public class RadarPulse :Ability
{
    public override void Use(GameObject gameObject)
    {
        Debug.Log(“Activating Radar Pulse”);
        // Radar Pulse ロジックをここに実装する
    }
}

[CreateAssetMenu(fileName = “AirSupportAbility”, menuName = “Abilities/
AirSupport”)]
public class AirSupport :Ability
{
    public override void Use(GameObject gameObject)
    {
        Debug.Log(“Calling in Air Support”);
        // Air Support ロジックをここに実装する
    }
}

[CreateAssetMenu(fileName = “FirstAidAbility”, menuName = “Abilities/
```

```
FirstAid")]
public class FirstAid :Ability
{
    public override void Use(GameObject gameObject)
    {
        Debug.Log("Using First Aid");
        // ファーストエイド/ヒーリングロジックをここに実装する
    }
}
```

これらの ScriptableObject は、シリアライズしてプロジェクトアセットとして保存できます。これにより、Unity Inspector 内で割り当てと変更を簡単に行うことができます。

その後、クライアントオブジェクトはこれらのストラテジーオブジェクトを参照できます。ここでは、ランタイムに特定の `currentAbility` を動的に設定できるように、`AbilityRunner` クラスをリファクタリングします。この例では、スペースキーを押すと、アビリティロジックを実行する `Use` メソッドが呼び出されます。

```
public class AbilityRunner :MonoBehaviour
{
    // Unity エディターで割り当てる
    public Ability currentAbility;

    void Update()
    {
        if (Input.GetKeyDown(KeyCode.Space))
        {
            currentAbility.Use(gameObject);
        }
    }
}
```

自身のオブジェクトとしてカプセル化された各アビリティは、ゲームのコアコードに影響を与えることなく編集、追加、削除できます。これにより、ゲームの柔軟性が高まり、ランタイムに動的にアビリティを変更できるようになります。その結果、新しいアビリティ作成の管理性とスケーラビリティも向上します。

例：サンプルプロジェクト

このプロジェクトは、ストラテジーパターンの基本的な実装を示しています。プレイヤーはパワーアップを集めて目標とする特典を入手できます。ボタンは連勝数 (streak) に応じて更新され、連勝数の増加に応じてさまざまなアビリティが表示されます。ボタンをクリックすると、現在の特殊アビリティがストラテジーとしてアクティベートされます。

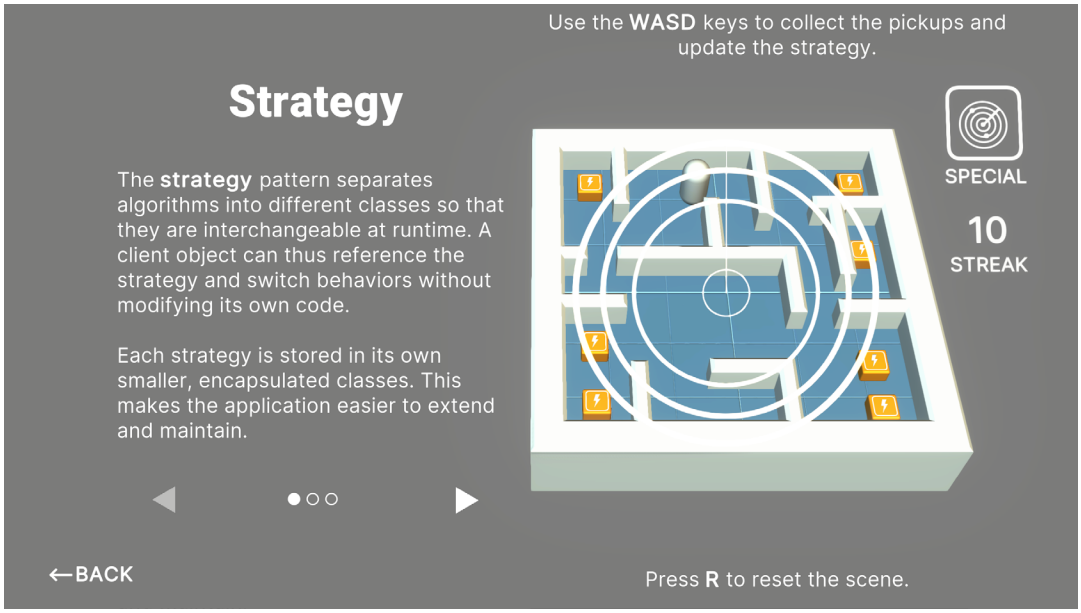
ボタンの実際の動作は ScriptableObject にラップされています。つまり、ランタイムに Inspector 内または別個のゲームロジックで交換できます。

このサンプルでは、連勝数のカウンターが関連する特典や特殊アビリティを UI に結び付けており、プレイヤーのパフォーマンスに動的に対応します。

交換可能な各ストラテジーは、自身のクラスにカプセル化されているため、アビリティを追加しても他のストラテジーには影響しません。単に、必要に応じて ScriptableObject アビリティを追加で作成します。

ここで、ボタンはいくつかの装飾要素 (パーティクルエフェクトやサウンドなど) をトリガーしますが、これは 1 つに限定されません。

カプセル化された各ストラテジーは、ゲーム固有のニーズに合わせた幅広いアクションを実行できます。ゲームプレイのメカニクスを変更したり、キャラクターのアビリティを強化したり、ゲーム環境を変更したりできます。



このプロジェクトでは、ストラテジーパターンを使用して、特殊アビリティ (特典) を実装しています。

長所と短所

ストラテジーパターンは、ランタイムにゲームの動作を変更する必要がある状況でうまく機能します。このパターンを使用すると、既存のコードを変更することなく新しい機能を追加できるため、SOLID の原則に従ってシステムをより柔軟に運用できます。各動作はそれぞれ独自のクラスに整理されており、テストも容易です。

デメリットは、管理するクラスが増えることで複雑さが増すことです。ストラテジーオブジェクトには少量のオーバーヘッドが伴うため、パフォーマンスが重視される場合は代替パターンや最適化を検討してください。

カプセル化されることは、これらのストラテジーがゲームプレイの他のシステム (イベントなど) と情報を共有し、コミュニケーションを取る方法を慎重にデザインする必要があることも意味します。ストラテジーを他のコンポーネントと緊密に結合することは避ける必要があります。そうしないと、パターンのメリットが損なわれてしまいます。

その他の例

ストラテジーパターンは、アビリティを管理するためだけのツールではありません。ゲームプレイのさまざまな側面に適用できます。実用的な例をいくつか紹介します。

キャラクターの移動ストラテジー:環境やパワーアップに応じてプレイヤーキャラクターの移動アビリティをアップグレードできるプラットフォームゲームを制作しているとしましょう。プレイヤーは最初は歩いたりジャンプしたりすることしかできませんが、後にダブルジャンプやダッシュ、さらには空を飛べるアビリティを獲得します。

AI の動作:ゲームの状態やプレイヤーのアクションに基づいて、AI の動作を切り替えることができます。プレイヤーに応じて、攻撃、防御、パトロールのストラテジー間で敵の状態を調整します。

ナビゲーションストラテジー:経路検索システムを作成した場合、ストラテジーパターンを使用して複数のアルゴリズム (A*、ダイクストラの最短経路など) を定義し、コンテキストに応じてゲームプレイ中にそれらを入れ替えることができます。

攻撃ストラテジー: MeleeAttack (近接攻撃)、RangedAttack (遠距離攻撃)、AreaEffectAttack (範囲攻撃) などのストラテジーにより、プレイヤーや AI が武器の種類を動的に切り替えることができます。または、残りの HP に応じてモードや独自の戦闘アビリティを切り替えられる敵ボスもあるでしょう。

難易度の調整:プレイヤーのパフォーマンスに基づいてゲームの難易度を自動的に調整します。リアルタイムで変化する "適応型難易度" ストラテジーを実装します。または、プレイヤーが一貫した課題に対して "固定難易度"ストラテジーを選択できるようにします。

フライウェイト (Flyweight) パターン

大規模なゲームワールドには、多数のゲームオブジェクトやコンポーネントが設定されたシーンが数多く存在します。類似のオブジェクトが同じデータフィールドを持つ場合、データの著しい重複を招き、メモリ使用量が増えるおそれがあります。樹木オブジェクトそれぞれが独自の設定データを格納する森のシーンを考えます。

ゲームシーン内の樹木には、以下のものが格納される可能性があります。

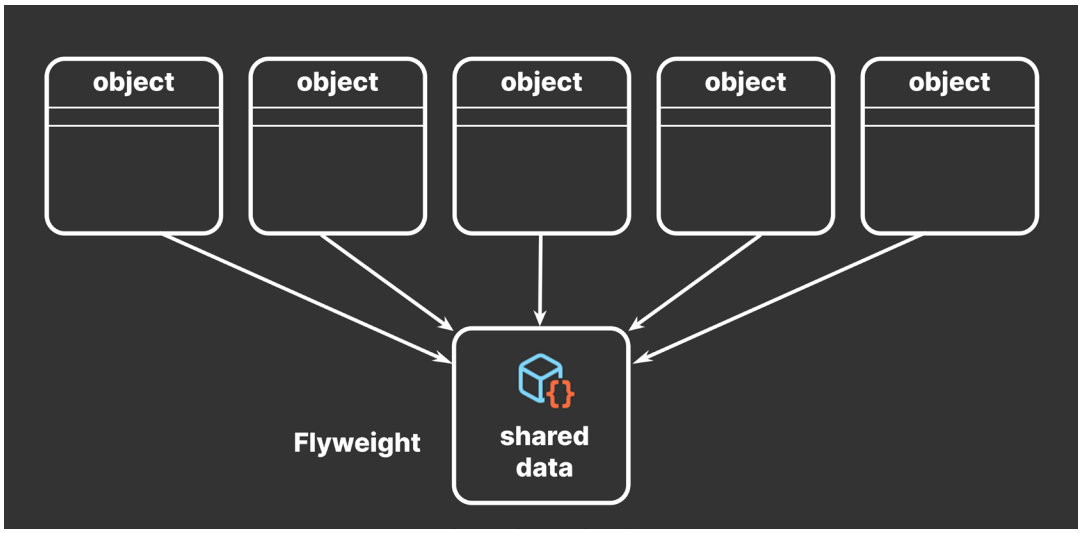
— 樹木を定義するための複雑なデータ構造。樹木をプロシージャルに生成している場合、これにはfloat、color、Vector3 値の配列またはリストが含まれることがあります。

— 風に揺れる樹木を定義するアニメーションカーブ。

— 他の物理特性やゲームプレイのメタデータを定義するカスタムクラスインスタンス。

個々のフィールドは比較的小さいかもしれませんが、ゲームオブジェクトを複製すると、そのさまざまなコンポーネントと格納されているデータフィールドもコピーされます。フィールドが値型 (構造体、プリミティブ型、配列など) の場合、複製された各ゲームオブジェクトには独自のデータのコピーがあります。ゲームワールドの森に樹木を植えると、冗長性はすぐに増します。

フライウェイトは、共有データを一元化することで、ゲーム内の重複データの量を削減できる最適化パターンです。これにより、個々のオブジェクトが自身のコピーを保存する代わりに、この共有データを参照するようになります。



類似のオブジェクト間でデータを共有するには、このフライウェイトパターンを使用する。

Unity のプレハブワークフローに慣れている方なら、このアイデアはもうご存知のことと思います。類似のオブジェクト間で可能な限り多くのデータを共有し、全体的なメモリフットプリントを削減できます。

リファクタリングされていない例

ゲームプレイユニットが多数登場するストラテジーゲームを考えましょう。それぞれ、HP、攻撃、防御、移動などの属性を持つことができます。各ユニットを識別するために、チームのラベルとアイコンでタグ付けすることもできます。

したがって、最適化されていないゲームプレイユニットには、例えば次のようなクラスがあります。

```
public class UnrefactoredUnitInstance :MonoBehaviour
{
    public string factionName;
    public Sprite factionIcon;
    public int baseHealth;
    public int baseAttack;
    public int baseDefense;
    public int baseMovement;

    // このユニットインスタンスの一意の状態
    public int health;
    public int attack;
    public int defense;
    public int movement;
    public Vector3 position;
```

```
    private void Start()
    {
        RefreshUnitStats();
    }

    private void RefreshUnitStats()
    {
        health = baseHealth;
        attack = baseAttack;
        defense = baseDefense;
        movement = baseMovement;
        // ...陣営データに基づいて他のユニットコンポーネントを更新する
    }

    public void SetFactionData(string factionName, Sprite factionIcon,
    int baseHealth, int baseAttack, int baseDefense, int baseMovement)
    {
        this.factionName = factionName;
        this.factionIcon = factionIcon;
        this.baseHealth = baseHealth;
        this.baseAttack = baseAttack;
        this.baseDefense = baseDefense;
        this.baseMovement = baseMovement;

        RefreshUnitStats();
    }
}
```

ユニットは、自身の数値情報に加えて、追加のペイロードデータを保持しており、特定の陣営やチーム全体で一定であることが理想的です。SetFactionData メソッドで新しいユニットインスタンスを作成するときは、共通のすべての陣営データを渡し、そのコピーを保存します。ユニットが増えるほど、冗長なデータが重複して保存されます。

これは単にメモリを消費するだけでなく、ゲーム内ですべてを更新し、一貫性を保つことも難しくなります。多数のオブジェクト間でデータを手動で同期するとエラーが発生しやすく、不整合が生じるおそれがあります。

オブジェクトが少数であれば問題ありません。しかし、同じ陣営に属するユニットの数が多いと、メモリ使用量の増加が顕著になり、少なくとも管理が困難になることがあります。

フライウェイトパターンの実装

簡単な方法は、共有データを中央リポジトリ、つまり**フライウェイト**オブジェクトに格納することです。

ScriptableObject は、ランタイムに変更の必要がないデータ (設定、構成データなど) を格納するのに理想的であり、この目的に適しています。

共有データを別のクラスにリファクタリングすると、次のように冗長性を減らすことができます。

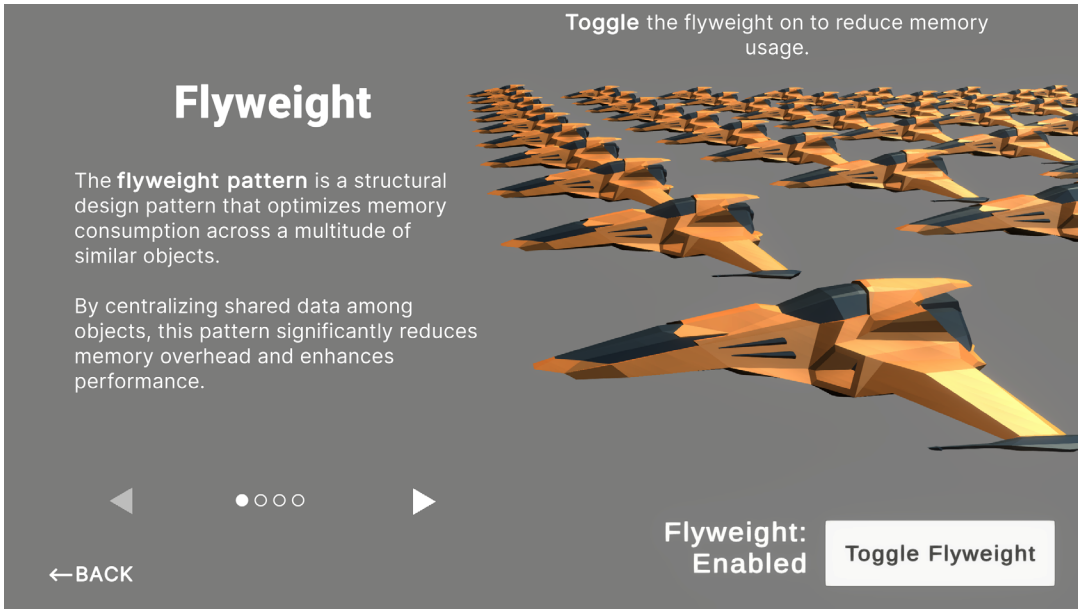
```
// Flyweight object (ScriptableObject)
[CreateAssetMenu]
public class FactionData :ScriptableObject
{
    public string factionName;
    public Sprite factionIcon;
    public int baseHealth;
    public int baseAttack;
    public int baseDefense;
    public int baseMovement;
}

// コンテキストオブジェクト
public class UnitInstance :MonoBehaviour
{
    public FactionData factionData;

    private void Start()
    {
        RefreshUnitStats();
    }

    private void RefreshUnitStats()
    {
        health = factionData.baseHealth;
        attack = factionData.baseAttack;
        defense = factionData.baseDefense;
        movement = factionData.baseMovement;

        // ...陣営データに基づいて他のユニットコンポーネントを更新する
    }

    // このユニットインスタンスの一意の状態
    public int health;
    public int attack;
    public int defense;
    public int movement;

    public Vector3 position;

    // ...ここに他の一意のステートを追加する
}
```

このパターンを使用すると、FactionData クラスは共有の陣営データを表す ScriptableObject です。これには、陣営名、アイコン、および基本ユニット統計のフィールドが含まれます。

UnitInstance クラスは、FactionData ScriptableObject への参照を保持するコンテキストオブジェクトです。共有された陣営データに基づいて統計が更新されます。共有データと固有データを分離することで、メモリ使用量を減らし、ゲームオブジェクト間の潜在的な不整合を減らします。

例：サンプルプロジェクト

提供された例では、宇宙船群のメモリ使用量を最適化することで、フライウェイトパターンが示されています。



フライウェイトパターンを使用して類似のオブジェクト間でデータを共有する。

中心となる原則は、intrinsic (状況に依存せず共有できる) 状態データと extrinsic (状況に依存し、固有で共有できない) 状態データに区別することです。intrinsic データは不変であり、インスタンス間で共有されるため、メモリ使用量を削減できます。extrinsic データはインスタンス間で異なり、個別に保存されます。

この例では、**ShipData** は ScriptableObject であり、ユニット名、速度、攻撃性能、防御など、すべての機体の本質的な共有データが含まれています。すべての宇宙船がこれらのプロパティで同じデータセットを参照するため、メモリフットプリントが最小限に抑えられます。

```
[CreateAssetMenu(fileName = “ShipData”, menuName = “Flyweight/ShipDa-
ta”, order = 1)]
public class ShipData :ScriptableObject
{
    public string UnitName;
    public string Description;
    public float Speed;
    public int AttackPower;
    public int Defense;
}
```

CreateAssetMenu 属性で定義されたコンテキストメニューから ScriptableObject インスタンスを作成します。

Ship クラスは、船団内の個々の宇宙船を表すことができます。各宇宙船のインスタンスは共有された ShipData への参照を保持し、HP などの固有の状態を管理します。

```
public class Ship :MonoBehaviour
{
    [SerializeField] private ShipData m_SharedData;

    [SerializeField] private float m_Health;

    public void Initialize(ShipData data, float health)
    {

        m_SharedData = data;
        m_Health = health;
    }

    public void DisplayShipInfo()
    {
        Debug.Log($”Name: {sharedData.UnitName}, Health: {m_Health}”);
    }
}
```

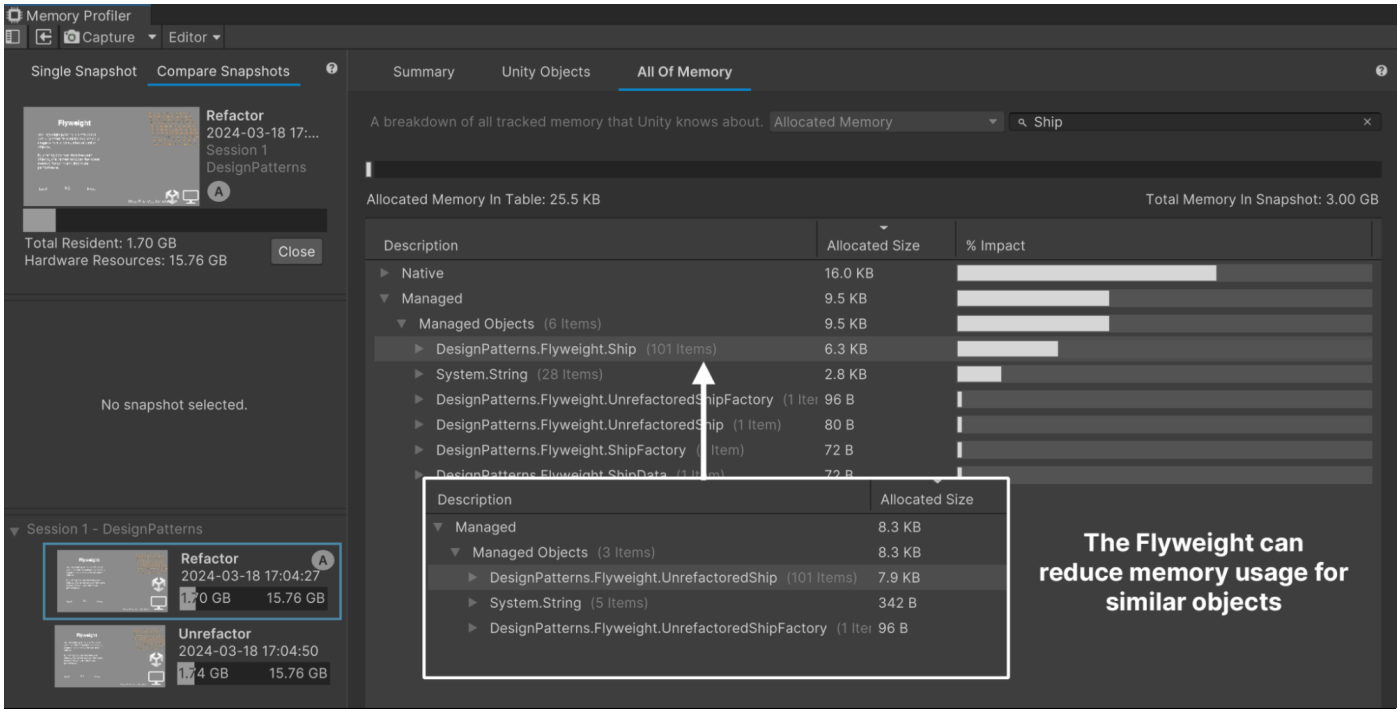
ShipFactory は、一連の宇宙船を生成します。プレハブゲームオブジェクトと共有 ShipData を使用して各宇宙船を初期化します。

```
public class ShipFactory :MonoBehaviour
{
    [SerializeField] private Ship shipPrefab;
    [SerializeField] private ShipData sharedShipData;
    [SerializeField] private float spacing = 1.0f;

    void Start()
    {
        GenerateShips(10, 10);
    }

    public void GenerateShips(int rows, int columns)
    {
        for (int i = 0; i < rows; i++)
        {
            for (int j = 0; j < columns; j++)
            {
                Vector3 position = new Vector3(i * spacing, 0, j *
                spacing);
                Ship newShip = Instantiate(shipPrefab, position,
                Quaternion.identity, transform);
                newShip.Initialize(sharedShipData, 100);
                // 100 が基本数と仮定
                newShip.name = $”Ship_{i * columns + j}”;
            }
        }
    }
}
```

共有データを ScriptableObject に一元化することで、多数の宇宙船インスタンスにわたってメモリフットプリントを削減できます。一方、HP などの固有データは引き続き各宇宙船インスタンスによって個別に管理されます。

これはゲームオブジェクトあたりの節約量としては多くありませんが、シーンに追加する宇宙船が増えるほど効率の良さが顕著になります。類似のオブジェクトが多数ある場合は、このパターンを最大限に活用できます。

Package Manager から Memory Profiler パッケージを追加して、メモリ節約について理解を深めてください。Memory Profiler ウィンドウ (**Window > Analysis > Memory Profiler**) でマネージオブジェクトを検索し、割り当てサイズを決定します。フライウェイトパターンを実装する前と後を比較します。



節約されたメモリを Memory Profiler で比較する。

サンプルシーンでは少数の共有フィールドしか表示されませんが、RPG やストラテジーゲームなど、画面上に多数のユニットが配置された大規模なプロジェクトでは、フライウェイトパターンのメリットが大いに発揮される可能性があります。

基本クラスに共通のプロパティを持つオブジェクトが多数ある場合、フライウェイトパターンを使用して、リソースを節約します。

プレハブとフライウェイト

プレハブシステムはフライウェイトパターンの実装とみなすことができますが、アプローチとスコープが異なります。

— プレハブは、すべてのコンポーネントや階層を含むゲームオブジェクト構造全体を共有し、一方フライウェイトパターンは、特定のデータフィールドやプロパティを選択的に共有できます。

— フライウェイトパターンは特定のゲームオブジェクト構造に縛られていないため、共有データをより柔軟に分離して管理できます。

プレハブは複雑なゲームオブジェクトの再利用に適していますが、フライウェイトは多数のオブジェクトがプロパティのサブセットのみを共有している場合にさらに最適化できます。2 つのアプローチは互いに補完でき、プレハブは全体的な構造の再利用を扱い、フライウェイトはそれらの構造内の共有データフィールドを最適化します。

長所と短所

フライウェイトパターンは、多数のオブジェクトが共通の状態を共有するシナリオに適しています。これは、メモリ消費量を減らすことでパフォーマンスを改善できる、モバイルデバイスのような、リソースに制約のあるプラットフォームで特に便利です。多数のオブジェクトのインスタンス化を必要とするアプリケーションでは、フライウェイトをうまく活用することで、より効果的にスケーリングできます。

ただし、このパターンでは追加のオーバーヘッドと複雑さが生じることに注意してください。シーン内の個々のオブジェクトの管理に加えて、それらの共有状態の管理も必要です。追加のオーバーヘッドを正当化するのに十分なユニット数がある場合にのみ、フライウェイトパターンのメリットが明確に現れます。

また、同じ基本データを共有しなければならないため、ユニットの柔軟性が制限されます。各ユニットを固有のものにするためには、共有データをオーバーライドする必要があり、これは特定のデータセットに適用されたプレハブワークフローに似ています。

その他の例

フライウェイトパターンが提供する機能の多くはプレハブを使用して実現することもできますが、以下のようなケースに適しています。

群衆シミュレーション:例えば、背景に群衆がいるスポーツシミュレーションがあります。このパターンを使用して、モデル、アニメーション、テクスチャーを共有し、ダイナミックで大規模な群衆を構築できます。

キャラクター/武器のスキンとカスタマイズ:多くのゲームでは、プレイヤーがスキンやアタッチメントを使用して武器や装備をカスタマイズできます。これらのアイテムの基本プロパティは、フライウェイトで共有でき、カスタマイズ部分のみを個別に格納します。

レベルアート:森をデザインするときは、樹木の普遍的なプロパティをすべて取り入れ、ベースの Tree クラスに格納します。そのため、サブクラス (PineTree、MapleTree など) でこれらを繰り返す必要はありません。

共有データを持つ何千ものオブジェクトがゲームに含まれるシナリオ (激しいシューティングゲームにおける無数の弾丸や、アクションストラテジーゲームにおける軍隊など) では、代わりに Unity の Data-Oriented Technology Stack (DOTS) を使用することを検討してください。DOTS はマルチスレッディングとデータ依存関係の軽減に注力することで、より優れたパフォーマンスの最適化を実現できます。DOTS e-book for advanced Unity developers (Unity の上級開発者向け Data-Oriented Technology Stack (DOTS)) では、このスタックに含まれる各パッケージとテクノロジー、および関連する概念について詳しく解説しています。

他のデザインパターンと同様に、実装する前にプロジェクトの具体的なニーズを評価しましょう。そして、どのパターンが最も良い結果をもたらすかをチームで判断します。

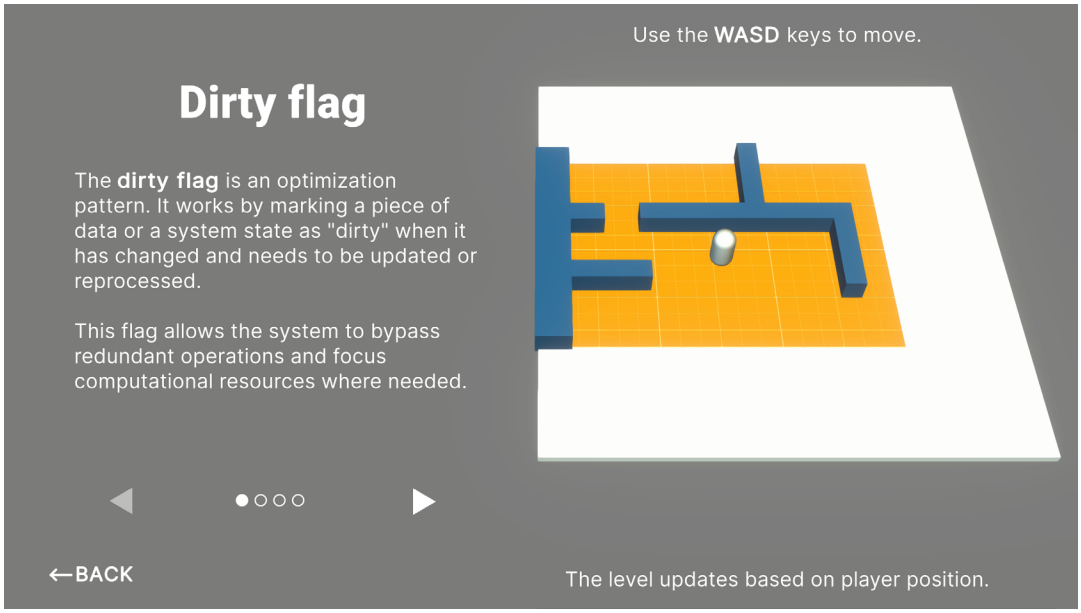
ダーティフラグ (Dirty flag)

ゲーム開発が複雑になり始めたり、そうでなくとも計算コストが高くなったりすることがあります。**ダーティフラグ** (Dirty flag) パターンは、シーンで大量の計算や更新が発生する場合に役立ちます。

ダーティフラグとは、オブジェクトが最後に処理またはレンダリングされてから、その状態が変更されているかどうかを示す単なるブール値です。オブジェクトが "ダーティ" であれば更新され、そうでなければスキップされ、計算リソースを節約できます。

一般的なユースケースは、複雑な階層を移動したり、サイズの大きなシーンファイルを管理することです。ダーティフラグは、特定のオブジェクトが "ダーティ" とマークされるまで計算を最小限に抑えるのに役立ちます。例えば、子 Transform は親 Transform またはルート Transform が必要になるまで更新を無視することができます。これにより、状態が頻繁に変化する動的オブジェクトの不要な計算を最小限に抑えることができます。

ダーティフラグパターンを使用すると、オブジェクトの状態が変化しそうな場所に戦略的にチェックポイントを配置できます。これはイベントハンドラー、物理演算のアップデート、アニメーションシステムといったところでしょう。チェックポイントを利用して、プレイヤーのアクションやゲームイベントに応じてゲームワールドのサブセットのみが更新されます。 節約されるリソースはすべて、オーバーヘッドの削減に役立ちます。



ダーティフラグパターンを使用して不要な更新を防ぐ。

例：サンプルプロジェクト

大規模なオープンワールドゲームを考えてみましょう。ゲーム環境全体を一度にロードすることは、多くの場合リソース的に不可能です。代わりに、一般的な方法は、プレイヤーが現在見ているゲームワールドの一部だけをロードすることです。

これを管理するために、ゲームワールドを小さな Unity シーンに分割し、必要に応じてそれぞれをロードします。ただし、このシーンのロードプロセスは比較的時間がかかり、短い一時停止によってゲームプレイが中断してしまうことがあります。

そのため、必要なときにのみゲームワールドを更新するべきです。この例では、特定の条件が満たされた場合にのみ実行される仕組みを導入しています。更新の準備が整うと、ワールドにダーティフラグを立てます。そうでない場合は、更新ループは負荷の高いロジックをスキップします。

プレイヤーがステージ内を移動すると、このゲームワールドはプレイヤーが現在のセクターの境界付近に移動したときにのみ更新されます。これにより、コンピューティングリソースを節約し、シームレスなゲームプレイ体験を実現します。

サンプルの実装で、このパターンを適用する方法の 1 つを確認してください。

- ゲームワールドは複数のセクターや地域に分かれており、それぞれに関連コンテンツがあり、プレイヤーが近くにいるときにロードされる必要があります。
- マネージャースクリプトはプレイヤーの動きを追跡し、プレイヤーの現在地に基づいて関連セクターを判断します。
- 各セクターにはダーティフラグがあり、プレイヤーの接近度やインタラクションに基づいて、そのコンテンツをロードまたはアンロードの必要があるかを示します。

シーンのロードは比較的負荷の高い操作であり、ダーティフラグパターンを使用すると、負荷の高いゲームワールドの更新が必要なときにのみ実行されるようになります。ここでは、ゲームワールドのどの部分をランタイムにロードするかを GameSectors スクリプトが管理します。

```
public class GameSectors:MonoBehaviour
{
    public Player player;
    public Sector[] sectors;

    private void Update()
    {
        foreach (Sector sector in sectors)
        {
            bool isPlayerClose = sector.IsPlayerClose(player
            .transform.position);

            // セクターの状態を変更する必要があるかどうかをチェックする
            If (isPlayerClose != sector.IsLoaded)
            {
                sector.MarkDirty();
            }

            // ダーティフラグに基づいてセクターを更新する
            if(sector.IsDirty)
            {
                If (isPlayerClose)
                {
                    sector.LoadContent();
                }
                else
                {
                    sector.UnloadContent();
                }

                // ダーティフラグをリセットする
                sector.Clean();
            }
        }
    }
}
```

このサンプルシーンは、プレイヤーの接近に応じてマテリアルの変化やシーンのロード/アンロードを視覚化する方法を示しています。ダーティフラグパターンを使用することで、コンテンツのロードとアンロードの処理は、プレイヤーがセクターに十分近づいたり離れたりしたときにのみ実行されます。

これによって、アプリケーションが一度に表示するゲームワールドのセクションが限られるため、不要な処理やメモリ使用量を最小限に抑えます。

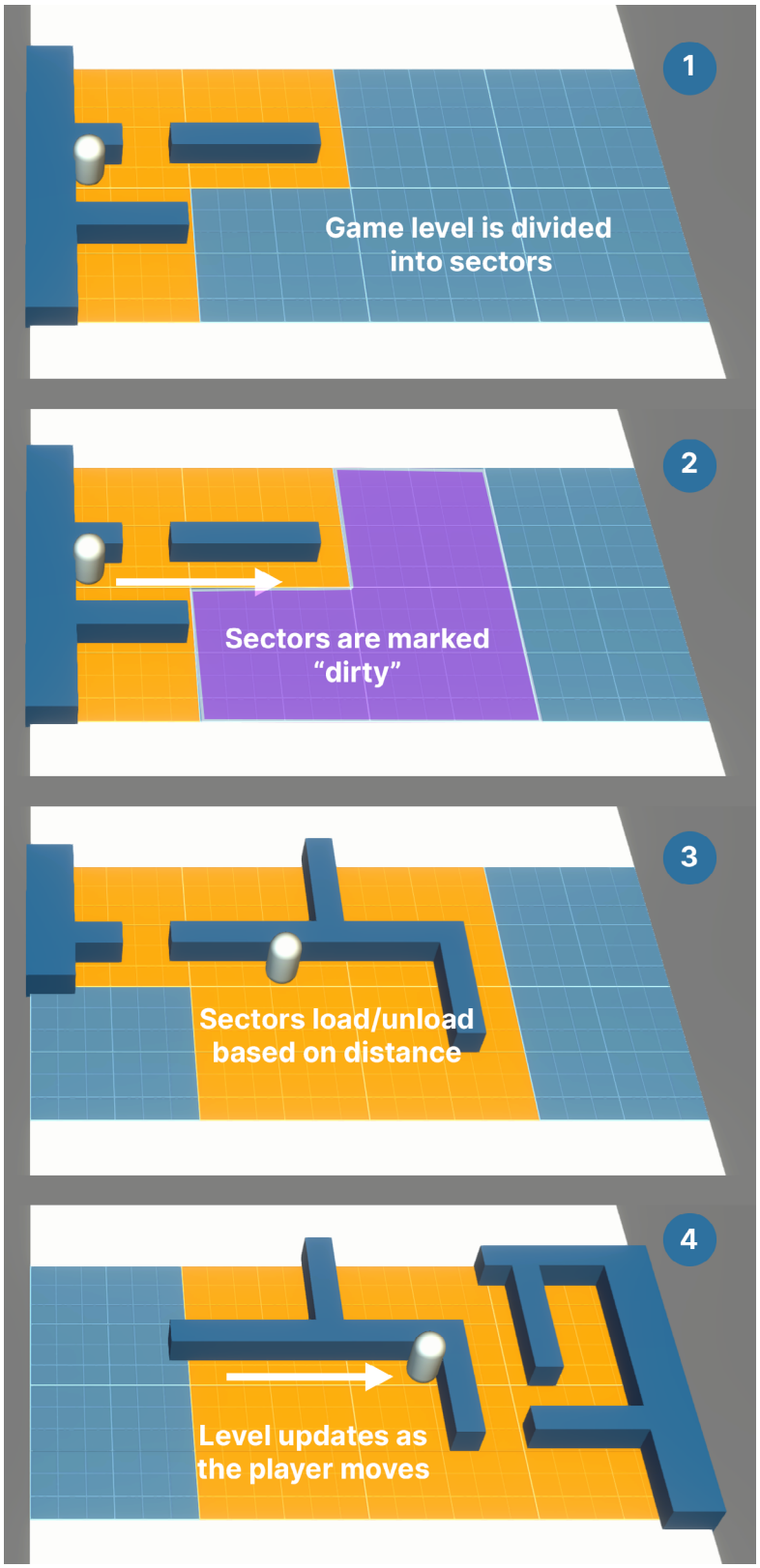
この例の各 Sector は、アセットをロードまたはアンロードするしきい値として、プレイヤーとの独自の距離を保持しています。サンプルの一部を次に示します。

```
public class Sector :MonoBehaviour
 {

     [Tooltip(“Minimum distance to load”)]
     public float m_LoadRadius;
     ...

     public bool IsLoaded { get; private set; } = false;
     public bool IsDirty { get; private set; } = false;

     void Awake()
     {
         ...
         Clean();
         IsLoaded = false;
     }

     public void MarkDirty()
     {
         IsDirty = true;
         Debug.Log($”Sector {gameObject.name} is marked dirty”);
         ...
     }

     public void LoadContent()
     {
         IsLoaded = true;
         ...
         // プロジェクトからシーンコンテンツをロードするロジック
     }

     public void UnloadContent()
     {
         IsLoaded = false;
         ...
         // シーンコンテンツをアンロードするロジック
     }

     public bool IsPlayerClose(Vector3 playerPosition)
     {
```

```
            return Vector3.Distance(playerPosition, transform.position
            + m_CenterOffset) <= m_LoadRadius;
        }

        public void Clean()
        {
            IsDirty = false;
        }
        ...
```

この設定では、プレイヤーがステージを進めるにつれて、SceneLoader コンポーネントがゲームの一部を選択的にロードおよびアンロードします。

このシーンのカメラはトップダウンビューで、このシステムの仕組みを説明します。実際のアプリケーションでは、カメラがプレイヤーを追いかけ、限られた有効視野 (FOV) しか映さないところを想像してみてください。このシナリオでは、プレイヤーの視界に直接入るエアリのみを選択的にロードして表示できました。



"ダーティ" とマークされた後のステージ更新部分。

さらに最適化することもできますが (Update を一切使用しないことも)、この例ではダーティフラグを使用して SceneManagement API への高コストな呼び出しを減らすシンプルなケースを示しています。

長所と短所

ダーティフラグパターンは、大規模なシミュレーションやプロシージャルコンテンツを生成するシステムで特に有用です。コストのかかる処理は必要なときにのみ実行されるため、不要な計算を最小限に抑え、メモリ効率を高めることができます。モバイルデバイスなど、メモリが限られているプラットフォームでは不可欠です。

マイナス面として、ダーティフラグはコンポーネント間の密結合を引き起こす可能性があることに注意してください。これにより、依存関係とリスクのレイヤーが追加されます。また、ダーティフラグがセットされるまで更新が延期されるため、必要な更新が行われるまで、アプリケーションの状態が古くなったように見えることがあります。

ダーティフラグ対ダーティビットとキャッシング

"ダーティビット" や "キャッシュ" という用語は、ダーティフラグパターンと並んでよく話題になります。コンピューティングでは、それぞれが異なる役割を果たします。

ダーティフラグパターンは、アプリケーションの不要な更新を減らすことを目的とした、高レベルのソフトウェアデザイン戦略です。

一方ダーティビットは、システムのプログラミングで使用される下位レベルのインジケーターで、変更されたメモリページに、置き換え前に更新が必要であることを知らせる信号です。

キャッシュのより広範な手法では、データの一時的なコピーを保存することでデータ取得速度を向上させます。多くの場合、ダーティフラグやダーティビットを利用して、キャッシュされたデータを最新の状態に保ちます。

その他の例

ダーティフラグは、高コストの計算や処理の影響を最小限に抑えたい以下のような場合に使用します。

複雑な階層変換。アニメーション化されたキャラクターの子 Transform がある場合は、親 Transform が更新されるまでアニメーションを最小化します (上半身が動いている場合は下半身のみ更新する、など)。ユニットが編成されているストラテジーゲームの場合、ユニットの動きを更新するのは、グループの編成全体にダーティのマークが付けられた後です。

物理シミュレーション:複雑な相互作用や多数のオブジェクトが関与する物理シミュレーションでは、パフォーマンスを最適化するためにオブジェクトの状態 (位置、速度、外力など) が変化したときにのみ再計算します。

経路検索:AI エージェントの経路の再計算は高コストになることがあります。ダーティフラグを使用して、障害物が移動したときやターゲットの場所が変わったときにのみパスを更新します。

プロシージャルコンテンツの生成:シーンロードの例と同様に、このパターンは、プレイヤーの動きやゲームイベントなどの特定のトリガーに基づいて、プロシージャルテレインを再生成するタイミングをシステムに指示できます。

UI レイアウト:複雑な UI システムでは、特定の条件が変わる (ウィンドウのサイズ変更、コンテンツの更新など) と、要素の配置を変える必要がある場合があります。ダーティフラグパターンを使用すると、必要なときにのみレイアウトを更新できるため、一定の再計算を回避できます。例えば、UI Toolkit の VisualElement には MarkDirtyRepaint メソッドが含まれています。同様に、EditorUtility には ClearDirty メソッドと SetDirty メソッドがあります。

まとめ

ソフトウェアパターンに触れるのが初めての方にとって、Unity 開発で直面する可能性がある最も一般的ないくつかのパターンの理解を深めるために、このガイドが役に立てば幸いです。

プレハブがスポーンされるファクトリーであっても、AI 用のステートパターンであっても、これらの手法を必要に応じてすぐに活用できるようにしておいてください。デザインパターンを適用するタイミングと方法を認識しておくことで、Unity から課せられる次の課題に対処するのに役立ちます。もちろん、特定のパターンを無理に当てはめることに陥らないようにしてください。パターンを使用しないことも、パターンを使用することと同じくらい重要です。

デザインパターンはワークフローをスピードアップするのに役立ち、正しく適用することで繰り返し発生する問題を手際よく解決できます。これにより、他にはない楽しい体験をプレイヤーのために制作するという重要なことに集中できます。

すでにあるものゼロから作成する必要はありませんが、そこに自分の持ち味を与えるのはもちろんかまいません。

その他のデザインパターン

このガイドで紹介しているのは、コンピューティングやゲーム開発においてよく知られているデザインパターンの一部をサンプリングしたものにすぎません。細かいところまでは述べませんが、ここでは皆さんにとって役立つ可能性があるその他のパターンを簡単に紹介します。

- **アダプター (Adaptor)**:2 つの関連性のないエンティティが連動するように、それらの間にインターフェース (ラッパーとも呼ばれます) を設定します。
- **ダブルバッファ (Double buffer)**:計算を終わらせる間に、2 セットの配列データを保持することができます。これにより、一方のセットのデータを、もう一方を処理している間に表示できます。これは、プロシージャルシミュレーション (セルオートマトンなど) や単に何かを画面にレンダリングしているときに便利です。
- **インタプリタ/バイトコード (Interpreter/Bytecode)**:MOD 作成のサポートを追加する、つまりプログラマーではないユーザーにゲームの拡張を許可する場合に、ユーザーが外部のテキストファイルで編集できる簡易言語を作成することができます。バイトコードコンポーネントによってそのインタプリタ型言語が C# のゲームコードに翻訳されます。
- **サブクラスサンドボックス (Subclass sandbox)**:動作が異なる類似のオブジェクトが複数ある場合に、それらの動作が親クラスで保護されるように定義できます。そうすることで、子クラスを組み合わせて新しい組み合わせを作成することができます。
- **タイプオブジェクト (Type object)**:さまざまなタイプのゲームオブジェクトがある場合は、それぞれにサブクラスを作成する代わりに、考えられるすべての動作を 1 つの抽象クラスや親クラスで定義します。個々のオブジェクトの特性を、コードを変更することなくカスタマイズできる、別個のデータファイル (ScriptableObject など) に分けます。例えば、これによってすべて同じクラスから派生する一見異なるアイテムのインベントリを作成することができます。ゲームデザイナーは、プログラマーの支援を受けることなく、そのデータファイルをカスタマイズして、各アイテム (RPG に登場する武器など) に個性を持たせることができます。
- **データ局所化 (Data locality)**:メモリに効率的に格納されるようにデータを最適化すると、パフォーマンス面で恩恵が得られます。クラスを構造体に置き換えることで、よりキャッシュに適したデータにすることができます。Unity の ECS と DOTS アーキテクチャにこのパターンが実装されています。
- **空間分割 (Spatial partitioning)**:大規模なシーンやゲームワールドで、特殊な構造を使用してそれぞれの位置ごとにゲームオブジェクトを整理します。Grid、Trie (Quadtree、Octree)、バイナリ検索ツリーはすべて、ユーザーがより効率的に分割して検索するのを手助けする手法です。
- **デコレーター (Decorator)**:既存の構造を変えることなく、オブジェクトに責任を追加することができます。デコレーターを使用すると、特別なアビリティを与えたり、ゲームオブジェクトに変更を加えたりすることができます。これにより、武器の基本クラスを変更することなく、その武器に特典を付与することができます。

- **ファサード (Facade)**:より複雑なシステムに、統一されたシンプルなインターフェースを提供します。1 つのゲームオブジェクトに AI、アニメーション、サウンドの各コンポーネントが別途存在する場合は、それらのコンポーネントをラッパークラスで囲むとよいでしょう (PlayerInput や PlayerAudio などの管理を担う Player コントローラークラスを想像してみてください)。このファサードにより、元のコンポーネントの細かい部分が隠され、使いやすくなります。
- **テンプレートメソッド (Template method)**:このパターンにより、アルゴリズムの完全なステップがサブクラスに委任されます。例えば、アルゴリズムやデータ構造の大まかな骨組みを抽象クラスに定義し、そのアルゴリズムの全体的な構造を変更することなく、特定の部分をオーバーライドすることをサブクラスに許可することができます。
- **合成 (Composite)**:この構造デザインパターンを使用して、オブジェクトをツリー構造に整理してから、結果として生成される構造を個々のオブジェクトのように扱います。単純要素と複合要素 (リーフとコンテナ) の両方からツリーを構築します。ツリー全体に対して同じ動作を繰り返し実行できるように、各要素に同じインターフェースが実装されます。

注:この e ブックのすべての Wikipedia のリファレンスは、Creative Commons のライセンスを通じて作成されました:https://creativecommons.org/licenses/by-sa/3.0/。本書で引用されている Wikipedia の執筆者が当社の取り組みを支持しているわけではありません。

Unity プログラマー向けの一連の高度なリソース

このプログラミングデザインパターンガイドは、経験豊富な Unity プログラマー向けに作成された一連のリソースの 1 つです。このシリーズのその他の e ブックは以下のとおりです。

1. Create a C# style guide:Write cleaner code that scales (C# スタイルガイドの作成: スケーラブルでクリーンなコードの記述)
2. Create modular architecture in Unity with ScriptableObjects (Unity で ScriptableObject を使用してモジュラー・ゲーム・アーキテクチャを作成する)

Unity の best practices hub (ベストプラクティスハブ) または Unity ドキュメントの the best practices page (ベストプラクティスページ) から、すべての e ブック (および多くのハウツー記事) を入手できます。

unity.com