R流・C# と日本語で簡単デザインパターン

2008年04月05日

R・田中一郎

http://blogs.wankuma.com/rti/

Microsoft MVP for Development Tools - Visual C#

アジェンダ

- はじめに
- デザインパターンとは？
- Observer　パターン
- Strategy　パターン
- Singleton　パターン
- まとめ

はじめに

- デザインパターンを知っていますか？
- 知らない人は、興味がありますか？
- 実際に活用しているパターンは何ですか？
- 本セッションでは、デザインパターンを身近なものに置き換えて考えてみます
- 実は僕もあまり得意じゃありませんw
- 今日は皆さんと意見交換をしながら一緒に勉強していきたいと思います！（初心者ですから）

デザインパターンとは？

WEB デザインのパターンのこと？

- 設計上の問題に直面したことはありますか？
- 解決策を考えるために多くの時間を費やしたことはありますか？
- 一度解決できたパターンを再利用することで以後の問題がスムーズに解決した経験はありますか？

デザインパターンとは？

さて、ここで疑問が湧いてきます

- この問題に直面したのは自分が最初なのか？
- 殆どの問題は、既に誰かが解決方法を発見しています。
- そして、それらの解決方法を再利用できるように、デザインパターンとしてまとめれられています。

デザインパターンとは？

- デザインパターンを知ると、.NET Framework を使う上でも、各クラスの設計思想が理解しやすくなります
- 各パターンに名前が付いているため、他者と設計に関する意見を交換する際にも、お互いの誤解が生じにくくなります
- 何より、既存のデザインを流用でき、理想的な問題解決ができます

観察者(オブザーバ) パターン

－ Observer Pattern －

観察者という名前を持つパターンです
実際、何を観察するのでしょうか？

Observer Pattern シナリオ

- あなたは泊りがけの出張に出かけました
- １日目の仕事を無事に終えホテルにチェックインしました
- 翌朝7時に起床しなければなりません

方法1　自分で時計を確認する

取りあえず寝るが、寝がえりをうったり何かのはずみで目覚めた時、自発的に時計を確認する

```
public void 就寝() {
                俺.眠る();
                時計.七時になるまで待つ();
                俺.起きる();
}
class 時計 {
    public static void 七時になるまで待つ() {
        while(現在時刻を取得() < 7) {}
    }
    private static int 現在時刻を取得() {
        return DateTime.Now.Hour;
    }
}
class 俺 {
    public static void 起きる() { Console.WriteLine("起きた！"); }
    public static void 眠る() {   Console.WriteLine("Zzz..."); }
}
```

方法1　自分で時計を確認する

- 安心して眠れない
- 非常に危険をともなう
- 再利用性が低い
- 7時になるまで待つ部分のオブジェクトの分離ができていない

方法2　目覚まし時計で起きる

方法2　目覚まし時計で起きる

```
interface I起床できる {
    void 起床する();
}
class 目覚まし時計 {
        public I起床できる お前 { get; set; }
        public void 開始() {
                時計.七時になるまで待つ();
                お前.起床する();
        }
}
class 目覚まし時計を使う賢い俺 : I起床できる {
        public void 就寝() {
                var 時計 = new 目覚まし時計();
                時計.お前 = (I起床できる)this;
                時計.開始();
                俺.眠る();
        }
        public void 起床する() { 俺.起きる(); }
}
```

時計をセットして寝る。アラームがなるまで寝ることに専念できる。

方法2　目覚まし時計で起きる

- 時間まで安心して眠れる
- 7時になるまでを監視するロジックが完全に分離できている
- 再利用性が高い
- 但し、自分専用で複数人を起こすことができない

方法3　モーニングコールシステム

目覚まし時計の代わりにモーニングコールシステムを使う。モーニングコールは依頼された人だけ起こす。

```
interface I電話を取れる {
        void 電話を取る();
}

interface Iモーニングコール {
        void 依頼(I電話を取れる お前);
        void キャンセル(I電話を取れる お前);
        void 起こす();
}

class モーニングコールを使う都会派の俺 : I電話を取れる {
        public void 就寝() {
                var モーニングコール = new モーニングコール();
                モーニングコール.始動();
                モーニングコール.依頼((I電話を取れる)this);
                俺.眠る();
        }
        public void 電話を取る() { 俺.起きる(); }
}
```

方法3　モーニングコールシステム

```
class モーニングコール : Iモーニングコール {
        private List<I電話を取れる> お前ら;

        public モーニングコール() {
                お前ら = new List<I電話を取れる>();
        }

        public void 依頼(I電話を取れる お前) {  お前ら.Add(お前);  }

        public void キャンセル(I電話を取れる お前) {
                お前ら.Remove(お前);
        }

        public void 起こす() {
                foreach(var お前 in お前ら) お前.電話を取る();
        }

        public void 始動() {
                時計.七時になるまで待つ();
                起こす();
        }
}
```

オブザーバーパターン

```
interface IObserver {
        void Update();
}
interface ISubject {
        void Add(IObserver observer);
        void Remove(IObserver observer);
        void Notify();
}
class Observer : IObserver {
        public Observer() {
                var subject = new Subject();
                subject.Add((IObserver)this);
        }
        public void Update() {}
}
class Subject : ISubject {
        private List<IObserver> observers;
        public Subject() {  observers = new List<IObserver>();  }
        public void Add(IObserver observer) {  observers.Add(observer);  }
        public void Remove(IObserver observer) {  observers.Remove(observer);  }
        public void Notify() {  foreach(var x in observers) x.Update();  }
}
```

オブザーバーパターンまとめ

```
class そんなの関係ねー {
        public void 就寝() {
                var モーニングコール = new モーニングコール2();
                モーニングコール.始動();
                モーニングコール.起こす += (sender, e) => 電話を取る();
                俺.眠る();
        }
        public void 電話を取る() { 俺.起きる(); }
}
class モーニングコール2 {
        public event EventHandler 起こす;
        public void 始動() {
                時計.七時になるまで待つ();
                if (起こす != null) 起こす(this, EventArgs.Empty);
        }
}
```

戦略(ストラテジ) パターン

－ Strategy Pattern －

戦略という名前を持つパターンです
実際、何の戦略を練るのでしょうか？

Strategy Pattern シナリオ

- あなたは、朝目覚めました
- 遅刻せずに出社しなければなりません
- 起床から出社までの行動に関する戦略を練ってみましょう

こんな感じでしょうか？

```
public enum 緊急度 { まったり, ぼちぼち, やばいかも };
class 遅刻しないための戦略その１ {
        public void 行動() {
                var 方法 = 緊急度.やばいかも;
                switch (方法) {
                case 緊急度.まったり:
                        Console.WriteLine("顔を洗う");
                        Console.WriteLine("歯を磨く");
                        Console.WriteLine("トイレに入る");
                        Console.WriteLine("お風呂に入る");
                        Console.WriteLine("着替える");
                        Console.WriteLine("徒歩で移動する");
                        break;
                case 緊急度.ぼちぼち:
                        Console.WriteLine("顔を洗う");
                        Console.WriteLine("歯を磨く");
                        Console.WriteLine("着替える");
                        Console.WriteLine("自転車で移動する");
                        break;
                case 緊急度.やばいかも:
                        Console.WriteLine("タクシーで移動する");
                        break;
                }
        }
}
```

戦略を練れていますか？

- 緊急度ごとの振舞い（戦略）が、オブジェクトとして独立していない
- 戦略の選出が複雑になったらどうするか？
- 戦略の変更に適していない！

遅刻しないための戦略を整理する

```
interface I行動できる {
        void 行動する();
}
class まったり : I行動できる {
        public void 行動する() {
                Console.WriteLine("顔を洗う");
                Console.WriteLine("歯を磨く");
                Console.WriteLine("トイレに入る");
                Console.WriteLine("お風呂に入る");
                Console.WriteLine("着替える");
                Console.WriteLine("徒歩で移動する");
        }
}
class ぼちぼち : I行動できる {
        public void 行動する() {
                Console.WriteLine("顔を洗う");
                Console.WriteLine("歯を磨く");
                Console.WriteLine("着替える");
                Console.WriteLine("自転車で移動する");
        }
}
```

遅刻しないための戦略を整理する

```
class やばいかも : I行動できる {
        public void 行動する() {
                Console.WriteLine("タクシーで移動する");
        }
}
class 遅刻しないための戦略その２ {
        public void 行動() {
                I行動できる 方法 = new やばいかも();
                開始(方法);
        }
        private void 開始(I行動できる 方法) {
                方法.行動する();
                // 戦略に応じてアルゴリズムを差し替える！
        }
}
```

Strategy Pattern

```
interface IStrategy {
        void Method();
}
class StrategyA : IStrategy {
        public void Method() {}
}
class StrategyB : IStrategy {
        public void Method() {}
}
class Strategy {
        public Strategy() {
                IStrategy x = new StrategyA();
                x.Method();
        }
}
```

単一のもの(シングルトン) パターン

- Singleton Pattern -

単一のものという名前を持つパターンです
実際、何が単一であることを保証してくれる
のでしょう？

Singleton Pattern シナリオ

- あなたは、お財布をたくさん持っています
- 必要な時、お財布を購入してしまいます
- お金は、複数のお財布に入っています
- 全財産がわかりません
- 必ず一つのお財布だけでお金を管理するには？

今の状態

```
class 財布をたくさん持つ  {
    public void お金の管理() {
        var 財布1 = new 財布 { お金 = 1000 };
        var 財布2 = new 財布 { お金 = 5000 };
        var 財布3 = new 財布 { お金 = 300 };
        var 財布4 = new 財布 { お金 = 10000 };
        var 全財産 =
            (new[] { 財布1, 財布2, 財布3, 財布4 })
           .Select(x => x.お金).Sum();
        var まとめた財布 = new 財布 { お金 = 全財産 };
}
class 財布 {
         public int お金 { get; set; }
}
```

財布は本当にひとつなのか？

- どこかでまとめても、新しく作って（購入）してお金を入れてしまうかもしれない
- そもそも、まとめる以前のお財布も、見つからないものがあるかもしれない
- 目前にあるお財布が本当にただ一つのものであることを保障して欲しい

唯一の財布であることを保障する

- 世の中に、財布というものが一つしかなければ良い
- 新しく購入することができなければ良い
- 一つだけは最初から存在している

今の状態

```
class 財布を一つにまとめる {
    public void お金の管理() {
        唯一の財布.Instance.お金 = 1000;
        唯一の財布.Instance.お金 += 5000;
        Console.WriteLine(唯一の財布.Instance.お金);
    }
}
class 唯一の財布 {
    private 唯一の財布(){}
    private static 唯一の財布 自分自身 = new 唯一の財布();
    public static 唯一の財布 Instance {
        get {  return 自分自身; }
     }
    public int お金 { get; set; }}
```

Singleton Pattern

```
class Singleton {
        private Singleton() {}
        private static Singleton singleton
                = new Singleton();
        public static Singleton Instance {
                get { return singleton; }
        }
}
```

まとめ

- C# ではイベントでオブザーバーパターンが簡単に実現できる！
- ストラテジーパターンで、アルゴリズムを差し替える
- シングルトンパターンで唯一のオブジェクトを利用する