インターフェイスを用いたif文を殺す設計

この勉強会で話すこと

- インターフェイスを使ったif文, switch文などの分岐を減らす設計方法について説明します
- インターフェイスを使った設計を行う上で有用なSOLID原則について説明します

既に理解されてる方も多いような気がしますが少しだけお付き合いください。
なお、サンプルコードは全部C#です。

参考文献

- 現場で役立つシステム設計の原則
- Unity開発で使える設計の話 (slideshare)

ifがあると何が悪いか

まずifがあると何が悪いかということについて実例を交えながら説明していきます。

```
// 銃弾クラス
public class Bullet
{
    // 衝突判定メソッド
    private void OnCollisionEnter(Collision collision)
    {
        var hit = collision.gameObject;

        // ぶつかった相手に合わせた処理を実行する
        switch(hit.tag)
        {
            case "Enemy":
                var e = hit.GetComponent<Enemy>();
                if(e != nul) e.ApplyDamage();
                break;

            case "Player":
                var e = hit.GetComponent<Player>();
                if(p != null) p.HitBullet();
                break;
        }
    }
}
```

このコードの悪い点は変更に弱いという部分です。 実際に変更が加えられたときのコードを見てみます。 新しくBossというクラスを作り、銃弾がBossに衝突したときの判定を追加します。

変更があったとき

```
public class Bullet
{
    private void OnCollisionEnter(Collision collision)
    {
        var hit = collision.gameObject;

        switch(hit.tag)
        {
            case "Enemy":
                var e = hit.GetComponent<Enemy>();
                if(e != nul) e.ApplyDamage();
                break;

            case "Player":
                var e = hit.GetComponent<Player>();
                if(p != null) p.HitBullet();
                break;

            // ボスに対する処理を追加
            case "Boss":
                var b = hit.GetComponent<Boss>();
                if(b != nul) b.ApplyDamage();
                break;
        }
    }
}
```

このコードには以下のような問題があります。

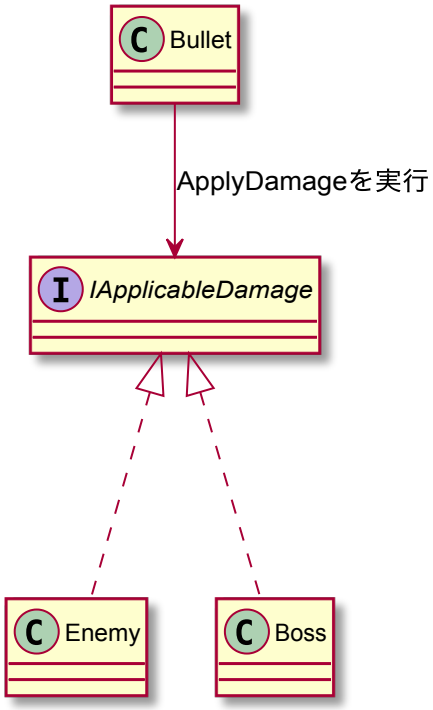
- Bossクラスを追加することでif, switch文を全部探して変更しなければならない(Bulletクラス以外も！)
- 修正漏れがあったときにエラーが出ない
- 同じような処理が複数回出てきて冗長
- 分岐が多いので処理が追いにくい

インターフェイスを使ってこれらの問題を解決していきます。

インターフェイスを使ってif文を減らし変更に強いクラス設計にする

上の例をインターフェイスを使って書き換えます。

```
private void OnCollisionEnter(Collision collision)
{
    var hit = collision.gameObject;

    var d = hit.GetComponent<IApplicableDamage>();
    if(d != null) d.ApplyDamage();
}
```



インターフェイスを使って抽象化したことでswitch文が消えました。

switch文を使っていたときは使う側（Bulletクラス）が相手の型を意識する必要があり、これによって変更に弱い不健全な設計になっていました。
インターフェイスを使うことによってBulletはEnemyもPlayerも知る必要はなく、操作対象を追加したい場合もBulletへの変更はいっさい必要なくなります。

SOLID原則

これまでの説明でインターフェイスを使うことで変更に強い設計を実現できることを理解していただけたと思います。
これ以降はインターフェイスを使う上で非常に重要な要素であるSOLID原則について実例を交えながら解説していきます。

まず、SOLID原則とは以下の5つの原則です。

- 単一責任の原則 (**S**ingle Responsibillity Principle)
- オープン・クローズドの原則 (**O**pen/Closd Principle)

- リスコフの置換原則 (**L**iskov Substitution Principle)
- インターフェイス分離の原則 (**I**nterface Segregation Principle)
- 依存性逆転の原則 (**D**ependency Inversion Principle)

順番に説明していきます。

単一責任の原則 (Single Responsibillity Principle)

単一責任の原則は以下のような原則です。

> 一個のクラスに役割は一つ

これは説明するまでもないと思うので定義だけ確認して次に行きます。
なんでもできるクラスは今更言うまでもなく厄介です。

オープン・クローズドの原則 (Open/Closd Principle)

オープン・クローズド原則は以下のような原則です。

> モジュールは
> **拡張**について**開いて**いなければならず、
> **修正**に対して**閉じて**いなければならない

要するに機能追加はしやすいようにする（オープン）
ただしそのときに修正が発生してはいけない（クローズ）
という原則です。

オープン・クローズドの原則を実現するためのポイントは抽象化を行うことです。
インターフェイスや規程クラスを使って抽象化を行い、使う側から使われる側の実装を隠蔽してやることで変更に強い設計を行うことができます。

リスコフの置換原則 (Liskov Substitution Principle)

リスコフの置換原則は以下のような原則です。

> 派生型はその基底型と置換可能でなければいけない

これについては言葉そのままで派生クラスは基底型に変換したときに正常に動くように作りましょうということです。
言い換えると**派生型は基底型で決めたルールを変更してはいけない**ということになります。

以下に例を示します。

よくない例1（というかできない例）

```
public interface ITest
{
    public int Sum(int a, int b);
}

class TestA: ITest
{
    // アクセス修飾子が違う
    private int Sum(int a, int b)
    {
        return a + b;
    }
}

class TestB: ITest
{
    // 引数, 返り値の範囲が継承元より厳しい
    public short Sum(short a, short b)
    {
        return a + b;
    }
}
```

上記の例はC#ではそもそもコンパイルできないので動かないコードなのであくまで例として見ておいてください。

TestAはインターフェイスではpublicで定義されているSumメソッドに対して実装ではprivateになってしまっています。
TestBはインターフェイスでは引数がintになっているのに対して実装はshortで定義されています。

TestA、TestBについても外から呼び出した場合問題が起こるであろうことは予想できると思います。

このように、

- アクセス修飾子を派生で勝手に上書きしてはいけない
- メソッドを実行するのに必要な判定を基底より強化してはいけない
- メソッドの実行結果を基底より緩くしてはいけない

というのは型安全性を保つ上で重要なことになってきます。

次によくない例をもう一つあげます。

よくない例2

```
interface IItem
{
    int Price{ get; }

    // 定価を返すメソッド
    int GetPrice();
}

class ItemA() : IItem
{
    private int Price { get; } = 200;

    // 販売額を返してしまっている
    public int GetPrice()
    {
        var total = Price * 1.1;
        return (int)total;
    }
}
```

インターフェイスIItemは定価を返すメソッドしてGetPriceを定義していますが、これを実装するItemAクラスでは販売額を返してしまっています。
こちらはコンパイルは通りますが、このGetPriceメソッドを用いた計算結果は本来得たい結果とずれてしまいます。

リスコフの置換原則を適用した例

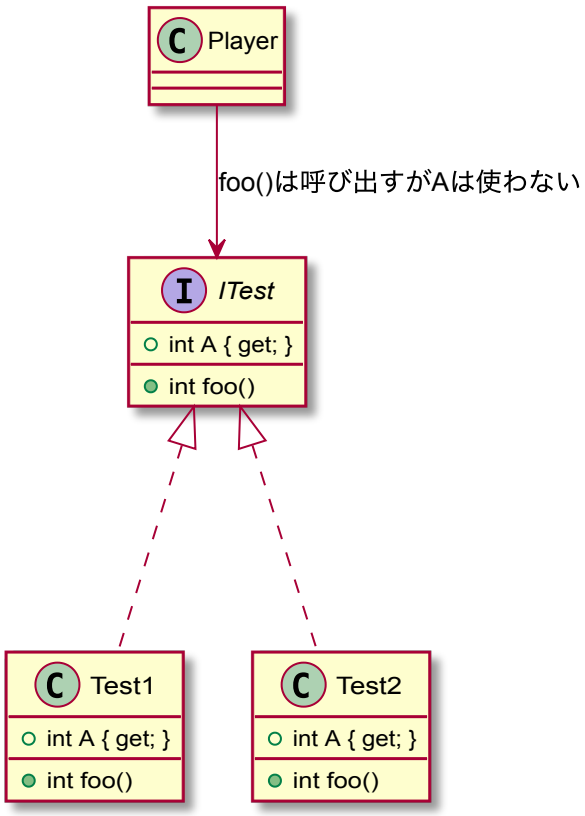
```
interface ITest
{
    int Price{ get; }

    // 定価を返すメソッド
    int GetPrice();
}

// 正常なクラス
class TestA() : ITest
{
    private int Price{ get; } = 100;

    public int GetPrice()
    {
        return Price;
    }
}
```

派生型の実装は基底型の決めたルールにきちんと従うようにしましょうというのがリスコフの置換原則になります。

インターフェイス分離の原則 (Interface Segregation Principle)

> クライアントが利用しないメソッドへの依存を強制してはいけない

要するにインターフェイスには必要なメソッドやプロパティだけ定義しようという話です。



この場合Aは特に参照されていないのでわざわざインターフェイスで定義されている必要がないです。

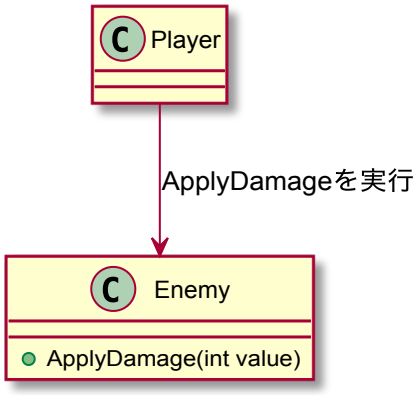
このへんは特に説明はいらない気がするので飛ばします。

依存性逆転の原則 (Dependency Inversion Principle)

> 上位モジュールが下位モジュールに依存してはいけない
> どちらも抽象に依存すべきである

上位が仕様を決め、下位がそれに従うのが正しい設計

よくない例

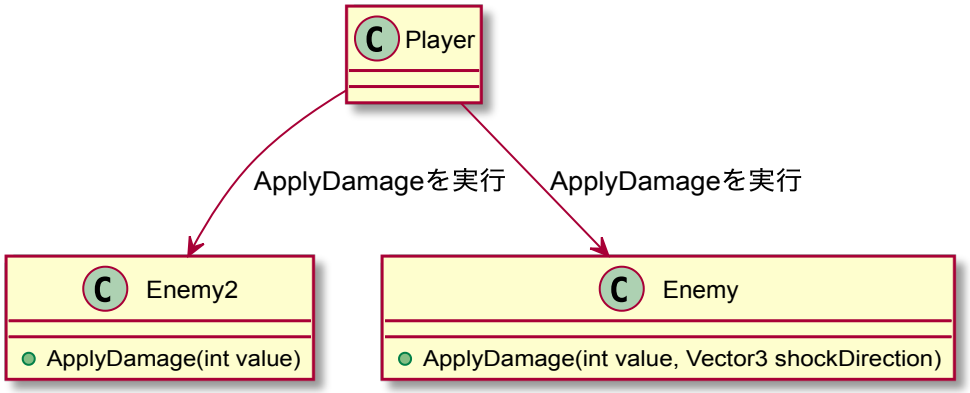


PlayerクラスとEnemyクラスがあり、PlayerがEnemyの持つメソッドApplyDamageを実行しています。

この設計だとPlayerの実装がEnemyの実装に左右されてしまう、つまり依存してしまっているので、変更に弱い仕様になってしまっています。

変更の例

実際にこのコードに変更を加えてみます。

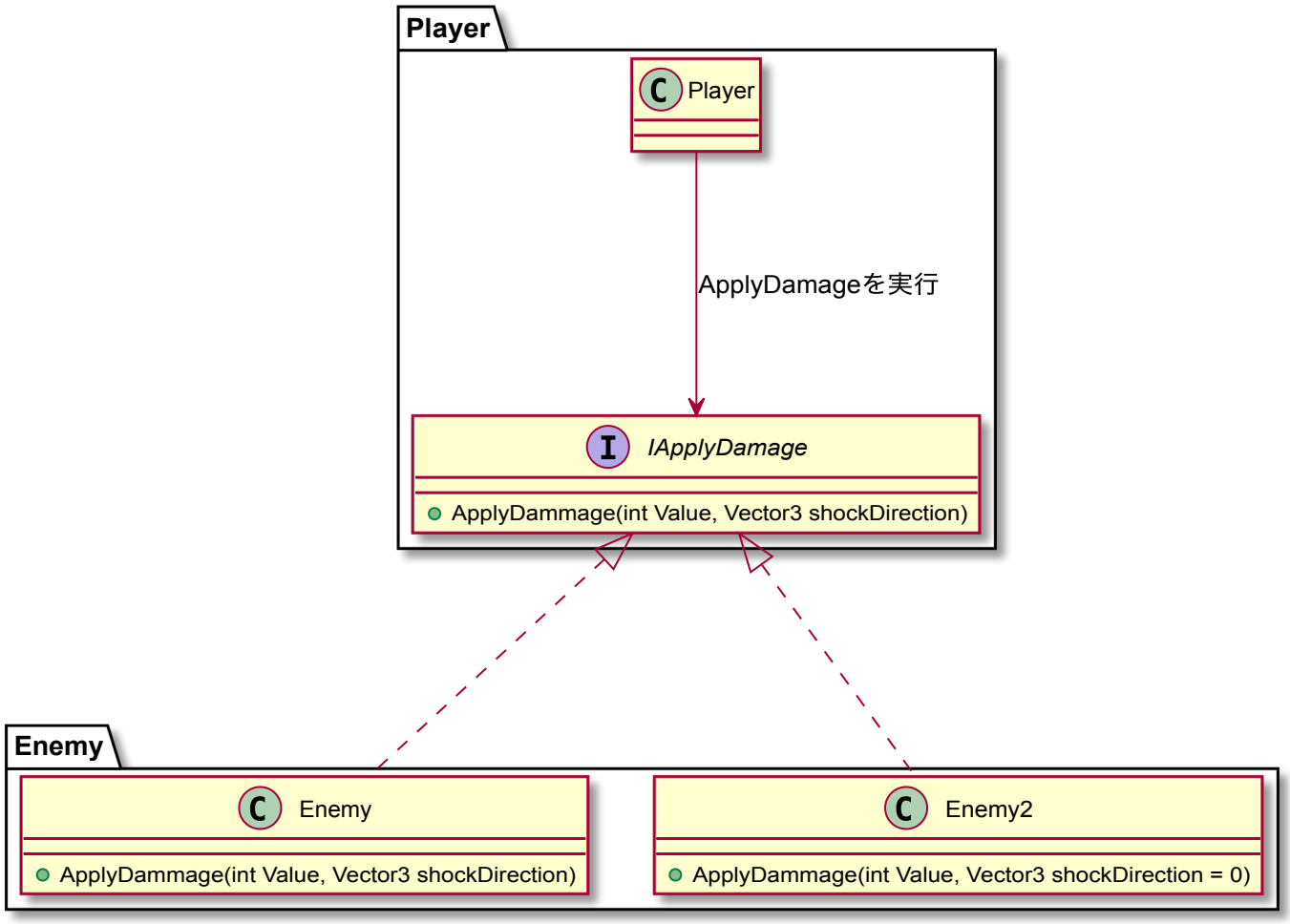


新しくEnemy2クラスが追加されており、Enemy2のApplyDamageに引数が増えています。
この場合、Enemy2の追加、Enemyの変更に合わせてPlayerクラスを修正する必要があります。
これはオープン・クローズド原則にも反しており、変更に弱い設計になってしまっています。

また、Enemy2が増えたことで処理の分岐も必要になり、よい設計とはいえません。

DIPを適用した例

PlayerパッケージにIApplyDamageというインターフェイスを定義し、Enemyクラスがそれを実装しています。



まずEnemyからIApplyDamageに伸びている矢印を見てください。
元々は「Player」が「Enemy」に依存していたのが、「Enemy」が「Player」に依存するようになっている、つまり依存性が「逆転」していることがわかると思います。

こうして依存性が逆転したことにより、EnemyがIApplyDamageに依存するようになりました。
つまり、Playerが仕様としてIApplyDamageを定め、Enemyがそれに従うという関係が成り立っています。
このように、インターフェイスを用いて依存性を逆転させることで下位クラスが上位クラスの仕様に合わせるのではなく、上位クラスが下位クラスの仕様を決定することができました。

まとめ

今回の勉強会では以下について学びました。

- インターフェイスを使うことでif文を減らすことができる
- SOLID原則を適用することでより変更に強い設計を実現することができる

インターフェイスを使ってどんどんif文を減らしていきましょう。