UNITY FOR GAMES -BOOK

ゲームプログラミングパターンで あなたのコードをレベルアップ

2021 LTS EDITION

Unity

内容

Unityで作業する場合、車輪の再発明をする必要はありません。誰かがすでに発明してくれている可能性が高いのです。

あなたが遭遇するソフトウェア設計の問題には、1000人の開発者が過去に遭遇したことがあります。彼らに直接アドバイスを求めることはできませんが。デザインパターンを通じて、彼らの決断から学ぶことができます。

デザインパターンは、ソフトウェア工学で見られる一般的な問題に対する一般的な解決策です。これらは、あなたのコードにコピー&ペーストできるような完成された解決策ではありませんが、デザインパターンはあなたの道具箱の中の追加の道具と考えることができます。しかし、デザインパターンは、あなたの道具箱の中の特別なツールと考えることができます。

本書は、Unityの開発においてよく知られているデザインパターンをまとめたものです。このガイドでは、C#の基本的な知識がある方が対象ですが、より理解しやすいように、例を簡略化し、専門用語を減らしています。

デザインパターンにまだ慣れていない方や、すぐに復習したい方には、ゲーム開発でデザインパターンを適用できる一般的なシナリオもご紹介しています。他のオブジェクト指向言語（Java、C++など）からC#に乗り換える人のために、これらのサンプルは、パターンをUnityに特別に適応させる方法を示しています。

デザインパターンは単なるアイデアです。すべての場面で適用できるわけではありません。しかし、正しく使用すれば、スケールの大きなアプリケーションを構築するのに役立ちます。コードの可読性を高め、コードベースをよりクリーンにするために、あなたのプロジェクトにそれらを統合してください。パターンを使いこなすうちに、どのような場合にパターンが開発プロセスを加速させるのかがわかるようになります。

そうすれば、車輪の再発明をやめて、何か新しいことに取り組めるようになります。

投稿者

このガイドは、映画やテレビで 15 年以上の経験を持つ 3D および視覚効果アーティストであり、現在は独立したゲーム開発者および教育者として活動している Wilmer Lin が執筆したものです。また、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の Thomas Krogh-Jacobsen とシニア Unity エンジニアの Peter Andreasen と Scott Bilas も多大な貢献をしています。

このガイドとKISSの原則を使って

このガイドの目的は、コードについて考え、組織化する新しい方法を提示することです。このガイドで紹介されているソフトウェア設計のいくつかのパターンは、Unityの開発に適応されています。

付属のサンプルプロジェクトでは、コードの一部を文脈に沿って示しています。対応するシーンを使用して、これらのデザインパターンとその基礎となる原則を探求してください。

しかし、これらの例を検討する際には、問題に取り組むための一律の「正しい方法」が存在しないことを忘れないでください。サンプルコードは、数あるソリューションの中の1つです。

迷ったら、このガイドのすべてをKISSの原則でフィルタリングしてください："Keep it simple, stupid".必要な場合のみ、複雑さを追加してください。

どのデザインパターンにもトレードオフがあります。それは、維持するための構造が増えることであれ、最初の設定が増えることであれ、です。実装する前に、その利点が余分な作業を正当化するかどうかを決定してください。

もし、あるパターンがあなたの特定の問題に適用できるかどうかわからない場合は、より自然にフィットすると感じられる状況を待つ方がよいかもしれません。新しいから使うのではなく、必要なときに使う。

そうすれば、デザインパターンは本来の目的である、より良いソフトウェアを開発するための助けとなるはずです。

さっそく始めてみましょう。

SOLID原則

パターンそのものに触れる前に、パターンの動作に影響を与える設計原理をいくつか見ておきましょう。

SOLIDとは、ソフトウェア設計の核となる5つの基本事項の頭文字をとったニーモニックである。

- 単一責任
- オープンクローズド
- リスコフ置換
- インターフェース分離
- 従属性の反転

それぞれの概念を検証し、コードをより理解しやすく、柔軟で、保守しやすいものにするためにどのように役立つか見てみましょう。

単一責任原則

クラスが変わる理由は、その単一の責任だけであるべきです。

SOLIDの原則のうち、まず最も重要なのが「単一責任原則（SRP）」である。これは、各モジュール、クラス、関数が1つのことに責任を持ち、その部分のみをロジックとしてカプセル化するというものである。

一枚岩のクラスを構築するのではなく、多くの小さなものからプロジェクトを組み立てましょう。より短いクラスとメソッドは、説明、理解、実装がより簡単になります。

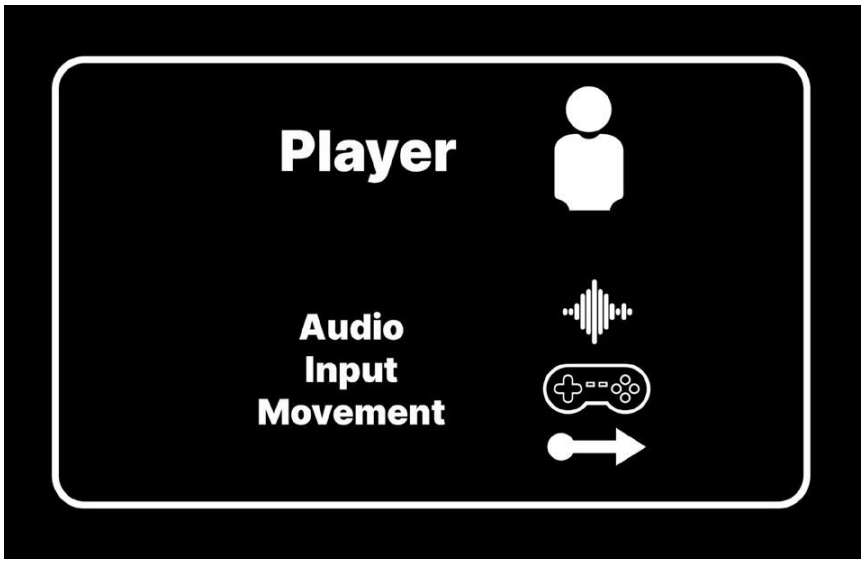
Unityをしばらく使っていれば、この概念にはすでに馴染みがあることでしょう。GameObjectを作成すると、その中には様々な小さなコンポーネントが含まれています。例えば、以下のようなものです。

- 3Dモデルへの参照を格納するMeshFilter
- モデル表面の画面上での見え方を制御するレンダラ
- スケール、回転、位置を格納する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リジッドボディ

各コンポーネントは、1つのことをしっかりと行います。GameObjectからシーン全体を構築するのです。その構成要素間の相互作用が、ゲームを可能にするのです。

スクリプト部品も同じように組み立てます。それぞれを明確に理解できるように設計します。そして、それらが連動して複雑な動作をするようにします。

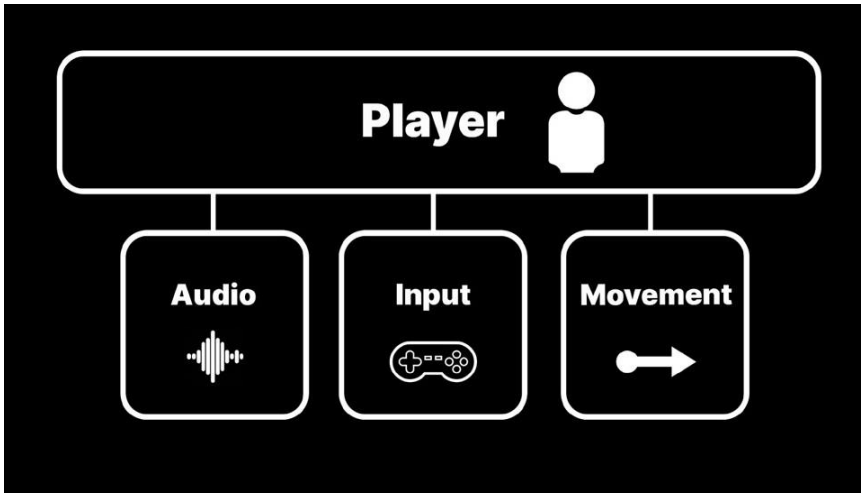
単一責任を無視するのであれば、これを行うカスタムコンポーネントを作成してもよいかもしれません。



複数の責任を負うプレイヤースクリプト

```
public class UnrefactoredPlayer : MonoBehaviour.
{

    [SerializeField] private string inputAxisName;
    [SerializeField] private float positionMultiplier;
    private float yPosition.SerializeFieldは、入力軸の位置を指定
    します。
    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, yPosi-)
tion * positionMultiplier, transform.position.z);
    }

    private void OnTriggerEnter(Collider other)
    {
        bounceSfx.Play()を実行します。
    }
}
```

このUnrefactoredPlayerクラスは様々な役割を担っています。プレイヤーが何かに衝突したときに音を鳴らし、入力を管理し、移動を処理します。今は比較的短いクラスでも、プロジェクトが発展するにつれてメンテナンスが面倒になるでしょう。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。



プレイヤー、単一の責任を持つクラスへのリファクタリング

```
[RequireComponent(typeof(PlayerAudio), typeof(PlayerInput),
typeof(PlayerMovement))] とする。
public class Player : MonoBehaviour (プレイヤー)
{
    SerializeField] private PlayerAudio playerAudio;
    [SerializeField] private PlayerInput playerInput;
    [SerializeField] private PlayerMovement
    playerMovement.SerializeField; [SerializeField] private
    PlayerMovement playerMovement;

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

public class PlayerAudio : MonoBehaviour (プレイヤーオーディオ)
{
    ...
}

public class PlayerInput : MonoBehaviour.
{
    ...
}

public class PlayerMovement :モノビヘイビア
{
    ...
}
```

プレイヤースクリプトは、他のスクリプトコンポーネントを管理することはできますが、各クラスは1つのことしか行いません。この設計により、特にプロジェクトの要件が時間の経過とともに変化した場合に、コードを修正することがより身近になります。

しかし一方で、単一責任原則と常識のバランスをとる必要があります。極端に単純化しすぎて、1つのメソッドしか持たないクラスを作らないようにしましょう。

これらの目的を念頭に置きながら、一任主義に取り組んでください。

> **読みやすさ。**短いクラスは読みやすい。明確なルールはありませんが、多くの開発者は200-300行を上限としています。自分で決める
> あるいはチームとして、何をもって "短い "と判断するか。この閾値を超えたら、より小さなパーツにリファクタリングできるかどうか判断します。
>
> **拡張性がある。**小さなクラスからより簡単に継承することができます。
> 意図しない機能が壊れる心配がなく、修正・交換が可能です。
>
> **再利用性。**クラスを小さくモジュール化し、ゲームの他の部分で再利用できるように設計します。

リファクタリングするときは、コードを整理することで自分や他のチームメンバーの生活の質がどのように向上するかを考えてください。最初に多少の努力をすることで、後で多くの問題を回避することができます。

シンプルは簡単ではない

ソフトウェア設計でよく語られる「シンプルさ」は、信頼性の前提条件です。あなたのソフトウェア設計は、生産中の変化に対応できますか？アプリケーションを長期にわたって拡張し、維持することができますか？

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、シンプルさを強制するのに役立ちます。そうすることで、コードのスケーラビリティや柔軟性、可読性が高まります。しかし、これらには特別な作業と計画が必要です。「シンプル "は "簡単 "とイコールではありません。

パターンを使わなくても同じ機能を作ることはできますが（多くの場合、より速く）、速く簡単なものが必ずしもシンプルなものになるとは限りません。単純なものを作るということは、それを集中させるということです。1つのことをするために設計し、他のタスクで複雑にしすぎないようにします。

Rich Hickeyの講義「Simple Made Easy」では、シンプルであることがいかに優れたソフトウェアを作るのに役立つかについて説明しています。

オープンクローズの原理

SOLID設計におけるOCP（open-closed principle）とは、クラスは拡張に対してはオープンであるが、変更に対してはクローズでなければならないというものです。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。

その典型的な例が、図形の面積を計算することです。矩形や円の面積を返すメソッドを持つAreaCalculatorというクラスを作ることができます。

面積を計算するために、RectangleクラスはWidthとHeightを持っています。円はRadiusとπの値だけが必要です。

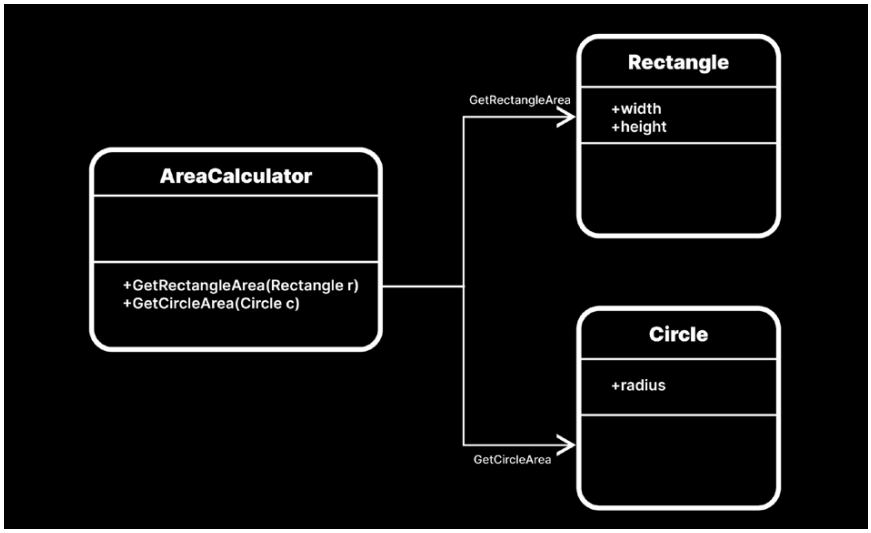
```
public class AreaCalculator
{
    public float GetRectangleArea(Rectangle rectangle)
    {
        rectangle.width * rectangle.height を返します。
    }

    public class GetCircleArea(Circle circle)
    {
        Circle.radius * Math.PI を返します。
    }
}
```

```
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、もしAreaCalculatorにさらに形状を追加したい場合は、新しい形状ごとに新しいメソッドを作成する必要があります。例えば、後で五角形や八角形を渡したいとしたらどうでしょう？さらに 20 個の図形が必要になったらどうしますか？AreaCalculator クラスはすぐに制御不能になります。

Shapeという基底クラスを作り、形状を処理するメソッドを一つ作ることも可能です。しかし、そうすると、ロジックの内部で各タイプの形状を処理するために複数のif文が必要になります。これでは、うまく拡張できない。

元のコード（AreaCalculatorの内部）を変更することなく、プログラムを拡張（新しい図形を使用する機能）するために開放したいのです。機能的には問題ないのですが、現在のAreaCalculatorはオープンクローズの原則に反しています。



新しい形状を取り込むために、AreaCalculatorをどのように設計すればよいのでしょうか。

代わりに、抽象的なShapeクラスを定義することを検討してください。

これには、CalculateAreaという抽象メソッドが含まれています。そして、RectangleとCircleをShapeから継承させると、それぞれのShapeが独自の面積を計算し、以下のような結果を返すことができるようになります。

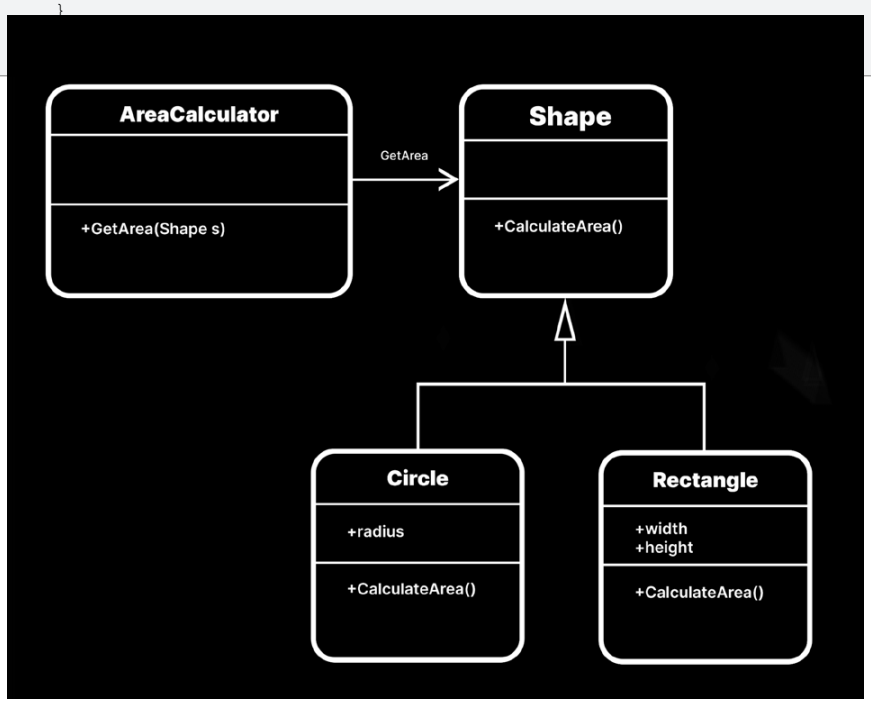
```
public abstract class Shape
{
    public abstract float CalculateArea();
}
```

```
public class Rectangle : Shape
{
    public float width;
    public float height;

    public override float CalculateArea()
    {
        width * height を返す。
    }
}

public class Circle :図形
{
    public float radius;

    public override float CalculateArea()
    {
        半径 * 半径 * Mathf.PI を返します。
    }
}
```

AreaCalculatorは、これを単純化することができます。

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

改良された`AreaCalculator`クラスは、抽象的な`Shape`クラスを適切に実装している任意の形状の面積を取得することができるようになりました。これにより、元のソースを変更することなく、`AreaCalculator`の機能を拡張することができます。



オープン・クローズド原理のためのクラスの見直し

新しい多角形が必要な場合は、`Shape`を継承した新しいクラスを定義するだけでよい。サブクラス化された各`Shape`は`CalculateArea`メソッドをオーバーライドして正しい面積を返します。

この新しいデザインは、デバッグを容易にします。新しい形状でエラーが発生した場合、`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ので、新しいコードに欠陥のあるロジックがないかどうかだけを調べればよいのです。

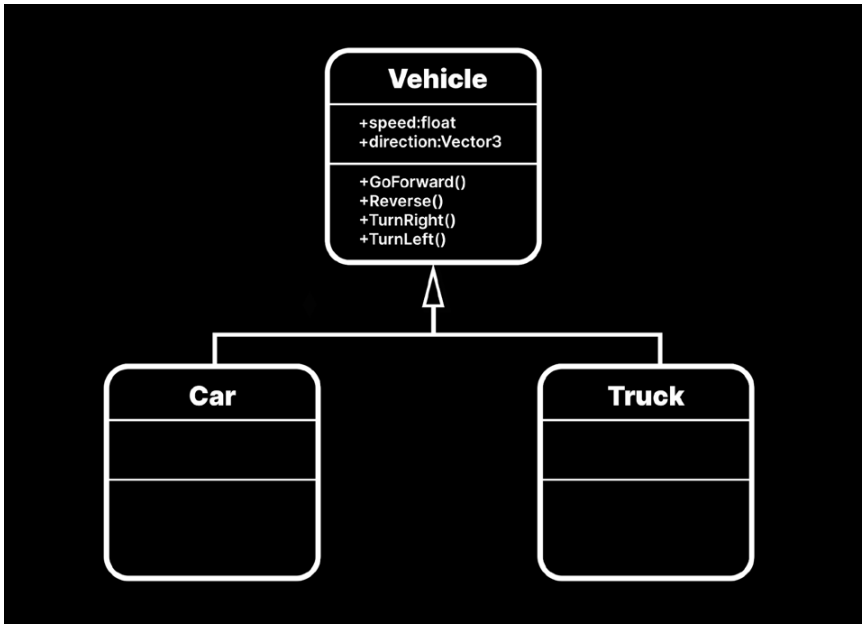
Unityで新しいクラスを作成するときは、インターフェースと抽象化を活用しましょう。これにより、ロジックに扱いにくい`switch`文や`if`文が含まれ、後で拡張するのが難しくなるのを避けることができます。OCPを尊重したクラスの設定に慣れれば、長期的に新しいコードを追加することが簡単になります。

リスコフ置換の原理

リスコフ置換原理（LSP）は、派生クラスはその基底クラスに対して置換可能でなければならないとするものである。オブジェクト指向プログラミングにおける継承は、サブクラスを通じて機能を追加することを可能にします。しかし、これは注意しないと不必要に複雑化する可能性があります。

SOLIDの3つ目の柱であるリスコフ置換の原則は、継承を適用してサブクラスをより堅牢で柔軟なものにする方法を説いたものである。

あなたのゲームでは、`Vehicle`というクラスが必要だとします。これは、アプリケーションで作成する乗り物のサブクラスのベースとなるクラスです。例えば、車やトラックが必要かもしれません。

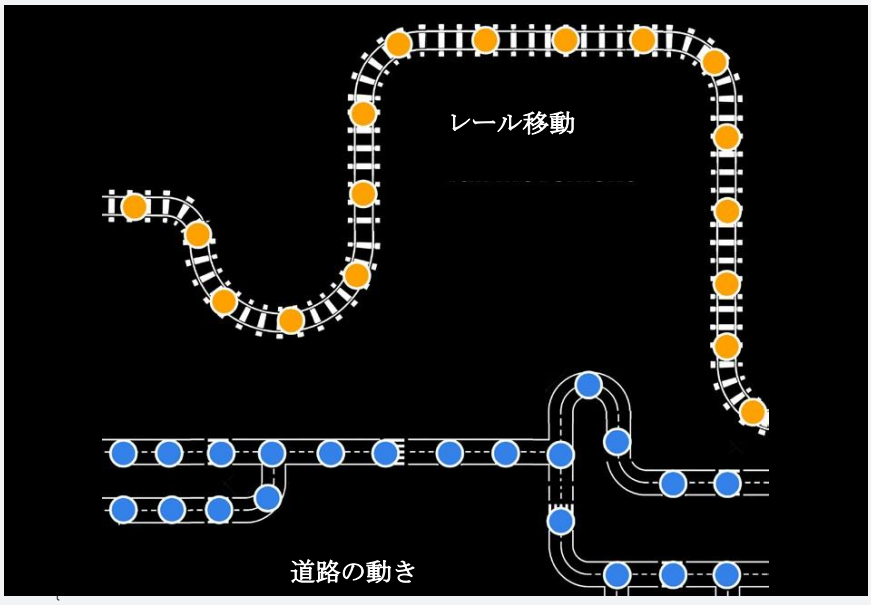


すべてはVehicleから継承される。

ベースクラス（`Vehicle`）が使えるところならどこでも、Carや`Truck`のようなサブクラスがアプリケーションを壊すことなく使えるはずです。

Vehicleクラスは次のようなものです。

例えば、乗り物をボード上で動かすターン制のゲームを作るとします。

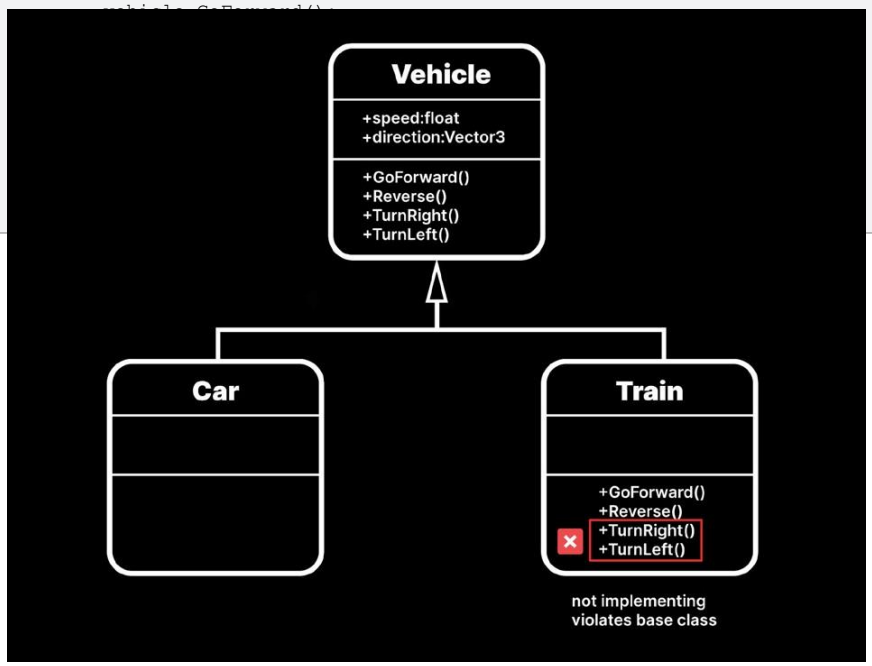


自動車対鉄道のゲーム例

また、Navigatorという別のクラスを用意して、所定の経路に沿って車両を操縦させることもできます。

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

このクラスはNavigatorのMoveメソッドにどんな乗り物でも渡すことができ、車やトラックでも問題なく動作することが期待されます。しかし、Trainというクラスを実装したい場合はどうなるでしょうか？



A列車は、あなたのベースクラスを侵害することになります。

列車は線路から離れることができないので、TurnLeft と TurnRight メソッドは列車クラスでは機能しません。もしNavigatorのMoveメソッドに列車を渡したとしても、これらの行に到達すると未実装のExceptionを投げる（あるいは何もしない）でしょう。型をそのサブタイプに置き換えることができない場合、リスコフ代入の原則に違反することになります。

列車はVehicleのサブタイプなので、Vehicleクラスを受け付けるところならどこでも使えると思うでしょう。そうでない場合は、コードが予測不可能な動作をする可能性があります。

Liskov置換の原則をより忠実に守るためのいくつかのヒントを考えてみましょう。

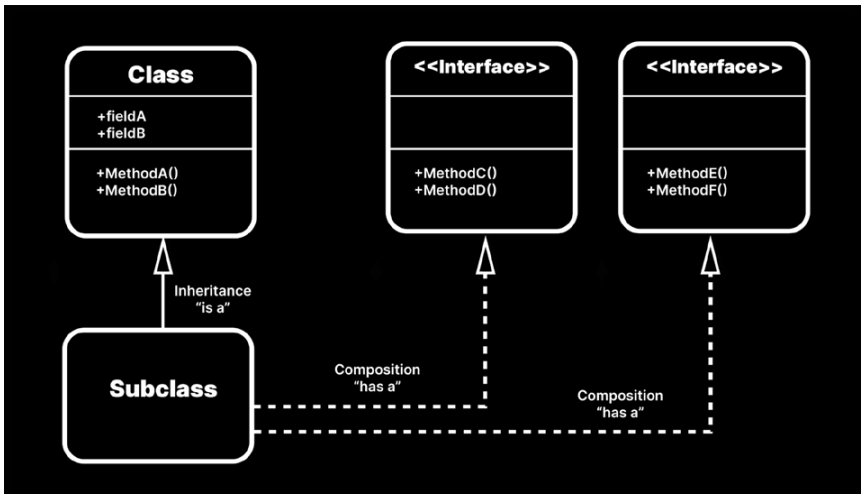
サブクラス化する際に機能を削除している場合、リスコフ置換を破っている可能性があります。NotImplementedExceptionは、この原則を破っていることを示す決定的な証拠です。メソッドを空白にした場合も同様です。サブクラスがベースクラスのように動作しない場合、LSPに従っていないことになります。たとえ、明示的なエラーや例外がなくてもです。

抽象化をシンプルにする。ベースクラスに多くのロジックを入れるほど、LSPが壊れる可能性が高くなります。ベースクラスは、派生するサブクラスに共通する機能のみを表現すればよい。

サブクラスは、ベースクラスと同じパブリックメンバを持つ必要があります。これらのメンバは、同じシグネチャと呼び出すときの動作も同じである必要があります。

クラス階層を確立する前に、クラスAPIを検討する。車と電車は別々の親クラスから継承する方が合理的かもしれません。現実の分類は、必ずしもクラス階層に反映されるとは限りません。

継承よりもコンポジションを優先する。継承によって機能を受け渡すのではなく、特定の動作をカプセル化するインターフェースや別のクラスを作成する。そして、異なる機能を組み合わせて「コンポジション」を構築する。

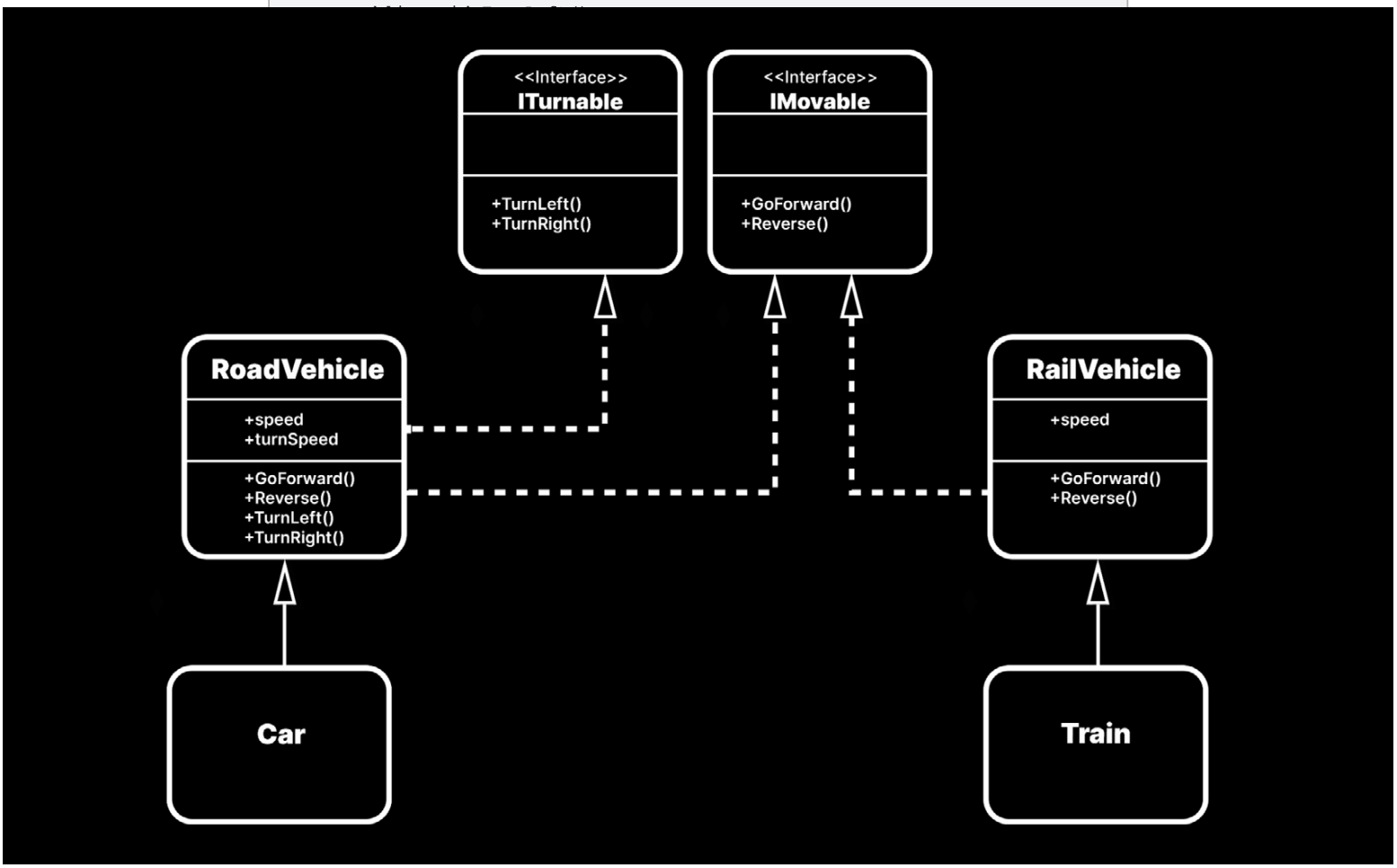


継承より合成

この設計を修正するには、オリジナルのVehicleタイプを破棄し、機能の多くをインターフェースに移行させます。

LSPの原則をより忠実に守って、RoadVehicle型とRailVehicle型を作成します。そして、CarとTrainはそれぞれのベースクラスを継承する。

```
    public void TurnRight();
```



リスコフ代入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 : ロードビークル
{
    ...
}

public class Train : RailVehicle (鉄道車両)
{
    ...
}
```

このように、機能は継承ではなく、インターフェースを通じて提供されます。CarとTrainは同じ基底クラスを共有しないので、LSPを満たすようになりました。RoadVehicleとRailVehicleを同じ基底クラスから派生させることもできますが、この場合、あまり必要ありません。

このような考え方は、現実の世界についてある種の仮定を持っているため、直感に反することがあります。ソフトウェア開発では、これを「サークルエリプス問題」と呼んでいます。実際の「is a」関係がすべて継承につながるわけではありません。ソフトウェア設計は、現実の知識ではなく、クラス階層を駆動させたいことを忘れないでください。

リスコフ置換の原則に従って、コードベースの拡張性と柔軟性を保つために、継承の使用方法を制限します。

インターフェース分離の原理

インターフェース分離の原則（ISP）は、どのクライアントも自分が使わないメソッドに依存することを強制されるべきではないとするものです。

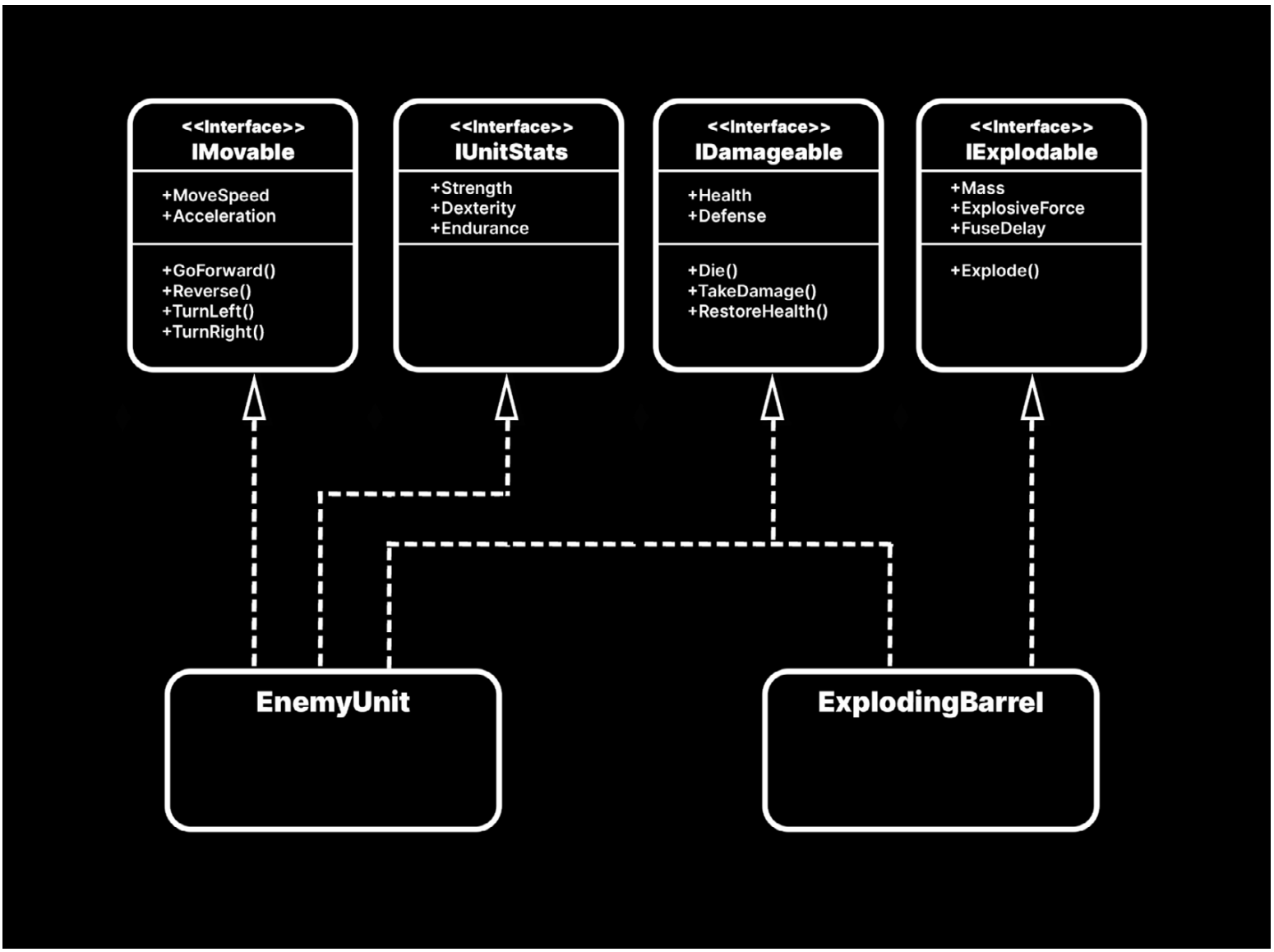
つまり、大きなインターフェースは避けるということです。単一責任の原則と同じ考えで、クラスとメソッドを短くするように指示します。これにより、柔軟性を最大限に高め、インターフェイスをコンパクトにし、集中させることができます。

様々なプレイヤーユニットが登場するストラテジーゲームを作ることを想像してみてください。各ユニットは、体力やスピードなどのステータスが異なります。すべてのユニットが同じような機能を実装していることを保証するためのインターフェイスを作りたいと思うかもしれません。

```
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; }
    public int Endurance { get; set; }.

}
```

例えば、壊せる樽や木箱のような破壊可能なプロップを作りたいとします。この小道具には、動かないにもかかわらずヘルスという概念も必要です。また、木箱や樽は、ゲーム内の他のユニットに関連する多くのアビリティを持ちません。

壊れやすいプロップに多くのメソッドを与える1つのインターフェイスを作るよりも、いくつかの小さなインターフェイスに分割してください。それらを実装したクラスは、必要なものだけを取り混ぜて使うことができます。



インターフェースを小さく分割する。

```
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; }
    public int Endurance { get; set; }.
}
```

また、爆発する樽のためにIExplodableインターフェイスを追加することができます。

```
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る。

爆発する樽は、IDamageableとIExplodableを使うことができ、他のインターフェースの不必要なオーバーヘッドを必要としません。

```
public class Barrel : MonoBehaviour, IDamageable, IExplodable
{
}

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

ここでもLiskov置換の例と同様に、継承よりも合成を優先します。インターフェース分離の原則は、システムを切り離し、修正や再展開を容易にするのに役立ちます。

依存関係逆転の原理

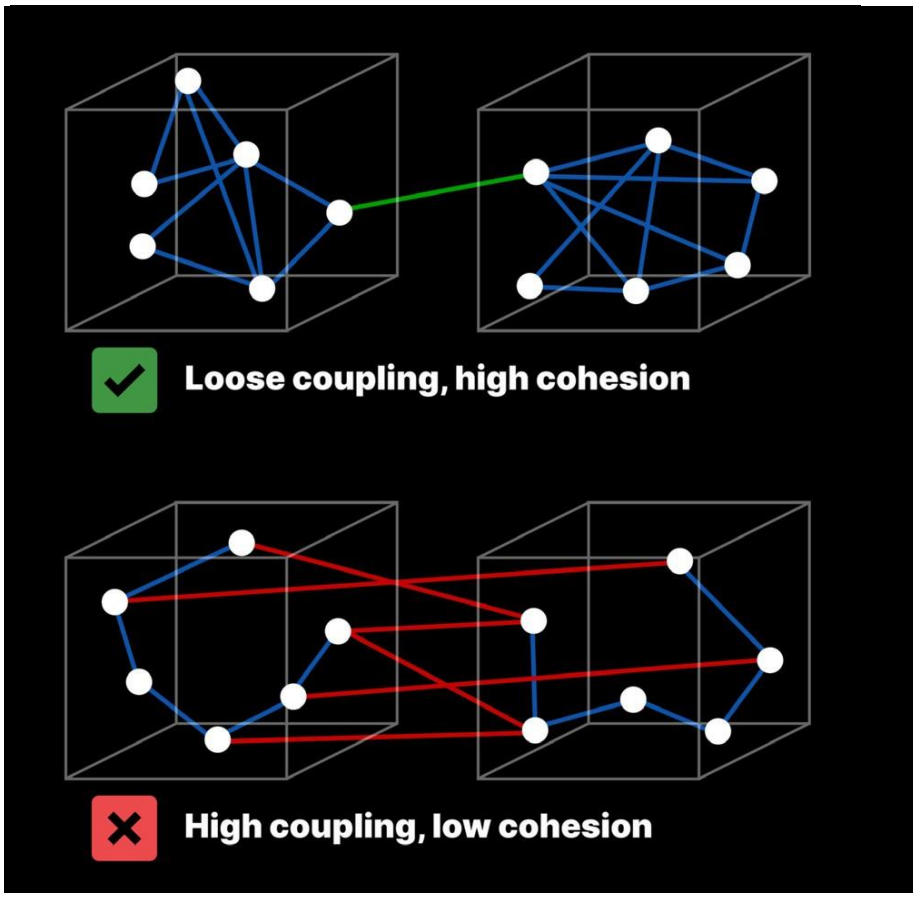
依存関係逆転の原則（DIP）は、高レベルのモジュールは低レベルのモジュールから直接何かをインポートすべきではない、と言っています。どちらも抽象化されたものに依存すべきです。

それが何を意味するのか、紐解いてみましょう。あるクラスが他のクラスと関係を持つとき、それは依存関係またはカップリングを持っています。ソフトウェア設計における依存関係は、それぞれ何らかのリスクを伴います。

あるクラスが他のクラスの動作について知りすぎていると、最初のクラスを修正することで2番目のクラスがダメージを受けたり、その逆が起こったりします。高度なカップリングは、不潔なコードの実践と見なされます。アプリケーションの1つの部分のエラーが雪だるま式に増えていくことがあります。

クラス間の依存関係はできるだけ少なくするのが理想的です。また、各クラスは外部との接続に頼るのではなく、内部の部品が一体となって動作する必要があります。オブジェクトは、内部ロジックやプライベートロジックで機能するとき、結束力があるとみなされます。

最良のシナリオは、疎結合と高結合力を目指すことです。

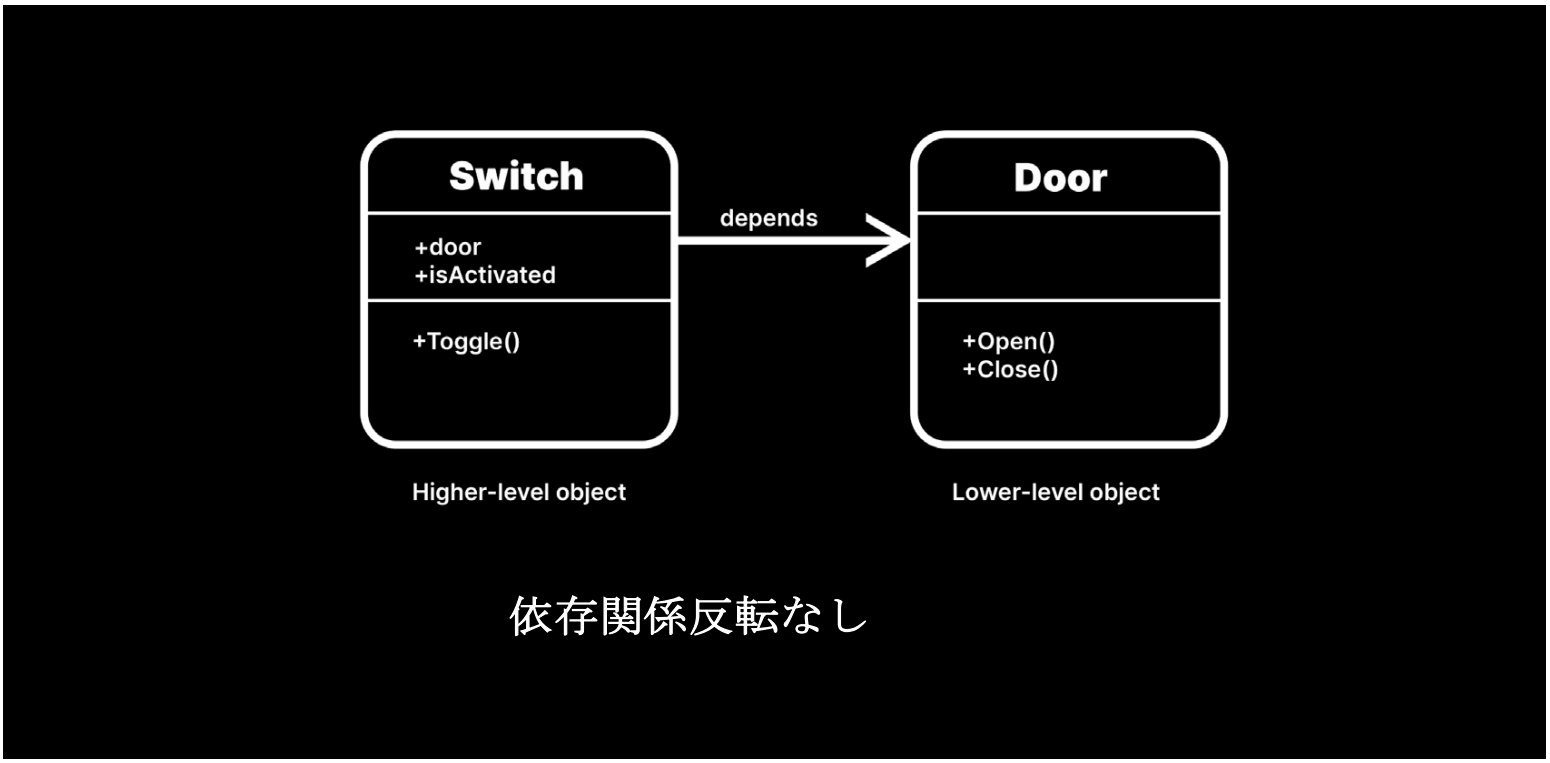


高い結合力を持つ緩い結合を目指す。

ゲームアプリケーションを修正し、拡張できるようにする必要があります。もし、もろくて修正に抵抗があるようなら、現在どのような構造になっているかを調査してください。

依存関係逆転の原則は、このようなクラス間の緊密な結合を減らすのに役立ちます。アプリケーションでクラスやシステムを構築する場合、当然ながら "高レベル "のものと "低レベル "のものがあります。高レベルのクラスは、何かを成し遂げるために低レベルのクラスに依存します。SOLID は、これを切り替えるように指示します。

キャラクターがレベルを探索し、ドアを開くトリガーとなるようなゲームを作っているとします。Switchというクラスとdoorというクラスを作りたいと思うかもしれません。



Switch（高レベル）は、Door（低レベル）クラスに直接依存します。

高次元では、キャラクターが特定の場所に移動して、何かが起こることを望んでいます。それを担ってくれるのがSwitchです。

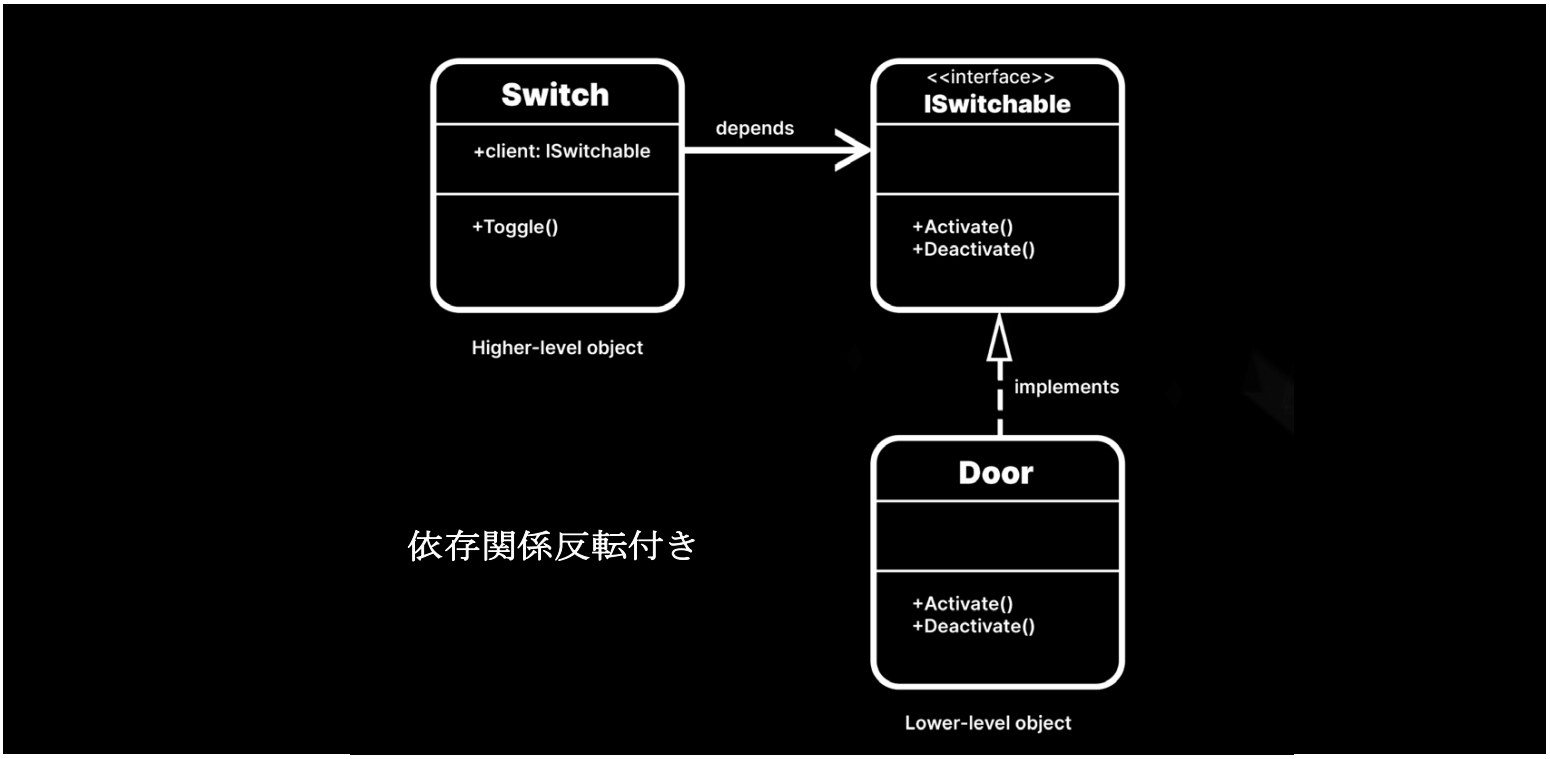
低レベルには、ドアジオメトリを開く方法の実際の実装を含む別のクラス、Doorがあります。簡略化のため、Debug.Logステートメントを追加して、ドアの開閉のロジックを表現しています。

```
public class Switch : MonoBehaviour.
{
    public Door ドア。
    public bool isActivated;

    public void Toggle()
    {
        if (isActivated)
        {
            isActivated = false;
            door.Close();
        }
        さもなくば
        {
            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Toggleメソッドを呼び出して、ドアを開閉することができます。これはうまくいくのですが、問題は、ドアからスイッチに直接、依存関係が配線されていることです。例えば、照明や巨大ロボットを作動させるなど、Switchのロジックがdoor以外にも必要な場合はどうでしょうか？

Switchクラスにメソッドを追加することは可能ですが、オープンクローズの原則に反することになります。機能を拡張するたびに、元のコードを修正しなければならない。

またしても抽象化の出番です。ISwitchableというインターフェイスをクラスの間に挟むことができる。



2つのクラス間のインターフェースISwitchable

ISwitchableは、アクティブかどうかを知るためのパブリックプロパティと、アクティブ化および非アクティブ化のためのいくつかのメソッドを必要とするだけです。

```
public interface ISwitchable
{
    public bool IsActive { get; }.

    public void Activate();
    public void Deactivate();
}
```

そうすると、Switchは、直接ドアではなく、ISwitchableなクライアントに依存する、こんな感じになります。

```
public class Switch : MonoBehaviour.
{
```

一方、ISwitchableを実装するためには、Doorを作り直す必要があります。

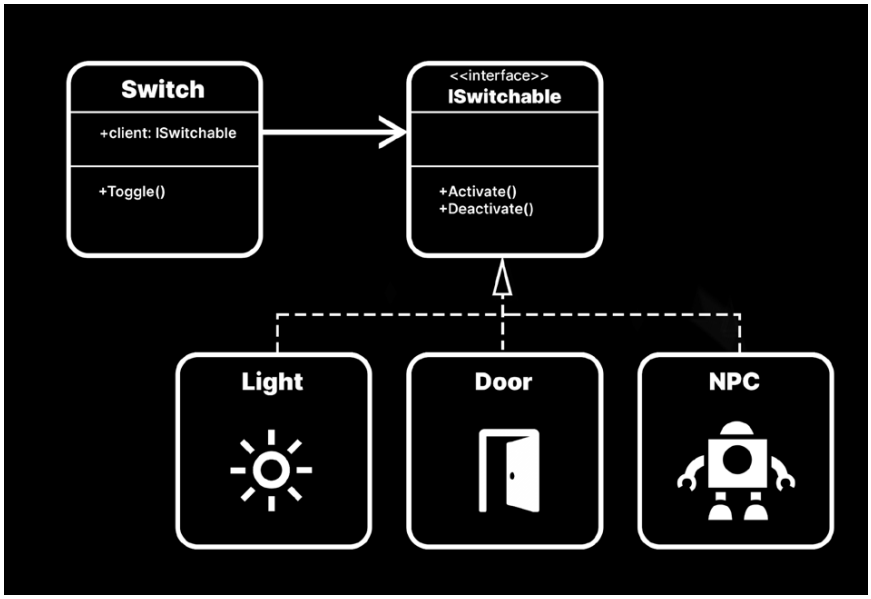
```
public class Door : MonoBehaviour, ISwitchable
{
    private bool isActive;
    public bool IsActive => isActive;

    public void Activate()
    {
        isActive = true とする。
        Debug.Log("The door is open.");
    }

    public void Deactivate()
    {
        isActive = false とする。
        Debug.Log("The door is closed.");
    }
}
```

これで依存関係が逆転しましたね。インターフェイスは、スイッチをドアだけに固定配線するのではなく、それらの間に抽象化を作成します。スイッチは、もはやドア特有のメソッド（OpenとClose）に直接依存することはありません。

代わりにISwitchableのActivateとDeactivateを使用します。

この小さな、しかし重要な変更は、再利用性を促進します。以前はSwitchはDoorでしか動作しませんでしたが、現在はISwitchableを実装しているものであれば何でも動作します。

これにより、Switchが作動するクラスをより多く作ることができます。高レベルのSwitchは、トラップドアであろうとレーザービームであろうと動作します。ただ、ISwitchableを実装した互換性のあるクライアントが必要なだけです。



スイッチは、任意のISwitchableオブジェクトを起動できるようになりました。

SOLIDの他の部分と同様に、依存関係の逆転の原則は、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ように求めます。ルースカップリングでプロジェクトを便利に拡張する。

インターフェースと抽象クラス

このガイドでは、「継承よりも構成」を優先するという理念のもと、多くの例でインターフェイスを使用しています。しかし、抽象クラスでも多くの設計原則やパターンを踏襲することができます。

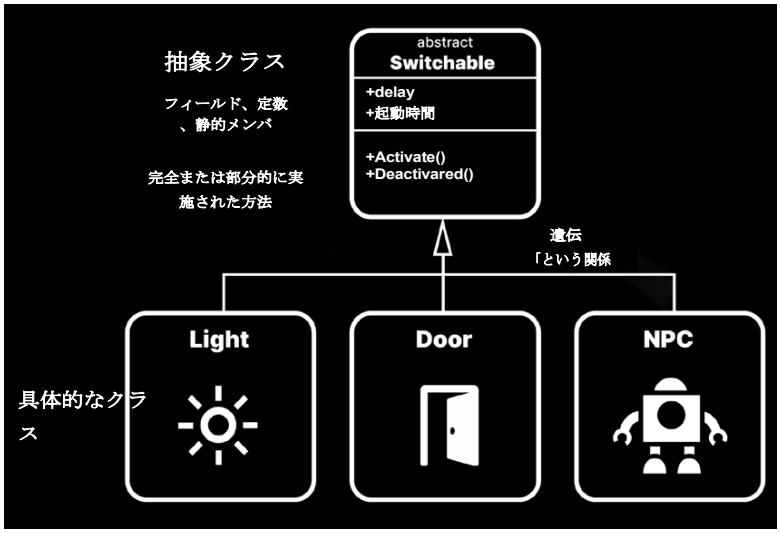
どちらもC#で抽象化を実現するための有効な方法です。どちらを使うかは、状況に応じたニーズ次第です。

抽象クラス

abstractキーワードで基底クラスを定義することで、継承によって共通機能（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なります。

抽象クラスは直接インスタンス化することができません。その代わり、具象クラスを派生させる必要があります。

前述の例では、抽象クラスは、異なるアプローチで、同じ依存関係の逆転を達成することができました。つまり、インターフェースを使うのではなく、`Switchable`という抽象クラスから具象クラス（例えば、Lightや`Door`）を派生させます。

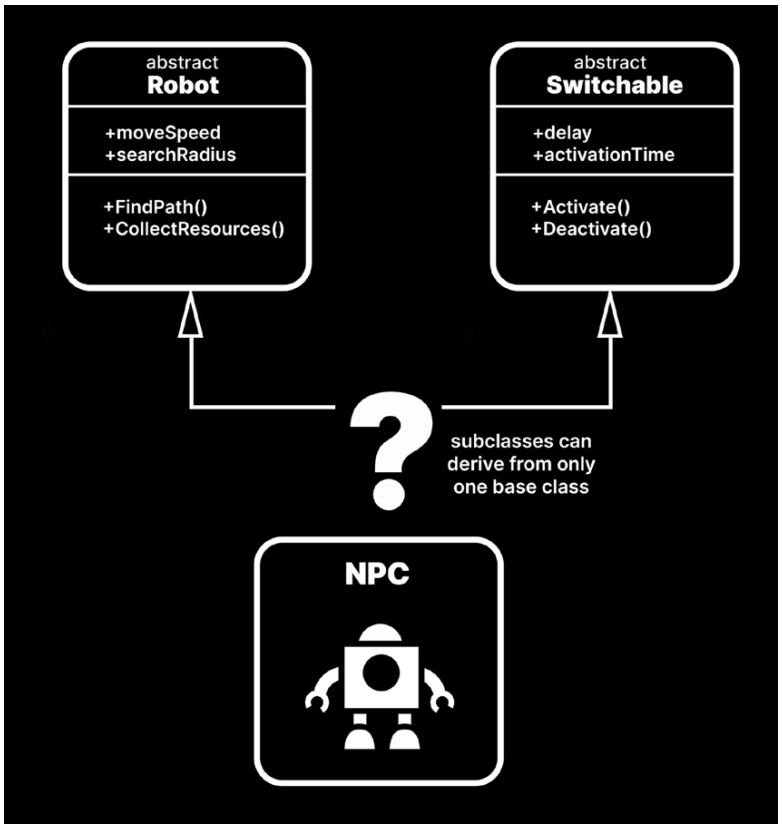


抽象クラスの使用

継承は「is a」の関係を定義する。上の図に示されているのは、オンとオフを切り替えることができる「スイッチャブル」なものばかりです。

抽象クラスの利点は、静的メンバだけでなく、フィールドや定数を持つことができることです。また、protectedやprivateといった、より限定されたアクセス修飾子を適用することができます。インターフェイスとは異なり、抽象クラスでは、具象クラス間でコア機能を共有するためのロジックを実装することができます。

継承は、2つの異なる基底クラスの特徴を持つ派生クラスを作成するまではうまく機能します。C#では、複数の基底クラスから継承することはできません。



ベースクラスの選択

もし、ゲーム内のすべてのRobotに対して別の抽象クラスがあったとしたら、何を派生させるか決めるのは難しいです。Robotの基底クラスを使うか、Switchableの基底クラスを使うか？

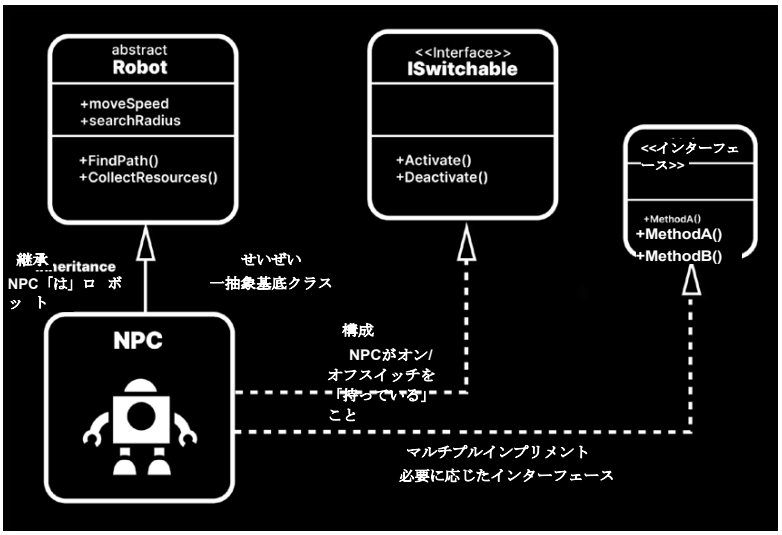
インターフェイス

インターフェース分離の原則に見られるように、インターフェースは、何かが継承のパラダイムにうまく当てはまらない場合に、より柔軟性を与えてくれます。has a "の関係では、より簡単に選択することができます。

しかし、インターフェースにはそのメンバーの宣言しかありません。実際にインターフェイスを実装するクラスは、具体的なロジックを具体化する責任を負います。

このように、常にどちらか一方を選択するわけではありません。コードを共有したい部分には、抽象クラスを使って基本機能を定義します。柔軟性が必要な周辺機能を定義する場合は、インターフェースを使用します。

この例では、Robotの基底クラスからNPCを派生させてそのコア機能を継承し、さらにISwitchableというインターフェースを使ってNPCのON/OFFを切り替える機能を追加しています。



の両方を使用するNPCロボット

抽象クラスとインターフェースの違いとして、以下の点に留意してください。

抽象クラス	インターフェース
メソッドを完全または部分的に実装している	メソッドを宣言しているが、実装できない
変数とフィールドの宣言/使用	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）。
静的メンバを持つ	静的メンバを宣言/使用できない
コンストラクタを使用する	コンストラクタを使用できない
すべてのアクセス修飾子（protected、privateなど）を使用する。	アクセス修飾子が使えない（すべてのメンバが暗黙のうちに公開される）

覚えておいてください。クラスは最大で1つの抽象クラスを継承することができますが、それは
は複数のインタフェースを実装することができます。

SOLIDの理解

SOLIDの原則を知ることは、日々の練習の積み重ねの問題です。SOLIDの原則は、コーディング中に常に心に留めておくべき5つの基本ルールだと考えてください。以下、簡単にまとめてみました。

> **単一の責任**。クラスは1つのことだけを行い、変更する理由は1つだけであることを確認する。
>
> **オープンクローズドであること**。クラスの機能を拡張しても、そのクラスがすでにどのように動作しているかを変更する必要があります。
>
> **リスコフ代入**。サブクラスはベースクラスと置換可能であるべきです。
>
> **インターフェースの分離**。インターフェイスを短く、メソッドを少なくする。クライアントは必要なものだけを導入する。
>
> **依存関係の反転**。抽象的なものに依存する。ある具象クラスから別のクラスに直接依存しないようにする。

SOLIDの原則は、よりクリーンなコードを書くことで、保守や拡張を効率的に行うためのガイドラインである。SOLIDの原則は以下を支配してきました。
これは、拡張性を必要とする大規模なアプリケーションに適しているためです。

場合によっては、SOLIDに準拠することで、前もって追加作業が発生することがあります。機能の一部を抽象化またはインターフェースにリファクタリングする必要があるかもしれません。しかし、多くの場合、長期的な節約という見返りがあります。

この原則は絶対的なものではなく、どの程度厳格に適用するかはご自身でお決めください。絶対的なものではありません。ニュアンスの違いや、ここではカバーしきれない多くの実装方法があります。原則の背後にある考え方は、特定の構文よりも重要であることを忘れないでください。

使い方に迷ったら、「KISSの原則」を参考にしてください。シンプルにすること。そして、その原則を無理にスクリプトに取り込もうとしないこと。必要性に応じて有機的に作用するようにしましょう。

詳しくは、Unite Austinの「Unity SOLID」のプレゼンテーションをぜひご覧ください。

ゲーム開発のための
デザインパターン

SOLIDの原則を理解したら、デザインパターンをもっと深く知りたくなるはずです。

デザインパターンは、日常的なソフトウェアの問題に対して、よく知られた解決策を再利用することができます。しかし、パターンは、既製のライブラリやフレームワークではありません。また、ある結果を得るための具体的な手順の集合であるアルゴリズムでもありません。

デザインパターンは、設計図のようなものだと考えてください。デザインパターンは一般的な計画であり、実際の構築はあなたに委ねられます。同じパターンを使っていても、2つのプログラムはまったく異なるコードになります。

開発者が同じ問題に遭遇したとき、その多くは必然的に似たような解決策を思いつく。そのような解決策が十分に繰り返されるようになると、誰かがパターンを「発見」して、正式に
名前をつける

ザ・ギャング・オブ・フォー

今日のソフトウェアデザインパターンの多くは、エーリッヒ・ガンマ（Erich Gamma）、リチャード・ヘルム（Richard Helm）、ジョン・リサイド（Johnlissides）らによる代表作『*デザインパターン*』から生まれた。Erich Gamma、Richard Helm、Ralph Johnson、John Vlissidesによる「***Elements of Reusable Object-Oriented Software**（再利用可能なオブジェクト指向ソフトウェアの要素）」である。*本書では、日々の様々なアプリケーションで確認された23のパターンを解説している。

原著者はしばしば「Gang of Four」（GoF）と呼ばれ、オリジナルのパターンがGoFパターンと呼ばれているのを耳にすることもあるかと思います。引用されている例はほとんどC++（とSmalltalk）ですが、彼らのアイデアはC#など、どのオブジェクト指向言語にも適用できます。

1994年にG4が*デザインパターンを発表して以来*、開発者はさまざまな分野でさらに数多くのオブジェクト指向パターンを発見してきた。多くのエンジニアリングの専門分野では、確立されたパターンがある。
ゲーム開発も同じです。

デザインパターンの学習

デザインパターンを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、デザインパターンを学ぶことは、より良い開発者になるために役立ちます。デザインパターンは、よくある問題に対する一般的な解決策であるため、このようなラベルが付けられているのです。

ソフトウェア技術者は、通常の開発において、これらのパターンを常に再発見しています。あなたも知らず知らずのうちに、これらのパターンのいくつかを実装しているかもしれません。

それを探す訓練をする。そうすることで、あなたの助けになります。

オブジェクト指向プログラミングを学ぶ。デザインパターンは、難解なStackOverflowの投稿に埋もれている秘密ではありません。開発における日常的なハードルを克服するための一般的な方法です。他の多くの開発者が同じ問題にどのように取り組んでいるかを知ることができます。あなたがパターンを使っていなくても、他の誰かが使っていることを忘れないでください。

他の開発者と話す。パターンは、チームとしてコミュニケーションを図る際の略語として役立ちます。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開発者は、あなたが何を実装しようとしているのか、正確に理解することができます。

新しいフレームワークを探求する。ビルトインのパッケージやアセットストアから何かをインポートするとき、必然的にここで説明したパターンの一つや複数に出くわすことになります。デザインパターンを認識することで、以下のことが可能になります。
新しいフレームワークがどのように機能し、どのような思考プロセスで作られたかを理解することができます。

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適用されるわけではありません。マズローのハンマーで探すのではなく、釘を探すようにしましょう。

他の道具と同様に、デザインパターンの有用性は文脈に依存します。それぞれのデザインパターンは、特定の状況において利益をもたらしますが、同時に欠点も伴います。ソフトウェア開発におけるすべての決断には、妥協がつきものです。

大量のGameObjectをオンザフライで生成していませんか？それはパフォーマンスに影響しますか？コードを再構築することで解決できますか？

これらのデザインパターンを意識し、適切なタイミングでゲーム開発者のバッグからそれらを取り出して、目の前の問題を解決してください。

さらに詳しく

Gang of Fourの*Design Patterns*に加え、*Robert NystromのGame Programming Patterns*も注目の書である。*再利用可能なオブジェクト指向ソフトウェアの要素」に加えて、*もうひとつ注目すべきは、Robert Nystromによる「*Game Programming Patterns」である。*著者は、さまざまなソフトウェア・パターンをわかりやすく説明している。このウェブ版は、gameprogrammingpatterns.comで無料で入手できる。

ユニティ内のパターン

Unityはすでにいくつかの確立されたゲーム開発パターンを実装しており、自分で書く手間を省くことができます。以下がその例です。

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千差万別であるため、すべてのゲームの核となるのはクロックスピードに依存しない無限ループである。コンピュータの速度が異なる場合、固定タイ

ムステップ（1秒あたりのフレーム数を設定）と可変タイムステップ（前のフレームからどれだけ時間が経過したかをエンジンが測定する）を使用する必要があります。

Unityがこれを担ってくれるので、自分で実装する必要はありません。Update、LateUpdate、FixedUpdateといったMonoBehaviourのメソッドを使ってゲームプレイを管理すればよいのです。

更新：ゲームアプリケーションでは、各オブジェクトの挙動を1フレームずつ更新することがよくあります。Unityではこれを手動で再現することができますが、MonoBehaviourクラスはこれを自動的に行ってくれます。適切なUpdate、LateUpdate、FixedUpdateメソッドを使うだけで、GameObjectやコンポーネントをゲームクロックの1刻みに合わせて変更することができます。

プロトタイプ。しばしば、元のオブジェクトに影響を与えることなく、オブジェクトをコピーする必要があります。この作成パターンは、オブジェクトを複製してクローンを作成し、それ自体に似た他のオブジェクトを作成するという問題を解決するものです。この方法では、ゲーム内のあらゆる種類のオブジェクトを生成するために、別のクラスを定義する必要がなくなります。

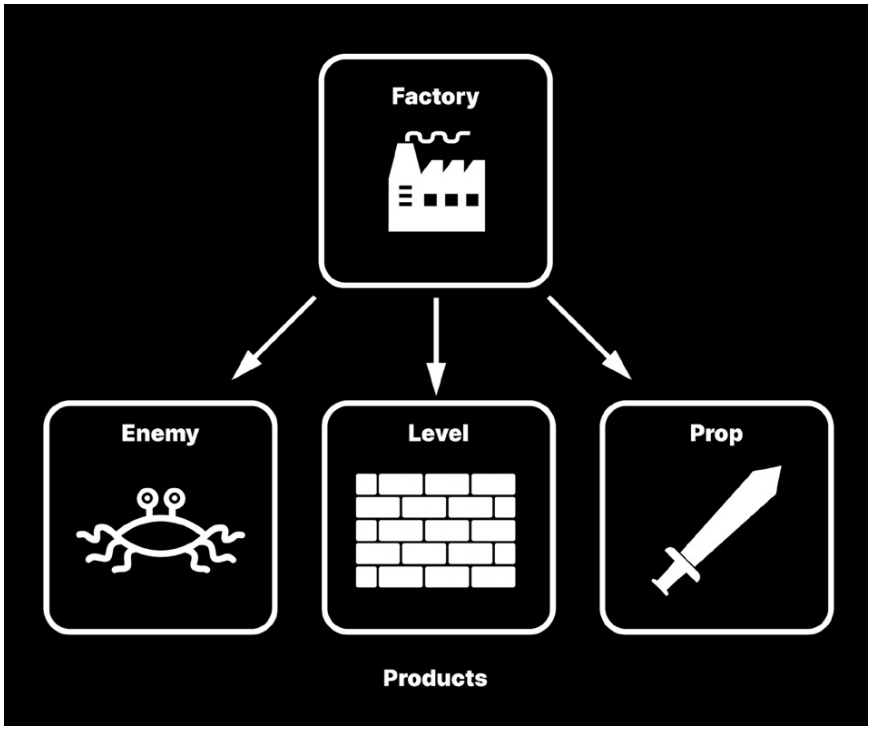
UnityのPrefabシステムは、GameObjectのプロトタイピングの一種を実装しています。これにより、テンプレートオブジェクトを複製し、その構成要素を完全に再現することができます。特定のプロパティをオーバーライドして Prefab Variant を作成したり、Prefab を他の Prefab の中にネストして階層を作成したりすることができます。特別なプレハブ編集モードでは、プレハブを単独で、またはコンテキストに沿って編集することができます。

コンポーネントです。Unityに携わる人であれば、ほとんどの人が知っているパターンです。複数の責任を持つ大きなクラスを作る代わりに、それぞれが1つのことを行う小さなコンポーネントを構築します。

コンポジションでコンポーネントを選ぶと、それらを組み合わせて複雑な挙動をする。物理演算のためのRigidbodyとColliderコンポーネントを追加します。3Dジオメトリには、MeshFilterとMeshRendererを追加します。GameObjectはコンポーネントの集合体であり、その集合体であるからこそ、豊かでユニークなものになるのです。

もちろん、Unityがすべてやってくれるわけではありません。必然的に、ビルトインされていない他のパターンが必要になります。次の章では、そのいくつかを探ってみましょう。

ファクトリーパターン



工場は1つまたは複数の製品を生み出すことができます。

他のオブジェクトを生成する特殊なオブジェクトがあると便利なことがあります。多くのゲームでは、ゲームプレイ中に様々なものが生み出され、実際に必要になるまで、実行時に何が必要なのか分からないことがよくあります。

ファクトリーパターンは、この目的のために、ご想像の通り、ファクトリーと呼ばれる特別なオブジェクトを指定します。あるレベルでは、ファクトリーは「製品」を生成するのに関わる多くの詳細をカプセル化します。直接的な利点は、コードの整理整頓です。

しかし、各製品が共通のインターフェースやベースクラスに従っている場合、さらに一歩進んで、ファクトリー自身から隠れるように、独自の構築ロジックを含むようにすることが可能です。こうして、新しいオブジェクトの作成は、より拡張性の高いものになります。

また、ファクトリーをサブクラス化することで、特定の商品専用のファクトリーを複数作ることも可能です。こうすることで、敵や障害物などをランタイムに生成しやすくなります。

例シンプルな工場

ゲームレベルの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ことを想像してください。GameObjectの作成にはPrefabを使うことができるが、各インスタンスを作成するときに、いくつかのカスタム動作を実行したいかもしれない。

このロジックを維持するために、`if`文や`switch`を使うのではなく、`IProduct`

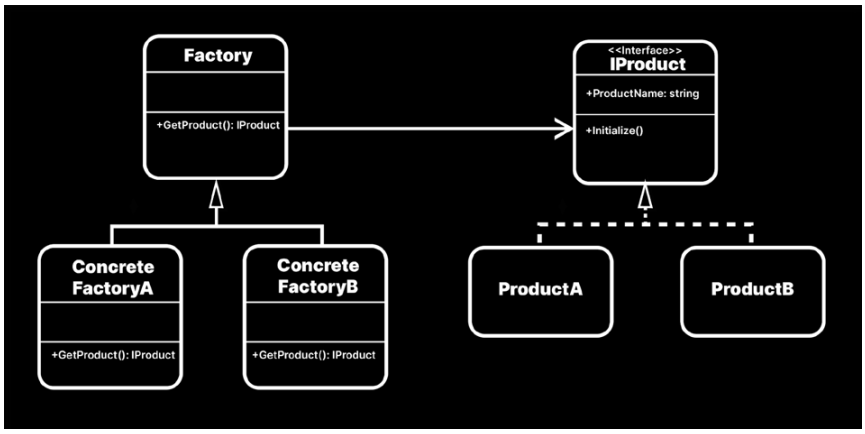
というインターフェースとFactoryという抽象クラスを作成します。

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

    public void Initialize();
}

public abstract class Factory : MonoBehaviour (ファクトリー：モノビジュアル)
{
    public abstract IProduct GetProduct(Vector3 position);

    // すべてのファクトリーで共有されるメソッド
    ...
}
```

プロダクトは、そのメソッドについて特定のテンプレートに従う必要がありますが、それ以外の機能は共有されません。そこで、IProductインタフェースを定義しています。

ファクトリーには共有の共通機能が必要かもしれないので、このサンプルでは抽象クラスを使っています。ただ、サブクラスを使用する際には、SOLIDの原則からリスコフ代入に注意してください。

このような構造になることがあります。



インターフェイスを利用して、製品間で共有するプロパティやロジックを定義します。

IProductインターフェースは、製品間で共通するものを定義します。
この場合、単純にProductNameプロパティがあり、Initializeで製品が実行されるロジックがあります。

そして、IProductインターフェースに従う限り、必要な数の製品（ProductA、ProductBなど）を定義することができる。

基本クラスであるFactoryには、IProductを返すGetProductメソッドがあります。これは抽象クラスなので、Factoryのインスタンスを直接作ることはできません。具体的なサブクラス(ConcreteFactoryAとConcreteFactoryB)をいくつか派生させて、実際に様々な製品を取得することになります。

この例のGetProductはVector3の位置を取るので、より簡単に特定の場所にPrefab GameObjectをインスタンス化することができます。また、各Concrete Factoryのフィールドには、対応するテンプレートPrefabが格納されます。

以下は、サンプルのProductAとConcreteFactoryA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()
    {
        // この製品に固有のロジックがある場合
        gameObject.name = productName;
        particleSystem = GetComponentInChildren<ParticleSystem>();
        particleSystem?.Stop();
        particleSystem?.Play()。
    }
}

public class ConcreteFactoryA : ファクトリー
{

    [SerializeField] private ProductA productPrefab;

    public override IProduct GetProduct(Vector3 position)
    {
        // Prefabのインスタンスを生成し、商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>();

        // 各製品は独自のロジックを含む
        newProduct.Initialize();

        return newProduct;
    }
}
```

ここでは、製品クラスをMonoBehavioursとして実装していますね。IProductは工場でプレハブを活用しています。

各製品がどのように独自のバージョンのInitializeを持つことができるかに注意してください。サンプルの ProductA Prefab には ParticleSystem が含まれており、ConcreteFactoryA がコピーをインスタンス化するときに再生されます。ファクトリー自体にはパーティクルをトリガーする特定のロジックはなく、すべての製品に共通する `Initialize` メソッドを呼び出すだけです。

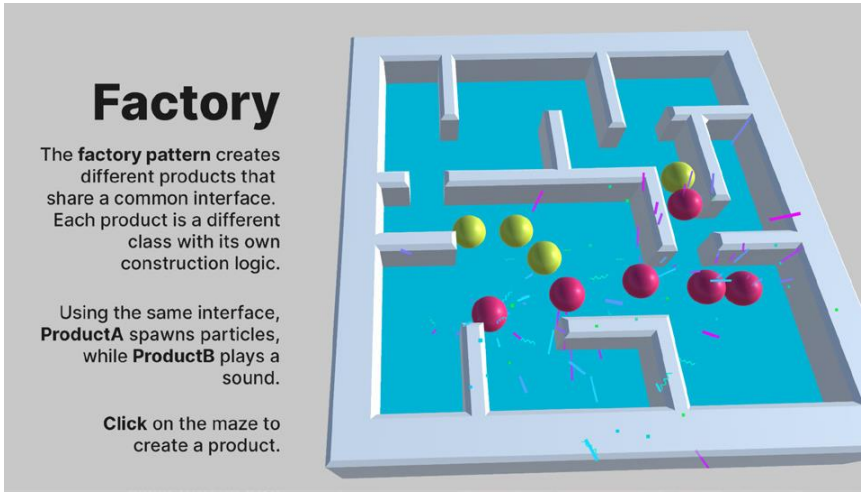
サンプルプロジェクトでは、`ClickToCreate`コンポーネントがファクトリーを切り替えて、異なる動作をするProductAとProductBを生成している様子をご覧いただけます。ProductBは生成時に音を鳴らし、ProductAはパーティクル効果を発生させます。

長所と短所

ファクトリーパターンが最も恩恵を受けるのは、多くの製品をセットアップするときです。アプリケーションで新しい製品タイプを定義しても、既存のものを変更したり、以前のコードを修正する必要はありません。

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することができます。各ファクトリーは、各製品の `Initialize` を呼び出すことだけを知っていればよく、基本的な詳細については知ることができません。

欠点は、このパターンを実装するために、多くのクラスとサブクラスを作成することです。他のパターンと同様、これは少しのオーバーヘッドをもたらしますが、製品の種類が多くない場合は不要かもしれません。



ある製品は音を再生し、別の製品は粒子を再生します。どちらも同じインターフェイスを使用しています。

改善点

ファクトリーの実装は、ここに示したものとは大きく異なる可能性があります。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。

商品の検索には辞書を使用します。商品をキーと値のペアとして辞書に格納したい場合があります。キーとして一意の文字列識別子（例えば、名前や何らかのID）を使用し、値としてタイプを使用します。この
を使えば、製品や工場の検索がより便利になります。

ファクトリー（またはファクトリーマネージャー）を静的なものにします。これは使いやすくなりますが、追加の設定が必要です。静的なクラスはインスペクタに表示されないので、製品のコレクションも静的にする必要があります。

GameObjectやMonoBehaviours以外にも適用してください。PrefabsやUnity固有のコンポーネントに限定しないでください。ファクトリーパターンは、あらゆるC#オブジェクトで動作します。

オブジェクトプールパターンと組み合わせる。ファクトリーは必ずしも新しいオブジェクトをインスタンス化したり作成したりする必要はない。また、階層にある既存のオブジェクトを取得することもできる。一度に多くのオブジェクトをインスタンス化する場合（例：武器の発射物）、オブジェクトプールパターンを使用すると、より最適なメモリ管理が可能になります。

工場は、あらゆるゲームプレイ要素を必要に応じて生み出すことができます。しかし、製品を作ることだけが目的ではないことが多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として使用する場合もあります（たとえば、ゲームレベルの一部のダイアログボックスにUI要素を設定するなど）。

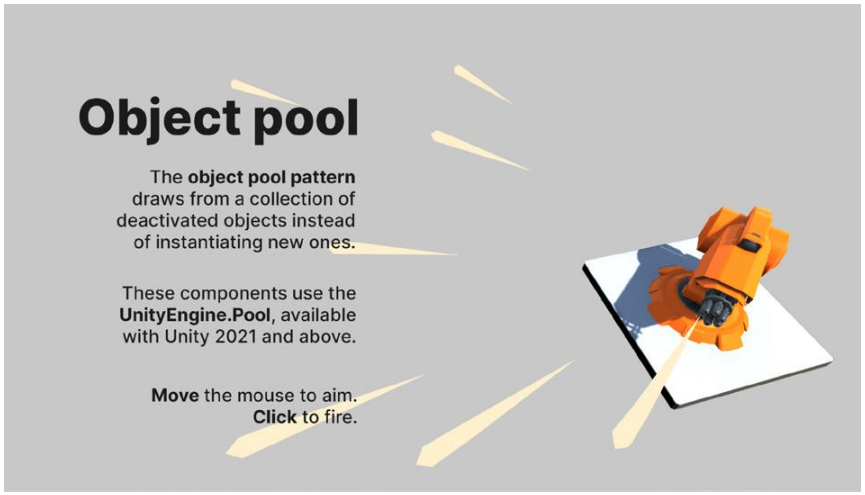
オブジェクトプール

オブジェクトプールとは、大量のGameObjectを生成・破棄する際に、CPUの負担を軽減するための最適化手法です。

オブジェクト・プール・パターンでは、初期化されたオブジェクトのセットを非活性化された "プール" に準備して待機させます。オブジェクトが必要なとき、アプリケーションはそれをインスタンス化しない。代わりに、プールからGameObjectを要求し、それを有効にする。

使い終わったら、オブジェクトを非アクティブにして、破壊する代わりにプールに戻す。

オブジェクトプールは、ガベージコレクションのスパイクに起因するスタッタリングを軽減することができます。GCスパイクは、メモリの割り当てのために大量のオブジェクトを作成または破壊することにしばしば伴います。ユーザーがスタッターに気づかないようなローディング画面などのタイミングを見計らって、オブジェクトプールを事前にインスタンス化しておくとよいでしょう。



オブジェクトプールを使えば、ゲームプレイに支障をきたすことなく弾丸を撃つことができます。

例シンプルなプールシステム

2つのMonoBehaviourが定義されたシンプルなプーリングシステムを考える。

描画対象となるGameObjectのコレクションを保持する`ObjectPool`

Prefabに追加された`PooledObject`コンポーネント。これは、各クローンアイテムがプールへの参照を保持するのに役立ちます。

ObjectPoolでは、プールのサイズを記述するフィールド、格納したいPooledObject Prefab、プール自体を形成するコレクション（この例ではスタック）を設定する。

SetupPool メソッドは、オブジェクト・プールにデータを入力します。PooledObjectの新しいスタックを作成し、objectToPoolのコピーをインスタンス化してinitPoolSizeの要素に満たします。ゲームプレイ中に一度だけ実行されるように、Start で SetupPool を呼び出します。

```
public class ObjectPool : MonoBehaviour (オブジェクトプール)
{
    [SerializeField private int initPoolSize;
    [SerializeField private PooledObject
    objectToPool.SerializeField.SerializeField.SerializeF
    ield.PrivatePoolSize。

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

    private void Start()
    {
        SetupPool()。
    }

    // プールを作成する (遅延が目立たないときに起動する) 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)を実行。
        }
    }
```

また、プールされているアイテムを取得するメソッド（GetPooledObject）やプールに戻すメソッド（ReturnToPool）も必要です。

GetPooledObjectは、プールが空の場合のみ、新しいPooledObjectを作成します。そうでなければ、単に次の利用可能な要素を返すだけである。プールの大きさが十分であれば、ほとんどの場合、既存のGameObjectへの参照しか得られないはずだ。

もろもろGetPooledObjectを呼び出したクライアントは、プールされたオブジェクトを所定の位置に移動/回転させる必要があります。

プールされた各要素は、ObjectPoolへの参照を参照するためだけの小さなPooledObjectユーティリティコンポーネントを持つことになります。

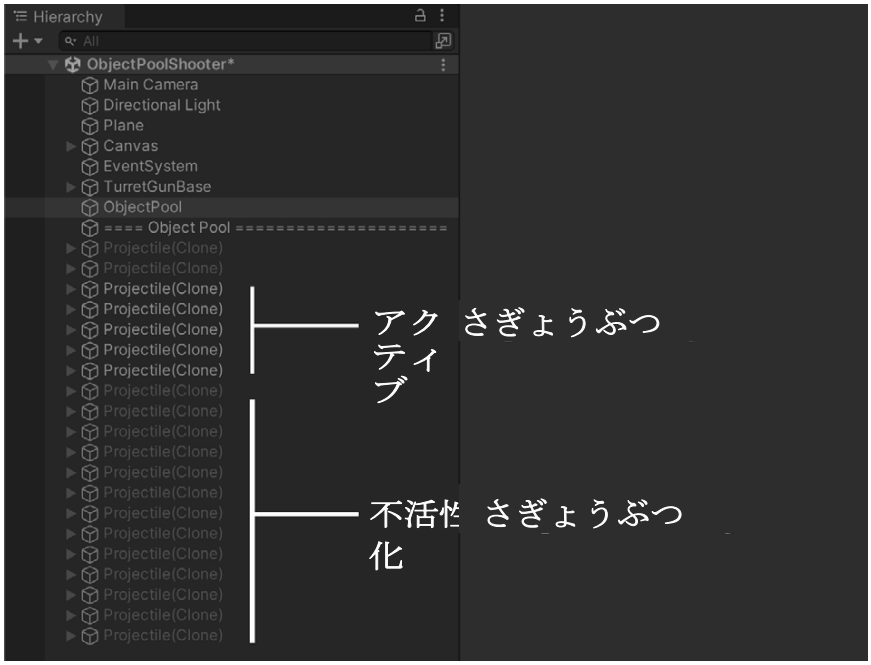
```
// プールから次のオブジェクトを返すpublic
// そうでなければ、GetNextの利用可能
// プールの大きさが十分でない場合、新しい PooledOb- をインスタンス化する。
if (stack.Count == 0)
{
    PooledObject newInstance = Instantiate(objectToPool);
    newInstance.ObjectPool = this;
    return newInstance;
}
```

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

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

Releaseを呼ぶとGameObjectは無効となり、プールキューに戻される。

付属のプロジェクトで、基本的な使用例を示します。ここでは、GameObjectにExampleGunスクリプトをアタッチしています。これは、オブジェクト・プールへの参照を保存するものである。ユーザーが撃つと、武器スクリプトは Object.Instantiate を呼び出す代わりに、その GetPooledObject メソッドを呼び出す。

弾自体は、ExampleProjectileスクリプトとPooledObjectスクリプトがあります。ExampleProjectileにはDeactivateメソッドがあり、発射された弾のGameObjectを数秒後に無効化し、利用可能なプールに戻すことができます。



プールされたオブジェクトの無効化と再利用

こうすることで、画面外では何百発もの弾丸を発射しているように見えても、実際は単に無効化してリサイクルしているだけということが可能になります。ただし、同時にアクティブになるオブジェクトを表示するために、プールのサイズが十分大きいことを確認してください。

プール・サイズを超える必要がある場合、プールは追加のオブジェクトをインスタンス化することができます。しかし、ほとんどの場合、既存の非アクティブなオブジェクトから取得する。

UnityのParticleSystemを使ったことがある人なら、オブジェクトプールを実際に体験したことがあるはずです。ParticleSystemコンポーネントには、パーティクルの最大数を設定する項目があります。これは単純に利用可能なパーティクルを再利用し、エフェクトが最大数を超えないようにするものです。オブジェクトプールも同様に、任意のGameObjectを使用することができます。

改善点

上の例は簡単なものです。実際のプロジェクトでオブジェクトプールを展開する場合は、以下のようなアップグレードを考慮してください。

静的またはシングルトンにする。様々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、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。この
を使用すると、アプリケーションのどこにでもアクセスできますが、インスペクタを使用することはできません。あるいは、オブジェクトプールパターンとシングルトンパターンを組み合わせて、グローバルにアクセスできるようにすると、使い勝手がよくなります。

複数のプールを管理するには、辞書を使用します。プールしたいPrefabが複数ある場合、それらを別々のプールに格納し、どのプールに問い合わせるかわかるようにキーと値のペアを格納します（PrefabのInstanceIDがユニークキーとして機能します）。

未使用のGameObjectをクリエイティブに削除する。オブジェクトプールを有効に活用するためには、未使用のオブジェクトを隠し、プールに戻すことが必要です。プールされたオブジェクトは、あらゆる機会を利用して無効化する（例：画面外、爆発で隠れるなど）。

エラーがないか確認する。すでにプールにあるオブジェクトの解放を避ける。そうでない場合は、実行時にエラーになる可能性があります。

最大サイズ/上限を追加する。プールされた多くのオブジェクトは、メモリを消費します。プールが多くのリソースを使用しないように、一定の制限を超えるオブジェクトを削除する必要があるかもしれません。

オブジェクトプールをどのように使うかは、用途によって異なります。このパターンは、弾幕シューティングのように、銃や武器が複数の弾丸を発射する必要がある場合によく登場します。

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が急増し、小休止が発生する危険性があります。オブジェクトプールは、この問題を軽減し、スムーズなゲームプレイを実現します。

2021年以降のバージョンのUnityを使用している場合、オブジェクトプーリングシステムが組み込まれているため、先ほどの例のようにPooledObjectや`ObjectPool`クラスを独自に作成する必要はありません。

UnityEngine.Pool（ユニティエンジン プール

オブジェクトプールパターンは非常にユビキタスで、Unity2021では独自のUnityEngine.Pool APIをサポートしています。これにより、スタックベースの`ObjectPool`を使用して、オブジェクトプールパターンでオブジェクトを追跡することができます。ニーズによっては、`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(シーン参照)では、カスタムプールコンポーネントは不要になりました。代わりに、ガンスクリプトを更新して、先頭に `using UnityEngine.Pool;` の行を追加します。これにより、ビルトイン `ObjectPool` を使用して発射体プールを作成することができます。

改善点

上の例は簡単なものです。実際のプロジェクトでオブジェクトプールを展開する場合は、以下のようなアップグレードを考慮してください。

静的またはシングルトンにする。様々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、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。この
を使用すると、アプリケーションのどこにでもアクセスできますが、インスペクタを使用することはできません。あるいは、オブジェクトプールパターンとシングルトンパターンを組み合わせて、グローバルにアクセスできるようにすると、使い勝手がよくなります。

複数のプールを管理するには、辞書を使用します。複数のPrefabをプールしたい場合、それらを別々のプールに格納し、どのプールに問い合わせるか分かるようにキーと値のペアを格納します（PrefabのInstanceIDはユニークキーとして機能します）。

未使用のGameObjectをクリエイティブに削除する。オブジェクトプールを有効に活用するためには、未使用のオブジェクトを隠し、プールに戻すことが必要です。プールされたオブジェクトは、あらゆる機会を利用して無効化する（例：画面外、爆発で隠れるなど）。

エラーがないか確認する。すでにプールにあるオブジェクトの解放を避ける。そうでない場合は、実行時にエラーになる可能性があります。

最大サイズ/上限を追加する。プールされた多くのオブジェクトは、メモリを消費します。プールが多くのリソースを使用しないように、一定の制限を超えるオブジェクトを削除する必要があるかもしれません。

オブジェクトプールをどのように使うかは、用途によって異なります。このパターンは、弾幕シューティングのように、銃や武器が複数の弾丸を発射する必要がある場合によく登場します。

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が急増し、小休止が発生する危険性があります。オブジェクトプールは、この問題を軽減し、スムーズなゲームプレイを実現します。

2021年以降のバージョンのUnityを使用している場合、オブジェクトプーリングシステムが組み込まれているため、先ほどの例のようにPooledObjectや`ObjectPool`クラスを独自に作成する必要はありません。

UnityEngine.Pool（ユニティエンジン プール

オブジェクトプールパターンは非常にユビキタスで、Unity2021では独自のUnityEngine.Pool APIをサポートしています。これにより、スタックベースの`ObjectPool`を使用して、オブジェクトプールパターンでオブジェクトを追跡することができます。ニーズによっては、`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(シーン参照)では、カスタムプールコンポーネントは不要になりました。代わりに、ガンスクリプトを更新して、先頭に `using UnityEngine.Pool;` の行を追加します。これにより、ビルトイン `ObjectPool` を使用して発射体プールを作成することができます。

```
UnityEngine.Poolを使用しています。

public class RevisedGun : MonoBehaviour (モノビヘイビア)
{
    ...

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

    // すでにプールに入っているアイテムを返そうとすると、例外が発生します。
    [SerializeField] private bool collectionCheck = true;
    // プールの容量と最大サイズを制御するための追加オプション [SerializeField]
    private int defaultCapacity = 20; [SerializeField] private int
    maxSize = 100.0; [SerializeField]は、プールの容量と最大サイズを制御する
    ためのオプションです。

    private void Awake()
    {
        objectPool = new ObjectPool<RevisedProjectile>(CreateProjec-)
タイルを使用します。
            OnGetFromPool, OnReleaseToPool, OnDestroyPooledObject,
            collectionCheck, defaultCapacity, maxSize)があります。
    }

    // オブジェクトプールを構成するアイテムを作成するときに呼び出される private
    RevisedProjectile CreateProjectile()
    {
        RevisedProjectile projectileInstance = Instantiate(projec-
tilePrefab);
        projectileInstance.ObjectPool = objectPool;
        return projectileInstance.ObjectPool =
        objectPool;
    }

    // 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ます。
    private void OnReleaseToPool(RevisedProjectile pooledObject)
    {
        pooledObject.gameObject.SetActive(false)を実行。
    }

    // 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)を実行。
    }

    // プールされているアイテムの最大数を超えたときに呼び出される（つまり、プールされてい
るオブジェクトを破壊する）。
    private void OnDestroyPooledObject(RevisedProjectile pooledObject)
    {
        Destroy(pooledObject.gameObject)を実行します。
    }

    private void FixedUpdate()
    {
        ...

    }
}
```

このスクリプトの多くは、オリジナルのExampleGunスクリプトで動作します。しかし、ObjectPoolコンストラクタは、いくつかのロジックを設定するための便利な機能を含むようになりました。

最初にプールに入れるアイテムを作成する

プールからアイテムを取り出す

プールにアイテムを返却する

プールされたオブジェクトの破棄（上限を超えた場合など）

次に、コンストラクタ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。

組み込みの ObjectPool には、デフォルトのプールサイズと最大のプールサイズのオプションがあることに注意してください。最大プールサイズを超えるアイテムは、自己破壊するアクションをトリガーし、メモリ使用量を抑制します。

Projectileスクリプトは、ObjectPoolへの参照を保持するように少し修正されました。これによって、オブジェクトをプールに戻すのが少し便利になる。

UnityEngine.Pool APIを使用すると、オブジェクトプールのセットアップが速くなり、パターンを一から作り直す必要がなくなります。これで車輪の再発明が一つ減りました。

```
public class RevisedProjectile : MonoBehaviour.

    private IObjectPool<RevisedProjectile> objectPool;

    // Projectile の Ob-jectPool への参照を与える public プロパティです。
    public IObjectPool<RevisedProjectile> ObjectPool { set => object-
Pool = value; }.

   ...
}
```

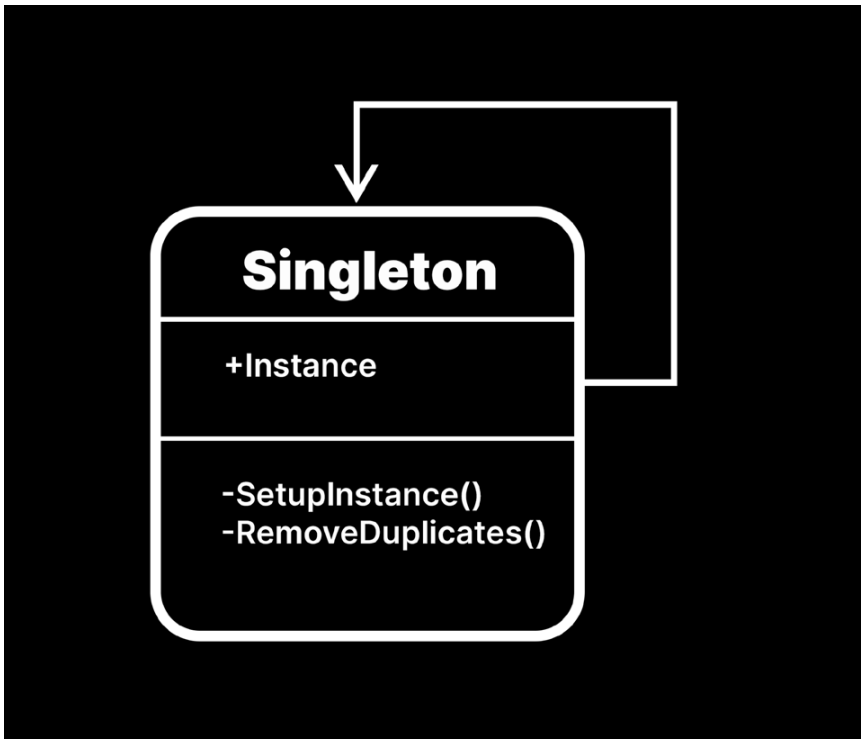
シングルトンパターン

シングルトンは評判が悪いです。Unityの開発を始めたばかりなら、シングルトンは最初に出会う認識できるパターンのひとつでしょう。また、最も悪名高いデザインパターンの1つでもあります。

本家Gang of Fourによると、シングルトン・パターン。

> クラスが自分自身のインスタンスを1つだけ作成できるようにします。
>
> その単一インスタンスに簡単にグローバルアクセスできる

この機能は、シーン全体のアクションを調整するオブジェクトを1つだけ用意する必要がある場合に便利です。例えば、メインゲームのループを制御するために、シーン内に1つのゲームマネージャが必要な場合があります。また、ファイルシステムに一度に書き込むファイルマネージャーは、1 つだけでよいでしょう。このような中央、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の良い候補になります。



SimpleSingletonは、最初のインスタンス以降のインスタンスをすべて破棄します。

*Game Programming Patterns*では、シングルトンは良いことよりも悪いことの方が多いと書かれており、アンチパターンとして挙げられています。この悪い評判は、このパターンの使いやすさが悪用につながるからだ。開発者はシングルトンを不適切な状況で適用し、不必要なグローバルステートや依存関係を導入しがちである。

Unityでシングルトンを構築する方法を検討し、その長所と短所を評価してみましょう。そうすれば、自分のアプリケーションに取り入れる価値があるかどうか判断できるはずです。

例シンプルなシングルトン

最も単純な一重項のひとつは、次のようなものである。

```
UnityEngineを使用しています。

public class SimpleSingleton : MonoBehaviour.
{
    public static SimpleSingleton Instance;

    private void Awake()
    {
        if (Instance == null)
        {
            Instance = this;
        }
        さもなければ
        {
            Destroy(gameObject)を実行します。
        }
    }
}
```

public static Instanceは、シーン内のSingletonのインスタンスを1つ保持することになります。

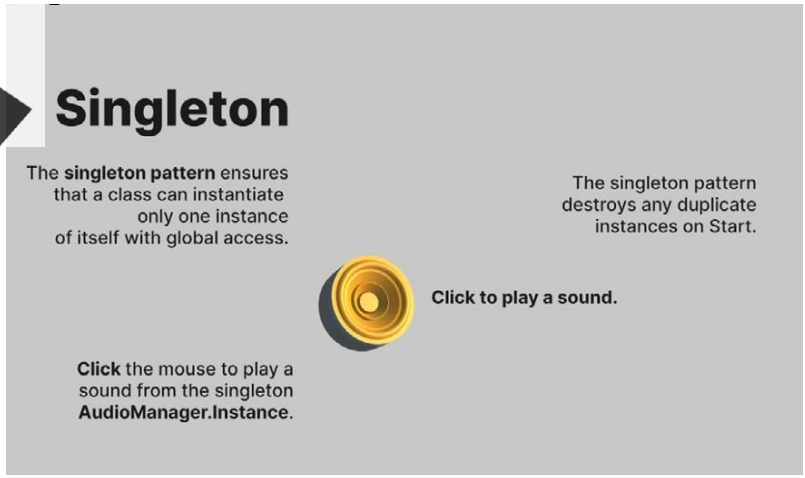
Awakeメソッドで、すでに設定されているかどうかをチェックします。Instanceが現在NULLの場合、Instanceはこの特定のオブジェクトに設定される。これは、シーン内の最初のシングルトンでなければなりません。

Destroy(gameObject)を呼び出すことで、シングルトンコンポーネントがシーン内に1つだけ存在することが保証されます。

実行時にスクリプトを階層内の複数のGameObjectにアタッチした場合、Awakeのロジックは最初のオブジェクトを保持し、残りを破棄します。

シングルトンパターンは1つのインスタンスしか許さない。

removes duplicate instances



`Instance`フィールドは**public**で**static**です。どのコンポーネントも、シーンのどこからでもローン シングルトンにグローバルにアクセスすることができます。

永続化と遅延インスタンス化

`SimpleSingleton`は書かれているとおりに動作します。しかし、2つの問題に悩まされています。

- 新しいシーンをロードすると、GameObjectは破壊されます。
- シングルトンを使用する前に、階層にセットアップする必要があります。

シングルトンはしばしば、どこにでもあるマネージャースクリプトとして機能するため、`DontDestroyOnLoad`を使用して永続的にすることで利益を得ることができます。

さらに、遅延インスタンス化を使えば、最初に必要になったときに自動的にシングルトンを構築することができる。GameObjectを作り、適切なSingletonコンポーネントを追加するロジックが必要なだけだ。

改良されたシングルトンは次のようなものである。

```
public class Singleton : MonoBehaviour
{
    private static Singleton instance;
    public static Singleton Instance
    {
        得る
        {
            if (instance == null)
                SetupInstance();
            }
            インスタンスを返します。
        }
    }

    private void Awake()
    {
        if (instance == null)
        {
            instance = this;
            DontDestroyOnLoad(this.gameObject);
        }
        さもなくば
        {
            Destroy(gameObject)を実行します。
        }
    }

    private static void SetupInstance()
    {
        instance = FindObjectOfType<Singleton>()。

        if (instance == null)
        {
            GameObject gameObj = new GameObject();
            gameObj.name = "Singleton "です。
            instance = gameObj.AddComponent<Singleton>();
            DontDestroyOnLoad(gameObj).AddComponent<Singleton>();
            DontDestroyOnLoad(gameObj);
        }
    }
```

Instanceは、プライベートなMonoBehaviorバッキングフィールドを参照するパブリック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は、Gettingで**Instance**が存在するかどうかをチェックする。存在しない場合は、SetupInstanceメソッドで適切なコンポーネントを持つ**GameObject**を作成する。

DontDestroyOnLoad(gameObject)は、シーンロードによって階層からシングルトンがクリアされるのを防ぎます。シングルトンインスタンスは永続的になり、ゲームのシーンが変わってもアクティブなままです。

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、同じシーン内で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、シングルトンがAudioManagerとGameManagerとしての別のシングルトンは、今すぐには共存できない。関連するコードを複製して、それぞれのクラスにロジックを貼り付ける必要があります。

その代わり、次のようなスクリプトの汎用版を作ってください。

```
public class Singleton<T> : MonoBehaviour where T : Component
{
    private static T instance;
    public static T Instance
    {
        得る
        {
            if (instance == null)
            {
                instance = (T)FindObjectOfType(typeof(T));
                if (instance == null)
                {
                    SetupInstance()。
                }
            }
            インスタンスを返します。
        }
    }
    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)です。
        }
        さもなくば
        {
            Destroy(gameObject)を実行します。
        }
    }
}
```

これにより、どんなクラスでもシングルトンにすることができます。クラスを宣言するときは、一般的なシングルトンを継承すればいいのです。例えば、GameManagerというMonoBehaviourをこのように宣言することで、シングルトンにすることができます。

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

そうすれば、常に `public static GameManager.Instance` を参照することができます。必要なときに必要な分だけ

長所と短所

シングルトンは、このガイドの他のパターンとは異なり、いくつかの点でSOLIDの原則に反しています。多くの開発者は、さまざまな理由からシングルトンを嫌います。

Singletonsはグローバルなアクセスを必要とします。グローバルなインスタンスとして使用するため、多くの依存関係を隠すことができ、バグのトラブルシューティングが非常に困難になります。

シングルトンはテストを困難にする。ユニットテストは互いに独立したものでなければなりません。シングルトンはシーン全体の多くのGameObjectの状態を変更することができるので、テストの邪魔になることがあります。

シングルトンは密結合を促します。このガイドにあるパターンのほとんどは、依存関係を切り離そうとするものです。シングルトンはその逆を行います。密結合はリファクタリングを難しくします。あるコンポーネントを変更すると、それに接続されているすべてのコンポーネントに影響し、汚れたコードになります。

シングルトーンに対する否定的な意見はかなりあります。もしあなたが、今後何年も維持することを想定した企業レベルのゲームを作っているのであれば、シングルトンは避けた方がいいかもしれません。

しかし、多くのゲームは企業レベルのアプリケーションではありません。ビジネスソフトのように継続的に拡張する必要はないのです。

実際、シングルトンには、拡張性を必要としない小さなゲームを作る場合に、魅力的に感じるかもしれない利点があります。

シングルトンは比較的早く習得できます。コアとなるパターン自体はとても簡単です。

シングルトンはユーザーフレンドリーです。他のコンポーネントからシングルトンを使用するには、単にパブリックで静的なインスタンスを参照するだけです。シングルトンインスタンスは、シーン内のどのオブジェクトからも常に利用可能です。

シングルトンはパフォーマンスが高い：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、遅くなりがちなGetComponentやFind操作の結果をキャッシュすることを避けることができるのです。

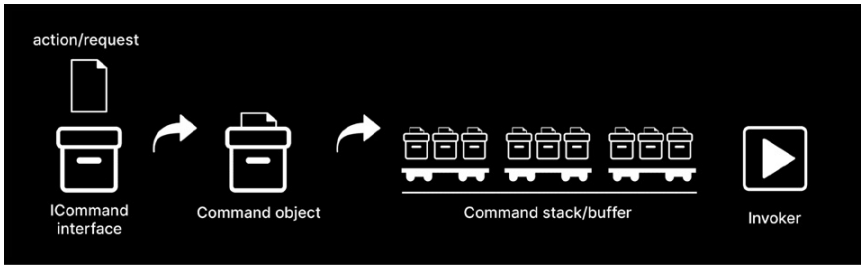
この方法で、シーン内の他の全てのGameObjectから常にアクセス可能なマネージャーオブジェクト（ゲームフローマネージャーやオーディオマネージャーなど）を作ることができます。また、オブジェクトプールを実装した場合、プーリングシステムをシングルトンとして設計することで、プールされたオブジェクトを簡単に取得することができます。

プロジェクトでシングルトンを使用する場合は、最小限にとどめてください。無差別に使用しないでください。シングルトンは、グローバルなアクセスから利益を得ることができる一握りのスクリプトのために予約してください。

コマンドパターン

Gang of Fourパターンの1つであるcommandは、特定の一連のアクションを追跡したいときに便利です。undo/redo機能を使用したり、入力履歴をリストに保存するようなゲームをプレイしたことがあれば、おそらくcommandパターンが機能しているのを見たことがあるはずです。ストラテジーゲームで、実際にプレイする前に何度もターンを計画できるようなものを想像してください。それがコマンドパターンです。

コマンドパターンは、メソッドを直接呼び出す代わりに、1つまたは複数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。



コマンドパターンでアクションを格納する

これらのコマンドオブジェクトをキューやスタックのようなコレクションに格納することで、コマンドの実行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。そして、一連のアクションを後で再生するために遅らせたり、元に戻したりすることができる可能性があります。

コマンドパターンを実装するには、アクションを格納する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は、どのようなロジックを実行するか、どのように元に戻すかを保持します。

コマンドオブジェクトとコマンドインボーカ

これを実装する方法はいくつもありますが、ここではインターフェースを使ったバージョンを紹介します。

この場合、すべてのゲームプレイアクションはICommandインターフェイスを適用します（抽象クラスで実装することも可能です）。

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

各コマンドオブジェクトは、それ自身の `Execute` と `Undo` メソッドに責任を持ちます。したがって、ゲームにコマンドを追加しても、既存のコマンドに影響を与えることはありません。

コマンドの実行と取り消しのために別のクラスが必要になります。`CommandInvoker` クラスを作成します。`ExecuteCommand` と `UndoCommand` メソッドに加えて、一連のコマンドオブジェクトを保持するための undo スタックを持っています。

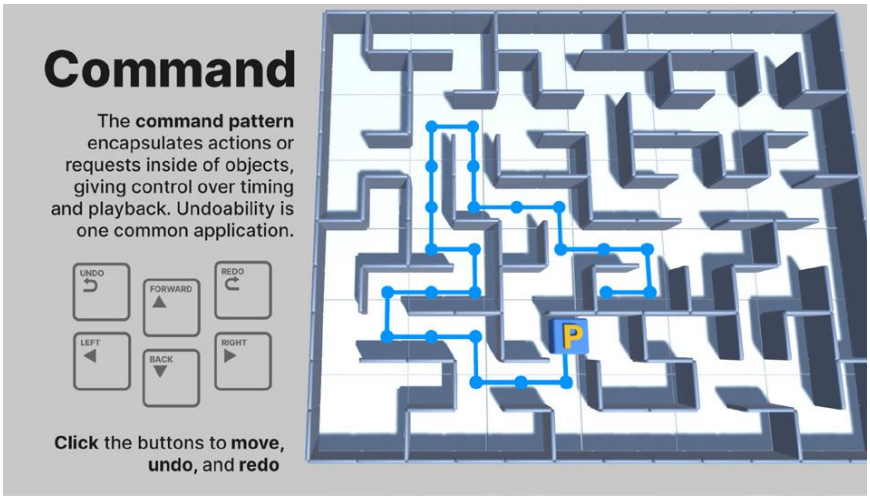
```
public class CommandInvoker
{
    private static Stack<ICommand> undoStack = new Stack<ICommand>();

    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()です。
        }
    }
}
```

例元に戻せない動き

アプリケーションの中でプレイヤーを迷路の中で動かしたい場合を考えてみましょう。プレイヤーの位置を移動させるPlayerMoverを作成することができます。

```
public class PlayerMover : MonoBehaviour (プレイヤー・コンパス)
{
    [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 を渡して、プレイヤーを 4 つのコンパスの方向に沿って誘導します。また、レイキャストを使って適切なLayerMaskにある壁を検出することができます。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のは、パターンそのものとは別の話です。



サンプルでの選手の動き

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捕捉する。Moveを直接呼び出すのではなく、ICommandインタフェースを実装したMoveCommandという新しいクラスを作成する。

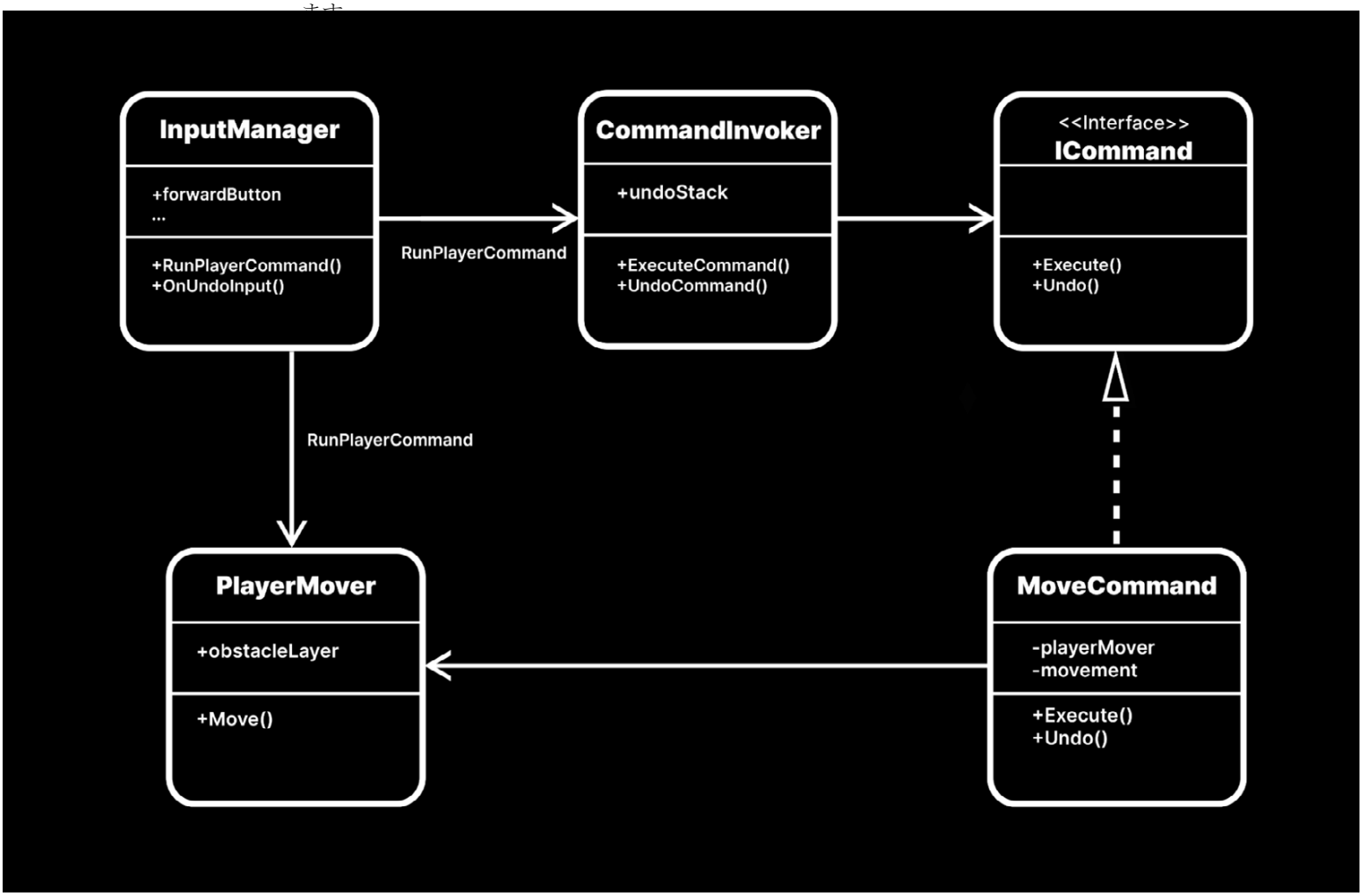
ICommandはExecuteメソッドに何を実現しようとしているかを格納する必要があります。達成したいロジックはすべてここに入るので、移動ベクトルを指定してMoveを起動する。

```
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(移動)。
    }

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

ICommand はシーンを以前の状態に戻すための Undo メソッドも必要です。この場合、Undo ロジックは移動ベクトルを減算し、実質的にプレイヤーを反対方向に押すことになります。

MoveCommandには、実行に必要なパラメータが格納されています。これらをコンストラクターで設定します。この場合、適切なPlayerMoverコンポーネントと移動ベクトルを保存します。

コマンドオブジェクトを作成し、必要なパラメータを保存したら、CommandInvoker の静的メソッド ExecuteCommand と UndoCommand 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、コマンドオブジェクトが undo スタックに追跡されます。



CommandInvoker、ICommand、MoveCommand。

InputManagerはPlayerMoverのMoveメソッドを直接呼び出さない。代わりにRunMoveCommandというメソッドを追加して、新しいMoveCommandを作り、それをCommandInvokerに送ります。

次に、UIボタンの様々なonClickイベントを設定し、以下のように呼び出します。4つの移動ベクトルを持つRunPlayerCommand。

```
private void RunPlayerCommand(PlayerMover playerMover, Vector3 move-
ment)
{
    if (playerMoverが有効な場合は)
    {
        を返します。
    }

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

InputManagerの実装の詳細は、サンプルプロジェクトをご覧ください。また、キーボードやゲームパッドを使って独自の入力を設定することもできます。これでプレイヤーは迷路の中を移動できるようになりました。Undoボタンをクリックすると、最初のマスに戻ることができます。

長所と短所

リプレイアビリティやアンドゥアビリティの実装は、コマンドオブジェクトのコレクションを生成するの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ができます。

例えば、格闘ゲームでは、特定のボタンを何度もクリックすることでコンボ技や攻撃技が発動します。プレイヤーのアクションをコマンドパターンで保存しておけば、コンボを組むのが非常に簡単になります。

一方、コマンドパターンは、他のデザインパターンと同様に、より多くの構造を導入しています。これらの余分なクラスやインターフェイスが、アプリケーションにコマンドオブジェクトを配置するのに十分な利益をもたらすかどうかを判断する必要があります。

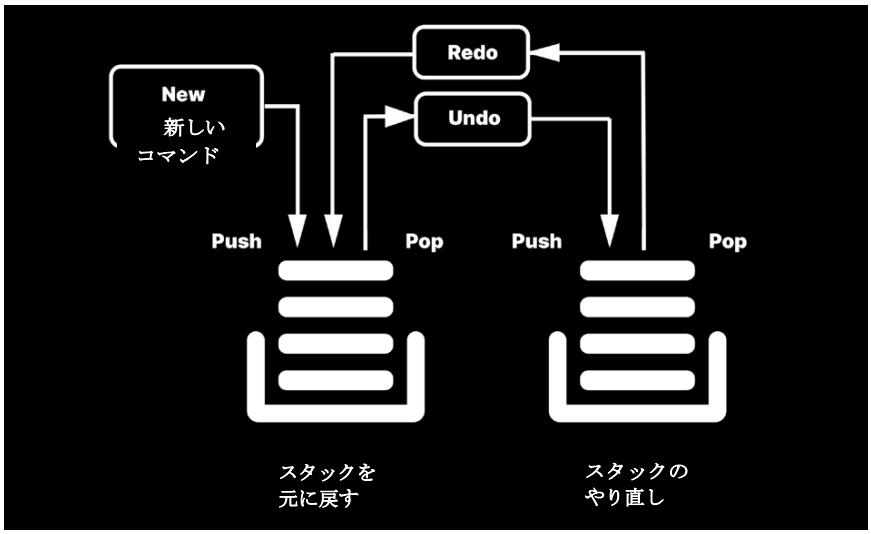
改善点

基本を覚えると、コマンドのタイミングに影響を与えたり、文脈に応じて連続再生や逆再生ができるようになります。

コマンドパターンを取り入れる際には、以下の点を考慮してください。

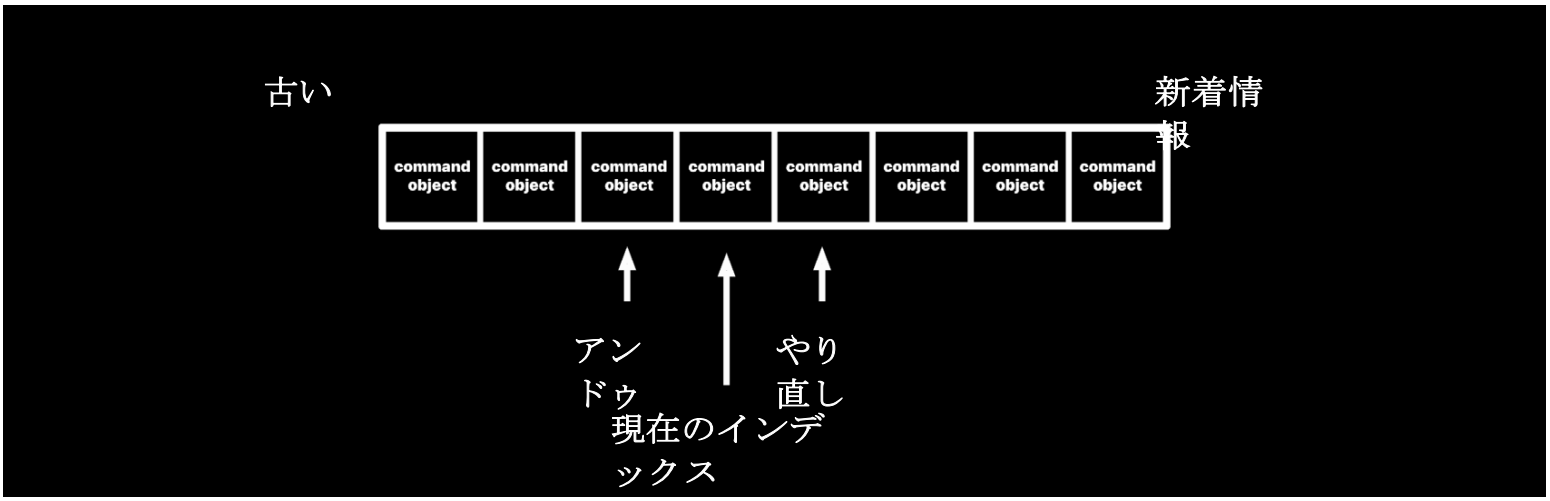
さらにコマンドを作成するサンプルプロジェクトには、`MoveCommand` という 1 種類のコマンドオブジェクトしか含まれていません。`ICommand`を実装したコマンドオブジェクトをいくつでも作成し、`CommandInvoker`を使用して追跡することができます。

やり直しの機能を追加するには、別のスタックを追加する必要があります。コマンドオブジェクトを元に戻すとき、それをやり直しの操作を追跡する別のスタックにプッシュします。こうすることで、取り消し履歴を素早く循環させたり、それらの操作をやり直したりすることができます。やり直しスタックは、ユーザーが
は、まったく新しいムーブメントを呼び出します（付属のサンプルプロジェクトに実装があります）。



UndoとRedoのスタック

コマンドオブジェクトのバッファには、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、キューの方が便利です。リストを使用する場合、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。アクティブなインデックスより前のコマンドは元に戻せます。インデックスより後のコマンドはやり直せます。



リストなどのコレクションは、コマンドバッファとして機能します。

スタックの大きさを制限する。UndoとRedoの操作は、すぐに制御不能に吹き飛ばされる可能性があります。スタックは直近のコマンド数に制限してください。

必要なパラメータは、コンストラクタに渡します。これは、MoveCommandの例に見られるように、ロジックをカプセル化するのに役立ちます。

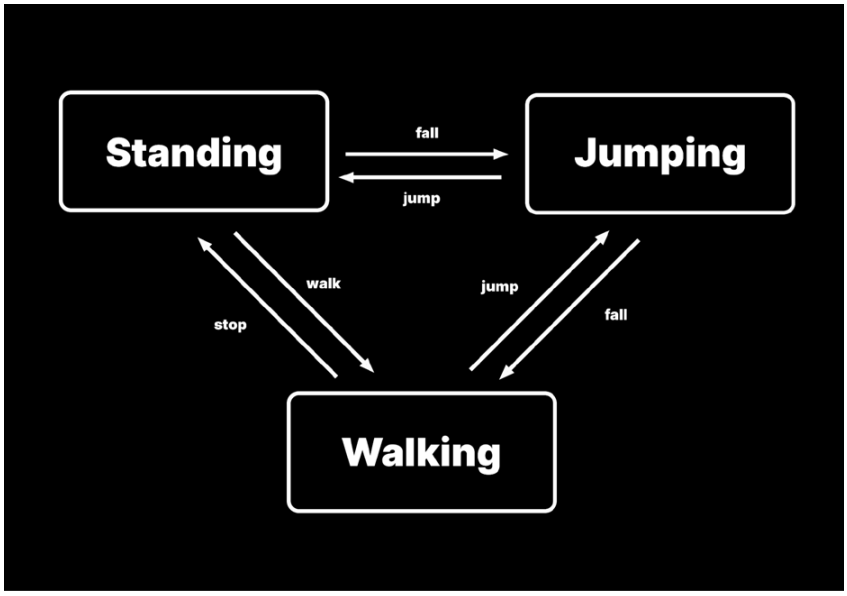
`CommandInvoker`は他の外部オブジェクトと同様、コマンドオブジェクトの内部構造を見ることはなく、ExecuteやUndoを呼び出すだけです。コンストラクタを呼び出すときに、コマンドオブジェクトに動作に必要なあらゆるデータを与えてください。

ステートパターン

プレイアブルキャラクターを作ることを想像してください。ある瞬間、キャラクターは地面に立っているかもしれません。コントローラーを動かせば、走ったり歩いたりするように見えます。ジャンプボタンを押すと、キャラクターは空中に飛び出します。数フレーム後に着地して、再び立っている状態に戻ります。

ステートとステートマシン

ゲームはインタラクティブなものであり、実行時に変化する多くのシステムを追跡することを余儀なくされます。キャラクターのさまざまな状態を表す図を描くと、次のようなものが出てくるかもしれません。



簡単な状態図

フローチャートに似ているが、いくつかの違いがある。

> この図は、いくつかの状態（Idling/Standing、Walking、Running、Jumpingなど）で構成され、ある時点で現在の1つの状態のみがアクティブとなる。
>
> 各状態は、実行時の条件に基づいて、他の1つの状態への遷移を引き起こすことができる。
>
> 遷移が発生すると、出力状態が新しいアクティブ状態になります。

この図は、有限状態マシン（FSM）と呼ばれるものを記述したものです。ゲーム開発では、典型的な使用例として、ゲームのアクターやプロップの内部状態を追跡することが挙げられます。

基本的なFSMをコードで記述する場合、単純なアプローチとして、`enum`と`switch`文があります。

```
public enum PlayerControllerState
{
    アイド
    ル、ウ
    ォーク
    、ジャ
    ンプ
}

public class UnrefactoredPlayerController : MonoBehaviour.
{
    private PlayerControllerState state;

    private void Update()
    {
        GetInput()。

        切り換える
        {
            case
                PlayerControllerState.Idle:Idl
                e();
                が壊れる。
            case
                PlayerControllerState.Walk:Wal
                k();
                が壊れる。
            case
                PlayerControllerState.Jump:Jum
                p();
                が壊れる。
        }
    }

    private void GetInput()
    {
        // ウォークとジャンプの制御を処理する
    }

    private void Walk()
    {
        // ウォークロジック
    }

    private void Idle()
    {
        // アイドルロジック
    }

    private void Jump()
    {
        // ジャンプロジック
    }
}
```

これでもいいのですが、PlayerControllerスクリプトはすぐにごちゃごちゃになってしまいます。状態を増やして複雑にすると、その都度PlayerControllerスクリプトの内部を見直す必要がある。

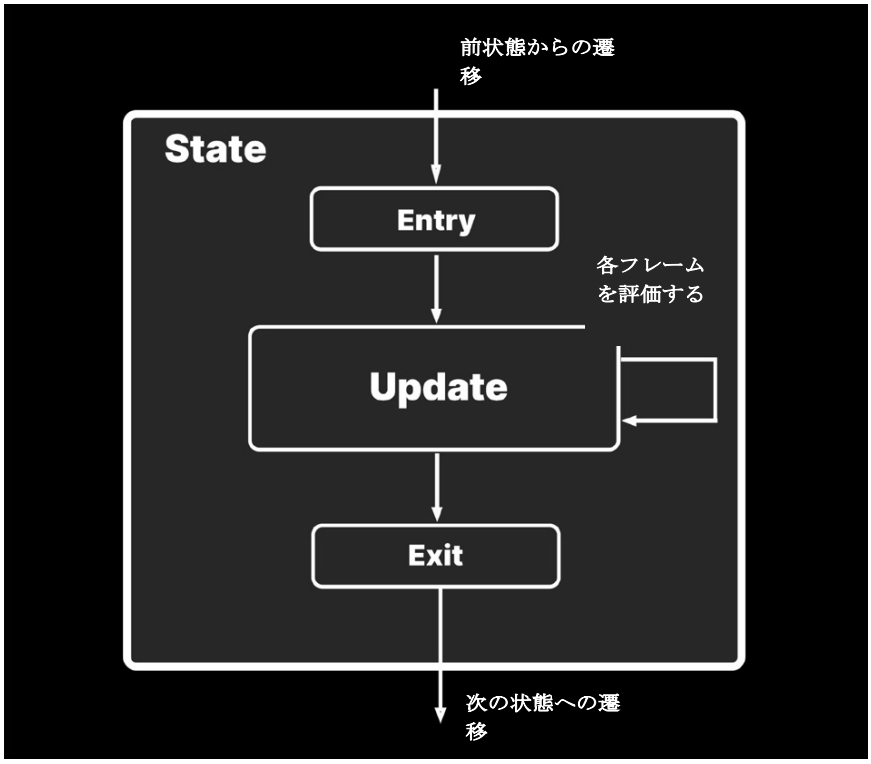
例シンプルな状態パターン

幸いなことに、ステートパターンはロジックを再編成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2つの問題を解決してくれる。

> オブジェクトは、その内部状態が変化したときに振る舞いを変える必要があります。
>
> 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影響を与えることはない。

上記の例の`UnrefactoredPlayerController`クラスは状態の変化を追跡することができますが、2番目の問題を満たしていません。新しい状態を追加する際に、既存の状態への影響を最小限に抑えたいものです。その代わりに、状態をオブジェクトとしてカプセル化することができます。

各状態をこのように構成することを想像してください。



Entry、Exit、Updateを持つカプセル化された状態。

ここでは、状態を入力し、条件によって制御フローが終了するまで、各フレームをループさせる。このパターンを実装するには、IStateというインターフェースを作成します。

```
public class 内の各具体的な状態は、IStateインターフェースを実装します。
{
    public void Enter()
    {
        // 最初に状態になったときに実行されるコード
    }

    public void Update()
    {
        // 状態変化のトリガーとなる条件が満たされるまで、各フレームで実行されるコード
    }

    public void Exit()
    {
        // ステート終了時に実行されるコード
    }
}
```

エントリー。このロジックは、最初にステートへ入るときに実行されます。

Update：このロジックは、フレームごとに実行されます（ExecuteやTickと呼ばれることもあります）。MonoBehaviourのようにUpdateメソッドをさらに細分化し、物理はFixedUpdate、物理はLateUpdateなどとすることができます。

アップデートの他の機能も、状態変化のトリガーとなる条件が満たされるまで、各フレームで実行されます。

Exit（終了）。ここでは、状態を終了し、新しい状態に移行する前にコードが実行されます。

IStateを実装した各状態のクラスを作成する必要があります。サンプルプロジェクトでは、WalkState、IdleState、JumpStateにそれぞれ別のクラスが設定されています。

そして、もう一つのクラスであるStateMachineは、制御フローがどのように状態に入り、出ていくかを管理する。3つの状態の例では、StateMachineは次のようになる。

このパターンに従い、StateMachineは管理下にある各状態（ここでは、walkState、jumpState、idleState）のpublicオブジェクトを参照する。StateMachineはMonoBehaviourを継承していないので、コンストラクタを使って各インスタンスをセットアップしてください。

```
[シリアライズ可能]です。
public class StateMachine
{
    public IState CurrentState { get; private set; }
```

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

```
    }

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

    public void Update()
    {
        if (CurrentState != null)
        {
            CurrentState.Update()を実行します。
        }
    }
}
```

コンストラクターには必要なパラメータを渡すことができます。サンプルプロジェクトでは、各ステートでPlayerControllerを参照しています。それを使ってフレームごとに状態を更新していく（以下の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属性によって、StateMachine（とそのpublicフィールド）をInspectorに表示することができる。そして他のMonoBehaviour（例えば、PlayerControllerやEnemyController）がStateMachineをフィールドとして使うことができるようになる。

`CurrentState` プロパティは読み取り専用である。StateMachine自身はこのフィールドを明示的に設定することはない。PlayerControllerのような外部オブジェクトは`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る。

各Stateオブジェクトは、現在アクティブな状態を変更するために`TransitionTo`メソッドを呼び出すための条件を独自に決定する。StateMachineインスタンスのセットアップ時に、各ステートに必要な依存関係（State Machine自体を含む）を渡すことができます。

サンプル・プロジェクトでは、PlayerControllerはすでにStateMachineへの参照を含んでいるので、`player`パラメータを1つだけ渡します。

各ステートオブジェクトは独自の内部ロジックを管理し、GameObjectやコンポーネントを記述するために必要な数だけステートを作成することができます。各ステートは、`IState`を実装した独自のクラスを持ちます。SOLIDの原則に則り、ステートを追加しても、以前に作成したステートへの影響は最小限である。

`IdleState`の例を示します。

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

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

    public void Enter()
    {
        // 最初に状態になったときに実行されるコード
    }

    public void Update()
    {
        // ここでは、以下の条件があるかどうかを検出するロジックを追加しています。
        // 別の状態に遷移する
        ...
    }

    public void Exit()
    {
        // ステート終了時に実行されるコード
    }
}
```

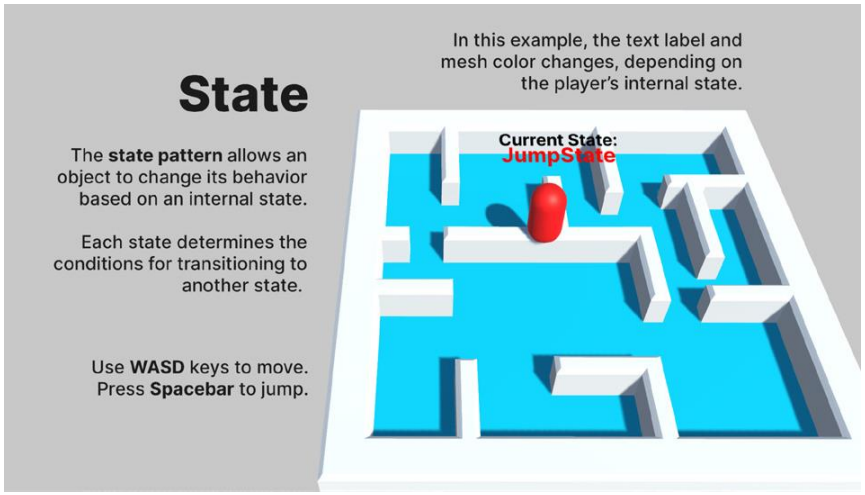
ここでもコンストラクタを使用して PlayerController オブジェクトを渡します。この例では、このプレイヤーにStateMachineへの参照とUpdateロジックに必要なものが含まれています。`idleState`はキャラクターコントローラーのベロシティやジャンプの状態を監視し、StateMachineの`TransitionTo`メソッドを適切に呼び出します。

WalkStateと`JumpState`の実装もサンプルプロジェクトで確認してください。動作を切り替える大きなクラスを1つ用意するのではなく、各ステートに独自の更新ロジックを持たせています。こうすることで、ステートは互いに独立して機能することができます。

長所と短所

ステートパターンは、オブジェクトの内部ロジックを設定する際に、SOLIDの原則を遵守するために役立ちます。各ステートは比較的小さく、別のステートに遷移するための条件を追跡するだけです。オープンクローズの原則に則り、既存の状態に影響を与えることなく状態を追加することができ、煩雑な`switch`文や`if`文を回避することができます。

一方、追跡する状態が数個しかない場合、余分な構造は過剰になる可能性がある。このパターンが意味を持つのは、状態がある程度複雑になることが予想される場合だけかもしれない。



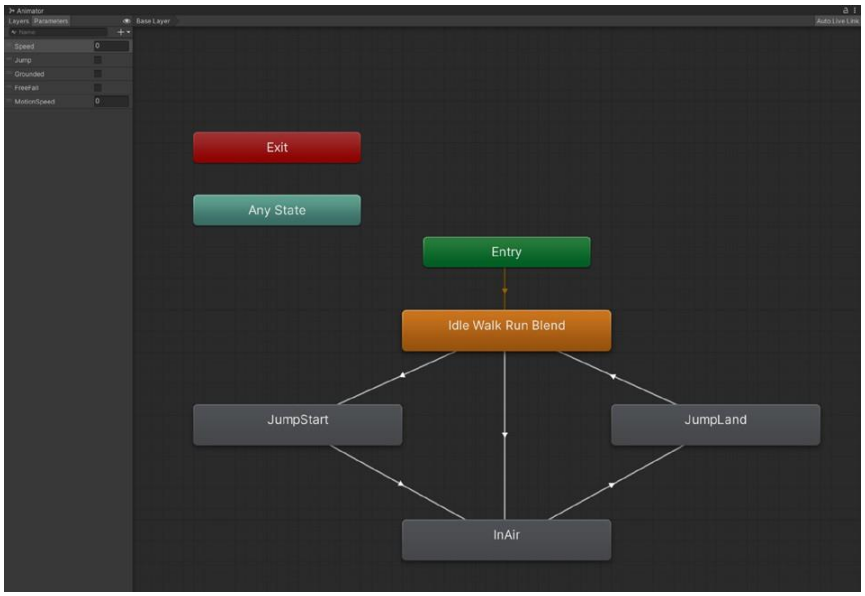
状態パターンサンプル

改善点

サンプルプロジェクトでは、カプセルの色が変化し、プレイヤーの内部状態に応じてUIが更新されます。実際の例では、状態の変化に伴って、もっと複雑なエフェクトをかけることができます。

ステートパターンとアニメーションを組み合わせるステートパターンの一般的な応用例として、アニメーションがある。プレイヤーや敵のキャラクターは、マクロレベルではプリミティブ（カプセル）として表現されることが多い。そして、内部の状態変化に反応するジオメトリをアニメーションさせることで、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る。

UnityのAnimatorウィンドウを使ったことがある人なら、そのワークフローがステートパターンと相性が良いことに気がつくでしょう。各アニメーションクリップは1つの状態を占有し、一度にアクティブになる状態は1つだけです。



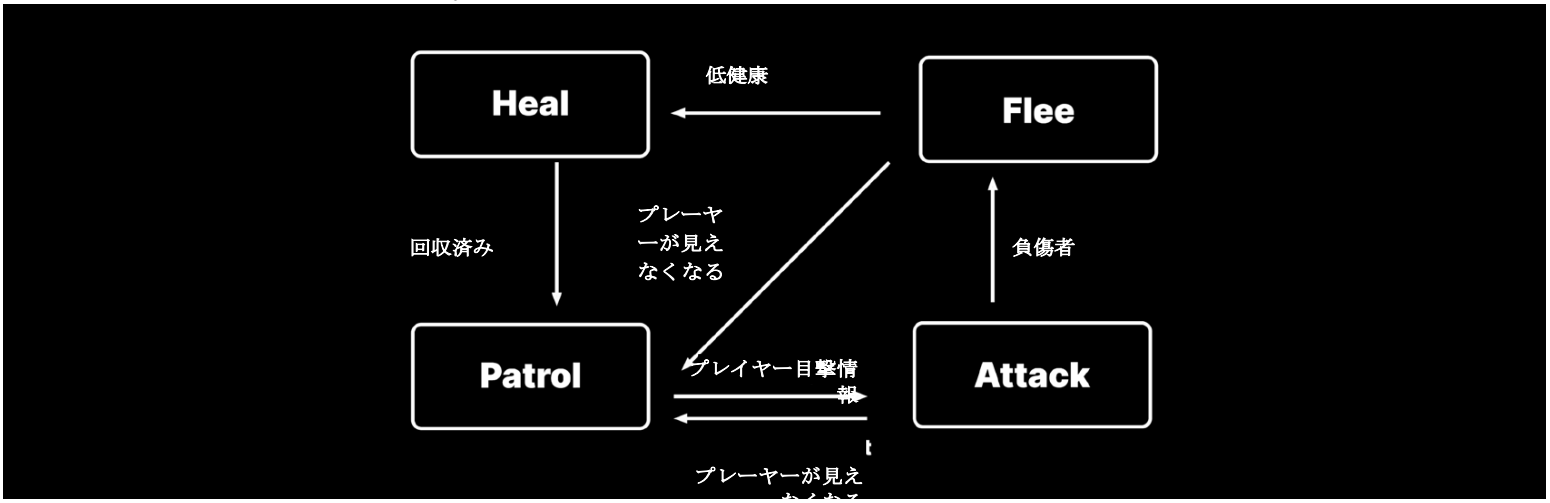
Animatorのステートグラフの例です。StateMachineと構造を比較する。

イベントを追加する。状態の変化を外部のオブジェクトに伝えるために、イベントを追加したい場合があります（observerパターンを参照）。ある状態になったときや、ある状態から抜け出したときにイベントを発生させることで、関連するリスナーに通知し、実行時に応答させることができる。

階層を追加するステートパターンを使ってより複雑なエンティティを記述するようになると、階層的なステートマシンを実装したくなることがあります。たとえば、プレイヤーやゲームアクターが地面にいる場合、WalkingStateでもRunningStateでも、ダッキングやジャンプをすることができます。

SuperStateを実装すれば、共通の振る舞いをまとめておくことができる。そして、継承を利用して、サブステートで特定のものをオーバーライドすることができる。例えば、最初にGroundedStateを宣言する。そして、そこからRunningStateやWalkingStateを継承することができる。

簡単なAIの実装：有限状態機械は、基本的な敵のAIを生成するのにも便利です。NPCの脳を作るためのFSMアプローチは、次のようになります。



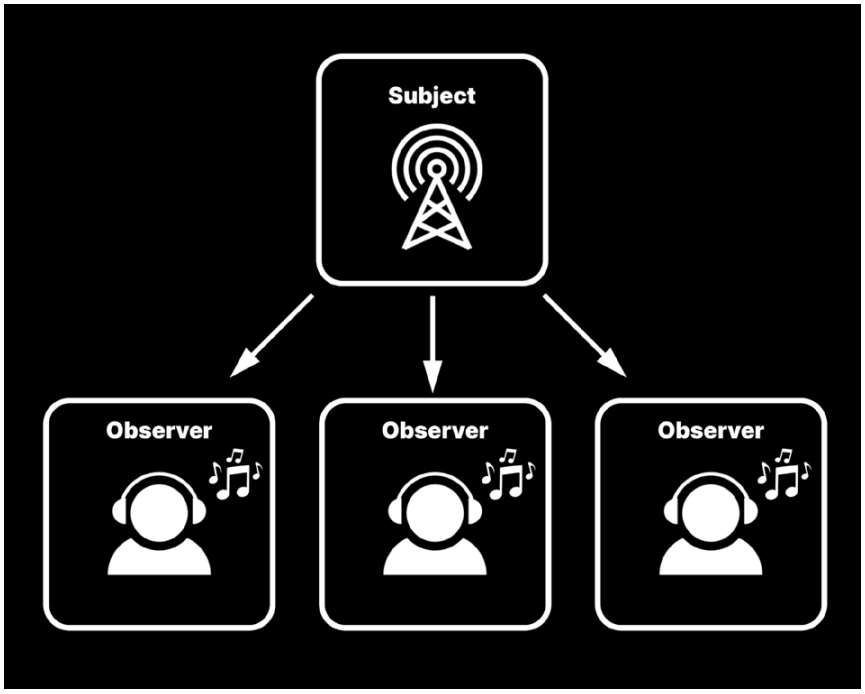
状態パターンに基づくシンプルなAI

ここでまた、全く別の文脈で状態パターンを見てみよう。すべてのステートは、攻撃、逃走、パトロールなどの行動を表します。一度にアクティブになるのは1つの状態のみで、各状態が次の状態への移行を決定する。

オブザーバーパターン

ランタイムでは、ゲーム内で様々なことが起こり得ます。敵を倒すとどうなるのか？パワーアップを集めたり、オブジェクトを完成させたときはどうでしょうか？ある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、他のオブジェクトに通知できるようなメカニズムが必要になることがよくあり、それによって不必要な依存関係が生まれます。

この種の問題に対する一般的な解決策として、オブザーバー・パターンがあります。これは、オブジェクトが通信を行いながらも、「一対多」の依存関係を用いて疎結合の状態を維持することを可能にします。あるオブジェクトが状態を変更すると、依存関係にあ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、多くの異なるリスナーに向けて放送する電波塔に似ています。



観測者パターンは電波塔のように機能する。被写体は観測者に向けて放送を行う。

放送している対象を主体（subject）と呼びます。聞いている他の物体はオブザーバーと呼ばれる。

このパターンでは、主体が緩やかに切り離され、観測者のことを本当に知らないし、信号を受け取った後の彼らの行動を気にもしない。観測者は被観測者に依存しているが、観測者自身は互いのことを知らない。

イベント情報

observerパターンは、C#言語に組み込まれているほど普及しています。Subject-Observerクラスを独自に設計することも可能ですが、通常は不要です。
車輪の再発明についての指摘を覚えていますか？C#はすでにイベントを使ってこのパターンを実装しています。

イベントとは、何かが起こったことを示す単なる通知です。これにはいくつかのパーツがあります。

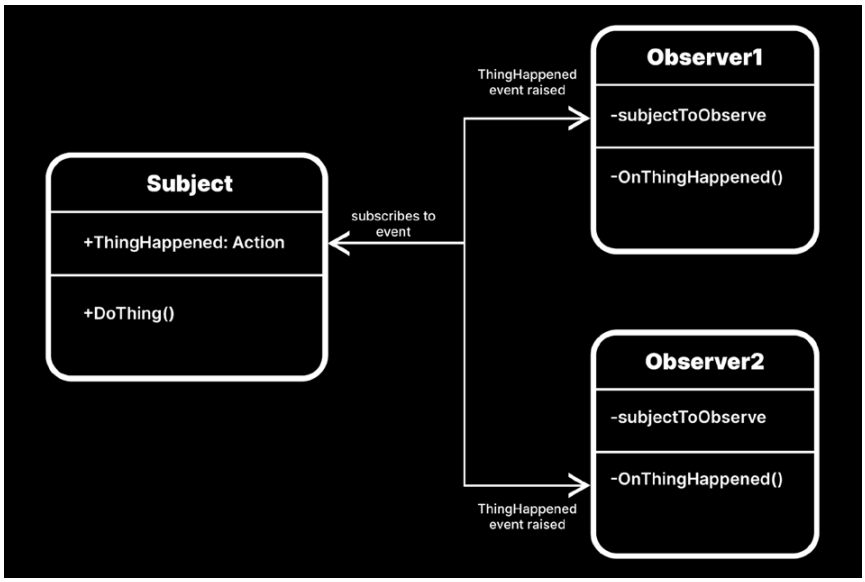
パブリッシャー(主体)はデリゲートに基づいてイベントを作成し、 特定の関数シグネチャを確立します。イベントとは、サブジェクトが実行時に行う何らかのアクションにすぎません (例えば、ダメージを受ける、ボタンをクリックする、などなど)。

そして、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と呼ばれるメソッドを作成し、デリゲートのシグネチャと一致させなければならない。

各オブザーバーのイベントハンドラはパブリッシャーのイベントを購読します。必要な数のオブザーバーをサブスクリプションに参加させることが出来ます。全てのオブザーバーはイベントが発生するのを待ちます。

パブリッシャーが実行時にイベントの発生を知らせることを、イベントを発生させると言いま す。これによってサブスクライバーのイベントハンドラが呼び出され、それに応じて内部ロジックが実行されます。

このように、被写体からの1つのイベントに対して、多くのコンポーネントを反応させることができます。例えば、ボタンがクリックされたことを示すと、オブザーバーはアニメーションやサウンドを再生したり、カットシーンを表示したり、ファイルを保存したりすることができます。このように、オブザーバーパターンは、オブジェクト間のメッセージ送信によく使われます。



主体がイベントを発生させ、観測者に通知する。

例単純な被写体と観察者

例えば、基本的な件名・出版社名はこのように定義します。

```
using UnityEngine;
using System;

public class Subject : MonoBehaviour
{
    public event Action ThingHappened;

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

ここでは、GameObjectに簡単にアタッチできるようにMonoBehaviourを継承しています。これは必須ではありません。

独自のデリゲートを定義するのは自由ですが、ほとんどの場合 System.Action が動作します。イベントと一緒にパラメータを送信する必要がある場合は、Action<T> デリゲートを使用し、角括弧の中に List<T> としてパラメータを渡します（最大 16 個のパラメータ）。

ThingHappenedは実際のイベントであり、サブジェクトがDoThingで呼び出すメソッドを使用します。

イベントを聞くには、Observerクラスのサンプルを作成します。ここでは、便宜上、MonoBehaviourを継承していますが、これは必須ではありません。

```
public class Observer : MonoBehaviour (オブザーバー)
{
    [SerializeField] private Subject SubjectToObserve;

    private void OnThingHappened()
    {
        // イベントに応答するロジックはすべてここに置く 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を参照する。を、ThingHappenedイベントをリッスンするためのInspectorの順序で実行します。

OnThingHappened メソッドには、オブザーバーがイベントに応答して実行するロジックを含めることができます。多くの場合、開発者はイベントハンドラを表すために "On" というプレフィックスをつけます (スタイルガイドにある命名規則に従ってください)。

AwakeやStartでは、+=演算子でイベントにサブスクライブすることができます。これは、オブザーバの OnThingHappened メソッドとサブジェクトの ThingHappened を組み合わせたものです。

もし何かで subject の DoThing メソッドが実行されると、イベントが発生します。そして、observerのOnThingHappenedイベントハンドラが自動的に起動し、デバッグ文が表示されます。

注意：ThingHappenedにサブスクライブしたまま、実行時にオブザーバーを削除したりすると、そのイベントの呼び出しがエラーになる可能性があります。したがって、MonoBehaviourのOnDestroyメソッドで、-=演算子でイベントの購読を解除することが重要です。

observerパターンは、ゲームプレイ中に発生するほぼ全ての出来事に適用できます。例えば、プレイヤーが敵を倒したり、アイテムを集めたりするたびに、イベントを発生させることができます。また、スコアや実績を記録する統計システムが必要な場合、Observer パターンを使えば、元のゲームプレイのコードに影響

を与えることなく、統計システムを作成することができます。

多くのUnityアプリケーションは、イベントを適用します。

目的または目標

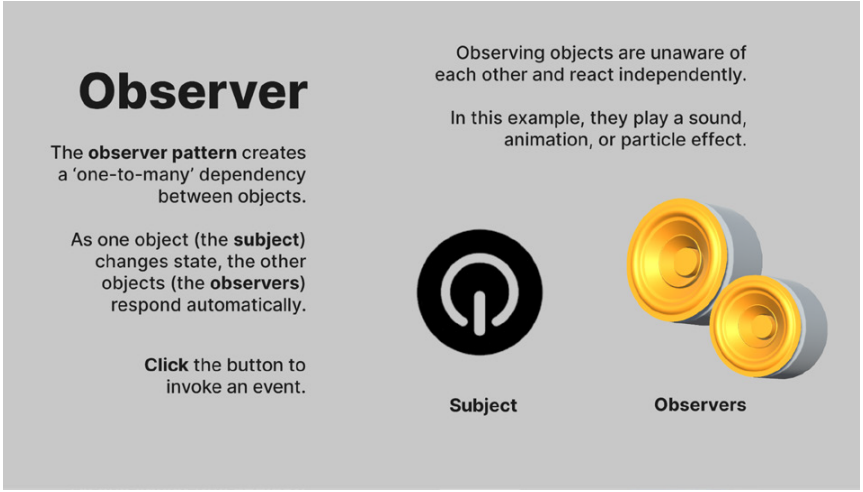
勝敗条件

プレイヤー死亡、エネミー死亡、ダメージのいずれか。

アイテムピックアップ

ユーザーインターフェース

主体は、適切なタイミングでイベントを発生させるだけで、何人ものオブザーバーがサブスクライブすることができます。



オブザーバーのサンプルシーン

サンプルプロジェクトでは、ButtonSubjectによって、ユーザーがマウスボタンがクリックされたイベント。AudioObserverと`ParticleSystemObserver`コンポーネントを持つ他のいくつかのGameObjectは、そのイベントに対して独自の方法で反応することができます。

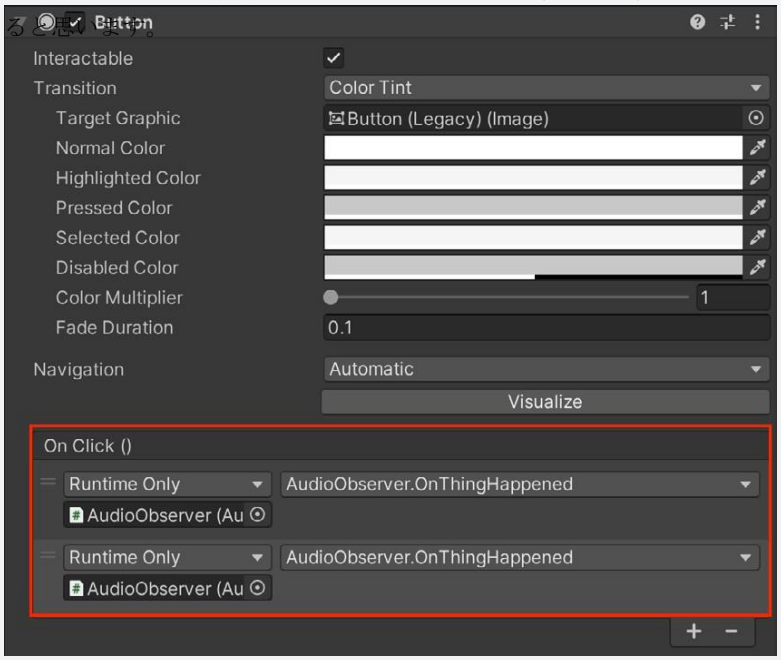
どの対象が「主体」で、どの対象が「観察者」であるかの判断は、使い方によって異なるだけです。イベントを発生させるものが主体、イベントに反応するものが観測者となる。同じGameObject上の異なるコンポーネントがsubjectにもobserverにもなり得ます。同じコンポーネントでも、あるコンテキストではサブジェクト、別のコンテキストではオブザーバーになることがあります。

例えば、この例の`AnimObserver`は、クリックされたボタンにちょっとした動きを加えています。これはButtonSubject GameObjectの一部でありながら、オブザーバーとして動作しています。

UnityEventsとUnityActions

Unityには、**UnityEngine.Event** APIのUnityActionデリゲートを使用するUnityEventsという独立したシステムもあります。

UnityEvents は observer パターンのためのグラフィカルなインターフェイスを提供します。UnityのUIシステム（例えば、UIボタンのOnClickイベントの作成）を使ったことがある人は、すでに経験があると思います。



UnityEventsには、セットアップ用のグラフィカルコンポーネントがあります。

この例では、ボタンの OnClick イベントが、2つの AudioObservers の OnThingHappened メソッドを呼び出し、レスポンスをトリガーしています。このように、コード無しでサブジェクトのイベントを設定することができます。

UnityEvents は、デザイナーや非プログラマーがゲームプレイのイベントを作成する場合に便利です。しかし、System 名前空間からの同等のイベントまたはアクションよりも遅いかもしれないことに注意してください。

UnityEventsとUnityActionsを検討する際には、パフォーマンスと使用量を比較検討します。例として、Unity Learn の Create a Simple Messaging System with Events モジュールを参照してください。

長所と短所

イベントの実装には、多少の手間がかかりますが、その分メリットもあります。

observer パターンはオブジェクトの分離を助けます。イベントパブリッシャーはイベント購読者自身について何も知る必要はありません。あるクラスと別のクラスの間に直接的な依存関係を作る代わりに、サブジェクトとオブザーバーはある程度の分離を保ちながら通信を行います。

構築する必要はありません。C#には確立されたイベントシステムがあり、独自に定義する代わりにSystem.Actionのデリゲートを使用することができます。
のデリゲートです。あるいは、Unityには、UnityEventsとUnityActionsもあります。

各オブザーバは、独自のイベント処理ロジックを実装しています。このように、各オブザーバは応答するために必要なロジックを保持します。これにより、デバッグやユニットテストが容易になります。

ユーザーインターフェースに適しています。ゲームプレイのコアとなるコードは、UIロジックとは別に存在させることができます。UI要素は、特定のゲームイベントやコンディションをリッスンして、適切に応答します。MVPやMVCパターンでは、このような目的で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は、このような注意点があることを認識しておいてください。

複雑さが増す他のパターンと同様、イベント駆動型アーキテクチャを作るには、より多くの設定を前もって行う必要があります。また、SubjectやObserverの削除には注意が必要です。OnDestroyでobserversの登録を解除することを忘れないように。

オブザーバはイベントを定義しているクラスへの参照を必要とします。オブザーバは、イベントを発行しているクラスへの依存性をまだ持っています。すべてのイベントを処理する静的な EventManager (下記) を使用することで、オブジェクトとオブジェクトを切り離すことができます。

パフォーマンスが問題になることがある。イベントドリブンアーキテクチャは余分なオーバーヘッドを追加します。大規模なシーンや多くのGameObjectはパフォーマンスの妨げになることがあります。

改善点

ここでは、observerパターンの基本バージョンだけを紹介しましたが、これを拡張してゲームアプリケーションのあらゆるニーズに対応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、これらの提案を考慮してください。

ObservableCollectionクラスを使用します。C#は、特定の変更を追跡するための動的なObservableCollectionを提供します。これは、アイテ

ムが追加、削除されたとき、またはリストが更新されたときに、オブザーバに通知することができます。

引数に一意のインスタンスIDを渡す。階層にある各GameObjectは一意のインスタンスIDを持っている。もし、複数のObserverに適用されるようなイベントを発生させる場合は、その一意なIDをイベントに渡す（Action<int>型を使う）。そして、そのIDに一致するGameObjectがある場合のみ、イベントハンドラ内のロジックを実行する。

静的なEventManagerを作成します。イベントはゲームプレイの大部分を駆動するため、多くの Unity アプリケーションは静的またはシングルトンの EventManager を使用します。この方法では、オブザーバがゲームイベントの中心的なソースを参照することができ、セットアップが容易になります。

FPS Microgameは、カスタムのGameEventを実装し、リスナーを追加または削除するための静的ヘルパーメソッドを含む静的EventManagerの良い実装を持っています。

Unity Open Projectでは、ScriptableObjectを使用してUnityEventを中継するゲームアーキテクチャも紹介されています。オーディオを再生したり、新しいシーンをロードしたりするためにイベントを使用します。

イベントキューを作成する。シーン内にたくさんのオブジェクトがある場合、一度にイベントを発生させたくないことがあります。1つのイベントを呼び出すと、1000個のオブジェクトが音を再生する不協和音を想像してみてください。

observerパターンとcommandパターンを組み合わせることで、イベントをイベントキューにカプセル化することができます。そうするとコマンドバッファを使用して、イベントを1つずつ再生したり、必要に応じて選択的に無視することができます（一度に音を出せるオブジェクトの数が最大である場合など）。

observerパターンは、次章で詳しく説明するModel View Presenter (MVP) アーキテクチャパターンに大きく関わってきます。

モデル　ビュー
プレゼンター[MVP]

MVC（Model View Controller）は、ユーザーインターフェースを開発する際によく使われるデザインパターン群です。

MVCの背後にある一般的な考え方は、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分離することです。これにより、不要な依存関係を減らし、スパゲッティ・コードを削減することができます。

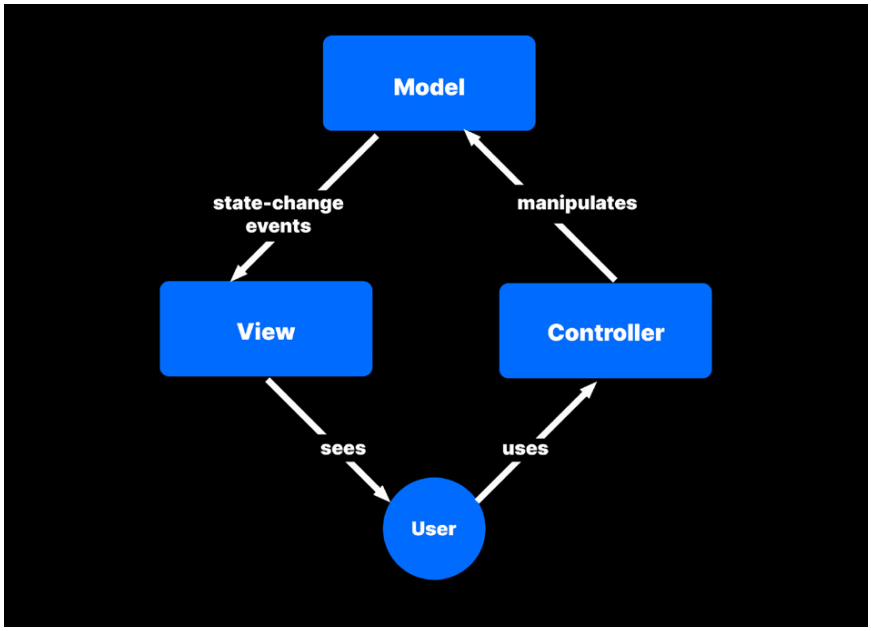
MVC（Model View Controller）デザインパターン

MVCパターンは、その名の通り、アプリケーションを3つの層に分割します。

モデルはデータを格納します。モデルは厳密に言えば、値を保持するデータコンテナです。ゲームプレイのロジックを実行したり、計算を実行したりすることはありません。

ビューはインターフェースです。ビューは、データをフォーマットし、画面上にグラフィカルなプレゼンテーションを表示します。

コントローラーは、ロジックを処理します。頭脳のようなものだと思ってください。ゲームデータを処理し、実行時にどのように値が変化するかを計算します。



モデル、ビュー、コントローラ

この関心事の分離は、これら3つの部分が互いにどのように相互作用するかを明確に定義するものでもあります。モデルはアプリケーションのデータを管理し、ビューはそのデータをユーザーに表示します。Controller は入力を処理し、ゲームデータに対して何らかの決定や計算を行います。そして、その結果を Model に送り返します。

したがって、Controllerはそれ自体にはゲームデータを持ちません。また、Viewも同様です。MVCデザインは、各レイヤーが行うことを制限しています。ある部分はデータを保持し、別の部分はデータを処理し、最後の部分はそのデータをユーザーに表示します。

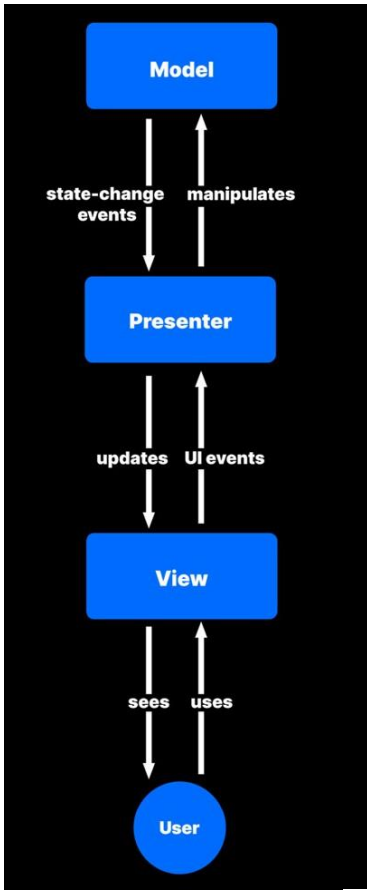
表面的には、単一責任の原則の延長線上にあると考えることができます。各パーツが一つのことをしっかりと行う、これがMVCアーキテクチャの利点の一つです。

モデルビュープレゼンター(MVP)とUnity

MVCでUnityプロジェクトを開発する場合、既存のUIフレームワーク（UI ToolkitまたはUnity UI）は当然ながらViewとして機能します。エンジンは完全なユーザーインターフェースの実装を提供するため、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。

しかし、従来のMVCパターンに従うと、実行時にモデルのデータの変更をリッスンするためのビュー固有のコードが必要になります。

これは正しいアプローチですが、多くの Unity 開発者は MVC のバリエーションとして、Controller を仲介役として使用することを選択します。この場合、View は直接 Model を観察しません。その代わり、次のような処理を行います。



MVP：MVCのバリエーション

このMVCのバリエーションは、Model View Presenterデザイン、またはMVPと呼ばれています。MVPは、3つの異なるアプリケーション層による関心事の分離をまだ保持しています。しかし、各パートの責任を少し変えます。

MVPでは、Presenter（MVCではControllerと呼ばれる）は他のレイヤーの仲介役として機能します。モデルからデータを取得し、ビューに表示するためにデータを整形します。MVPでは、どの層が入力を処理するかを切り替えます。コントローラではなく、Viewがユーザの入力を処理する責任を負います。

イベントとオブザーバーパターンがこのデザインにどのように関わっているかに注目してください。ユーザーは、Unity UI の Button、Toggle、Slider コンポーネントと対話することができます。View レイヤーはこの入力をが UI イベントを介して Presenter を操作し、Presenter がモデルを操作します。モデルからの状態変化イベントは Presenter にデータが更新されたことを知らせます。Presenter は変更されたデータを View に渡し、UI を更新します。

例健康インターフェース

MVPの例として、キャラクターやアイテムの健康状態を表示する簡単なシステムを想像してください。データとUIを混在させた一つのクラスにすべてを詰め込むこともできますが、それではうまくスケールしないでしょう。さらに機能を追加して拡張する必要があるため、より複雑になります。

その代わり、Healthコンポーネントをより MVPを意識した方法で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コンポーネントはこのような感じになります。

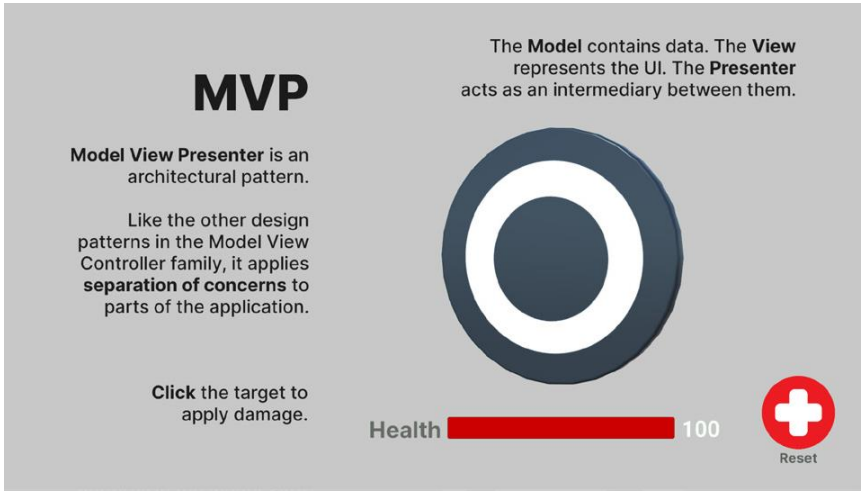
```
public class Health:MonoBehaviour
{
    public event Action HealthChanged;

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

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

    public void Increment(int amount)
    {
        currentHealth += amount;
        currentHealth = Mathf.Clamp(currentHealth、minHealth、max-)
健康）です。
        UpdateHealth()です。
    }

    public void Decrement(int amount)
    {
        currentHealth -= amount;
        currentHealth = Mathf.Clamp(currentHealth、minHealth、max-)
健康）です。
        UpdateHealth()です。
    }

    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。実際の健康状態を保存し、その値が変化するたびにイベント「HealthChanged」を呼び出します。Healthはゲームプレイロジックを含まず、データのインクリメントとデクリメントを行うメソッドのみです。

しかし、ほとんどのオブジェクトは、Healthそのものを操作することはありません。を予約することになります。
そのタスクのためのHealthPresenter。

```
public class HealthPresenter : MonoBehaviour.
{
    [SerializeField] 健康 health;
    [SerializeField] スライダー healthSlider;

    private void Start()
    {
        if (health != null)
        {
            health.HealthChanged += OnHealthChanged;
        }
        UpdateView()を実行します。
    }

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

    public void Damage(int amount)
    {
        健康？.Decrement(量)。
    }

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

    public void Reset()
    {
        health?.Restore()を実行します。
    }

    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()を実行します。
    }
}
```

他のGameObjectは、Damage、Heal、Resetを使用してヘルス値を変更するために、HealthPresenterを使用する必要があります。HealthPresenterは通常を使用して、Health が HealthChanged イベントを発生させるまで、UpdateView でユーザインタフェースを更新します。これは、モデル内の値の設定に短い時間がかかる場合（例えば、値をディスクに保存したり、データベースに保存したりする場合）に便利です。



MVPを使用したヘルスインターフェイスのサンプル

サンプルプロジェクトでは、ユーザがクリックすることで、ターゲットオブジェクトにダメージを与えたり、ボタンでHealthをリセットすることができます。これらは、Healthを直接変更するのではなく、HealthPresenter（DamageまたはResetを呼び出す）に通知します。UI TextとUI Sliderは、Healthがイベントを発生させ、HealthPresenterにその値が変更されたことを通知したときに更新されます。

長所と短所

MVP（およびMVC）は、大規模なアプリケーションで真価を発揮します。もし、あなたのゲームの開発に大規模なチームが必要で、発売後も長期間維持することを想定しているなら、次のようなメリットがあるかもしれません。

> **スムーズな作業分担**ViewとPresenterを分離したことにより、ユーザーインターフェースの開発や更新を他のコードベースからほぼ独立させることができます。
>
> これにより、専門の開発者に分業することができます。あなたのチームには、専門のフロントエンド開発者がいますか？彼らにViewを任せましょう。彼らは他のメンバーから独立して作業することができます。
>
> **MVPとMVCによるユニットテストの簡素化：**これらのデザインパターンは、ゲームプレイのロジックをユーザーインターフェースから分離します。そのため、Editor で実際に Play モードを入力しなくても、コードで動作するオブジェクトをシミュレートすることができます。これにより

、かなりの時間を節約することができます。

読みやすく、メンテナンスしやすいコードこのデザインパターンでは、より小さなクラスを作る傾向があり、読みやすくなります。依存関係が少ないということは、通常、ソフトウェアが壊れる場所が少なく、バグが隠されている可能性のある場所が少ないということです。

MVCとMVPはWeb開発や企業向けソフトウェアで広く使われていますが、多くの場合、アプリケーションが十分なサイズと複雑さ に 達するまで、その利点は明らかになりません。Unityのプロジェクトでどちらのパターンを実装する場合でも、事前に以下の点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ります。このガイドで説明する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ーのパターンです。これらのいずれかを使用するには、クラスを責任ごとに分割する必要があり、これにはある程度の整理が必要で、前もってより多くの作業を行う必要があります。

Unityプロジェクトのすべてがこのパターンに当てはまるわけではありません。純粋な」MVC や MVP の実装では、画面にレンダリングするものはすべて、本当に View の一部です。すべてのUnityコンポーネントが、データ、ロジック、インターフェイスに簡単に分けられるわけではありません（例：MeshRenderer）。また、単純なスクリプトは、MVC/MVPの恩恵をあまり受けないかもしれません。

このパターンから最も恩恵を受けることができる場所を判断する必要があります。通常は、ユニットテストのガイドに従うとよいでしょう。もしMVC/MVPがテストを容易にするのであれば、アプリケーションのその側面について検討しましょう。
それ以外の場合は、無理に柄をつけようとしないことです。

結論

ソフトウェアパターンに初めて触れる方は、このガイドを読んで、Unityの開発で遭遇する最も一般的なパターンを理解して頂ければと思います。

プレファブを生産するための工場や、AIの状態パターンなど、必要に応じてこれらのテクニックを利用してください。デザインパターンをいつ、どのように適用するかを認識することは、次のUnityの課題に取り組む際に役立ちます。もちろん、特定のパターンを無理やり当てはめることに迷う必要はありません。パターンを使わないことも、パターンを使うことと同じくらい重要なのです。

デザインパターンは、正しく適用することで、ワークフローをスピードアップし、繰り返し発生する問題をエレガントに解決することができます。そして、プレイヤーに楽しく、ユニークな体験を提供するという、重要なことに集中できるようになります。

だから、車輪を再発明する必要はないけれど、自分なりの工夫は絶対にできるはずです。

その他のデザインパターン

このガイドは、コンピュータやゲーム開発でよく知られているいくつかのデザインパターンのほんの一部に過ぎません。それらの詳細については説明しませんが、ここでは、あなたにとって有用と思われる他のいくつかの概要について説明します。

アダプタ。関連性のない2つのエンティティの間にインターフェイス（ラッパーともいう）を提供し、一緒に動作できるようにするもの。

フライウェイト：多数のオブジェクトがある場合、ベースクラスで共通のプロパティを共有し、リソースを節約する。例えば、森をデザインする場合、木の普遍的なプロパティをすべて取り出して、ベースとなるTreeクラスに格納する。そうすれば、サブクラス（PineTree、MapleTreeなど）でそれらを繰り返す必要はない。

ダブルバッファ。計算が終了するまでの間、2組の配列データを保持することができます。これは、セル・オートマトンなどの手続き型シミュレーションや、画面へのレンダリングに便利です。

Dirtyフラグ。このテクニックは、ゲーム内で何かが変更されたが、コストのかかる操作（ディスクへの保存や物理シミュレーションの実行など）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。

インタプリタ/バイトコード。改造をサポートしたり、プログラマーでなくてもゲームを拡張できるようにしたい場合、ユーザーが外部テキスト ファイルで編集できる簡易言語を作成できます。バイトコードコンポーネントは、その解釈された言語をC#ゲームコードに変換します。

サブクラスのサンドボックス：様々な動作をする類似のオブジェクトがある場合、それらの動作を親クラスでprotectedとして定義することができます。そうすれば、子クラスが混ざって新しい組み合わせを作ることができる。

オブジェクトを型付けする。GameObjectの種類が多い場合、それぞれのサブクラスを作るのではなく、1つの抽象クラスまたは親クラスで、可能なすべての動作を定義します。個々のオブジェクトの特別な特性は、別のデータファイル（ScriptableObjectなど）で区別し、コードを変更することなくカスタマイズできるようにします。例えば、同じクラスから派生した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データファイルをカスタマイズして、各アイテムをユニークなものにすることができます（例：RPG用の武器など）、すべてプログラマーの手を借りずにできます。

データローカリティ。データを最適化し、メモリに効率よく格納できるようにすれば、パフォーマンスの向上という恩恵を受けることができます。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しやすくすることができます。UnityのECSとDOTSアーキテクチャは、このパターンを実装しています。

空間的なパーティショニング。大きなシーンやゲーム世界では、特別な構造を使ってGameObjectを位置ごとに整理します。グリッド、トライ（四分木、八分木）、二分木はすべて、分割と検索をより効率的に行うためのテクニックです。

デコレーター。オブジェクトの構造を変更することなく、そのオブジェクトに責任を持たせることができる。例えば、武器の基本クラスを変更することなく、武器に特典を追加することができます。

ファサード。より複雑なシステムに対して、シンプルで統一されたインターフェイスを提供するものです。AI、アニメーション、サウンドの各コンポーネントを持つGameObjectがあったとして、それらのコンポーネントの周りにラッパークラスを追加します（PlayerInput、PlayerAudioなどを管理するPlayerコントローラクラスを想像してください）。このファサードは、元のコンポーネントの詳細を隠し、使い方を簡素化します。

テンプレートメソッド。このパターンは、アルゴリズムの正確なステップをサブクラスに委ねるものです。例えば、抽象クラスでアルゴリズムやデータ構造の大まかな骨格を定義し、サブクラスがアルゴリズム全体の構造を変えずに特定の部分をオーバーライドできるようにすることができる。

ストラテジーです。この動作パターン（ポリシーパターンとも呼ばれる）は、アルゴリズムのファミリーを定義し、それぞれをクラス内にカプセル化するのに役立つ。これにより、各アルゴリズム（ストラテジー）は実行時に交換可能なものとなる。例えば、経路探索システムを作る場合、Strategyパターンを使って複数のアルゴリズム（A*、Dijkstraの最短経路など）を定義し、ゲームプレイ中に文脈に応じて交換できるようにすることができる。

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編成し、その結果できた構造を個々のオブジェクトと同じように扱います。ツリーは、単純要素と複合要素（葉とコンテナ）の両方から構成されます。どの要素も同じインターフェースを実装しているので、ツリー全体に対して再帰的に同じ動作を実行することができます。

ユニティ
ドットコ
ム