ゲームのための UNITY

E ブック

# ゲームプログラミングのパターンを活用してコードをレベルアップさせる

2021 LTS EDITION

Unity®

# コンテンツ

# デザインパターン入門

Unity を使用して制作を進める際に、すでにあるものを一から作成する必要はありません。ほとんど、他の誰かによって既に使おうとしているものが作成されているためです。

ソフトウェアデザインに関して皆さんが翻弄されているどの問題についても、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。毎回先人へアドバイスを直接求めることはできませんが、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は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いて見つかる一般的な問題に対して汎用的に使える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へ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、デザインパターンとは、手持ちのツールボックスへの追加できるツール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簡略化されており、誰でも手に取りやすいように技術専門用語の使用は減らすよう努めているものの、実際に使用する前に C# の基礎に関する実用的な知識が求められます。

まだデザインパターン初心者の方や、簡単にさらっておきたい方向けに、このガイドにはゲーム開発のどのような場面でそのデザインパターンを適用できるかに関しての一般的なシナリオもご紹介しています。別のオブジェクト指向言語（Java、C++ など）からの切り替えを検討している方向けに、ここのサンプルにおいては、とりわけ明確に Unity に合わせてパターンを導入する方法について紹介していきます。

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より規模の大きなアプリケーションを構築するのに役立ちます。プロジェクトに組み込むことで、コードの可読性を高め、コードベースが整理された状態に保たれます。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がわかります。

こうすることで、すでにあるものを一から作成することを止めて、新しいものの制作に取り組むことができます。

**協力者**

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 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。

## 当ガイドの使用方法と KISS の原則

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皆さんに提示することです。このガイドで紹介しているソフトウェアデザインのいくつかのパターンは、Unity 開発用に構成されています。

ここに含まれているサンプルプロジェクトには、文脈内におけるコードの一部が提示されています。対応するシーンを使用して、デザインパターンとその土台となる原則を色々探ってみてください。

ただし、そういった例を確認する際には、一括全面的に問題へ対処する「正しい方法」は存在しない、ということを忘れないようにしてください。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、このガイドのすべてを KISS の原則（「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。複雑さを追加するのは、必要な場合のみに留めるようにしましょう。

すべてのデザインパターンには、追加の構造を保持しなければならなかったり、最初に設定する項目が増加してしまう、など、いわゆるトレードオフがつきものです。実装する前に、それ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が定かでない場合は、そのパターンがより自然にはま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、奇抜であるからという理由でパターンを使用せず、必要なときに使用してください。

こうすることで、より優れたソフトウェアの開発を手助け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 の原則

いきなりパターンの紹介に入る前に、それらの働きに影響するいくつかのデザイン原則について見て行きましょう。

SOLID はソフトウェアデザインに関わる以下の 5 つの核となる原則の頭文字を取ったものです。

- Single responsibility（単一責任）
- Open-closed（オープン / クローズド）
- Liskov substitution（リスコフの置換）
- Interface segregation（インターフェース分離）
- Dependency inversion（依存性逆転）

各コンセプトについて確認し、コードをより柔軟でわかりやすく、メンテナンスもしやすくするのに、こういったコンセプトがどう役立つの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（その単一の責任のみ）であるべきです。

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 つの責任を担い、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。

すべて分割されず一体化した一枚岩なクラスを構築するのではなく、より小さな部品からプロジェクトを組み立てます。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。

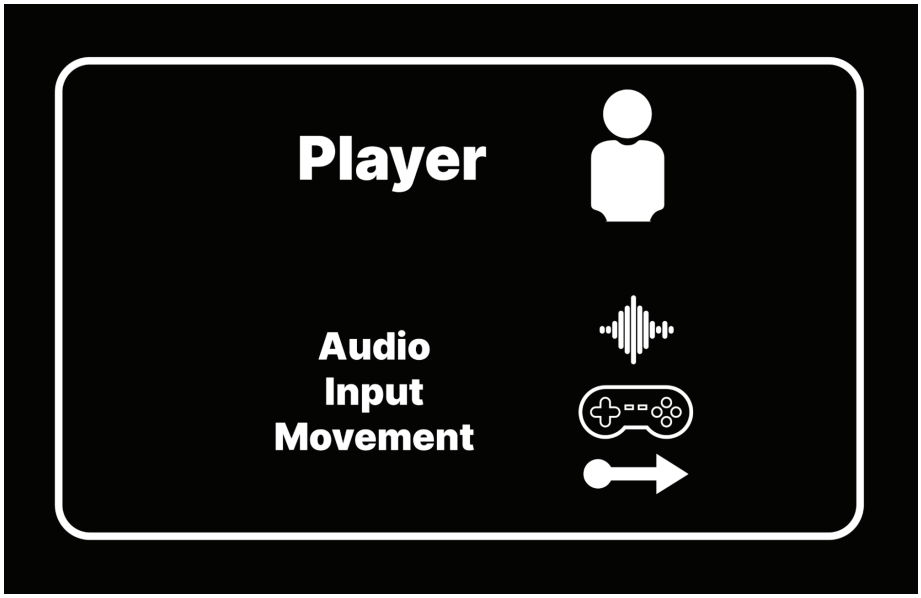
Unity の使用経験がある方であれば、このコンセプトにすでに慣れていることと思われます。ゲームオブジェクトを作成すると、その中にはより小さな各種コンポーネントが保たれます。例えば、以下のようになる場合があります。

- 3D モデルへのリファレンスが格納される 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、回転、位置が格納される Transform コンポーネント
- 物理演算シミュレーションとの対話が必要な場合は 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。1 つのシーンはすべてゲームオブジェクトで構成されます。ゲームはコンポーネント間の対話により成り立っています。

スクリプト化されたコンポーネントはそれと同じように構築します。それぞれを明確に理解できるようにデザインします。その後、それらを連動させて複雑な動作を作成します。

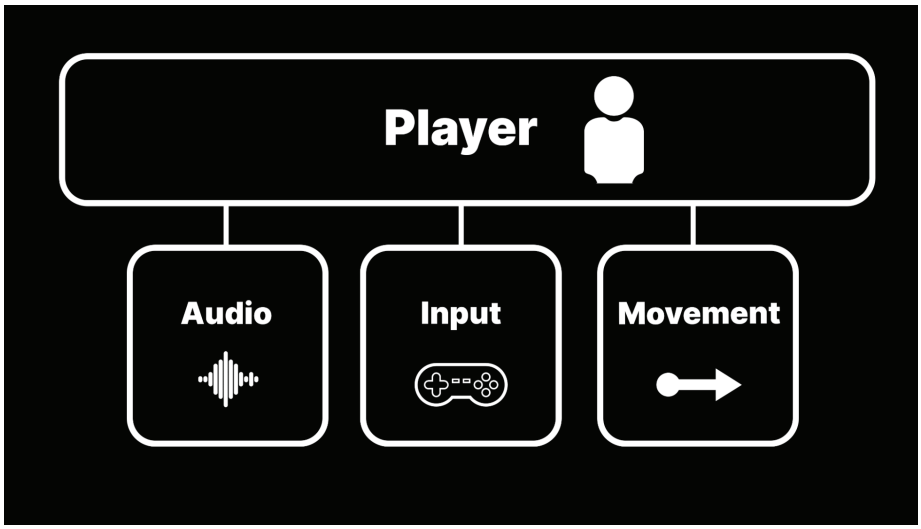
単一責任の原則を無視すると、以下のようなカスタムコンポーネントになってしまうおそれがあります。




複数の責任を担う 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責任が混在しています。プレイヤーが何かに衝突すると音を再生し、入力の管理、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、メンテナンスが難しくなります。Player クラスをより小さなクラスに分割することを検討してください。




単一の責任を担う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 スクリプトによって他のスクリプト化されたコンポーネントが管理されるのは変わりませんが、各クラスは 1 つのことのみを担うようになっています。特にプロジェクトの要件は時間とともに変化するため、このデザインによりコードを改める際に触りやすくなります。

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 つのメソッドのみを使用してクラスを作成するなど、極端に簡略化することは避けてください。

単一責任の原則を適用する際には、以下のことを肝に銘じておくようにしてください。

- **読みやすさ：**クラスは短いほうが読みやすくなります。はっきりとしたルールはありませんが、多くの開発者は 200 行から 300 行を上限に設定しています。どの程度を「短い」と見なすかを、自分やチームの中で決めるようにしてください。このしきい値を超えるときは、より小さなパーツにリファクタリングできないかどうかを判断してください。
- **拡張性：**小さなクラスのほうがより簡単に継承できます。機能を意図せず壊してしまうことを恐れることなく、変更を加えたり置き換えたりすることができます。
- **再利用性：**クラスを小さくかつモジュール式になるように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にとって QOL（生活の質）の向上にどれほどつなが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が、後でトラブルを回避することにつながることが多々あります。

### 「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実際の制作において変更を処理できるか？時間の経過に伴い、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るか？

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。こうすることで、コードの柔軟性を高め、よりスケーラブルかつ読みやすくします。ただし、それには追加作業や計画を立てることが求められます。「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成することは、よりフォーカスしたものに仕上げるということです。1 つのことを担うようにデザインし、他のタスクも担う過度に複雑なものにはしないでください。

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を構築するの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。

## 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。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にクラスを構成します。

古くからある例として、ある形状の面積を計算することが挙げられます。矩形または円の面積を返すメソッドが備わった AreaCalculator というクラスを作成する方法があります。

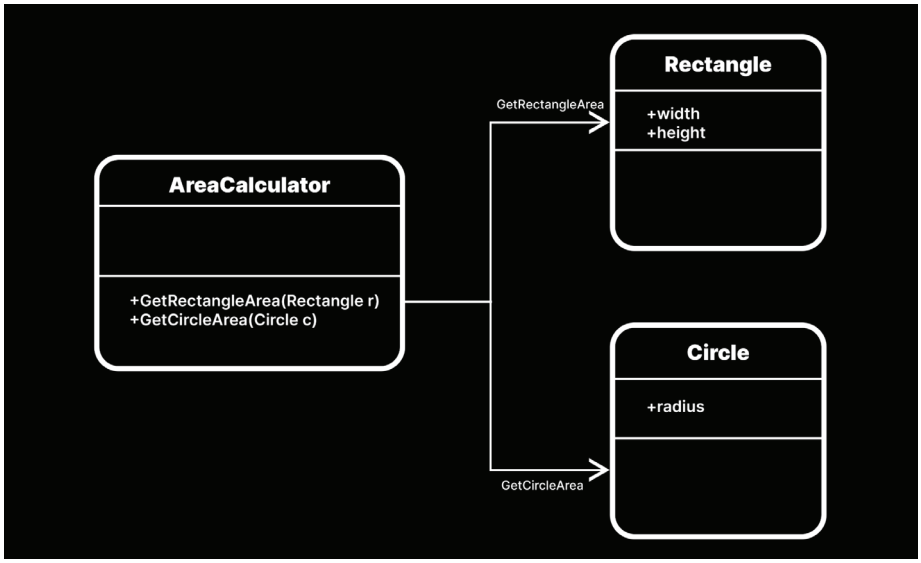
面積を計算するために、Rectangle クラスには Width と Height があります。Circle に必要なのは Radius と円周率の値のみです。

```
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;
}
```

これは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？あと 20 個の形状が必要な場合は？この AreaCalculator クラスではすぐに膨れ上がり、手に負えなくなってしまいます。

Shape というベースクラスを作成し、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）は修正することなく、プログラムを拡張（新しい形状を使用する機能）に対して開く必要があります。現状の AreaCalculator は機能しますが、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でしょうか？

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();
}
```

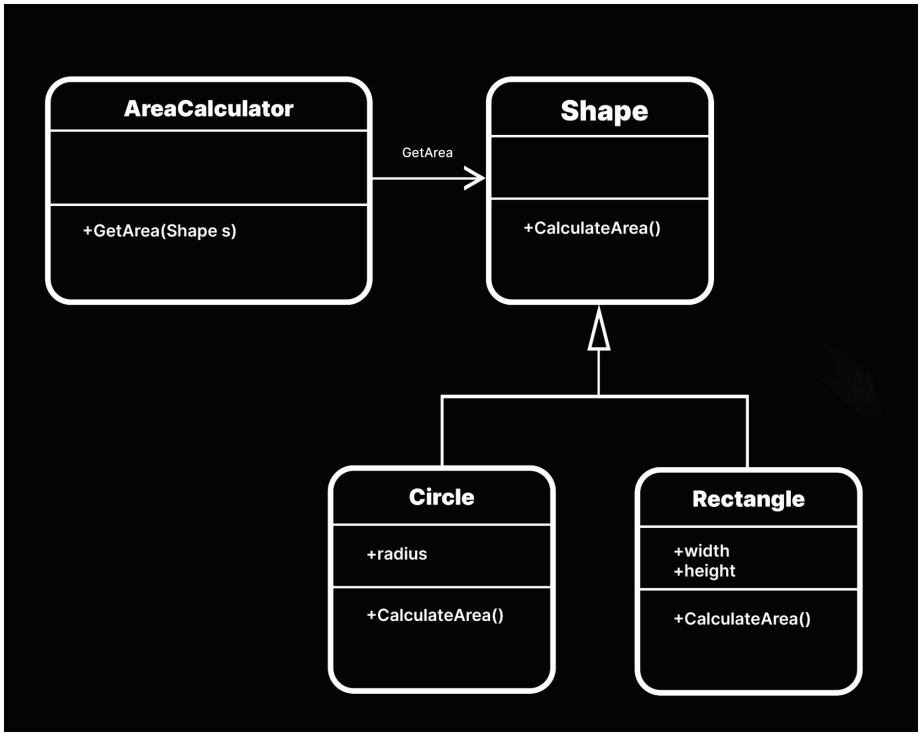
これには CalculateArea という抽象メソッドが含まれています。その後、Rectangle と Circle が Shape から継承されるように設定すると、各形状で独自に面積を計算できるようになり、以下の結果が返されます。

```
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することができます。

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

修正された AreaCalculator クラスで、抽象クラス Shape が適切に実装された任意の形状の面積を取得できるようになりました。これで、元のソースは一切変更せずに、AreaCalculator 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にクラスを改訂する

新しいポリゴンが必要になるたびに、Shape から継承され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により CalculateArea メソッドがオーバーライドされ、正しい面積が返されます。

この新しいデザインにより、デバッグがより簡単になります。新しい形状によりエラーが発生した場合でも、AreaCalculator を再検討する必要はありません。古いコードは変更されないままであるため、新しいコードに正しくないロジックがないかを確認するだけで済みます。

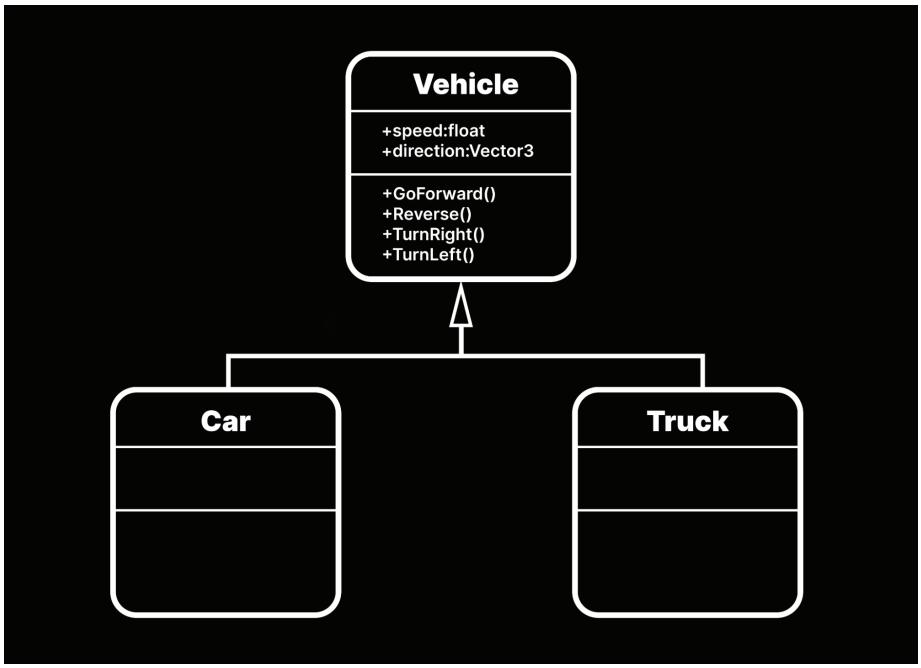
Unity で新しいクラスを作成するときは、インターフェースや抽象化を利用してください。これにより、後で拡張を難しくする扱いにくい switch ステートメントや if ステートメントをロジック内に作成せずに済みます。OCP に配慮したクラスを設定することに慣れることで、長期的に見て新しいコードを追加することが簡単になります。

## 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、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、継承を適用してサブクラスをより堅牢かつ柔軟にする方法を示します。

ゲームに `Vehicle` というクラスが必要であるとします。これがアプリケーションに作成する乗り物のサブクラスの基本クラスになります。例えば、自動車またはトラックが必要になることがあります。



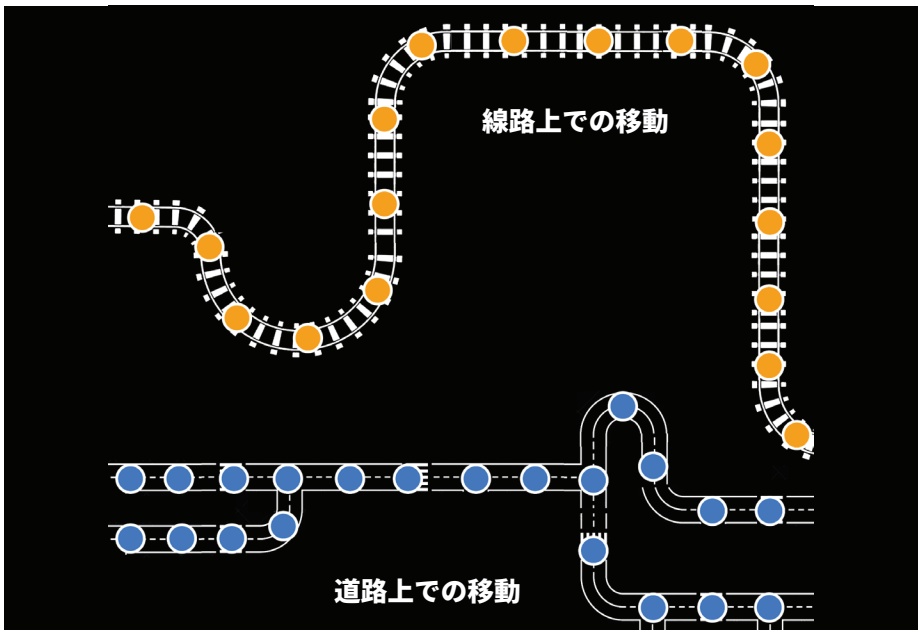

すべて Vehicle から継承されます。

基本クラス（`Vehicle`）が使用できる場所ではすべて、`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必要があります。

Vehicle クラスは以下のようになることがあります。

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

    public void GoForward()
    {
            ...
    }

    public void Reverse()
    {
            ...
    }

    public void TurnRight()
    {
            ...
    }

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

盤面で乗り物を動かすターンベースのゲームを制作しているとします。

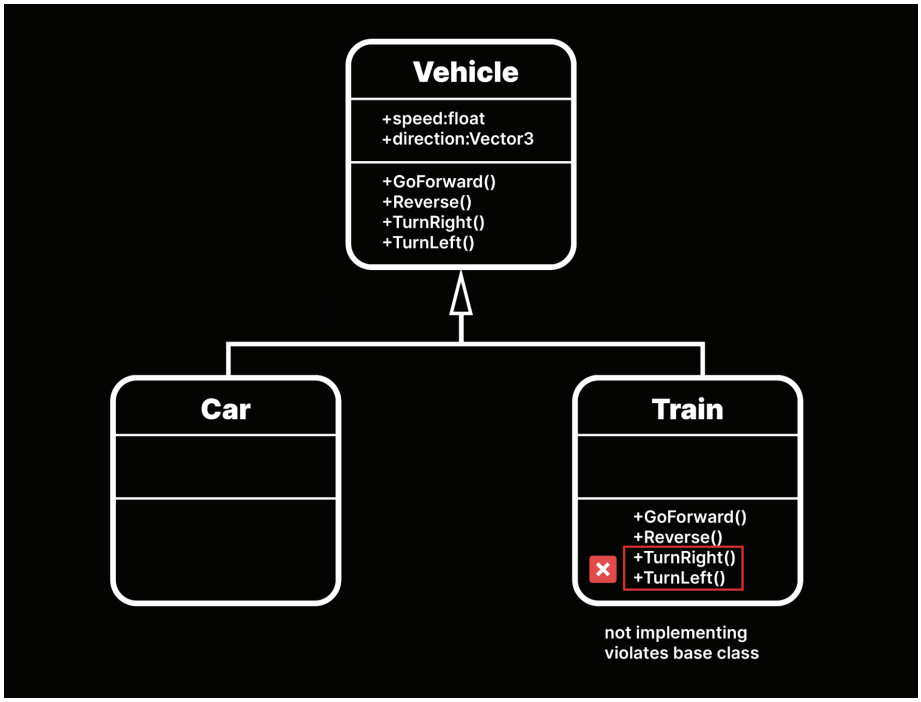



自動車と電車で競うサンプルゲーム

定められた道に沿って乗り物を操縦するための Navigator という別のクラスを作成する方法があります。

```
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、あらゆる乗り物を Navigator の Move メソッドに渡せるようになることが期待されます。そして、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可能性があります。

電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 クラスでは機能しない可能性があります。Navigator の Move メソッドに電車を渡すと、その行に到達したときに、未実装の Exception が渡されます（または何も行われません）。サブタイプをその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 クラスを受け取る場所であればどこにでもそれを使用することを期待します。そうしないと、コードの動作が予測不能にな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。

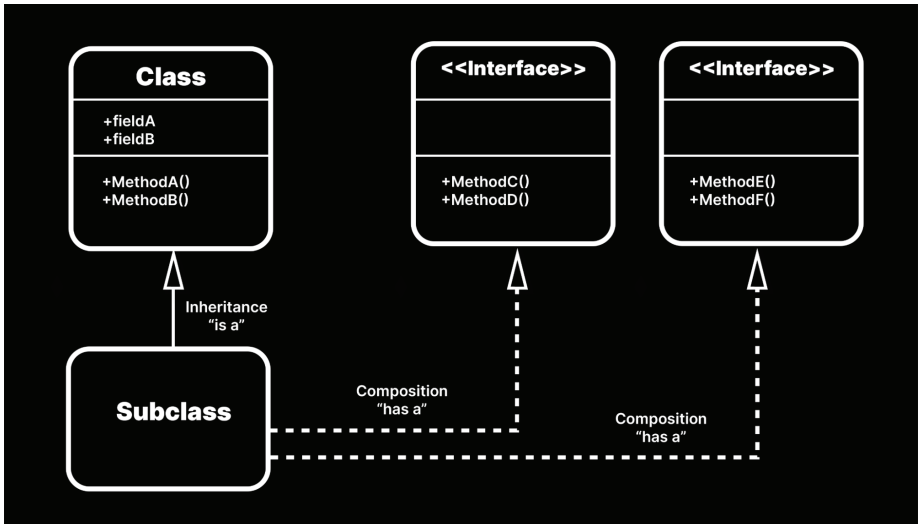
— **サブクラス化するときに機能を削除すると、リスコフの置換原則に違反する可能性が高まる：**NotImplementedException はこの原則に違反した決定的な証拠です。メソッドを空のままにする場合も同様です。サブクラスがその基本クラスのように動作しない場合は、たとえ明示的なエラーや例外が発生していない場合でも、LSP に従っていません。

— **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。基本クラスで表現されるのは、その派生サブクラスの一般的な機能のみである必要があります。

— **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、同じ署名と動作も必要です。

— **クラス階層を確立する前にクラス API を検討する：**すべてを乗り物と見なしていますが、Car と Train は別個の親クラスから継承するほうが理にかなっているかも知れません。実際の分類が常にクラス階層になるとは限りません。

— **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。その後、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。

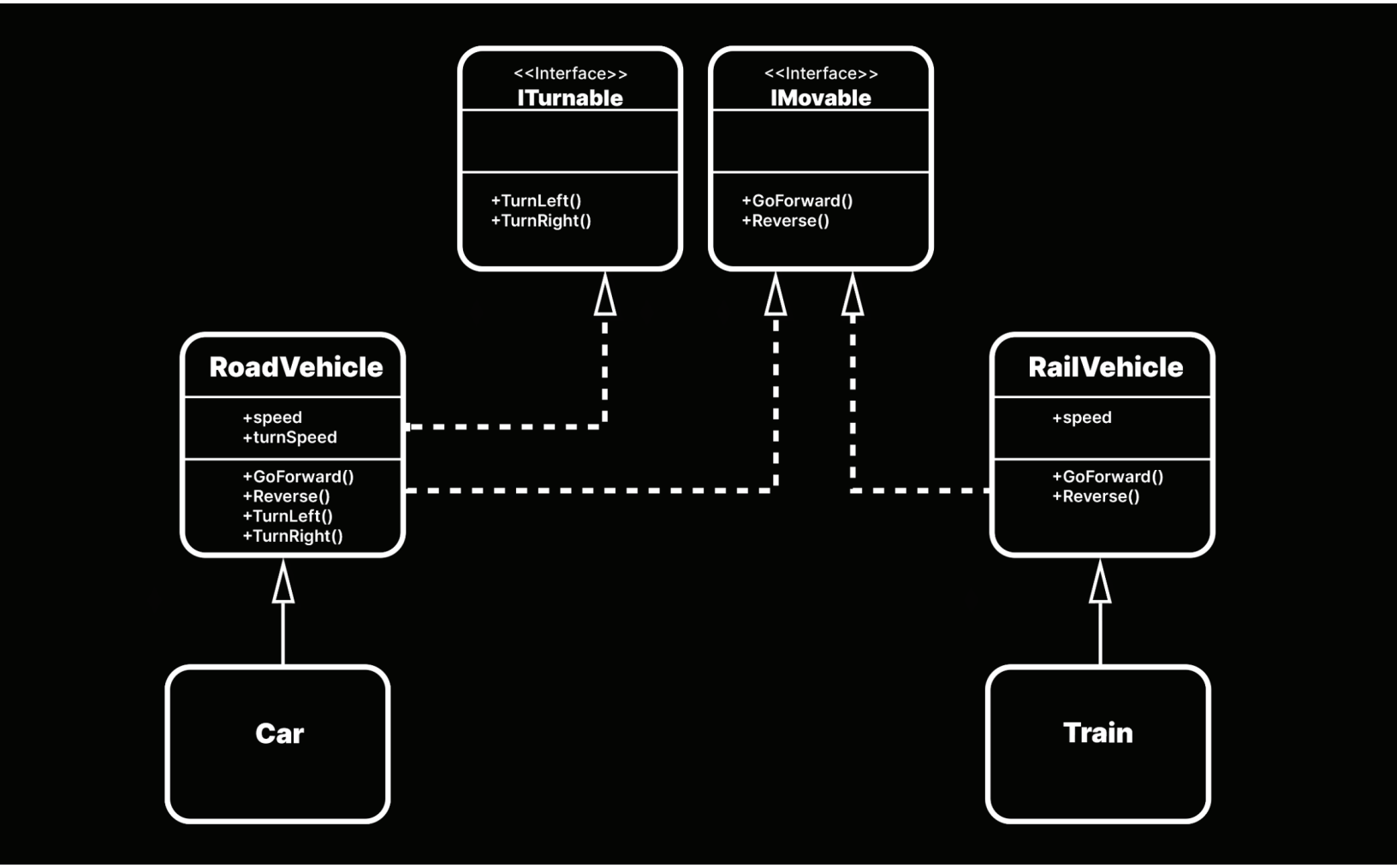



継承よりも合成を優先する

このデザインを修正するには、元の Vehicle 型を解体してから、その機能の大多数をインターフェースに移行します。

```
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から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ようにする必要はそれほどありません。

このような考え方は、直感的には理解しにくいかも知れません。現実世界に関して一定の思い込みがあるからです。ソフトウェア開発において、これは「円 - 楕円問題」と呼ばれます。実際の「is-a」の関係のすべてが継承に変換されるとは限りません。ソフトウェアデザインに担わせる必要があるのは、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。

## 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、クライアントが使用しないメソッドへの依存をクライアントが強制されてはならないという原則のことです。

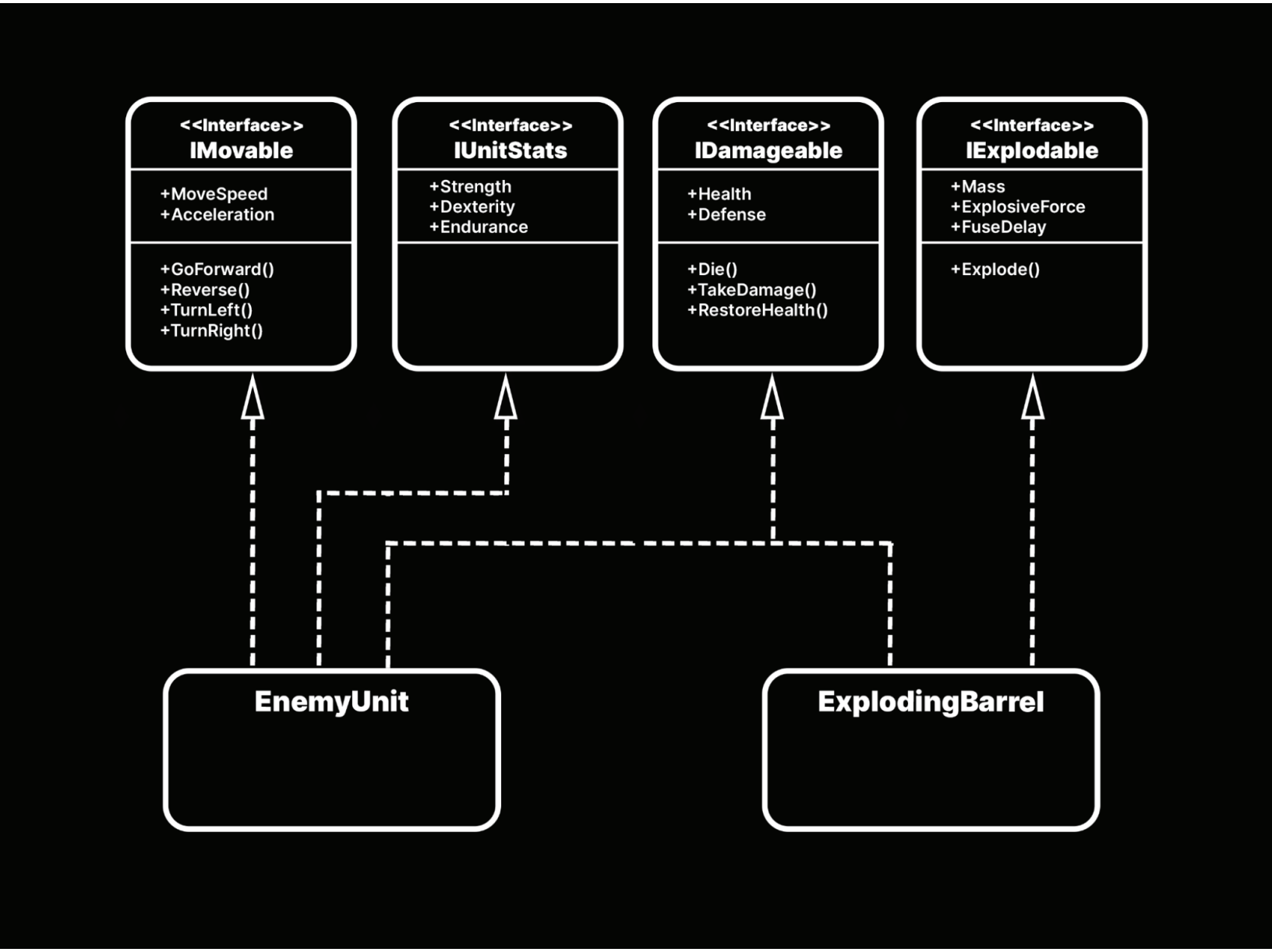
言い換えると、インターフェースが莫大になることを避けることです。クラスやメソッドを短く保つことを示す単一責任の原則と同じ考え方に従います。これにより、柔軟性が最大限に高まり、インターフェースがコンパクトかつフォーカスされている状態に保たれます。

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。ユニットごとに体力やスピードなどのステータスが異なります。以下のように、すべてのユニットに類似の機能を実装することを 1 つのインターフェースに担わせることをお勧めします。

```
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; }

}
```

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、耐久力の概念も必要です。また、木箱や樽には、ゲーム内の他のユニットに紐づ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 1 つのインターフェースを作成する代わりに、よりサイズの小さい複数のインターフェースに分割します。これにより、それらが実装されるクラスで、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より小さなインターフェースに分割します。

```
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 つのクラスには複数のインターフェースを実装できるため、IDamageable、IMoveable、IUnitStats から敵ユニットを構成することができます。

爆発する樽では、他のインターフェースの不要なオーバーヘッドを必要とすることなく、IDamageable と IExplodable を使用できます。

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

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

繰り返しになりますが、リスコフの置換の例と同様に、ここでは継承よりも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、修正を加え、再展開するのが容易になります。

## 依存性逆転の原則

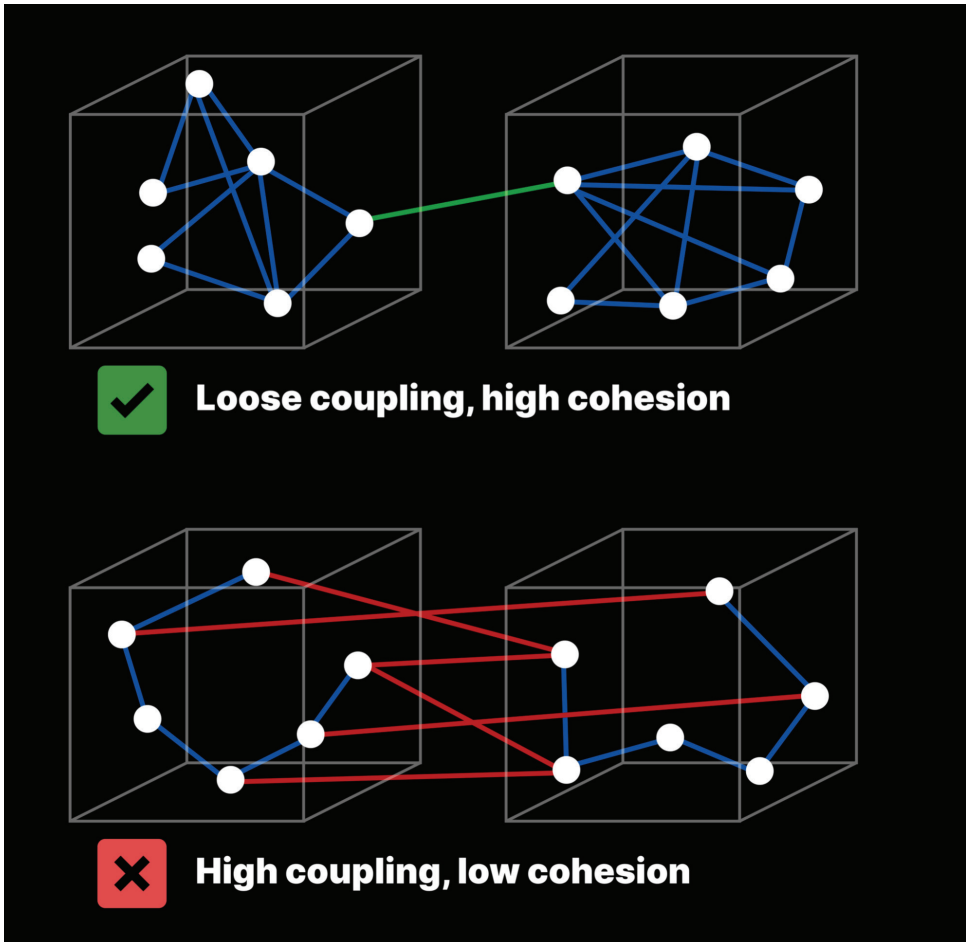
依存性逆転の原則（DIP）とは、上位のモジュールは下位のモジュールから直接インポートしてはならないという原則のことです。両方とも抽象化に依存する必要があります。

これが意味することを解き明かしてみましょう。あるクラスに他のクラスとのリレーションシップがある場合、そこには依存関係または結合が存在します。ソフトウェアデザインにおける各依存関係は、一定のリスクをはらんでいます。

あるクラスに別のクラスのしく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 番目のクラスが破損する（または 2 番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慣行と見なされています。アプリケーションの一部に発生した 1 つのエラーが雪だるま式に増えてしまいます。

理想としては、クラス間の依存関係はできるだけ少なくすることを目指してください。また、各クラスが同時に機能するには、外部への接続に依存するのではなく、そのクラス内部のパーツが必要になります。また、内部またはプライベート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。

最良のシナリオとして、結合を疎にし、凝集度を高めることを目指してください。

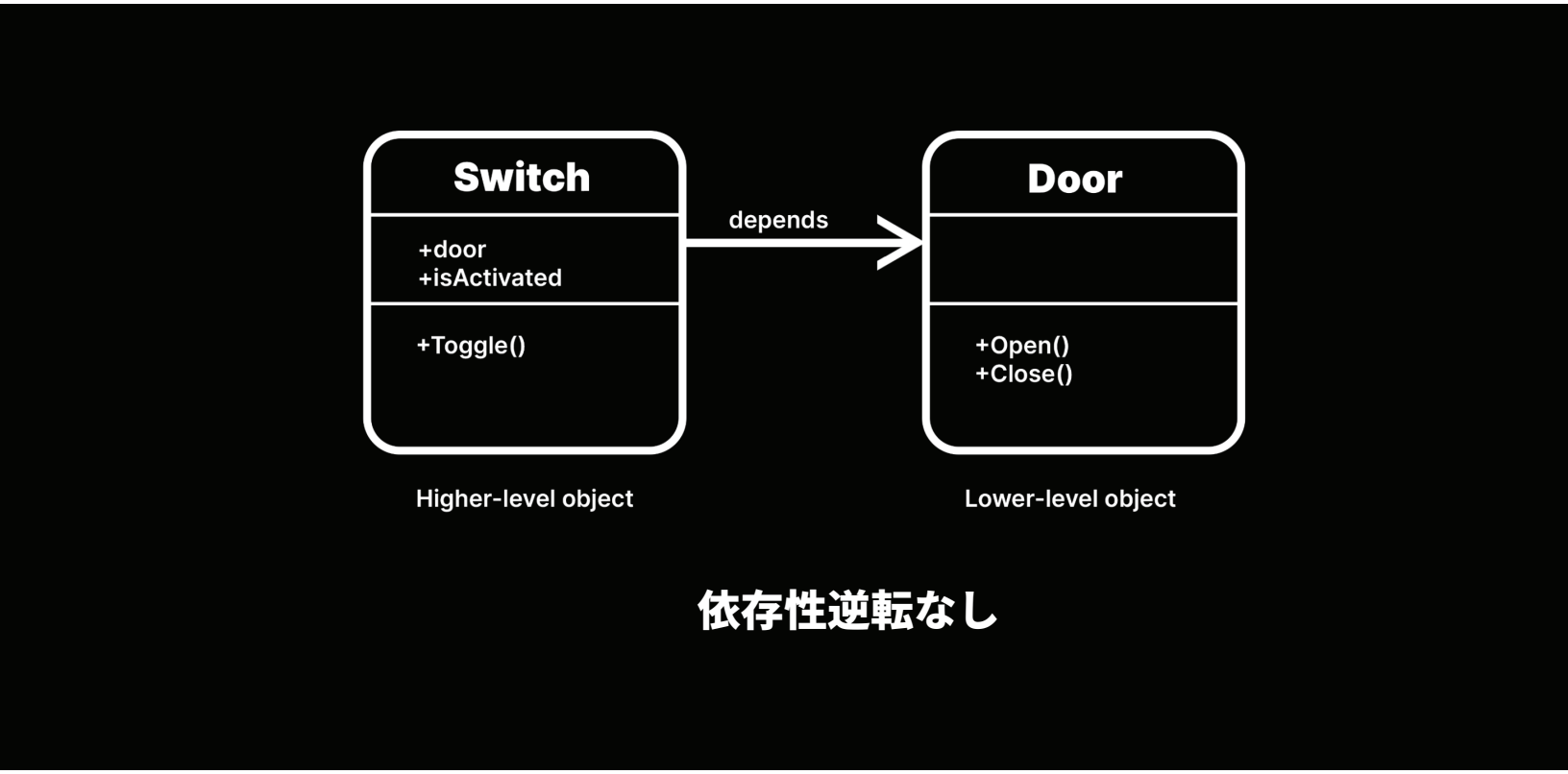



結合を疎にし、凝集度を高めることを目指します。

制作したゲームアプリケーションは修正および拡張できる必要があります。壊れやすく、修正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を調べてください。

依存性逆転の原則は、この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、一部は最初から「上位」であり、一部は「下位」です。上位のクラスでは、何かを実行するのに、下位のクラスに依存します。SOLID によると、これを逆転するとのことです。

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ことをお勧めします。




Switch（上位）は Door（下位）クラスに直接依存します。

上位では、キャラクターを特定の場所に動かし、何かを起こす必要があります。Switch がその役割を担います。

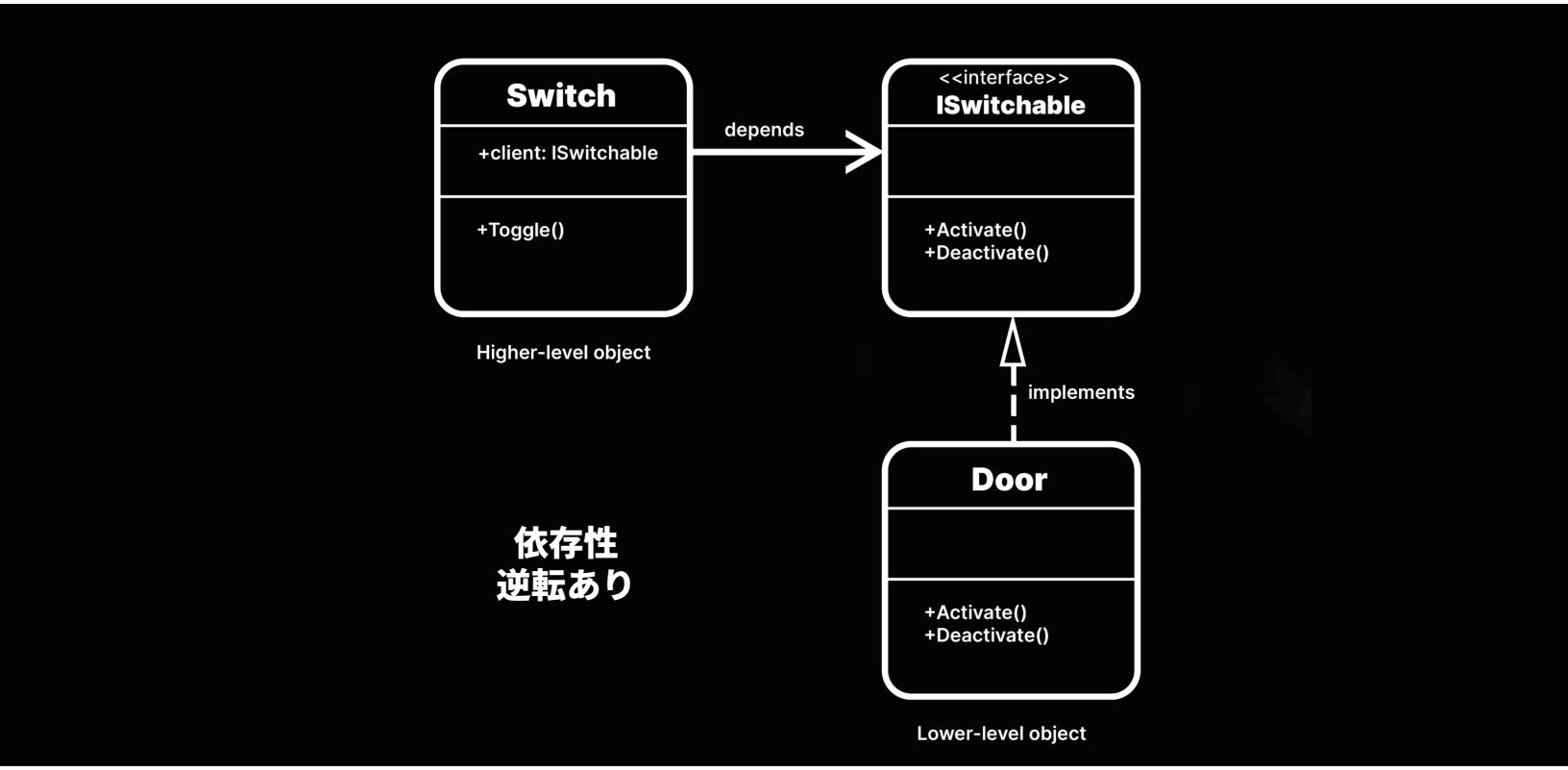
下位にはもう 1 つのクラス Door があり、ドアのジオメトリを開く方法に関する実際の実装が含まれます。シンプルにするために、ドア開閉のロジックを表す Debug.Log ステートメントが追加されています。

```
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 では、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よいでしょうか？

追加のメソッドを 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、元のコードに修正を加えなければなりません。

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 つのクラスの間にある 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 つの 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 が、ドアと直接ではなく、ISwitchable client に応じて、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();
        }
    }
}
```

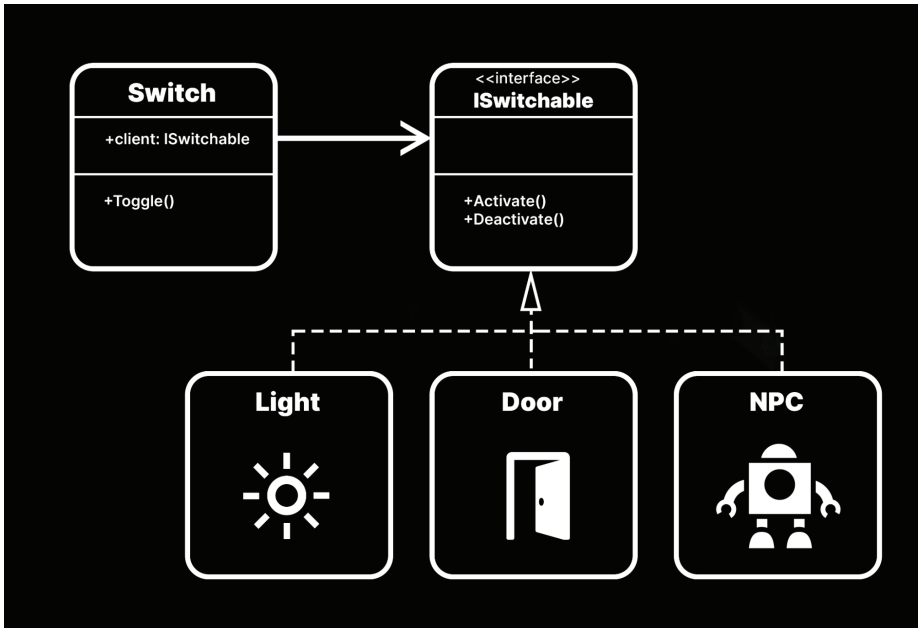
一方で、Door を以下のように手直しして、ISwitchable を実装する必要があります。

```
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。スイッチをドアに排他的にハード配線するのではなく、インターフェースによってそれらの間に抽象化が作成されます。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 でのみ機能していましたが、今では ISwitchable を実装すればどこででも機能します。

これにより、Switch によってアクティベートされるクラスをさらに作成することができます。落とし戸であろうとも、レーザービームであろうとも、上位の Switch が機能します。必要なのは、互換性のある client と、それに実装される ISwitchable だけです。




これで、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りました。

SOLID の他の原則と同じように、依存性逆転の原則で求められていることは、クラス間のリレーションシップを設定するいつものやり方を吟味することです。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。

### 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、同様に抽象クラスを使用したデザイン原則やパターンの多くにも従うことができます。

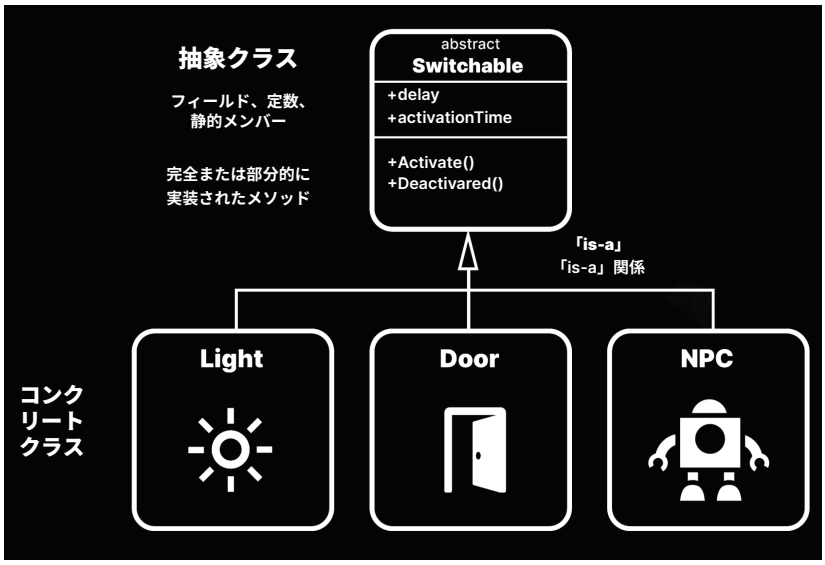
両方とも C# で抽象化を実現する有効な方法です。どちらを使用するかは、そのときのニーズによって変わります。

**抽象クラス**

abstract キーワードを使用すると、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るように、基本クラスを定義できます。

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、コンクリートクラスを派生する必要があります。

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、コンクリートクラス（例：`Light` または `Door`）を `Switchable` という抽象クラスから派生します。

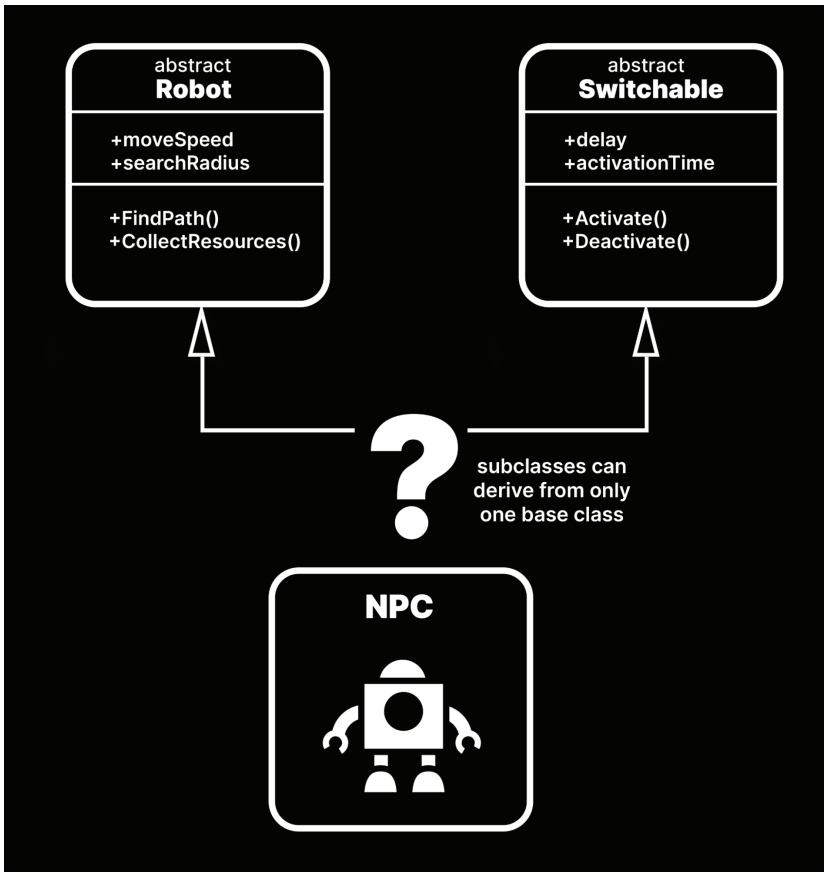



抽象クラスの使用

継承によって「is-a」関係が定義されます。上の図に示すのは、すべてオンとオフの「切り替えが可能」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点です。また、より制約のあるアクセスモディファイア（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、コンクリート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徴を備えた派生クラスを作成する必要性が出てくるまでは、問題なく機能します。C# では、複数の基本クラスから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

ゲーム内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、何を派生す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？

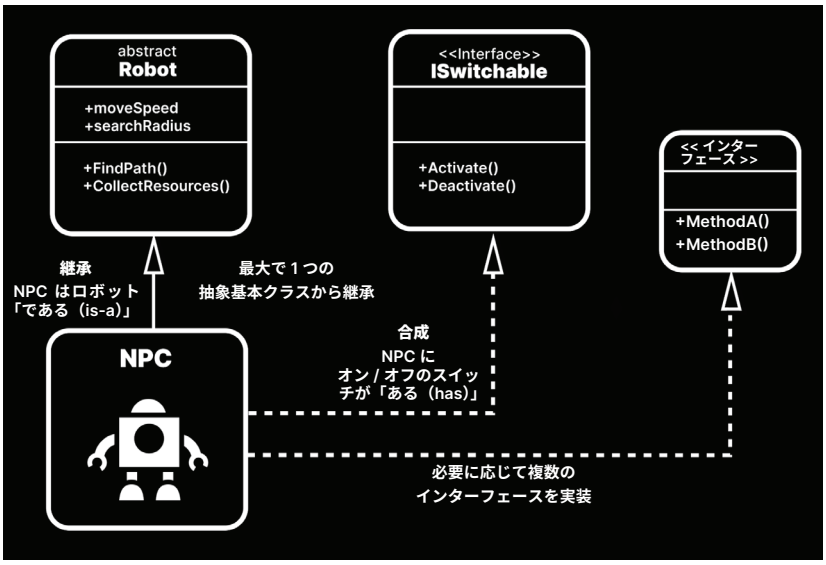
**インターフェース**

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別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のみが含まれます。そのインターフェースが実際に実装されるクラスが、その特定のロジックに肉づけ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（ただし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 |
| すべてのアクセスモディファイア（protected、private など）を使用する | アクセスモディファイアは使用できない（すべてのメンバーは暗示的に public） |

1 つのクラスは最大で 1 つの抽象クラスから継承できますが、複数のインターフェースを実装できることを忘れないでください。

## SOLID について理解を深める

SOLID の原則は、日常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 つのみであるようにしてください。
- **オープン / クローズド**：クラスにすでに備わっている機能は変えることなく、機能を拡張できるようにする必要があります。
- **リスコフの置換**：サブクラスは基本クラスと置換可能でなければなりません。
- **インターフェース分離**：インターフェースに含まれるメソッドは少なくし、短く保つようにしてください。クライアントには必要なものだけを実装します。
- **依存性逆転**：抽象化に依存するようにします。あるコンクリートクラスから別のコンクリートクラスに直接的には依存しないようにしてください。

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にするのを手助けするガイドラインです。SOLID の原則は、スケーリングが必須となるような大規模なアプリケーションと大変相性が良いため、エンタープライズレベルのソフトウェアデザインでは 20 年近く普遍的なものとして扱われています。

場合によっては、SOLID の原則を忠実に守ることで、事前に追加の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となる場合もあります。ただし、往々にして長期的視点での労力の節約とは、つまりその分だけコストを負うことになります。

この原則にどの程度厳密に従うかについては、プロジェクトごとでの自身の判断となります。これは絶対的なものではありません。ここで説明されていない微妙な差異についてや、扱われていない各自の実装方法はあふれるほどの数で存在しています。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するようにしてください。

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みてください。シンプルに保つようにしましょう。しかし、ただスクリプトへ対象の原則を利用するという、その目的のためだけに適用するようなことはしないでください。必要な場面で有機的な効果を発揮させるように努めましょう。

詳細については、Unite Austin での Unity による SOLID に関するプレゼンテーションをご覧ください。

# ゲーム開発のための デザインパターン

SOLID の原則について理解したら、デザインパターンに深く踏み込んでいくことをお勧めします。

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、パターンとは、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というわけではありません。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ブループリント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の付けられた存在となるのです。

## Gang of Four（ギャングオブフォー、GoF）

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）』に端を発します。この著書では、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。

この 4 人の著者は「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の言語全般に適用できます。

GoF が『Design Patterns』の初版を出版したのは 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。エンジニアリング専門分野の多くは、確立されたパターンというものを有しています。ゲーム開発も同様です。

## デザインパターンについて学ぶ

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、学習しておけばそれだけでより優れた開発者になるのに役立ちます。結局、デザインパターンとはよくある問題に対しての一般的なソリューションとなるからこそ、そういった形で分類されているものと言えます。

ソフトウェアエンジニアであれば、通常開発しているその過程でどれも目にしたものだと再確認するものばかりでしょう。すでにそういったパターンのいくつかを無意識のうちに実装しているかもしれません。

それらを自分で見つけ出すよう心がけてください。以下を実践することが役立つ場合があります。

— **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難解な StackOverflow の投稿に埋もれた秘密ではありません。日々の開発で出てく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、どれほどの開発者が取り組んできたかがわかるものです。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、チームとして伝えようとしているときに、略称として機能することがあります。「コマンドパターン」または「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 1 つ以上のパターン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しくみやその作成に関わる思考プロセスについて理解するのに役立ちます。

もちろん、すべてのデザインパターンがすべてのゲームアプリケーションに当てはまるとは限りません。マズローの金槌を手にしてそれらを探すことはしないでください。そのような形で探しても、見つかるのはいわゆる釘だけになってしまうでしょう。

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するのです。それぞれが一定の状況下でメリットをもたらすほか、ある程度の短所も付随します。ソフトウェア開発において下すあらゆる判断には、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？それにより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それを直すことはできますか？

こういったデザインパターンを頭に入れておき、来るべき時が来たら、ゲーム開発における虎の巻としてパターンを取り出して、その場で問題を解決してしまいましょう。

**参考資料**

GoF 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）』に加えて、注目の一冊が Robert Nystrom 氏による著書『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について、必要なことだけが詳しく述べられています。ウェブ版は gameprogrammingpatterns.com にて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。以下のパターンが含まれます。

— **ゲームループ：**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可能性があるため、すべてのゲームにおいて核となるのは、クロック周波数と独立して機能する必要がある無限ループです。速度が異なるさまざまなコンピューターに対応するために、ゲーム開発者は固定時間ステップ（設定した 1 秒あたりのフレーム数）と変動時間ステップを使用する必要がある場合が多く、これによってエンジンで前のフレームからどれだけの時間が経過したかが計算されます。

これは Unity によって処理されるため、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。

— **更新：**ゲームアプリケーションにおいて、各オブジェクトの動作を 1 フレームずつ更新する必要があるケースはよくあります。Unity ではこれを手動で再作成できますが、MonoBehaviour クラスを使用すれば自動的に行われます。Update、LateUpdate、FixedUpdate の中から該当するメソッドを使用して、ゲームオブジェクトとコンポーネントをそのゲームクロックの 1 ティックに変更するだけです。

— **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よくあります。この創作パターンは、他の類似のオブジェクトを作成する目的で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。

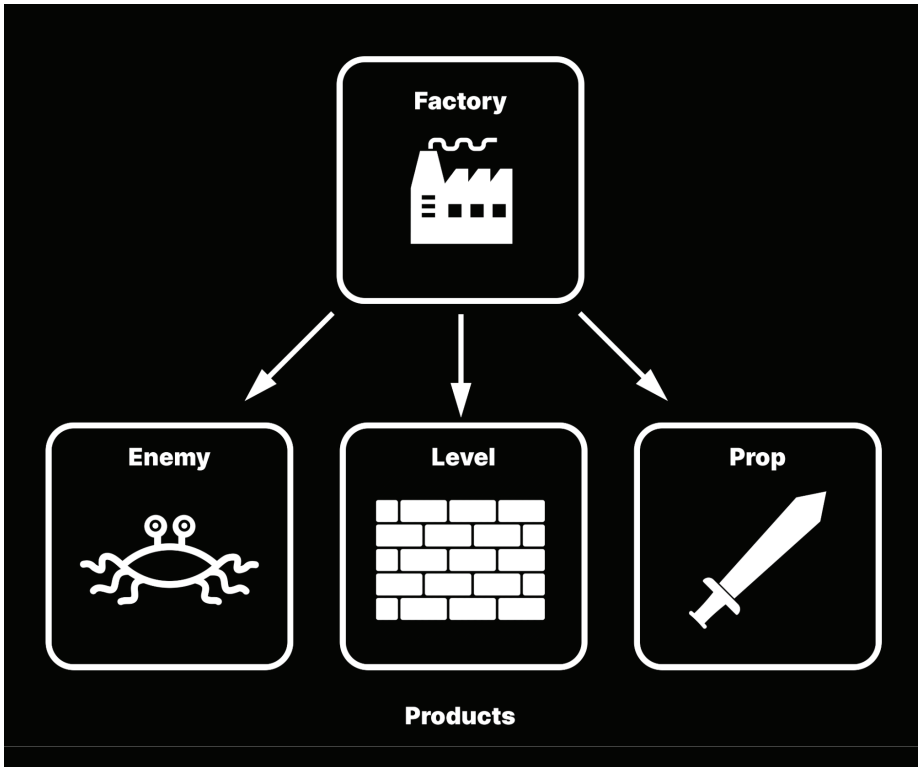
  Unity のプレハブシステムにより、ゲームオブジェクト用の一形態のプロトタイピングが実装されます。これにより、コンポーネントがすべて設定されたテンプレートオブジェクトを複製できます。特定のプロパティをオーバーライドし、他のプレハブの中にプレハブバリアントまたはネストプレハブを作成して、階層を作成します。特別なプレハブ編集モードを使用して、単独またはコンテキスト内でプレハブを編集します。

— **コンポーネント：**Unity を使用して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。複数の責任を担う大きなクラスを作成する代わりに、それぞれ 1 つのことを担うより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別する場合は、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。必然的にビルトインでないその他のパターンが必要になります。次の章で、それらのいくつかについて見て行きましょう。

# FACTORY（ファクトリー）パターン




1 つのファクトリー（工場）では、1 つ以上の製品をスポーン（生産）できます。

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。大半のゲームは、ゲームのプレイ中にさまざまなものがスポーンされるものですが、ランタイムに何が必要であるかは本当な必要な局面になるまでわからないことが多々あります。

ファクトリーパターンでは、皆さんのご想像どおりに、ファクトリーという特別なオブジェクトを指定します。ある視点から捉えた形で、その「製品」のスポーンにまつわる数多くの詳細がカプセル化されます。まず得られる利点としては、コードが片付いてくれます。

しかしながら各製品が 1 つの共通のインターフェースまたは基本クラスに従っているという場合であるのならば、さらに一歩先に進め、独自の組立ロジックがもっと含まれるようにして、ファクトリー自体から隠してしまうこともできます。これにより、新しいオブジェクトの作成がさらに拡張できるようになります。

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。

## ファクトリーのわかりやすい例

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こともお勧めします。

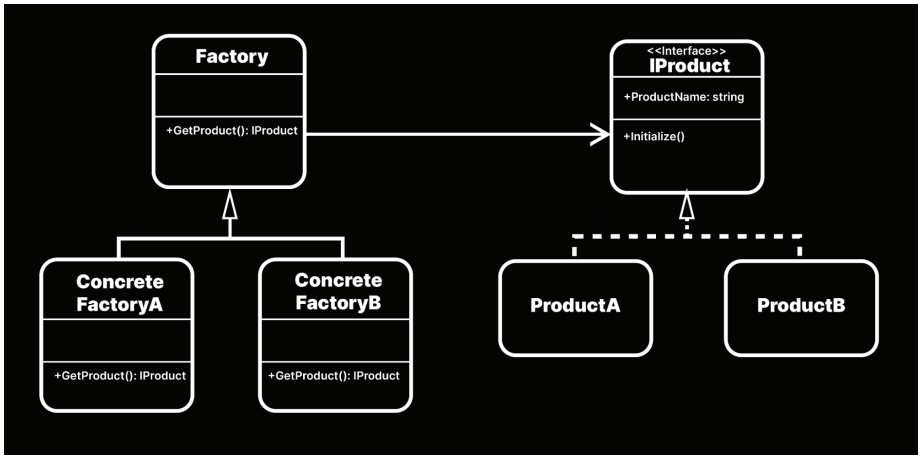
`if` ステートメントや `switch` を使用してこのロジックを保つよりも、`IProduct` というインターフェースと `Factory` という抽象クラスを作成します。

```
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メソッドを共有した
    …
}
```

製品はメソッドのために特定のテンプレートに従う必要がありますが、それ以外に機能は一切共有することがありません。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。

以下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 に対して実行する任意のロジックのみです。

その後、IProduct インターフェースに従っている限りは、必要な数だけ製品を定義することができます（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。いくつかのコンクリートサブクラス（ConcreteFactoryA と ConcreteFactoryB）を派生します。それぞれ実際に異なる製品を取得します。

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。各コンクリートファクトリー内のフィールドには、対応するテンプレートプレハブも格納されます。

サンプルの ProductA と ConcreteFactoryA はこちらです。

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

    private ParticleSystem particleSystem;

    public void Initialize()
    {
        // この製品で一意の任意のロジック
        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>();

        // 各製品に独自のロジックが含まれる
        newProduct.Initialize();

        return newProduct;
    }
}
```

ここでは、ファクトリー内のプレハブを利用するように `IProduct` が実装された製品クラス MonoBehaviour を設定しました。

各製品に独自のバージョンの `Initialize` がどのように設定されているか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 でコピーが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一切含まれていません。すべての製品に共通の `Initialize` メソッドを呼び出すだけです。

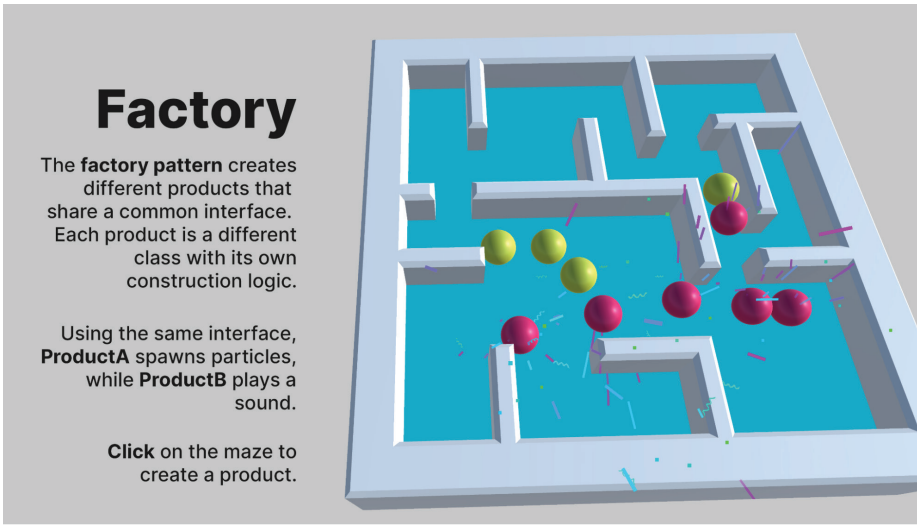
サンプルプロジェクトの詳細を確認し、`ClickToCreate` コンポーネントがファクトリー間でどのように切り替わり、それぞれ動作が異なる ProductA と ProductB が作成されるかをご覧ください。ProductB がスポーンされると音が鳴り、ProductA からはパーティクルエフェクトが発せられます。

## 長所と短所

ファクトリーパターンで一番恩恵を受けることができるのは、たくさん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ことがあった場合に、既存の製品が変更されることも、前のコードを修正する必要もなくなります。

各製品の内部ロジックを独自のクラスに分割することで、ファクトリーコードが比較的短く保たれます。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、数々のクラスやサブクラスを作成することです。他のパターンと同じように、これによって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。




一方の製品では音が鳴り、もう一方ではパーティクルが再生されます。両方とも同じインターフェースを使用します。

## 改善策

ファクトリーの実装は、ここで紹介している実装と比較して、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、以下の調整を加えること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ことをお勧めします。一意の文字列識別子（名前、ID など）をキーとして使用し、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。静的クラスはインスペクターには表示されないため、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囚われ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する可能性があります。
- **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することができます。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より大きなタスクの一部（例：ゲームステージの一部であるダイアログボックスの UI 要素を設定する）として使用することがあります。

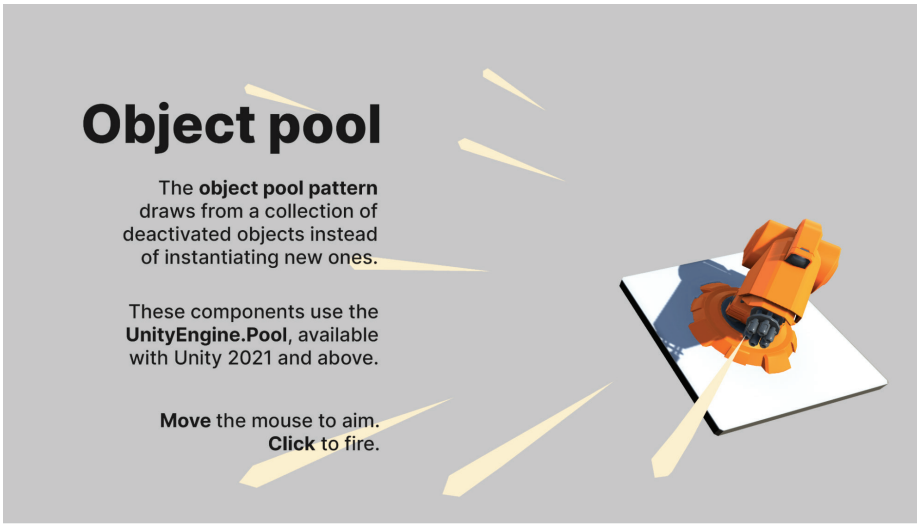
# OBJECT POOL（オブジェクトプール）

オブジェクトプーリングとは、大量のゲームオブジェクトを作成および破壊する際に CPU を解放するための最適化手法のことです。

オブジェクトプールパターンでは、非アクティブ化された「プール」に使用準備が整った状態で待機している一連の初期化されたオブジェクトを使用します。オブジェクトが必要なときに、アプリケーションではそれをインスタンス化しません。代わりに、プールからゲームオブジェクトを要求して、それを有効にします。

使用が終わった際には、そのオブジェクトを破壊する代わりに、非アクティブ化してプールに戻します。

オブジェクトプールにより、ガベージコレクションのスパイクから発生する可能性がある、スタッターが減ることがあります。GC のスパイクには、よくメモリの割り当てによる大量のオブジェクトを作成および破壊することが伴います。オブジェクトプールを、ユーザーがスタッターに気づかない良いタイミング（画面のロード中など）で事前インスタンス化することができます。




オブジェクトプールにより、ゲームプレイ中にスタッターを発生させることなく、弾を発射することができます。

## プールシステムのわかりやすい例

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`

— プレハブに追加された `PooledObject` コンポーネント。これにより、クローンが作成された各項目でプールへの参照が保たれます

ObjectPool で、プールのサイズが記述されるフィールド、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（この例の stack）を設定します。

```
public class ObjectPool : MonoBehaviour
{
    [SerializeField] private uint 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（ラグが目立たないときに呼び出す）
    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 メソッドにより、オブジェクトプールが入力されます。PooledObjects 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、それを 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 回のみ実行されるようにします。

また、プールされた項目を取得するメソッド（GetPooledObject）と、1 つ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()
    {

        // プールの大きさが十分でない場合は、新しい PooledObjects をインスタンス
化する
        if (stack.Count == 0)
        {
            PooledObject newInstance = Instantiate(objectToPool);
            newInstance.Pool = this;
            return newInstance;
        }
        // それ以外の場合は、リストから次のものをグラブする
        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 によって新しい PooledObject が作成されます。それ以外の場合は、単に次の有効な要素が返されます。プールの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。

クライアントで GetPooledObject が呼び出されると、プールされたオブジェクトを所定の場所に移動 / 回転する必要があります。

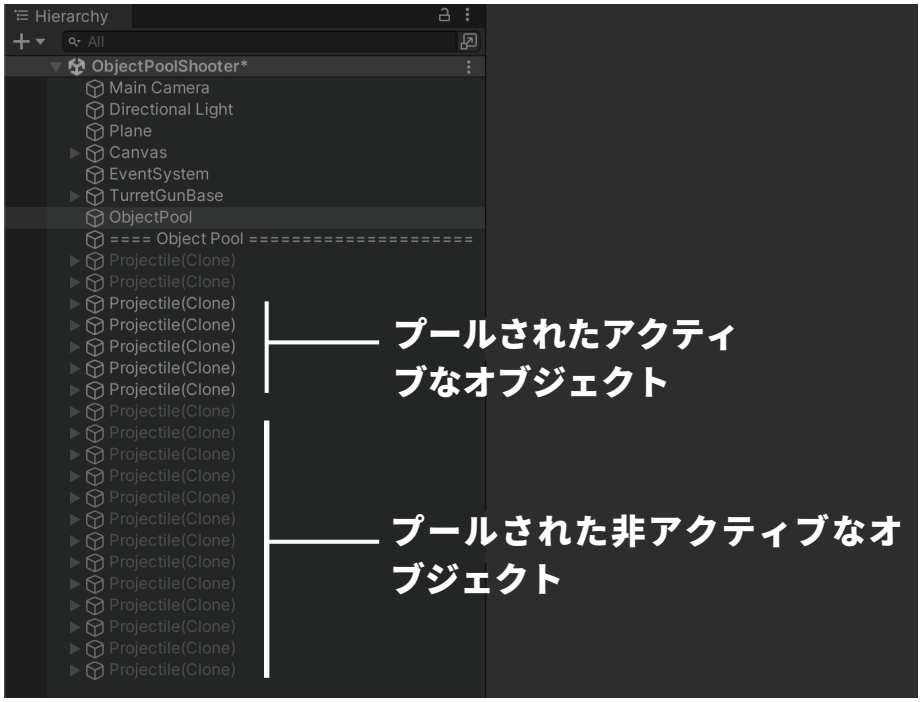
プールされた各要素には、ObjectPool を参照する役割のみを担う、小さな PooledObject コンポーネントが存在します。

```
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は、基本的な使用例を示します。ここでは、ExampleGun スクリプトがゲームオブジェクトにアタッチされています。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、武器のスクリプトによって、Object.Instantiate が呼び出される代わりに、そのスクリプトの GetPooledObject メソッドが呼び出されます。

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、発射された各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、何百発もの弾をオフスクリーン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なサイズであることを確認してください。

プールの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引っ張ってきます。

Unity の ParticleSystem を使用したことがある方であれば、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、それと同じように機能します。

## 改善策

上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展開する際には、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にすることを検討してください。これにより、インスペクターを使用することなく、アプリケーションのどこからでもアクセスできるようになります。また、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：**プールする必要があ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 が一意のキーとして機能することがあります）。
- **使用されていないゲームオブジェクトを都合の良いやり方で削除する：**オブジェクトプールを効果的に活用する方法として、使用されていないオブジェクトを非表示にし、それらをプールに返すという方法が挙げられます。あらゆる機会（例：オフスクリーン、爆発で隠すなど）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。
- **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。さもないと、ランタイムにエラーが発生するおそれがあります。
- **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、メモリの浪費につながります。プールでリソースが大量に消費されないように、一定限度を超えるオブジェクトは削除することをお勧めします。

オブジェクトプールの使用方法は、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出現します。

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から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。

2021 以降のバージョンの Unity を使用している場合は、オブジェクトプーリングシステムがビルトインされているため、前述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。

## UnityEngine.Pool

オブジェクトプールパターンはどこにでも存在するものであるため、Unity 2021 で独自の UnityEngine.Pool API に対応するようになりました。これにより、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するのに、スタックベースの `ObjectPool` を使用できるようになります。ニーズに応じて、`CollectionPool`（List、HashSet、Dictionary など）を使用することもできます。

このサンプルプロジェクト（シーンを参照）では、カスタムプールコンポーネントが不要になりました。代わりに、行の先頭に `using UnityEngine.Pool;` を追加することで、銃のスクリプトを更新します。これにより、ビルトインの `ObjectPool` を使用して発射される弾のプールを作成できます。

```
using UnityEngine.Pool;

public class RevisedGun : MonoBehaviour
{
    …

    // Unity 2021 以降で使用できるスタックベースの ObjectPool
    private IObjectPool<RevisedProjectile> objectPool;

    // すでにプール内にある既存の項目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>(CreateProjecti
le,
            OnGetFromPool, OnReleaseToPool, OnDestroyPooledObject,
            collectionCheck, defaultCapacity, maxSize);
    }

    // オブジェクトプールに入力する項目を作成するときに呼び出される
    private RevisedProjectile CreateProjectile()
    {
        RevisedProjectile projectileInstance =
Instantiate(projectilePrefab);
        projectileInstance.ObjectPool = objectPool;
        return projectileInstance;
    }

    // オブジェクトプールに項目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項目を取得するときに呼び出される
    private void OnGetFromPool(RevisedProjectile pooledObject)
    {
        pooledObject.gameObject.SetActive(true);
    }

    // プールされた項目の最大数を超える（プールされたオブジェクトを破壊する）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 の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。

- プールに入力するためにプールされた項目を初めて作成するとき
- プールから項目を取得するとき
- プールに項目を戻すとき
- プールされたオブジェクトを破壊するとき（例：上限に達した場合）

次に、コンストラクターに渡す対応する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。項目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の修正が加えられ、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 =>
objectPool = 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、一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一から作成する手間が 1 つ減ることになります。

# SINGLETON（シングルトン）パターン

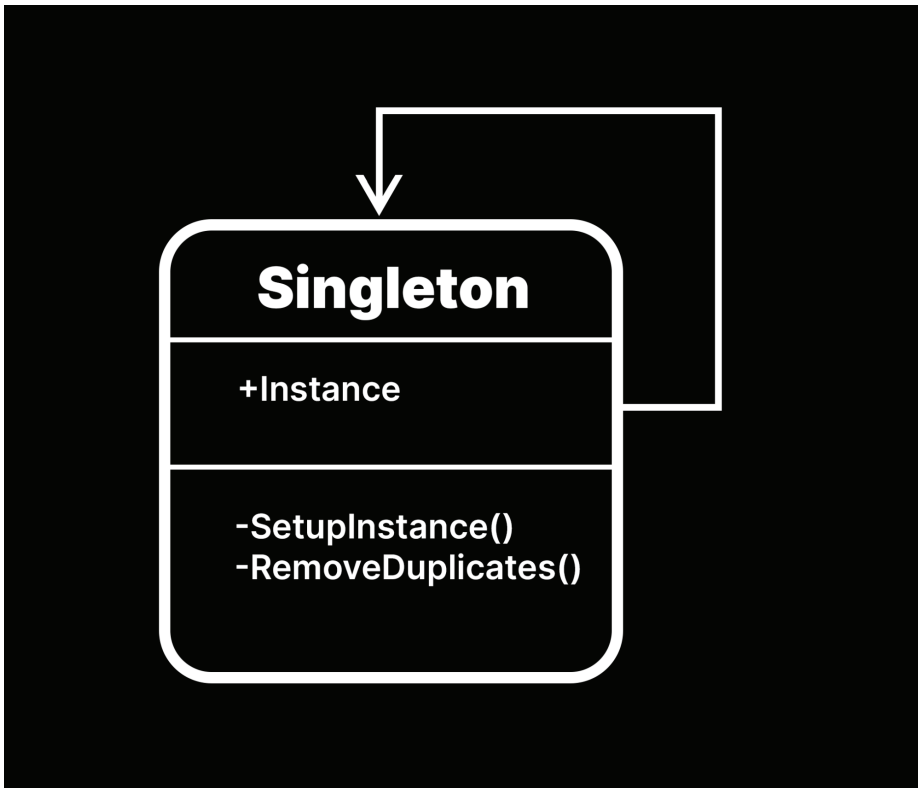
シングルトンはいわれのない非難を受けています。Unity 開発が初めての方にとって、おそらく自然に直面し最初に認識するパターンの 1 つがシングルトンでしょう。また、これは最も評判が悪いデザインパターンの 1 つでもあります。

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ように定義されています。

— 1 つのクラスにはそれ自体の 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簡単に提供する

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 つだけにすることをお勧めします。また、一度に 1 つのファイルマネージャーのみがファイルシステムに書き込めるようにすることをお勧めします。これら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壊します。

*ゲームプログラミングパターン*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、アンチパターンの 1 つに挙げられています。この悪評の理由は、同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。開発者はあまり適切で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。

## シングルトンのわかりやすい例

最もわかりやすいシングルトンの 1 つは、以下のようになります。

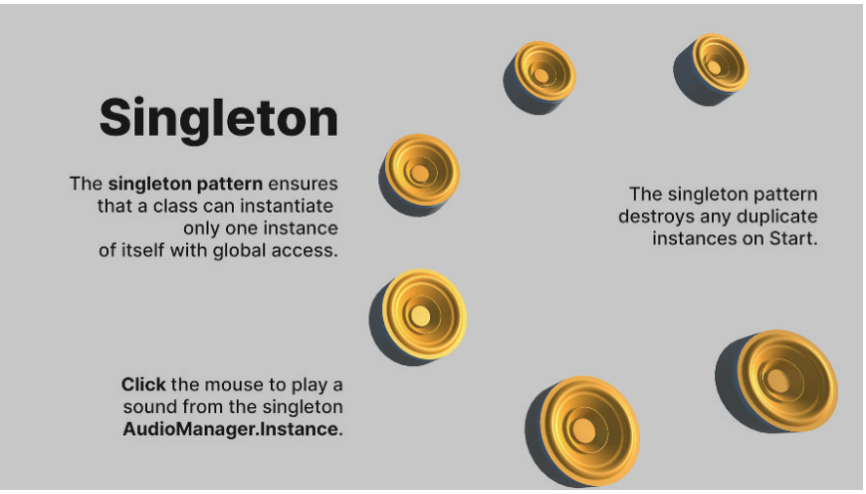
```
using UnityEngine;

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をチェックします。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。これがシーン内の最初のシングルトンに違いありません。

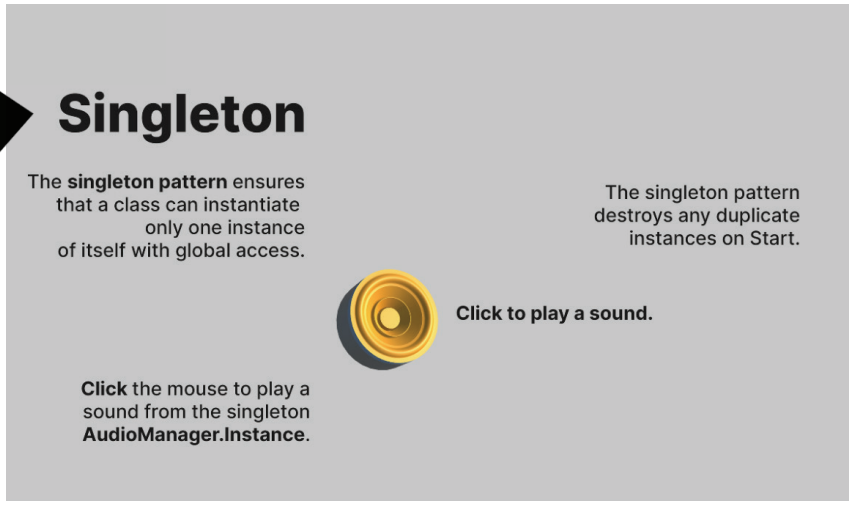
それ以外の場合、このインスタンスはおそらく複製です。Destroy(gameObject) を呼び出して、シングルトンにシーン内にそのようなコンポーネントが 1 つのみあることを保証します。

ランタイムにスクリプトを一度に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 のロジックで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します。

removes duplicate instances




`Instance` フィールドは public かつ static です。すべてのコンポーネントに、シーン内の任意の場所から唯一のシングルトンへのグローバルアクセスが備わっています。

## 永続化と遅延初期化

`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る。
- 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る。

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、遅延初期化を使用して、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、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のみです。

改善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 が、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、取得されているものの中から 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 メソッドによって適切なコンポーネントが備わったゲームオブジェクトが作成されます。

DontDestroyOnLoad(gameObject) を指定すると、シーンのロードによって階層からシングルトンが消去されなくなります。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 1 つのシングルトンと、GameManager として機能するもう 1 つのシングルトンが必要な場合、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 : Component
{
    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から継承するだけです。例えば、以下のように宣言することで、GameManager という MonoBehaviour をシングルトンにすることができます。

```
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` を参照することができます。

## 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、さまざまな面で SOLID の原則を破ります。以下のようなさまざまな理由から、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。
- **シングルトンにより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を及ぼし、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、以下のようなシングルトンに備わっているメリットを魅力的に感じるかも知れません。

— **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のに、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、常にシーン内のあらゆるオブジェクトから必要に応じて使用できます。

— **シングルトンは高性能：**常に静的なシングルトンインスタンスにグローバルにアクセスできるため、低速になる傾向がある 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。

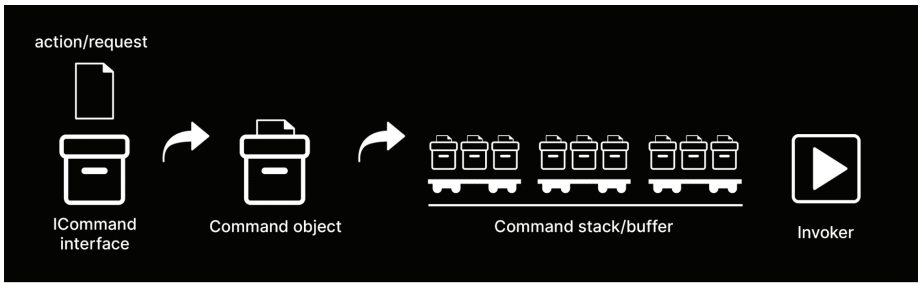
この方法により、シーン内の他のすべてのゲームオブジェクトから常にアクセス可能なマネージャーオブジェクト（例：ゲームフローマネージャーやオーディオマネージャー）を作成できます。また、オブジェクトプールを実装している場合は、プーリングシステムをシングルトンとしてデザインすることで、プールされたオブジェクトを簡単に取得できるようになります。

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。乱用はしないでください。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。

# COMMAND（コマンド）パターン

オリジナルの Gang of Four のパターンの 1 つであるコマンドは、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 / やり直し機能を使用する、またはユーザーの入力履歴をリストに残すゲームをプレイしたことがある方であれば、おそらく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って計画することができる戦略ゲームを想像してみてください。こ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」としてカプセル化することができます。




コマンドパターンを使用した格納アクション

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、それらの実行のタイミングを制御することができます。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潜在的に遅延させ、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、アクションが含まれる汎用の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の呼び出し元

これを実装する方法はたくさんありますが、インターフェースを使用するバージョンの 1 つがこちらです。

```
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（これを抽象クラスを使用して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。

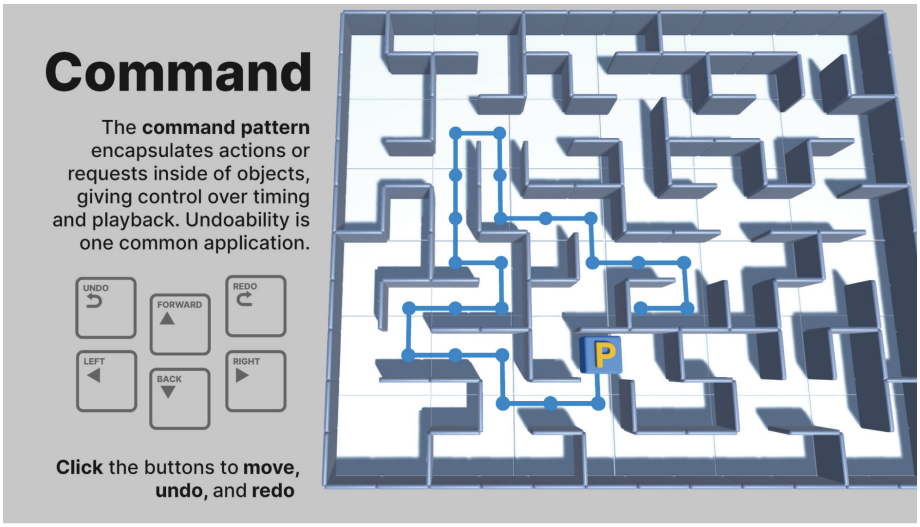
```
public class CommandInvoker
{
    private static Stack<ICommand> undoStack = new Stack<ICommand>();

    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：取り消すことのできる動作

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,
boardSpacing, 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する必要があるものを実装することは、そのパターン自体とは切り離されています。




サンプル内のプレイヤーの動き

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 インターフェースが実装されます。

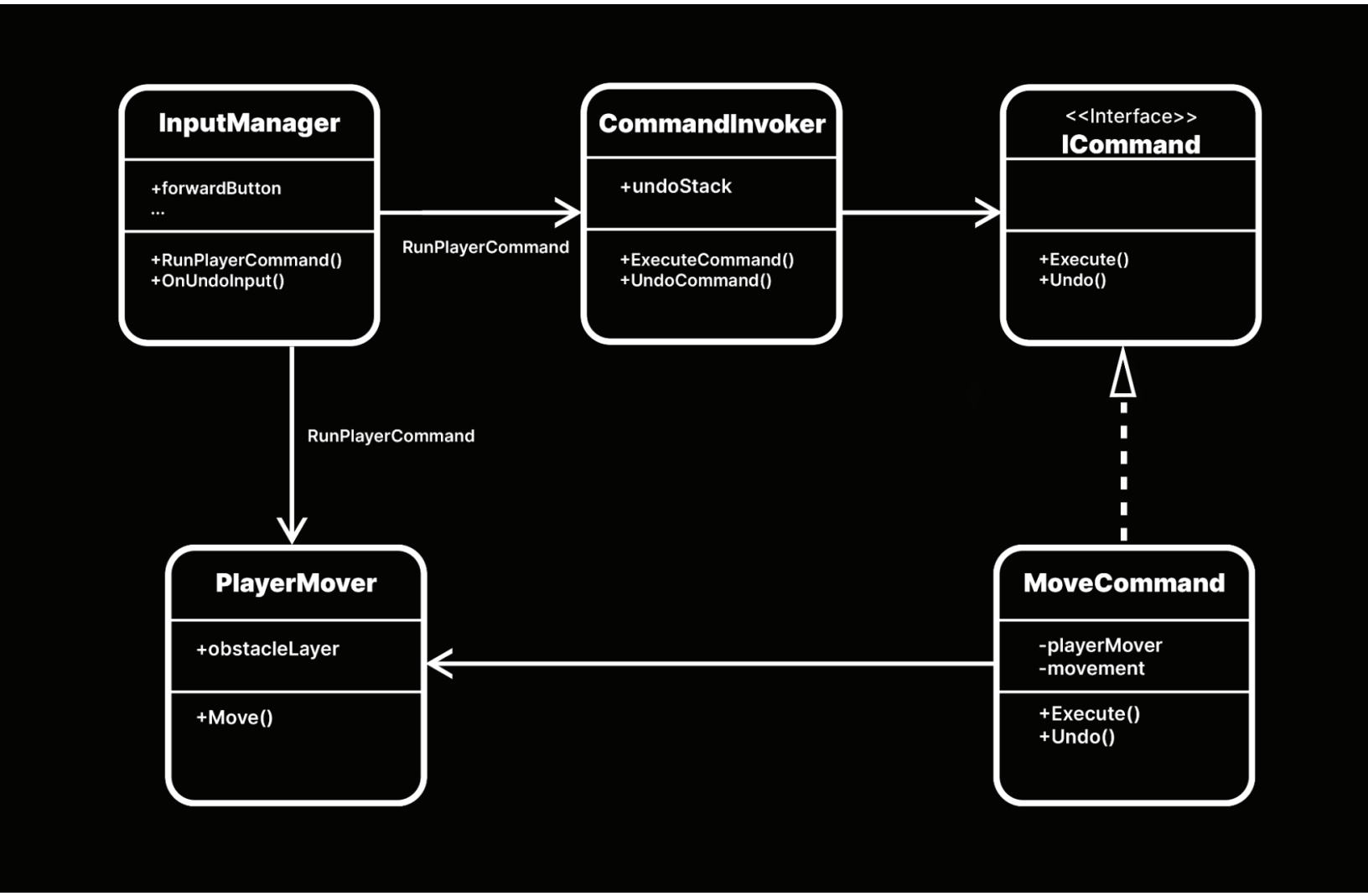
```
public class MoveCommand : ICommand
{
    PlayerMover playerMover;
    Vector3 movement;

    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。やろうとしていることで必要になるものが、どのようなロジックであってもここに入るため、Move を movement ベクターを指定して呼び出します。

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 movement ベクターが減算され、必然的にプレイヤーを反対方向に押し出します。

MoveCommand には、実行する必要があるすべてのパラメーターが格納されます。これらをコンストラクターを使用して設定します。このケースでは、該当する PlayerMover コンポーネントとその movement ベクターを保存します。

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、追加メソッド RunMoveCommand を追加し、新しい MoveCommand を作成してそれを 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 ベクターを使用して RunPlayerCommand を呼び出します。

InputManager 実装の詳細については、サンプルプロジェクトで確認を行います。そうでない場合、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ます。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す。取り消しボタンをクリックすると、スタート地点のマスに引き返すことができます。

## 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、特別な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はるかに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のにあたって、そういった余分な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かどうかを判断する必要があります。

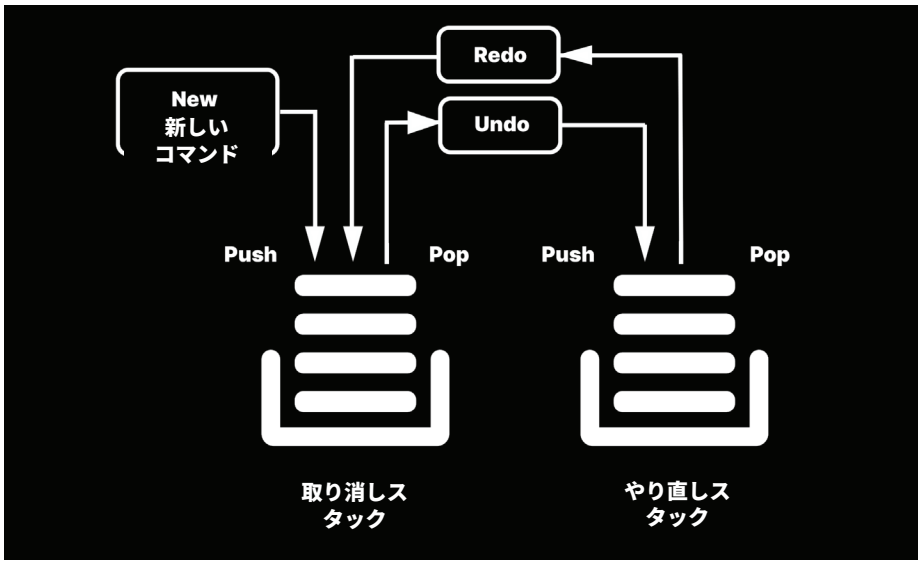
## 改善策

一度基本を習得すれば、コマンドのタイミングに影響を及ぼし、コンテキストに応じてそれらを連続再生したり、逆再生したりすることができます。

コマンドパターンを組み込む際には、以下のことを検討してください。

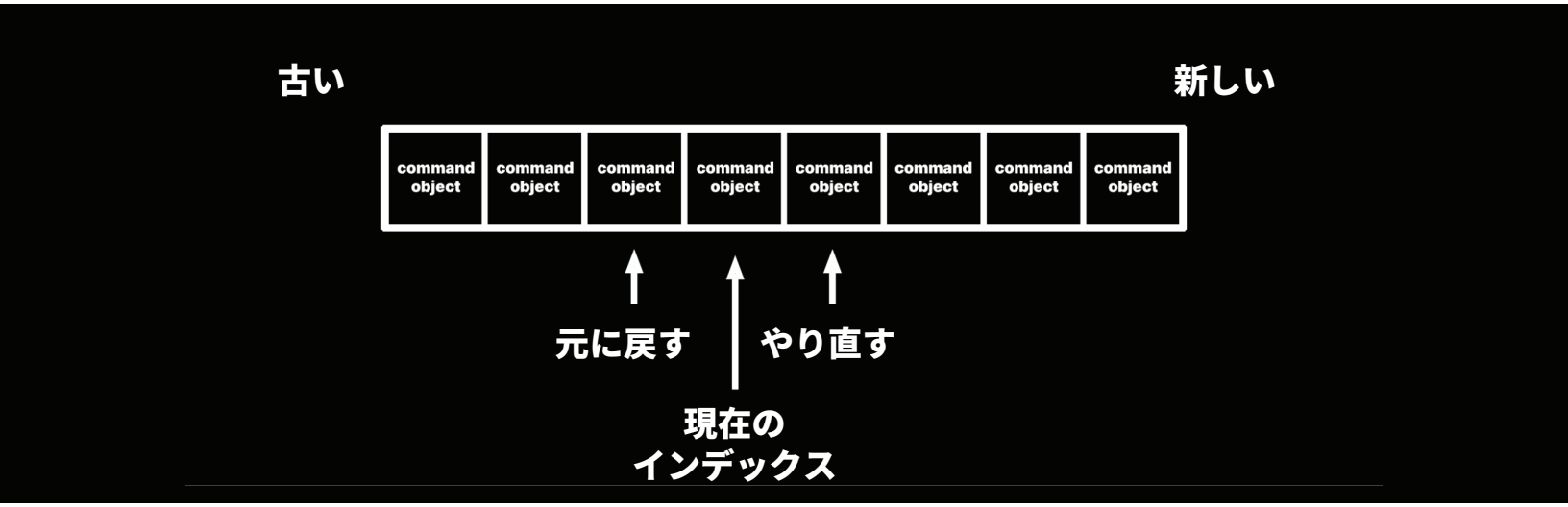
— **より多くのコマンドを作成する：**サンプルプロジェクトに含まれているコマンドオブジェクトは 1 種類（MoveCommand）のみです。ICommand を実装する任意の数のコマンドオブジェクトを作成し、それらを CommandInvoker を使用して追跡することができます。

— **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別個のスタックにそれをプッシュします。この方法により、取り消し履歴を時間をかけることなく循環させる､つまりそれらのアクションをやり直すことができます。ユーザーがまったく新しい動きを呼び出すときに、やり直しスタックを空にします（実装は付属のサンプルプロジェクトに見つかります）。




取り消しスタックとやり直しスタック

— **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します。**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取り消し可能です。インデックスより後のコマンドはやり直し可能です。




リストまたはその他のコレクションがコマンドバッファとして機能します。

— **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。スタックを数コマンドの範囲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。

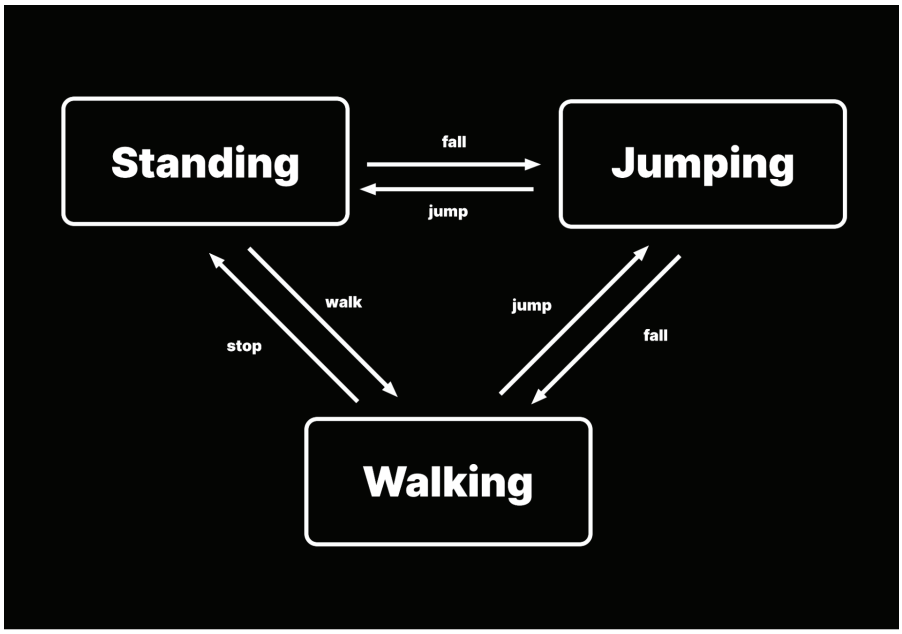
`CommandInvoker` で行われるのは、他の外部オブジェクトと同じように、Execute または Undo を呼び出すことのみで、コマンドオブジェクトの内部構造は見られません。コンストラクターを呼び出すときに、処理する必要があるすべてのデータをコマンドオブジェクトに渡します。

# STATE（ステート）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、キャラクターは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、キャラクターが宙に跳び上がります。数フレーム後に着地し、立って静止しているポジションに戻ります。

## 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、以下のようなものになる可能性があります。




簡略化したステートダイアグラム

フローチャートに似ていますが、多少の違いがあります。

- ダイアグラムはさまざまなステート（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、ある特定の時点で現在のステートが 1 つだけアクティブになります。
- 各ステートでは、ランタイム時の条件に基づいて、別の 1 つのステートへの遷移をトリガーできます。
- 遷移が発生すると、出力されたステートが新しいアクティブなステートになります。

このダイアグラムは、有限ステートマシン（FSM）と呼ばれるものを表します。ゲーム開発における一般的な用途の 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 FSM 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ナイーブアプローチを使用することをお勧めします。

```
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 スクリプトの収拾がすぐにつかなくなってしまいます。ステートと複雑さを追加するたびに、PlayerController スクリプト内部を再検討する必要があります。

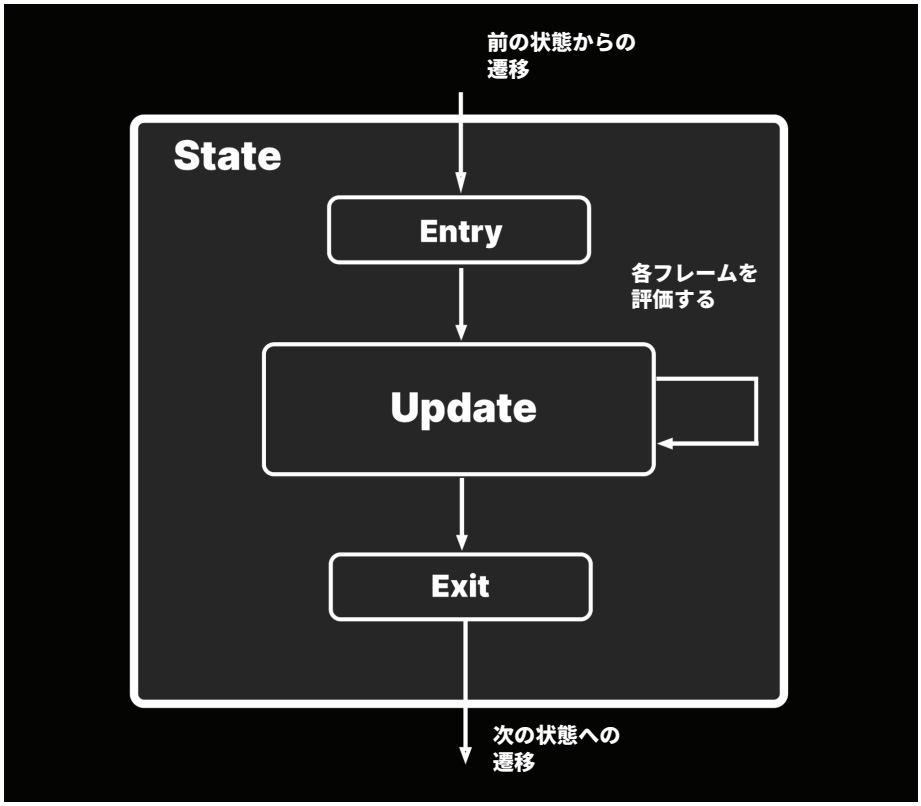
## ステートパターンのわかりやすい例

幸いなことに、ステートパターンはロジックを再整理するのに役立ちます。オリジナルの Gang of Four によると、ステートパターンは以下の 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が変わる必要があります。
- 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。

上の例の `UnrefactoredPlayerController` クラスではステートの変化は追跡できますが、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の既存のステートに及ぼす影響を最小限に抑える必要があります。代わりに、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ができます。

各ステートを以下のように構成するとします。




Entry、Exit、Update を使用してカプセル化されたステート

ここではステートを入力し、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 Update()
    {
        // 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
    }
}
```

ゲーム内の各コンクリートステートにより、IState インターフェースが実装されます。

- **1 つの Entry：**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。
- **Update：**このロジックは毎フレーム実行されます（Execute または Tick とも呼ばれることがあります）。MonoBehaviour と同じように、物理演算用の FixedUpdate や LateUpdate などを使用して、Update メソッドをさらにセグメント化することができます。

  Update 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。
- **1 つの Exit：**このコードはそのステートを離れ、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 は以下のようになります。

```
[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 Update()
    {
        if (CurrentState != null)
        {
            CurrentState.Update();
        }
    }
}
```

同パターンに従うために、StateMachine ではそのマネージメント下にある各ステートの public オブジェクトを参照します（このケースでは walkState、jumpState、idleState）。StateMachine は MonoBehaviour からは継承されないため、各インスタンスを設定するの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);
}
```

コンストラクターに必要とされるすべてのパラメーターを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（以下の IdleState の例を参照）。

StateMachine に関しては、以下のこと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）をインスペクターに表示することができます。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することができます。
- `CurrentState` プロパティは読み取り専用です。StateMachine 自体ではこのフィールドは明示的には設定しません。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 を設定することができます。
- 各 State オブジェクトによって、`TransitionTo` メソッドを呼び出して現在アクティブなステートを変更する、その独自の条件が決まります。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には、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。それぞれ `IState` が実装される独自のクラスを取得します。SOLID の原則に従って、ステートをさらに追加することが前に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 Update()
    {
        // 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
    }
}
```

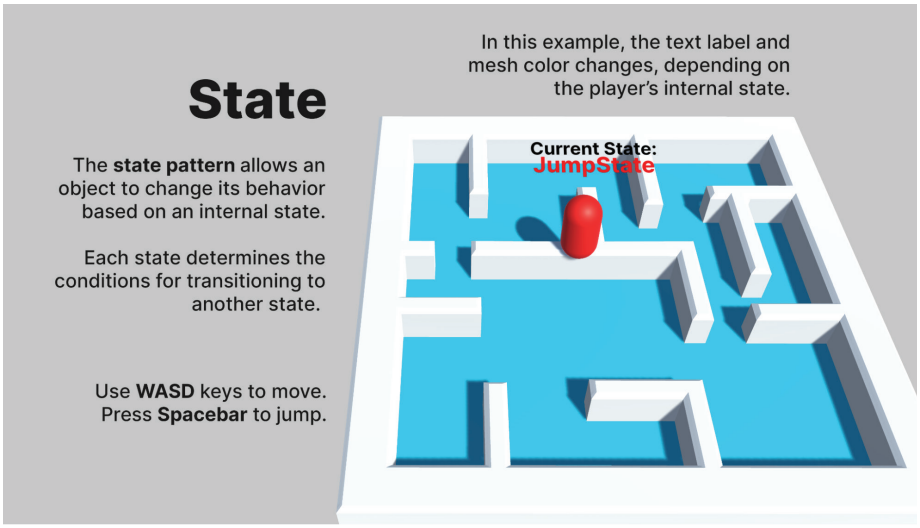
繰り返しになりますが、コンストラクターを使用して PlayerController オブジェクトを渡します。この例では、この `player` に StateMachine への参照と、Update ロジックに必要なその他すべてのものが含まれます。`idleState` によって、Character Controller の速度やジャンプステートが監視され、その後適宜 `StateMachine` の `TransitionTo` メソッドが呼び出されます。

このサンプルプロジェクトの `WalkState` と `JumpState` の実装を見直してください。動作が切り替わる 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、各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 の原則に準拠するのに役立ちます。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さらにステートを追加することができます。

一方で、追跡するステートが少数の場合、余分な構造が多くなり過ぎてしまうおそれがあります。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測される場合に限定される可能性があります。



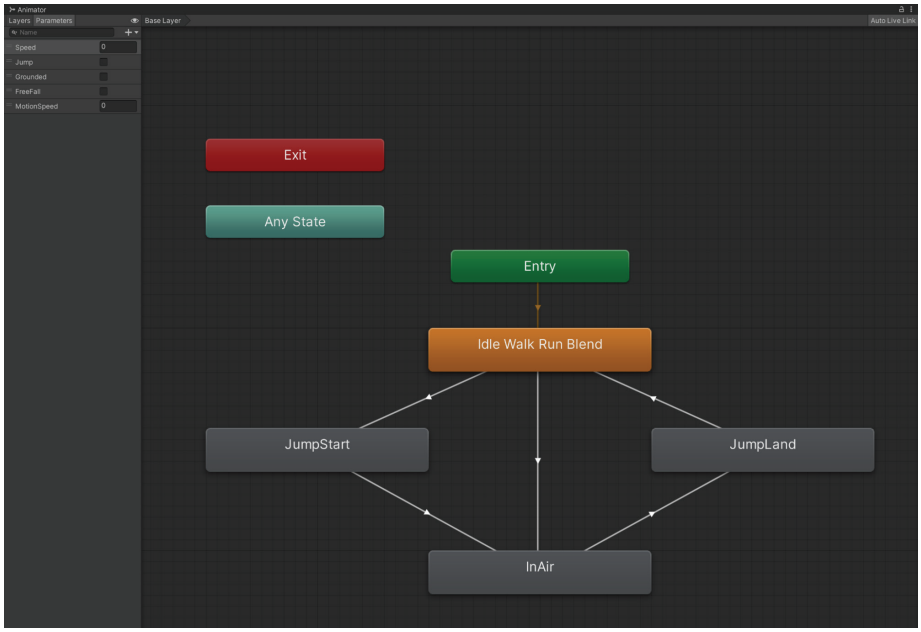

ステートパターンのサンプル

## 改善策

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。現実世界の例では、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、内部的な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、ゲームのアクターが走っている、ジャンプしている、泳いでいる、登っ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づくことと思います。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。



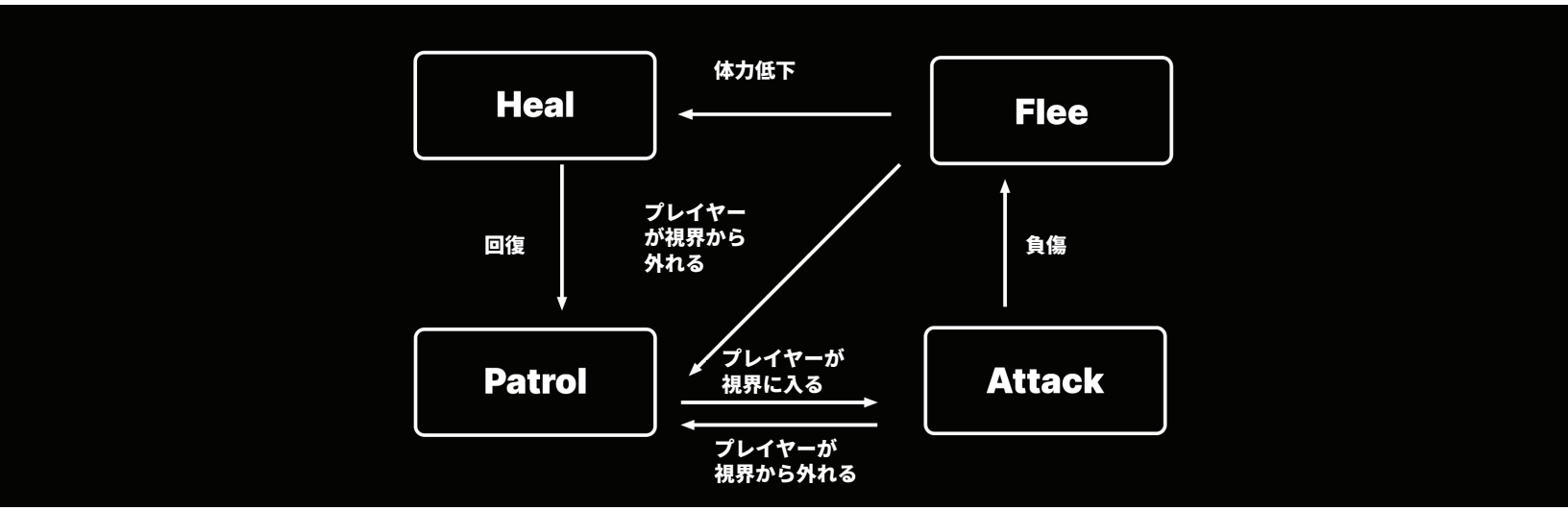

アニメーターのステートグラフの例。その構造を StateMachine と比較してください。

— **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ことをお勧めします（Observer（オブザーバー）パターンを参照）。ステートに入るまたはステートから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、ランタイムに応答するよう設定できます。

— **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を記述することを開始するにあたっては、階層化されたステートマシンを実装することをお勧めします。一部のステートは必然的に似てきます。例えば、プレイヤーまたはゲームのアクターが地面に立っている場合は、WalkingState でも RunningState でも屈むかジャンプすることができます。

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。こうすることで、継承を使用して具体的な何かをサブステートでオーバーライドすることができます。例えば、最初に GroundedState を宣言するとします。その後に、そこから RunningState または WalkingState を継承することができます。

— **簡単な AI を実装する：**有限ステートマシンは、基本的な敵 AI を生成するのにも便利です。NPC の知能を構築する FSM アプローチは、以下のようになる場合があります。



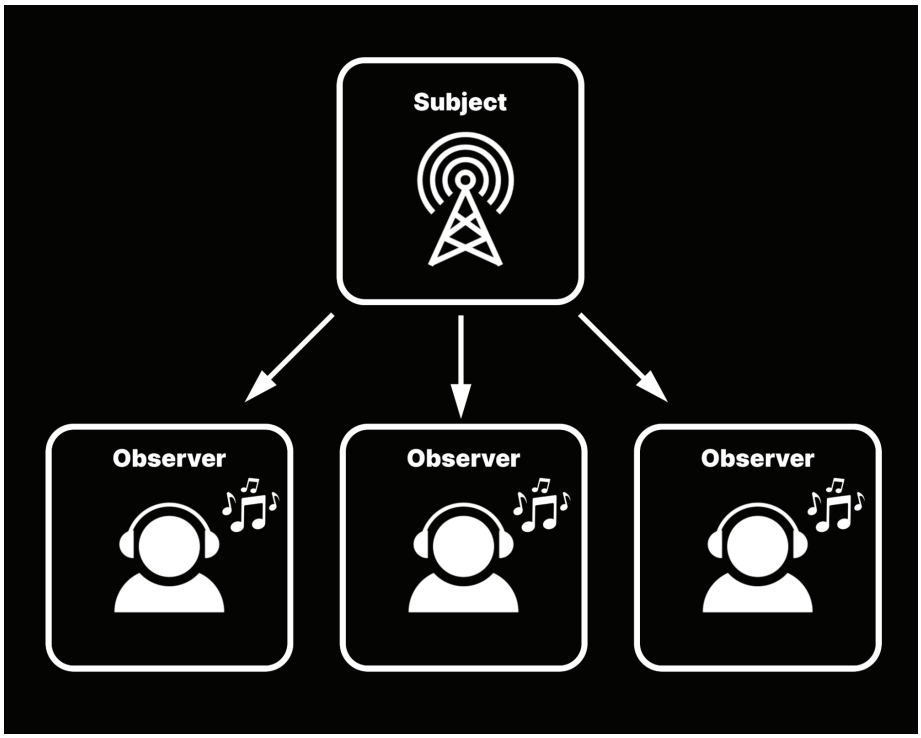

ステートパターンをベースとする簡単な AI

こちらが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。各ステートがアクション（攻撃、逃走、偵察など）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。

# OBSERVER（オブザーバー）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か？パワーアップを集めたり、オブジェクトを完成させたりしたらどうなるか？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。

オブザーバーパターンは、このような種類の問題を解決する一般的なソリューションです。「1 対多」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をやり取り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が行きます。これは多数の視聴者に向けて電波を発信するラジオ塔に似ています。




オブザーバーパターンはラジオ塔のように機能します。サブジェクトからオブザーバーに発信されます。

発信側のオブジェクトのことを、サブジェクトと呼びます。リッスンするその他のオブジェクトは、オブザーバーと呼びます。

このパターンによりサブジェクトが疎に切り離されます。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存在を認識していません。

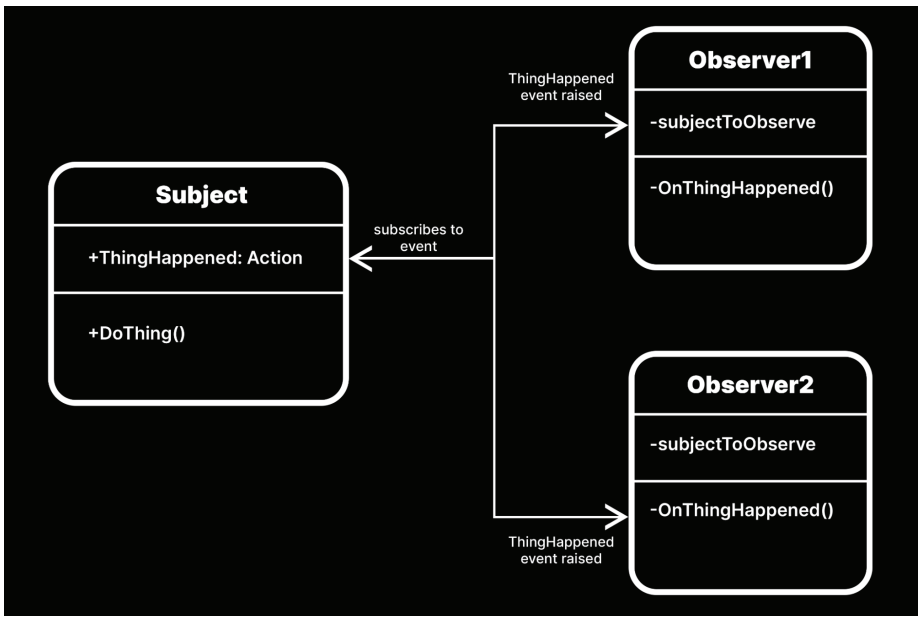
## イベント

オブザーバーパターンは広範囲にわたり、C# 言語に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、一般的には不要です。すでにあるものを一から作成しないようにすることについてお話ししたのを覚えていますか？ C# にはすでにイベントを使用したパターンが実装されています。

イベントとは単に何かが発生したことを示す通知のことです。以下のいくつかの部分が関与します。

- 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。イベントとは、単にサブジェクトでランタイムに実行されるアクション（例：ダメージを受ける、ボタンをクリックするなど）のことです。
- その後、サブスクライバー（オブザーバー）がそれぞれイベントハンドラーという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にサブスクライブします。必要に応じて、必要な数のオブザーバーをサブスクリプションに結合することができます。それら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の発生の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。

この方法により、数多くのコンポーネントをそのサブジェクトからの単一のイベントに反応するように設定します。サブジェクトによりボタンがクリックされたことが示されると、オブザーバーで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、ファイルの保存が行われることがあります。レスポンスはどのようなものにもなる可能性があります。これがオブジェクト間でメッセージを送信するのにオブザーバーパターンが見られることが多い理由です。




サブジェクトからイベントが発生し、オブザーバーに通知が行きます。

## サブジェクトとオブザーバーのわかりやすい例

例えば、以下のような基本的なサブジェクト / パブリッシャーを定義することがあります。

```
using UnityEngine;
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より簡単にアタッチするために MonoBehaviour から継承しますが、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それらを山かっこで囲んで List<T> として渡します（最大で 16 個のパラメーター）。

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

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 から継承しますが、それは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 Awake()
    {
        if (subjectToObserve != null)
        {
            subjectToObserve.ThingHappened += OnThingHappened;
        }
    }

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

このコンポーネントを GameObject にアタッチし、subjectToObserver をインスペクターの順序で参照して、ThingHappened イベントをリッスンします。

OnThingHappened メソッドには、イベントに応答してオブザーバーで実行されるあらゆるロジックを含めることができます。多くの場合、開発者はプレフィックス「On」を追加してイベントハンドラーであることを示します（お使いのスタイルガイドの名前付け規則をそのまま使用してください）。

Awake または Start では、+= 演算子を使用してそのイベントに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 メソッドが実行されると、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 にサブスクライブされた状態でランタイムにオブザーバーを削除または除去した場合、そのイベントを呼び出すと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。

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あらゆるものに適用できます。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は、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及ぼすことなく同システムを作成することができます。

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。

— 中短期的な目標または最終目標

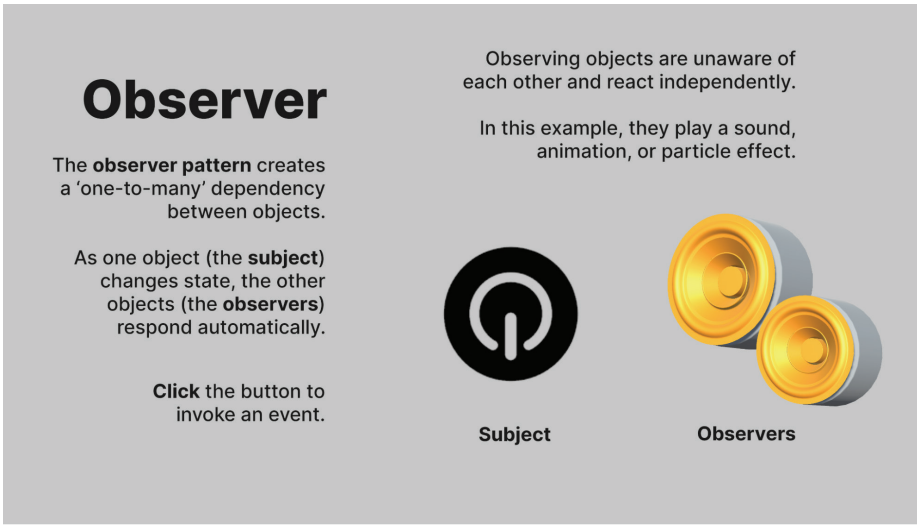
— 勝敗条件

— PlayerDeath、EnemyDeath、または Damage

— アイテムの拾得

— 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のみで、サブスクライブできるオブザーバーの数に上限はありません。




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。これで、`AudioObserver` コンポーネントと `ParticleSystemObserver` コンポーネントが指定されたその他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。

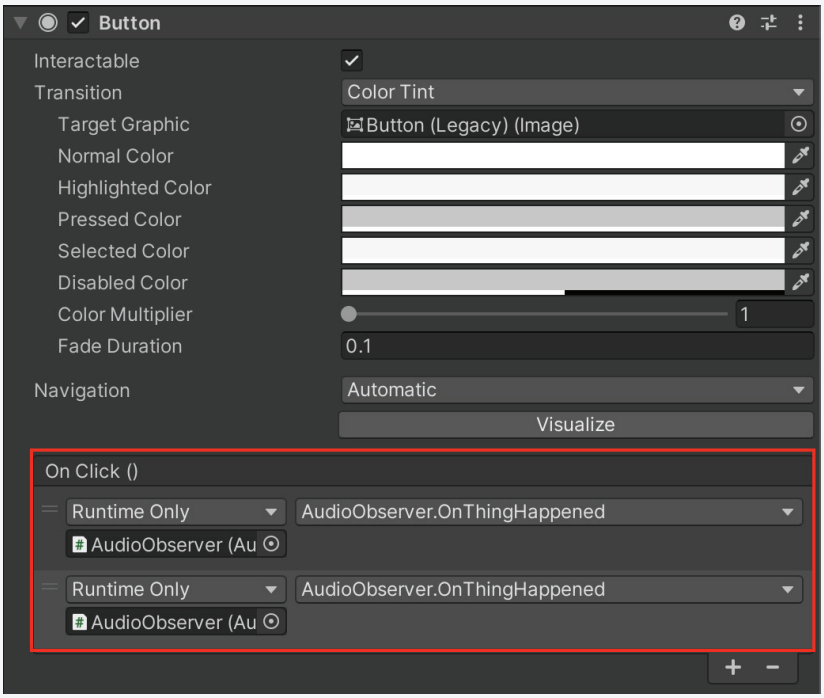
どのオブジェクトを「サブジェクト」または「オブザーバー」にするかは、用途によって変わります。イベントを発生させるオブジェクトはすべてサブジェクトとして機能し、イベントに応答するオブジェクトはオブザーバーになります。同じゲームオブジェクトにある異なるコンポーネントは、サブジェクトになることもオブザーバーになることもあります。同じコンポーネントでも、一方のコンテキストではサブジェクトになり、別ではオブザーバーになることがあります。

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。Unity の UI システムを使用したことがある（例：UI ボタンの OnClick イベントを作成する）方であれば、すでに使用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 AudioObservers の 2 つの OnThingHappened メソッドからのレスポンス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することができます。

UnityEvents は、デザイナーやプログラマー以外のユーザーがゲームプレイのイベントを制作できるようになる、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ためご注意ください。

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。例については、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）のモジュールを参考にしてください。

## 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が一切不要です。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 にも UnityEvents と UnityActions が備わっています。
- **各オブザーバーに独自のイベント処理ロジックが実装される：**この方法により、観察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。こうすることで、UI 要素が具体的なゲームイベントや条件をリッスンし、適宜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に関しては、以下のこと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は注意してください。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、イベントをパブリッシュ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（以下参照）を使用すると、オブジェクト間のもつれを解く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増えてしまいます。大きなシーンや多数のゲームオブジェクトにより、パフォーマンスが低下するおそれがあります。

## 改善策

ここで紹介しているのは基本的なオブザーバーパターンのみですが、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することができます。

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。項目が追加または削除されたとき、またはリストが更新されたときに、オブザーバーに通知することができます。
- **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。1 つ以上のオブザーバーに適用される可能性が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 を渡します（型 Action<int> を使用します）。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の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』には、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ます。

また、Unity Open Project にて、UnityEvents をリレーするために ScriptableObject が使用される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ことをお勧めします。単一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て使用することで、複数のイベントを 1 つのイベントキューにカプセル化することができます。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 回ずつ再生するか、必要に応じて選択的に無視することができます（例：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している場合）。

多くの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の一部に含まれます。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）

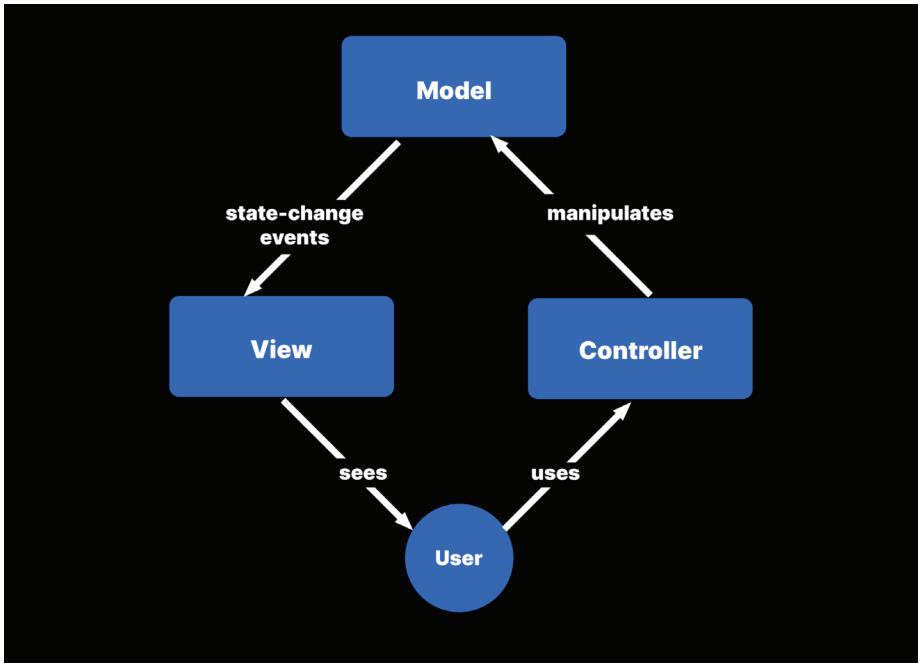
モデルビューコントローラー（MVC）は、ユーザーインターフェースの開発中に一般的に使用されるデザインパターンのファミリーです。

MVC の背景にある概念は、ソフトウェアの論理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。これにより不要な依存関係が減り、スパゲッティコードの削減に寄与することがあります。

## 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が実行されることも、計算が行われることもありません。
- **ビューとはインターフェースのこと：**ビューによってデータのグラフィカルな表現がフォーマットされ、画面上にレンダリングされます。
- **コントローラーによってロジックが処理される：**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

また、この関心の分離により、これら 3 つのパーツが互いにどのように対話するかが具体的に定義されます。モデルではアプリケーションデータを管理する一方で、ビューではそのデータをユーザーに表示します。コントローラーでは入力を処理し、ゲームデータに関わるあらゆる意思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。1 つのパーツがデータを保持し、もう 1 つのパーツがそのデータを処理して、最後のパーツがそのデータをユーザーに表示します。

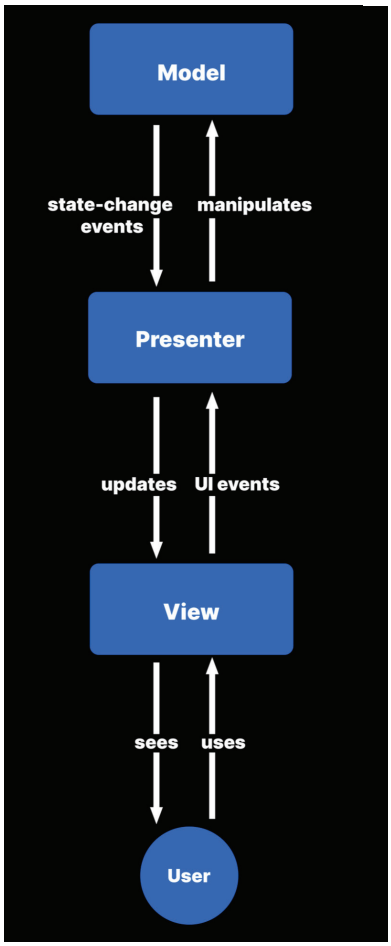
表面上は、単一責任の原則を拡張したものと見なすことができます。各パーツが 1 つのことを担います。これが MVC アーキテクチャの利点の 1 つです。

## 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 ツールキットまたは Unity UI）は必然的にビューとして機能します。エンジンには完全なユーザーインターフェースの実装が備わっているため、個々の UI コンポーネントを一から開発する必要はありません。

ただし、従来型の MVC パターンに従うことには、モデルのデータの変化をランタイムにリッスンするために、ビュー固有のコードが求められます。

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 のバリエーションの 1 つを使用することを選択します。ここでは、ビューによってモデルが直接的には観察されません。代わりに、以下のようなことが行われます。




MVP：MVC の 1 つのバリエーション

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 によって入力を処理するレイヤーが切り替わります。コントローラーの代わりに、ビューがユーザー入力を処理することを担います。

イベントやオブザーバーパターンがこのデザインにどのように含まれているかに注目してください。ユーザーは Unity UI の Button、Toggle、Slider の各コンポーネントと対話することができます。ビューレイヤーによりこの入力が UI イベントを介してプレゼンターに返され、今度はプレゼンターでこのモデルの操作が行われます。モデルからのステート変更イベントにより、そのデータが更新されたことがプレゼンターに伝わります。変更されたデータがプレゼンターからビューに渡され、そこで UI が最新の情報に更新されます。

## 例：体力インターフェース

MVP の例に形にするために、キャラクターやアイテムの体力（耐久力）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。機能を追加するほど、拡張が必要になったときに煩雑になります。

代わりに、より MVP を重視したやり方で体力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 `Health` と `HealthPresenter` に分割します。`Health` コンポーネントは以下のようになります。

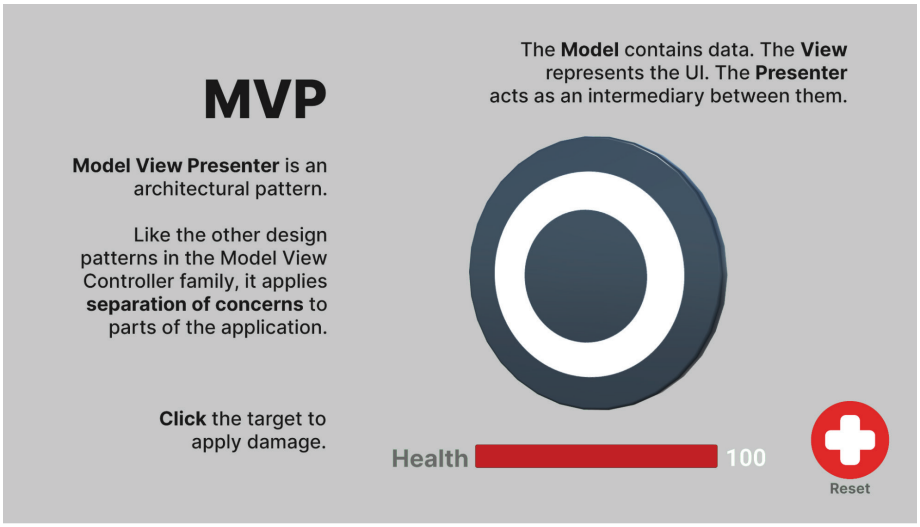
```
public class Health: MonoBehaviour
{
    public event Action HealthChanged;

    private const int minHealth = 0;
    private const int maxHealth = 100;
    private int currentHealth;

    public int CurrentHealth { get => currentHealth; set =>
currentHealth = value; }
    public int MinHealth => minHealth;
    public int MaxHealth => maxHealth;

    public void Increment(int amount)
    {
        currentHealth += amount;
        currentHealth = Mathf.Clamp(currentHealth, minHealth,
maxHealth);
        UpdateHealth();
    }

    public void Decrement(int amount)
    {
        currentHealth -= amount;
        currentHealth = Mathf.Clamp(currentHealth, minHealth,
maxHealth);
        UpdateHealth();
    }

    public void Restore()
    {
        currentHealth = maxHealth;
        UpdateHealth();
    }

    public void UpdateHealth()
    {
        HealthChanged?.Invoke();
    }
}
```

このバージョンでは、Health がモデルとして機能します。実際の体力値が格納され、その値が変化するたびにイベント HealthChanged が呼び出されます。Health にはゲームプレイのロジックは含まれません。データの増減を担うメソッドのみが含まれます。

ただし、ほとんどのオブジェクトでは Health 自体を操作しません。そのタスクは HealthPresenter が担います。

```
public class HealthPresenter : MonoBehaviour
{
    [SerializeField] Health health;
    [SerializeField] Slider healthSlider;

    private void Start()
    {
        if (health != null)
        {
            health.HealthChanged += OnHealthChanged;
        }
        UpdateView();
    }

    private void OnDestroy()
    {
        if (health != null)
        {
            health.HealthChanged -= OnHealthChanged;
        }
    }

    public void Damage(int amount)
    {
        health?.Decrement(amount);
    }

    public void Heal(int amount)
    {
        health?.Increment(amount);
    }

    public void Reset()
    {
        health?.Restore();
    }

    public void UpdateView()
    {
        if (health == null)
            return;

        if (healthSlider !=null && health.MaxHealth != 0)
        {
            healthSlider.value = (float) health.CurrentHealth / (float)
health.MaxHealth;
        }
    }

    public void OnHealthChanged()
    {
        UpdateView();
    }
}
```

その他のゲームオブジェクトでは、Damage、Heal、Reset を使用して体力値を変更するのに、HealthPresenter を使用する必要があります。HealthPresenter では通常、Health から HealthChanged イベントが発生するまで、UpdateView を使用したユーザーインターフェースの更新を待機します。これは、モデルで値を設定するのに少し時間がかかる場合に便利です（例：値をディスクに保存する、値をデータベースに格納する）。




MVP を使用した体力インターフェースのサンプル

このサンプルプロジェクトでは、ユーザーがターゲットオブジェクトをクリックしてダメージを与えたり、ボタンを使用して体力をリセットしたりすることができます。これらの操作により、Health が直接変更されるのではなく、HealthPresenter に情報が伝わります（Damage または Reset が呼び出されます）。Health からイベントが発生し、その値が変わったことが HealthPresenter に伝わると、UI テキストと UI スライダーが更新されます。

## 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輝きを放ちます。ゲームの開発にかなりの規模のチームが必要とされ、ローンチ後長期にわたってメンテナンスを行うことを予測している場合は、以下の点からメリットを享受でき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すことで、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る場合は、ビューについては任せてしまいましょう。チームから独立して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、エディターで実際に再生モードに入ることなく、オブジェクトのコードでの動作についてシミュレーションを行うことができます。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より小さなクラスを作成する傾向にあるため、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、バグが潜んでいる可能性が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にまで大きくならないと、そのメリットははっきりとはわかりません。いずれかのパターンを Unity プロジェクトに実装する前に、以下のことを検討する必要があります。

— **事前に計画を立てる必要がある：**このガイドで紹介するその他のパターンとは異なり、MVC と MVP はより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、ある程度の組織化と、事前により多くの作業が求められます。

— **Unity プロジェクト内のすべてがそのパターンに収まるわけではない：**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（例：MeshRenderer）。また、シンプルなスクリプトでは MVC/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について判断を下す必要があります。一般的には、単体テストを指針とすることをお勧めします。MVC/MVP によってテストが容易になる可能性がある場合は、アプリケーションのその側面のためにそれらを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。

# 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 開発において直面する可能性がある最も一般的なパターンのいくつかについて理解を深めるのに、このガイドがお役に立つことがあれば幸いです。

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。ときにはパターンを使用しない選択も重要です。

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、プレイヤーのために他にはない楽しい体験を制作するという重要なことに集中できます。

すでにあるもの一から作成する必要はありませんが、もちろんそこに自分の持ち味を与えてもかまいません。

## 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。

- **Adapter（アダプター）**：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。
- **Flyweight（フライウェイト）**：オブジェクトの数が多い場合に、共通のプロパティを基本クラスで共有し、リソースを節約します。例えば、森林をデザインしているときに、1 本の樹木のすべてのユニバーサルプロパティを基本クラス Tree に格納します。こうすることで、それらをサブクラス（PineTree、MapleTree など）で繰り返す必要がなくなります。
- **Double buffer（ダブルバッファ）**：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することができます。これは、プロシージャルシミュレーション（セルラーオートマタなど）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。
- **Dirty flag（ダーティフラグ）**：この手法により、ゲーム内で何か変化があったが、コストのかかる操作（ディスクに保存する、物理演算シミュレーションを実行するなど）がまだ実行されていない場合に、ブール値を設定することができます。
- **Interpreter/Bytecode（インタープリター / バイトコード）**：MOD 作成のサポートを追加する、つまりノンプログラマーにゲームを拡張することを許可する場合に、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。こうすることで、バイトコードコンポーネントによってそのインタープリター型言語が C# のゲームコードに翻訳されます。
- **Subclass sandbox（サブクラスサンドボックス）**：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。こうすることで、子クラスをいろいろと混ぜ合わせて新しい組み合わせを作成することができます。
- **Type object（タイプオブジェクト）**：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。ゲームデザイナーが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。

— **Data locality（データローカリティ）**：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、データをよりキャッシュフレンドリー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。

— **Spatial partitioning（スペーシャルパーティショニング）**：大規模なシーンやゲームの世界で、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。

— **Decorator（デコレーター）**：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、特別な機能を注入したり、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、その武器に特性を付与することができます。

— **Facade（ファサード）**：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ことをお勧めします（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。

— **Template method（テンプレートメソッド）**：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ますが、そのアルゴリズムの全体的な構造は変更することなく、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。

— **Strategy（ストラテジー）**：この行動パターン（ポリシーパターンとも呼ばれます）は、アルゴリズムのファミリーを定義し、それぞれを 1 つのクラスにカプセル化するのに役立ちます。これにより、各アルゴリズム（ストラテジー）をランタイムに置き換えることができるようになります。例えば、経路検索システムを作成した場合は、コンテキストに応じてゲームプレイ中にスワップできる複数のアルゴリズム（A*、ダイクストラの最短経路など）を定義するのに、ストラテジーパターンを使用することができます。

— **Composite（コンポジット）**：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（1 つのリーフと 1 つのコンテナ）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。

## Unity クリエイターのためのプロフェッショナルトレーニング

Unity プロフェッショナルトレーニングにより、Unity でより高い生産性を発揮し、効率的にコラボレーションを行うためのスキルと知識が得られます。Unity では、業界を問わず、あらゆるスキルレベルのプロフェッショナル向けに設計された、充実したトレーニングのカタログを複数の形式で提供しています。

すべての資料は、Unity の経験豊富なインストラクショナルデザイナーが Unity のエンジニアや製品チームと協力して制作しています。これは、常に Unity の最新テクノロジーに関する最新のトレーニングを受けることができることを意味します。

こちらで Unity Professional Training が皆さんとチームにとってどのように役立つかについての詳細を確認してください。


Unity®