「良いコード/悪いコードで学ぶ設計入門」を読んで

366394 Shota.Aizawa

0 章

はじめに

0. はじめに

きっかけ

- GW 中に「良いコード／悪いコードで学ぶ設計入門」を読みました
- まだ途中ですが順に皆さんに内容をかいつまんで展開したいと思います
- https://note.com/minodriven/n/n12af8005899f
- https://www.amazon.co.jp/dp/4297127830?tag=note0e2a-22&linkCode=ogi&th=1&psc=1

対象読者

- オブジェクト指向プログラミング言語の基礎知識はあるものの、設計がよくわからない/自信がない方、これから設計をしっかり学び始めようと考えている方

突然ですが、正方形の定義ってわかりますか？

四辺の長さがすべて等しい

内角がすべて直角

SW 開発で以下の経験はあるあるではないでしょうか？

- どこかのコードを変更すると、別の箇所でバグが発生した
- 変更の影響があるそうな箇所をあちらこちら探し回らなければならなくなった
- コードを読んでいるだけで日が暮れてしまった
- 簡単だと思っていた仕様変更やバグ修正に何日も費やしてしまった

これらの原因がわからない理由は正方形のときと違い変更に強いあるべき構造を知らないためです

SWの成長を阻害する設計や実装上の問題を「悪魔」と例え、正体を知覚し正しい対処ができるようになることを目的とした内容となっています

1章

悪しき構造の弊害を知覚する

設計を蔑ろにすると起こる弊害

- コードを読み解くのに時間がかかる
- バグを埋め込みやすくなる
- 悪しき構造が更に悪しき構造を誘発する

1.1. 意味不明な命名

技術駆動命名

```
class MemoryStateManager
{
    void ChangeIntValue01(int changeValue)
    {
        IntValue01 -= changeValue;
        if (IntValue01 < 0)
        {
            IntValue01 = 0;
            UpdateState02Flag();
        }
    }
}
```

- 型名を表す Int、メモリ制御を表す Memory や Flag など、プログラミング、コンピュータ用語に基づいた技術ベースでの命名を技術駆動命名と呼びます

連番命名

```
class Class01
{
    void Method001();

    void Method002();

    void Method003();
}
```

- クラスやメソッドに対して番号付けで命名することを連番命名と呼びます

- 技術駆動命名、連番命名は意図が全く読み取れない悪しき手法
- 意図や目的を表現した命名をすることで構造が簡明になります

1.2. 理解を困難にする条件分岐のネスト

何重にもネストしたロジック

```
        // 生存判定
        if(0 < member.HitPoint){
            // 行動可能判定
            if(member.CanAct()){
                // MP残存判定
                if(magic.ConstMagicPoint <= member.ConstMagicPoint){
                    member.ConsumeMagicPoint(magic.ConstMagicPoint);
                    member.Chant(magic);
                }
            }
        }
```

- 上記は RPG における魔法発動までの条件を実装した例です
- ネストしているとコードの見通しが悪くなりどこまでが if 文の処理ロジックなのか読み解きのが難しくなります

1.3. さまざまな悪魔を招きやすいデータクラス

- データクラスは単純な構造でありながら様々な悪魔を招きやすいです
- 業務契約を扱うサービスにて契約金額を扱う仕様を例にします

データしか持たないありがちなクラス構造

```
1   public class ContractAmount
2   {
3       public int AmountIncludingTax;   // 税込み金額
4       public decimal SalesTaxRate;     // 消費税率
5   }
```

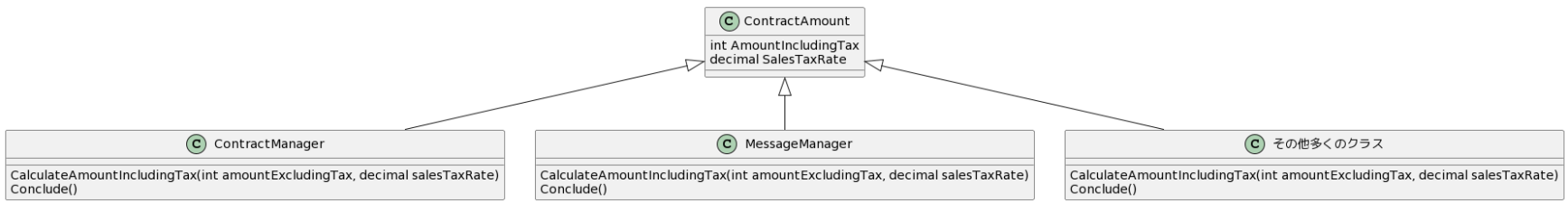
- 税込み金額と消費税率を public なインスタンス変数として持ち、自由にデータの出し入れが可能な構造です
- データの入れ物だけでなく税込み金額を計算するロジックが当然必要になります

CONTRACTMANAGER に書かれる金額計算ロジック

```
public class ContractManager
{
    public ContractAmount ContractAmount;

    // 税込金額を計算する
    public int
    CalculateAmountIncludingTax(int amountExcludingTax, decimal salesTaxRate)
    {
        decimal multiplier = Decimal.Add(salesTaxRate, new decimal(1.0));
        decimal amountIncludingTax =
            Decimal.Multiply(multiplier, new decimal(amountExcludingTax));

        return decimal.ToInt32(amountIncludingTax);
    }

    // 契約を締結する
    public void Conclude()
    {
        // 省略
        int amountIncludingTax =
            CalculateAmountIncludingTax(amountExcludingTax, salesTaxRate);
        ContractAmount = new ContractAmount();
        ContractAmount.AmountIncludingTax = amountExcludingTax;
        ContractAmount.SalesTaxRate = salesTaxRate;
        // 省略
```

1.3.1. 仕様変更時に牙をむく悪魔

- 先程のは別のクラスに税込金額の計算ロジックが実装されているパターンです
- このサービスにおいて消費税関係で仕様変更により、消費税の税率ロジックを変更しました
  - しかし、数日後に新しい消費税率になっていないとの障害報告があがりました
  - 同様のロジックが複数箇所にあるのではと疑い、調査をしたところ数十箇所で実装されていることが発覚しました



なぜ同様の計算ロジックが複数箇所に存在するのでしょうか？

税込み計算は金額を扱うあらゆるユースケースで必要になるため多くの箇所で実装されやすい

多くの場所で使うといいつつも、税込み計算ロジックをどこか一箇所にすればいいと考える方もいるでしょう

そうなると設計に無頓着だと税込み計算ロジックが実装されていることに気づかずに再実装されていしまう可能性があります

- これらの事態はデータを保持するクラスとデータを使って計算するロジックが離れているときに頻発します
- 離れているがゆえに複数実装されていても認知されず、再実装されてしまいます

関連するデータやロジック同士が分離し、バラバラになっているのを低凝集と言います
低凝集により引き起こされる弊害を次ページ以降にまとめます

1.3.2. 重複コード

- 先程の例で示した通り、実装済みの機能があるのに未実装と勘違いし、同じようなロジックをいたるところに複数実装してしまう可能性が高まります
- 意図せず重複コードが量産されることになります

1.3.3. 修正漏れ

- 重複コードが多く実装されている場合、仕様変更時にすべての重複コードを変更しなければならず、修正漏れが生じバグとなります

1.3.4. 可読性低下

- 関連するコード同士が分散していると重複コードも含めすべてを探し出すのに膨大な時間が必要となります

1.3.5. 生焼けオブジェクト

```
1       ContractAmount amount = new ContractAmount();
2       Console.WriteLine(amount.SalesTaxRate.ToString());
```

- ~~上記のコードを実行するとヌルポが発生します~~
  - C#だと SalesTaxRate が decimal だと初期値の 0、string だとヌルリになります
- ContractAmount は初期化の必要なクラスであること、未初期化状態が発生しうるクラスであることを利用者側が知らないとバグが生じてしまう不完全なクラス（生焼けオブジェクト）です

1.3.6. 不正値の混入

不正とは仕様として正しくない状態を指します

- 注文数がマイナスになっている
- ゲームにおいて HP の値が最大値を超えてしまっている

不正値を混入可能

- 負数の消費税率を代入するなどデータクラスは不整地を与えることが容易にできてしまいます

```
1       ContractAmount amount = new ContractAmount();
2       amount.SalesTaxRate = new decimal(-0.1);
```

- 不正値が混入しないようにデータクラスの利用側でバリデーションロジックを実装することがありますが、こちらも税込み計算ロジック同様に重複コードの発生原因となります

今までの話をまとめてできた CONTRACTAMOUNT クラス

```
public class ContractAmount
{
    public int AmountIncludingTax { get; }  // 税込み金額

    public int AmountExcludingTax { get; }  // 税抜き金額
    public decimal SalesTaxRate { get; }    // 消費税率

    public ContractAmount(int amountExcludingTax, decimal salesTaxRate)
    {
        if (salesTaxRate < 0)
        {
            throw new ArgumentException();
        }

        if (amountExcludingTax < 0)
        {
            throw new ArgumentException();
        }
        AmountExcludingTax = amountExcludingTax;
        SalesTaxRate = salesTaxRate;
        AmountIncludingTax = CalculateAmountIncludingTax(amountExcludingTax, salesTaxRate);
    }

    // 税込金額を計算する
    private int CalculateAmountIncludingTax(int amountExcludingTax, decimal salesTaxRate)
    {
```

2 章

設計の初歩

- クラス設計の前に設計の基本的な考えからから入ります。
- 簡単なコードを例に、`どういったことをするのが設計なのか理解することを目的とします。`

2.1. 省略せずに意図が伝わる名前を設計する

```
1       int d = 0;
2       d = p1 + p2;
3       d = d - ((d1 + d2) / 2);
4       if (d < 0)
5       {
6           d = 0;
7       }
```

- 何かの計算になっているが、何を計算しているのかがわからない。

- 実はゲームのダメージ計算のロジックで各変数下記の通りとなります。

変数	意味
d	ダメージ量
p1	プレイヤー本体の攻撃力
p1	プレイヤーの武器の攻撃力
d1	敵本来の防御力
d2	敵の防具の防御力

意図がわかる変数名に改善します

```
        int damageAmount = 0;
        damageAmount = playerPower + playerWeaponPower; // ①
        damageAmount = damageAmount - ((enemyDodyDefence    enemyArmorDefence) / 2); // ②
        if (damageAmount < 0)
        {
            damageAmount = 0;
        }
```

- 上記で見やすくはなりましたが課題があります

ダメージ量damageAmountに何度か値が代入されています

2.2.変数を使い回さない、目的ごとの変数を用意する

- 複雑な計算処理では計算の途中の結果を同じ変数に代入しがちです。
  - 上記を再代入と呼びます。
- コードの途中で変数の用途が変わってしまい、バグを埋め込んでしまう可能性があります。

```
1       int damageAmount = 0;
2       damageAmount = playerPower + playerWeaponPower; // ①
3       damageAmount = damageAmount - ((enemyBodyDefence enemyArmorDefence) / 2); // ②
4       if (damageAmount < 0)
5       {
6           damageAmount = 0;
7       }
```

- ① で damageAmount に代入されているのはプレイヤーの攻撃力の総量です。
- ② は敵の防御力の総量を計算しています。

```
    int totalPlayerAttackPower = playerPower playerWeaponPower;
    int totalEnemyDefence = enemyBodyDefence enemyArmorDefence;
    int damageAmount = totalPlayerAttackPower (totalEnemyDefence / 2);
    if (damageAmount < 0)
    {
        damageAmount = 0;
    }
```

- 全体としてどんな値を扱っているのか、ある値を算出するのにどんな値を使っているのか、関係性がわかりやすくなりました。

2.3.ベタ書きせず、意味のあるまとまりでメソッド化

- 攻撃力、防御力の総量の計算や計算結果を格納する変数を分けましたが、一連の処理の流れはすべてベタ書きになっています。
- 意味のあるまとまりでロジックをまとめメソッドとして実装しましょう。

```
1       private int SumUpPlayerAttackPower(int playerArmPower, intplayerWeaponPower)
2       {
3           return playerArmPower + playerWeaponPower;
4       }
5       private int SumUpEnemyDefence(int enemyBodyDefence, intenemyArmorDefence)
6       {
7           return enemyBodyDefence + enemyArmorDefence;
8       }
9       int EstimateDamage(int totalPlayerAttackPower, inttotalEnemyDefence)
10      {
11          int damageAmount = totalPlayerAttackPower -(totalEnemyDefence / 2);
12          if (damageAmount < 0)
13          {
14              damageAmount = 0;
15          }
16          return damageAmount;
17      }
```

- 先程のメソッドを呼び出す形に整理します。

```
        int totalPlayerAttackPower = SumUpPlayerAttackPow(playerBodyPower, playerWeaponPower);
        int totalEnemyDefence = SumUpEnemyDefen(enemyBodyDefence, enemyArmorDefence);
        int damageAmount = EstimateDama(totalPlayerAttackPower, totalEnemyDefence);
```

- 当初と同じ実行結果が得られるロジックですが見た目や構造がずいぶんと変わりました。

```
        int d = 0;
        d = p1 + p2;
        d = d - ((d1 + d2) / 2);
        if (d < 0)
        {
            d = 0;
        }
```

- このように保守しやすい、変更しやすいような変数名、ロジックに工夫を凝らすことも設計になります。

2.4. 関係し合うデータとロジックをクラスにまとめる

- ゲームを例に戦闘が伴うゲームでは主人公の HP があります。
- これがローカル変数など何らかの変数で定義されているとします。

```
1       int hitPoint;
```

- ダメージを受けて HP が減少するロジックが必要になり、どこかに実装されるでしょう。

```
1       hitPoint = hitPoint - damageAmount;
2       if (hitPoint < 0)
3       {
4           hitPoint = 0;
5       }
```

- そのうち回復アイテムでの HP 回復ロジックもどこかに実装されるでしょう。

```
1       hitPoint = hitPoint + recoveryAmount;
2       if (999 < hitPoint)
3       {
4           hitPoint = 999;
5       }
```

- こうした変数や変数を操作するロジックはゲームに限らず、バラバラに書かれがちです。
- 小さなプログラムですと問題ないですが大規模になればなるほど関係するロジックを探し回るだでも時間がかかります。
- また、変数 hitPoint に負数が入ってしまうなどの不正値が混入してしまうかもしれません。

上記の問題を解決するのがクラスです。クラスはデータをインスタンス変数として持ち、インスタンス変数を操作するメソッドをまとめることができます。

```
    public class HitPoint
    {
        private static readonly int Min = 0;
        private static readonly int Max = 999;
        public int Value { get; }
        public HitPoint(int value)
        {
            if (value < Min)
            {
                throw new ArgumentException();
            }
            if (Max < value)
            {
                throw new ArgumentException();
            }
            this.Value = value;
        }
        public HitPoint Damage(int damageAmount)
        {
            int damaged = Value - damageAmount;
            int corrected = damaged < Min ? Min : damaged;
            return new HitPoint(corrected);
        }
        public HitPoint Recover(int recoveryAmount)
        {
            int recovered = Value + recoveryAmount;
            int corrected = Max < recovered ? Max : recovered;
```

- ダメージは damage メソッド、回復は recover メソッドというように HitPoint クラスには HP に関するロジックが備わりました。
- コンストラクタでは 0~999 の範囲外は不正な値として弾くロジックにし、不正な値が紛れ込みバグにつながらないようなクラス構造になっています。