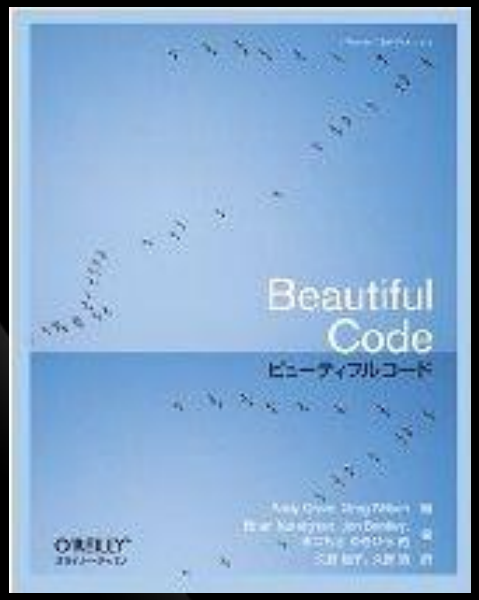
Microsoft®
tech·ed
Japan | 2009

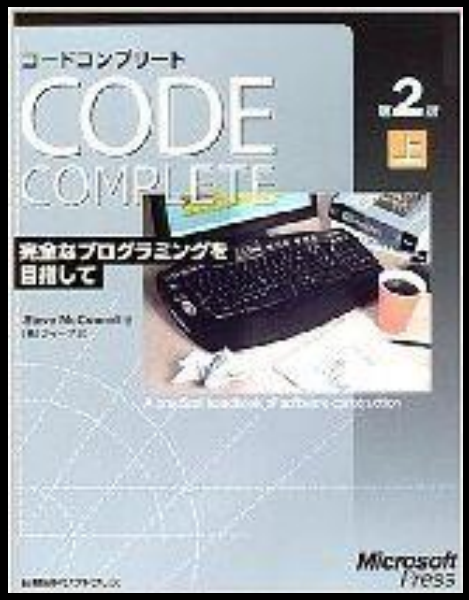
LEARNING LIVES HERE

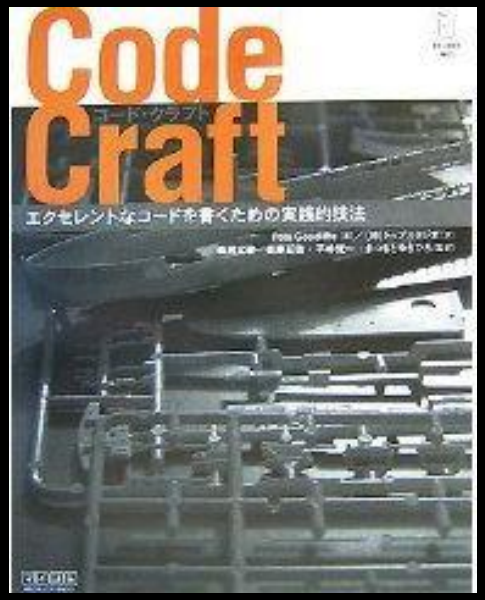
BoF-08
美しいソース コードのための考え方

～マルチパラダイム時代のプログラムの書き方～

Tech·Ed Japan 2009
2009年8月27日(木) 15:15～16:25


Beautiful
Code
ビューティフルコード
O'REILLY


コードコンプリート
CODE
COMPLETE
第2版
上
完全なプログラミングを目指して
Microsoft
Press


Code
Craft
コードクラフト
エクセレントなコードを書くための実践的技法


プログラム書法
第2版
The Elements of Programming Style


実装
パターン
Implementation Patterns


達人プログラマー
The Pragmatic Programmer


アジャイル
ソフトウェア
開発の奥義
第2版
オブジェクト指向開発の神髄と匠の技


ソフトウェア作法
Software
Tools

タイム・テーブル

1. プロローグ
2. 問題提起フェーズ
   - 『美しいソースコードのための考え方 コメント編』 by 原 敬一
   - 『美しいソース コードのための考え方 ～マルチパラダイム時代のプログラムの書き方～』 by 小島 富治雄 (Fujiwo)
   - 『コメントについて』 by 大澤 正
3. ディスカッション 『マルチパラダイム時代のプログラミング』
4. エピローグ

1.プロローグ

こみゅぷらす (COMU+)

コミュニティーの紹介

COMU+

こみゅぷらす

こみゅぷらす (COMU+) は、INETA の加盟コミュニティ
のリーダーや Microsoft MVP などこれまでもアクティブに
情報を発信してきたエンジニアが集まって結成した団体です。
マイクロソフト技術を中心に、さまざまな情報をオンライン、
オフライン勉強会を通じて配信しています。

http://comuplus.net

所属

Culminis 所属

Developers Summit 2007-2008 オフィシャル コミュニティ

お問い合わせ: *contacts@comuplus.net*

2006/12/01 こみゅぷらす発足
2006/12/19 勉強会
2007/01/31 勉強会
2007/02/13 勉強会
2007/02/14 Developers Summit 2007 コミュニティライブ
2007/03/27 勉強会
2007/04/24 勉強会
2007/05/19 勉強会
2007/05/19 Community Launch (渋谷・ピンクカウ)
2007/06/19 勉強会
2007/07/27 勉強会
2007/08/08 勉強会
2007/08 Tech・Ed BoF & Usergroup Street Live!
2007/09/08 Tech・Ed 報告会
2007/09/29 勉強会
2007/09/29 コミュニティ勉強会 第一回東京編
2007/10/18 勉強会
2007/11/22 勉強会
2007/12/21 勉強会
2008/01/25 勉強会

http://comuplus.net

Tech·Ed 2008 Yokohama

- BoF12
  『プログラミング! プログラミング! プログラミング!
  .NET 3.5 時代のコーディング
  ～ これからの実装技術について考えよう ～』
  http://www.event-marketing.jp/events/te08/special/bof/bof_12.htm

自己紹介

- 小島 富治雄 (Fujiwo)
- 原 敬一
- 大澤 正
- 亀川 和史 (めさいあ)

本 BoF は、

「きれいなソースコード」

がテーマですが、

好きな言語は

なんですか?

好きな言語アンケート

- C#/Visual Basic
- Java
- C++
- 関数型言語
- LL (Light Weight Language)
- その他

本日のテーマ

本日のテーマ

- 美しいソースコード
  - 美しいソースコードとは?
  - マルチパラダイム時代のソースコードの書き方

M言語や並列プログラミング・クラウド
・Visual Studio 2010などの
新たなマイクロソフトの技術に向け、
プログラマーの原点である
プログラミング技術に関して、
熱く語り合っていきたいと思います。

ありがとうございます!

- 本日はご参加いただき、
  本当にありがとうございます。
- お楽しみください。
- どうぞディスカッションに
  ご参加ください。
  色々な意見が聞けることを
  楽しみにしております。

2.問題提起フェーズ

LEARNING LIVES HERE

Developers Summit 2009

「美しいソースコードのための考え方」コメント編

原 敬一 (こみゅぷらす/codeseek)

今回は「コメントは不要か」に注目

- コメントは不要ですか？
  - コメントの必要性
  - しばしば議論される「美しいコードとは？」とも絡む永遠のテーマ。

不要！

- 不要という視点

コメントは不要という視点から

- コメントは書かないほうがいい理由
  - 入力と読むのに時間がかかる
  - 間違いが書かれていることがある
  - コメントかっこ悪い

入力と読むのに時間がかかる

- 入力には時間がかかる
- 読む量は多くなるが、把握する時間は短くなる
- トータルではコメントがない方が時間がかかる

間違いが書かれていることがある

- コメントの間違いは内部仕様書のバグと言える

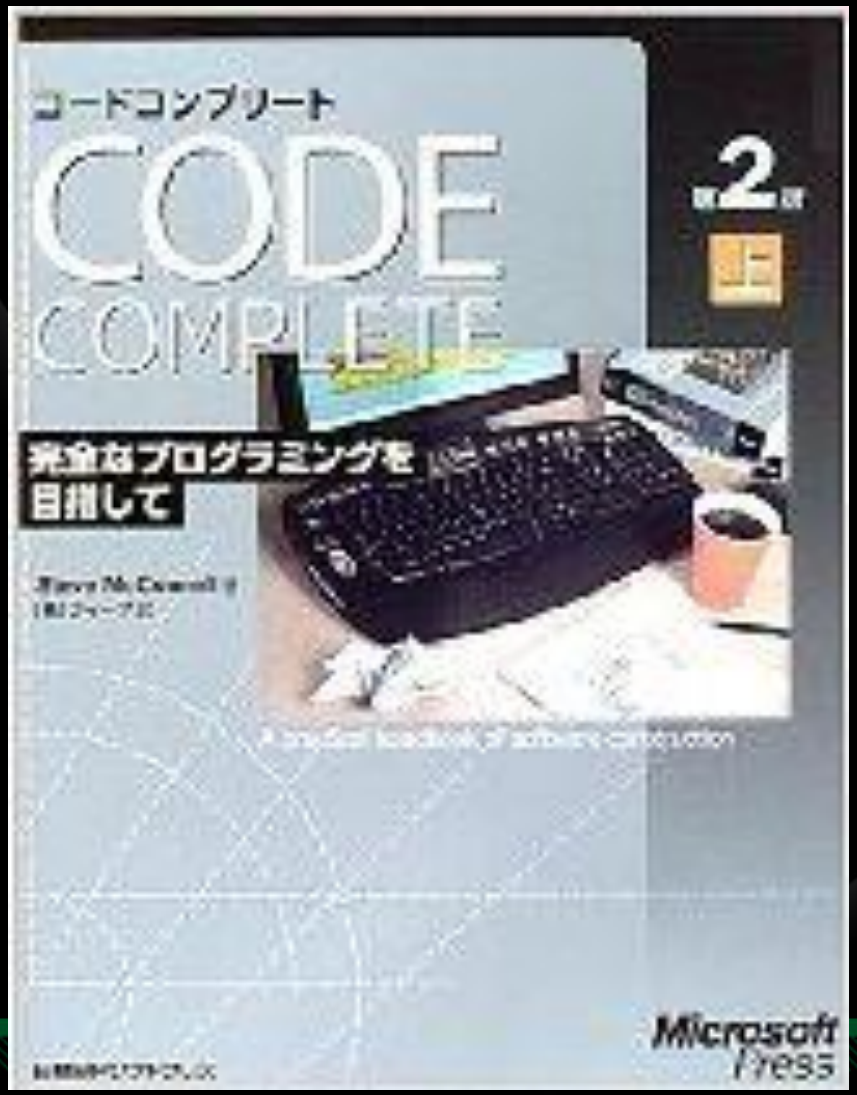
このバグがあっても動作してしまうという問題

- このバグを発見してくれるコンパイラはない
- このバグを発見するテストはコードレビュー　->効率が悪い

コメントかっこ悪い

- コメントなんてダサい、というおれかっこいい
- 本当は、「コメントなどなくともよいコードを書くのがかっこいい」

コードコンプリート


コードコンプリート
CODE COMPLETE
第2版
上
完全なプログラミングを目指して
Microsoft Press

コードコンプリート
第二版 上 384頁
第32章 第3節

「コメントを入れるか入れないか」

カリクレスの台詞より:

「プログラミング言語の文は短くて的を射ている。」

「コードを明確にできない者が、どうしてコメントを明確にできるんだい。」

「それに、コードを修正すればコメントは古くなる。期限切れのコメントを信じたら救いようがないぞ。」

トラシュマコスの台詞より:

(「コメントは無駄」というカリクラスの言を受けて)

「ちょっと待ってください。良いコメントは、コードを繰り返したり説明したりしませんよ。」

「何をしようとしているのかを、コードよりも抽象的なレベルで説明するのが、コメントなんです。」

コメントの種類

- コードの繰り返し
  - コードを別の言葉で言い換えただけ
- コードの説明
  - 複雑・トリッキーなコードを説明
- コードの目印
  - 作業の状況

コメントの種類

- コードの概要
  - コードの要約
- コードの意図の説明
  - コードの目的
- コード自体では表せない情報
  - 設計上の注意点、バージョン等

コメントありすぎ、なさすぎ

- コメントが必要なところに無い
- コメントが不要なところにある

- コメントがあまりにも多すぎる

→コメント不要論？

汎用言語だから

- 汎用言語にコメントなしは考えられない
- 要件とコードのインピーダンス不整合

- DSLメタ言語　M言語

ディスカッション

- コメントが無いのがかっこいいのではなく、
- コメントなどなくともよいコードを書くのがかっこいいのです。
- コメント書いてくださいね。

LIVES HERE

美しいソース コードのための考え方
～マルチパラダイム時代のプログラムの書き方～

小島 富治雄 (Fujiwo)

今日私が
お伝えしたいこと

今日私がお伝えしたいこと

プログラミング
という行為は
モデリング。

背景

背景

- マルチパラダイム プログラミングが現実的に。
  - C#、Visual Basic など言語の進化
  - Visual Studio などツールの進化

マルチパラダイム プログラミング

- 手続き型プログラミング
  ⇔ オブジェクト指向型プログラミング
  ⇔ 関数型プログラミング
- 命令型プログラミング
  ⇔ 宣言型プログラミング
- テキスト型プログラミング
  ⇔ 図解型プログラミング
- ジェネリック・プログラミング
- 並列プログラミング

などの組み合わせ

C# や Visual Basic は、
最新のものほど、
マルチパラダイム
プログラミング言語化

Q.パラダイムによって
ソースコードは
大きく変わる?

例. 命令型・手続き型

```
using System;

enum 性 { 男, 女 }

struct 日付
{
    public int 年, 月, 日;
}

struct モニター
{
    public string 氏名    ;
    public 日付   生年月日;
    public 性     性別    ;
}

class プログラム
{
    static 日付 日付の作成(int 年, int 月, int 日)
    {
        日付 日付;
        日付.年 = 年;
        日付.月 = 月;
        日付.日 = 日;
        return 日付;
    }

    static void モニターの初期化(out モニター モニター, string 氏名, 性 性別, int 生年,
       int 生月, int 生日)
    {
        モニター.氏名     = 氏名                          ;
        モニター.生年月日 = 日付の作成(生年, 生月, 生日);
        モニター.性別     = 性別                          ;
    }

    static モニター[] 集計データの作成()
    {
        モニター[] 集計データ = new モニター[6];
        モニターの初期化(out 集計データ[0], "宇野宗佑", 性.男, 1922, 8, 27);
        モニターの初期化(out 集計データ[1], "山岡久乃", 性.女, 1926, 8, 27);
        モニターの初期化(out 集計データ[2], "田中星児", 性.男, 1947, 8, 27);
        モニターの初期化(out 集計データ[3], "渡部絵美", 性.女, 1959, 8, 27);
        モニターの初期化(out 集計データ[4], "渡辺謙"  , 性.男, 1969, 8, 27);
        モニターの初期化(out 集計データ[5], "手島優"  , 性.女, 1984, 8, 27);
        return 集計データ;
    }

    static int 年齢の算出(日付 生年月日, 日付 今日)
    {
        int 年齢;
        if (今日.月 < 生年月日.月 || 今日.月 == 生年月日.月 && 今日.日 < 生年月日.日)
            年齢 = 今日.年 - 生年月日.年 - 1;
        else
            年齢 = 今日.年 - 生年月日.年    ;
        return 年齢;
    }

    static 日付 今日の日付()
    {
        DateTime now  = DateTime.Now;
        日付     今日 = 日付の作成(now.Year, now.Month, now.Day);
        return 今日;
    }

    static int 年齢の算出(日付 生年月日)
    { return 年齢の算出(生年月日, 今日の日付()); }

    static bool 範囲内かどうか(int 数, int 最低, int 最高)
    {
        if (数 >= 最低 && 数 <= 最高)
            return true ;
        else
            return false;
    }
```

例. 命令型・手続き指向型

```
static bool 条件に一致(モニター モニター)
{
    if (モニター.性別 == 性.女) {
        if (範囲内かどうか(
              年齢の算出(モニター.生年月日), 20, 49))
            return true;
    }
    return false;
}

static void 表示(モニター モニター)
{
    Console.WriteLine("[ 氏名: {0}, 性別: {1}, 生年月
日: {2:D} ]", モニター.氏名, モニター.性別, モニター.
生年月日);
}
```

```
static void 集計データの表示(モニター[] 集計デー
タ)
{
    foreach (モニター モニター in 集計データ) {
        if (条件に一致(モニター))
            表示(モニター);
    }
}

static void Main()
{
    モニター[] 集計データ =
                        集計データの作成();
    集計データの表示(集計データ);
}
}
```

例. 命令型・オブジェクト指向型

```
using System;
using System.Collections;
using System.Collections.Generic;

static class ユーティリティ
{
    public static bool 範囲内かどうか(int 数, int 最低, int 最高)
    { return 数 >= 最低 && 数 <= 最高; }

    public static int 年齢(DateTime 生年月日, DateTime 今日)
    {
        if (今日.Month < 生年月日.Month ||
                        今日.Month == 生年月日.Month && 今日.Day < 生年月日.Day)
            return 今日.Year - 生年月日.Year - 1;
        else
            return 今日.Year - 生年月日.Year;
    }

    public static int 年齢(DateTime 生年月日)
    { return 年齢(生年月日, DateTime.Today); }
}

enum 性 { 男, 女 }

class モニター
{
    public string   氏名     { get; set; }
    public DateTime 生年月日 { get; set; }
    public 性       性別     { get; set; }

    public int 年齢
    {
        get
        { return ユーティリティ.年齢(生年月日); }
    }
}
```

```
class 絞り込みオプション
{
    public 性  性別     { get; set; }
    public int 最低年齢 { get; set; }
    public int 最高年齢 { get; set; }

    public bool マッチするかどうか(モニター モニター)
    {
        if (モニター.性別 != 性別)
            return false;
        if (!ユーティリティ.範囲内かどうか(モニター.年齢, 最低年齢, 最高年齢))
            return false;
        return true;
    }
}

class アンケートデータ
{
    List<モニター> data = new List<モニター>();

    public アンケートデータ()
    {
        data.Add(new モニター() { 氏名 = "宇野宗佑", 性別 = 性.男,
                                  生年月日 = new DateTime(1922, 8, 27) });
        data.Add(new モニター() { 氏名 = "山岡久乃", 性別 = 性.女,
                                  生年月日 = new DateTime(1926, 8, 27) });
        data.Add(new モニター() { 氏名 = "田中星児", 性別 = 性.男,
                                  生年月日 = new DateTime(1947, 8, 27) });
        data.Add(new モニター() { 氏名 = "渡部絵美", 性別 = 性.女,
                                  生年月日 = new DateTime(1959, 8, 27) });
        data.Add(new モニター() { 氏名 = "渡辺鐘"  , 性別 = 性.男,
                                  生年月日 = new DateTime(1969, 8, 27) });
        data.Add(new モニター() { 氏名 = "手島優"  , 性別 = 性.女,
                                  生年月日 = new DateTime(1984, 8, 27) });
    }

    public IEnumerator<モニター> GetEnumerator()
    { return data.GetEnumerator(); }
}
```

例. 命令型・オブジェクト指向型

```
abstract class ビュー
{
    public 絞り込みオプション 表示オプション { get; set; }
    public アンケートデータ    データ         { get; set; }

    public void 表示()
    {
        foreach (var アイテム in データ) {
            if (表示オプション.マッチするかどうか(アイテム))
                表示(アイテム);
        }
    }

    protected abstract void 表示(モニター モニター);
}

class コンソール画面 : ビュー
{
    protected override void 表示(モニター モニター)
    { Console.WriteLine("[ 氏名: {0}, 性別: {1}, 生年月日: {2:D} ]", モニター.氏名, モニター.性別, モニター.生年月日); }
}
```

```
class プログラム
{
    static アンケートデータ 集計データ
                        = new アンケートデータ();
    static ビュー           画面
                        = new コンソール画面   ();

    static void Main()
    {
        画面.データ = 集計データ;
        画面.表示オプション
            = new 絞り込みオプション() {
                性別 = 性.女, 最低年齢 = 20,
                最高年齢 = 49 };
        画面.表示();
    }
}
```

例. 宣言型・関数型

```
using System;
using System.Collections.Generic;
using System.Linq;

static class プログラム
{
    enum 性 { 男, 女 }

    static void の各々について<T>(this IEnumerable<T> コレクション, Action<T> アクション)
    {
        foreach (var アイテム in コレクション)
            アクション(アイテム);
    }

    static IEnumerable<T> を絞り込み<T>(this IEnumerable<T> コレクション, Func<T, bool> マッチするかどうか)
    {
        foreach (var アイテム in コレクション) {
            if (マッチするかどうか(アイテム))
                yield return アイテム;
        }
    }
```

```
    static int 年齢(DateTime 生年月日, DateTime 今日)
    {
        return (今日.Month < 生年月日.Month ||
           今日.Month == 生年月日.Month && 今日.Day < 生年月日.Day)
               ? 今日.Year - 生年月日.Year - 1
               : 今日.Year - 生年月日.Year    ;
    }

    static int 年齢(DateTime 生年月日)
    { return 年齢(生年月日, DateTime.Today); }
    static bool 範囲内かどうか(int 数, int 最低, int 最高)
    { return 数 >= 最低 && 数 <= 最高; }

    static void 表示(dynamic モニター)
    { Console.WriteLine("[ 氏名: {0}, 性別: {1}, 生年月日: {2:D} ]",
      モニター.氏名, モニター.性別, モニター.生年月日); }

    static IEnumerable<dynamic> 集計データ()
    {
        yield return new { 氏名 = "宇野宗佑", 性別 = 性.男,
                           生年月日 = new DateTime(1922, 8, 27) };
        yield return new { 氏名 = "山岡久乃", 性別 = 性.女,
                           生年月日 = new DateTime(1926, 8, 27) };
        yield return new { 氏名 = "田中星児", 性別 = 性.男,
                           生年月日 = new DateTime(1947, 8, 27) };
        yield return new { 氏名 = "渡部絵美", 性別 = 性.女,
                           生年月日 = new DateTime(1959, 8, 27) };
        yield return new { 氏名 = "渡辺鐘"  , 性別 = 性.男,
                           生年月日 = new DateTime(1969, 8, 27) };
        yield return new { 氏名 = "手島優"  , 性別 = 性.女,
                           生年月日 = new DateTime(1984, 8, 27) };
    }
```

例. 宣言型・関数型

```
    static void Main()
    {
        集計データ().を絞り込み(モニター => モニター.性別 == 性.女)
                    .を絞り込み(
                        モニター =>
                        範囲内かどうか(年齢(モニター.生年月日), 20, 49)
                    )
                    .の各々について(表示);
    }
}
```

まったく同じ
実行結果だが…

美しさが異なる

- 保守性が異なる
  - 変更容易性
  - テスト容易性

別の例

アセンブリ言語

```
mov   dx, msg
mov   ah, 0x09
int   0x21

xor   al, al
mov   ah, 0x4C
int   0x21

msg db "Hello World!$"
```

を、なんで

C#

```
class Program
{
    static void  Main()
    {
        System.Console.WriteLine("Hello World!");
    }
}
```

F#

```
printfn "Hello world!"
```

のように書く
ようになってきたのか?

「動けばいい」のなら、
どっちでも良いはず。

また別の例

なんで
説明するときに
図を使うことが
あるのか?

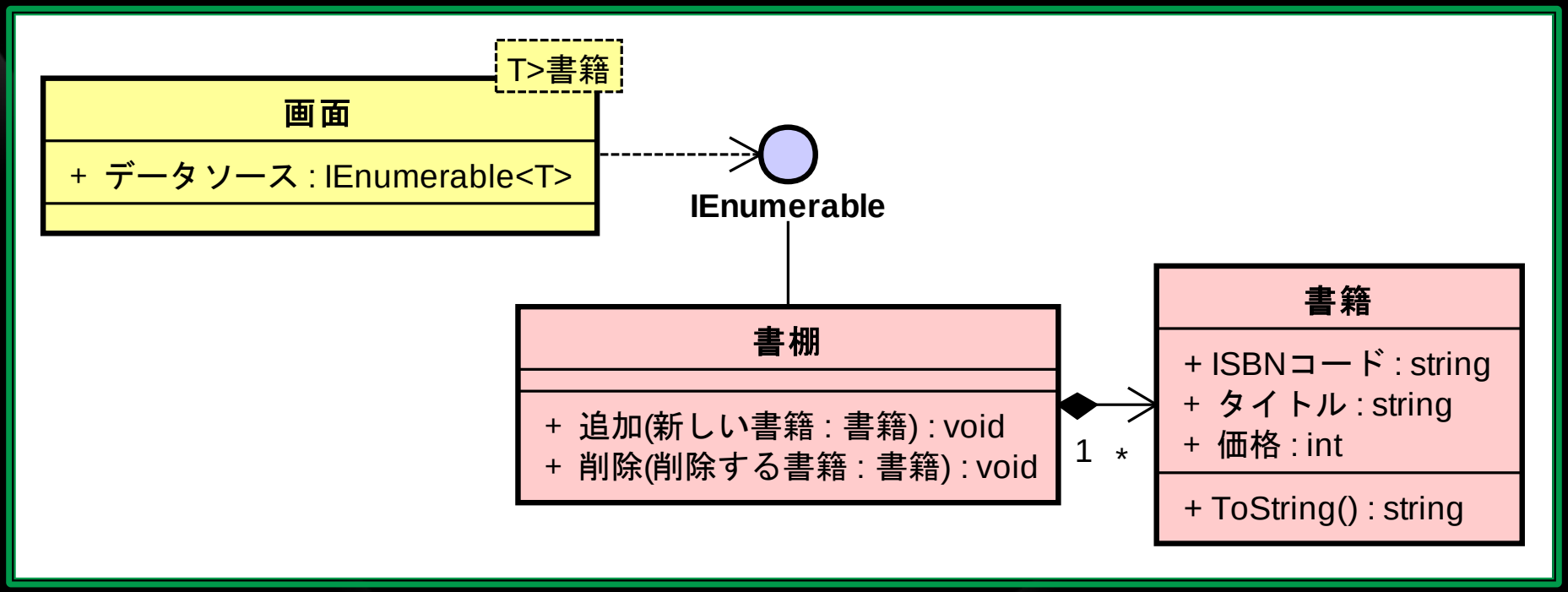
C#

```
class 書籍
{
    public string ISBNコード { get; set; }
    public string タイトル   { get; set; }
    public int    価格       { get; set; }

    public override string ToString()
    { return string.Format("ISBNコード: {0}, タイトル:
     {1}, 価格: {}", ISBNコード, タイトル, 価格); }
}

class 書棚 : IEnumerable<書籍>
{
    List<書籍> 書籍リスト = new List<書籍>();

    public void 追加(書籍 新たな書籍)
    { 書籍リスト.Add(新たな書籍); }

    public void 削除(書籍 削除する書籍)
    { 書籍リスト.Remove(削除する書籍); }

    public IEnumerator<書籍> GetEnumerator()
    { return 書籍リスト.GetEnumerator(); }

    IEnumerator IEnumerable.GetEnumerator()
    { return 書籍リスト.GetEnumerator(); }
}
```

```
class 画面<T>
{
    public IEnumerable<T> データソース
    {
        set
        {
            foreach (var アイテム in value)
                Console.WriteLine(アイテム);
        }
    }
}
```

ع

クラス図


T>書籍
画面
+ データソース : IEnumerable<T>
IEnumerable
書棚
+ 追加(新しい書籍 : 書籍) : void
+ 削除(削除する書籍 : 書籍) : void
1
*
書籍
+ ISBNコード : string
+ タイトル : string
+ 価格 : int
+ ToString() : string

の違いは何?

何の違いが
ソースコードの違いを
生むのか?

ソースコードが異なる

↑

プログラマーが意図したモデルが異なる。

モデル
(＝関心のある/コミュニケーションしたい部分を抽出したもの)
の記述に
より近いものが使われる。

アセンブリ言語

```
mov   dx, msg
mov   ah, 0x09
int   0x21

xor   al, al
mov   ah, 0x4C
int   0x21

msg db "Hello World!$"
```

より

C#

```
class Program
{
    static void  Main()
    {
        System.Console.WriteLine("Hello World!");
    }
}
```

F#

```
printfn "Hello world!"
```

の方が、

(この場合、) プログラマにとって、

「関心のある/コミュニケーション
したい部分を抽出」 (=モデル)

した記述になっている。

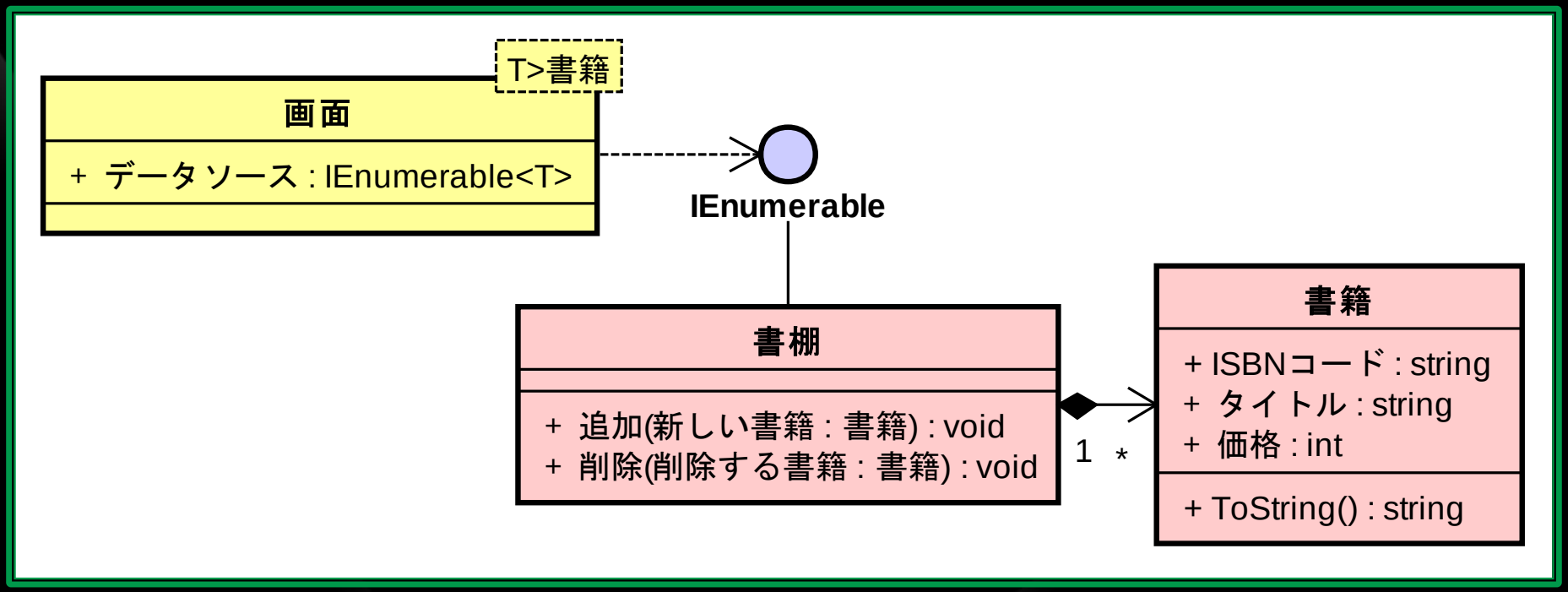
C#

```
class 書籍
{
    public string ISBNコード { get; set; }
    public string タイトル   { get; set; }
    public int    価格       { get; set; }

    public override string ToString()
    { return string.Format("ISBNコード: {0}, タイトル:
     {1}, 価格: {}", ISBNコード, タイトル, 価格); }
}

class 書棚 : IEnumerable<書籍>
{
    List<書籍> 書籍リスト = new List<書籍>();

    public void 追加(書籍 新たな書籍)
    { 書籍リスト.Add(新たな書籍); }

    public void 削除(書籍 削除する書籍)
    { 書籍リスト.Remove(削除する書籍); }

    public IEnumerator<書籍> GetEnumerator()
    { return 書籍リスト.GetEnumerator(); }

    IEnumerator IEnumerable.GetEnumerator()
    { return 書籍リスト.GetEnumerator(); }
}
```

```
class 画面<T>
{
    public IEnumerable<T> データソース
    {
        set
        {
            foreach (var アイテム in value)
                Console.WriteLine(アイテム);
        }
    }
}
```

より

クラス図


T>書籍
画面
+ データソース : IEnumerable<T>
IEnumerable
書棚
+ 追加(新しい書籍 : 書籍) : void
+ 削除(削除する書籍 : 書籍) : void
1
*
書籍
+ ISBNコード : string
+ タイトル : string
+ 価格 : int
+ ToString() : string

の方が、

(この場合、) プログラマにとって、

「関心のある/コミュニケーション
したい部分を抽出」 (=モデル)

した記述になっている。

私の認識

プログラミング
というのは、実装のみを行う
のではない。

プログラミング
は、
「設計＋実装＋テスト」。

プログラミング
というのは、
「検証可能な
設計/実装モデル」
を作ること。

プログラミング
という行為は
モデリング。

モデリングなので、

モデリングのための
言語やツールが重要。

プログラミング言語が重要

- モデルを書くのに適した言語
  - 進化した C#、Visual Basic など
  - DSL (ドメイン特化言語)

「プログラミングで作られるモデル」

=ソースコード

(が理想)

美しいソースコード

＝美しい「検証可能な設計/実装モデル」

モデルとは

- そのコンテキストでの関心事を抽出したもの。
- モデルの目的:
  - 関心事に限定したコミュニケーション。

誰とのコミュニケーション?

ソースコードで誰とコミュニケーションするのか?

- 古くは:
  - コンピュータとコミュニケーション
- 時代が進んで:
  - コンパイラとコミュニケーション
- 今は:
  - 人とコミュニケーション
  - 人=ステークホルダー

プログラミングでは:

- Q. 何をモデリングするのか?
  ＝何を関心事として抽出するか?
- A. 意図をモデリング。

意図がソースコード (＝モデル) で表現されるべき。

意図を
モデリング

例

例えば…

C# では、

「従業員名簿内の 全ての各従業員を
画面に出力する」

のソースコードは、

```
従業員名簿.ForAll(各従業員 => 各従業員.出力(画面));
```

と書かれたりする。

宣言型プログラミング

なんで

```
従業員名簿.ForAll(各従業員 => 各従業員.出力(画面));
```

このソースコードが、

```
for (int i = 0; i < 従業員名簿.Count; i++)
    出力(従業員名簿[i]);
```

より良いか?

命令型プログラミング

「この場合は」、

「整数 i を 0 にし、i が 従業員名簿の Count までの間、i をインクリメントしながら、従業員名簿の i 番目を出力」

という意図のモデルじゃなく、

「従業員名簿内の全ての各従業員を
画面に出力する」

のモデルだから。

つまり、

「従業員名簿内の全ての各従業員を
画面に出力する」

のモデルとしては、

```
従業員名簿.ForAll(各従業員 => 各従業員.出力(画面));
```

の方が (ベストではないが) ベター。

もし仮に、

「整数 i を 0 にし、i が 従業員名簿の Count までの間、
　i をインクリメントしながら、従業員名簿の i 番目を出力」

という意図のモデルだったら、

```
for (int i = 0; i < 従業員名簿.Count; i++)
    出力(従業員名簿[i]);
```

の方が、

```
従業員名簿.ForAll(各従業員 => 各従業員.出力(画面));
```

よりベター。

別の例

なんで

```
従業員名簿.ForAll(各従業員 => 各従業員.出力(画面));
```

このソースコードが、

```
l.ForAll(x => x.Func(s));
```

より良いか?

「この場合は」、

「I 内の全ての各 x を s を使って Func する」

という意図のモデルじゃなく、

「従業員名簿内の全ての各従業員を
画面に出力する」

のモデルだから。

もし仮に、

「l 内の全ての各 x を s を使って Func する」

という意図のモデルだったら、

```
l.ForAll(x => x.Func(s));
```

の方が、

```
従業員名簿.ForAll(各従業員 => 各従業員.出力(画面));
```

よりベター。

別の例

なんで

```
従業員名簿.ForAll(各従業員 => 各従業員.出力(画面));
```

このソースコードが、

```
employeeList.ForAll(employee => employee.Output(screen));
```

より良いか?

「この場合は」、

「employee 内の全ての各 employee を screen に Output する」

という意図のモデルじゃなく、

「従業員名簿内の全ての各従業員を画面に出力する」

のモデルだから。

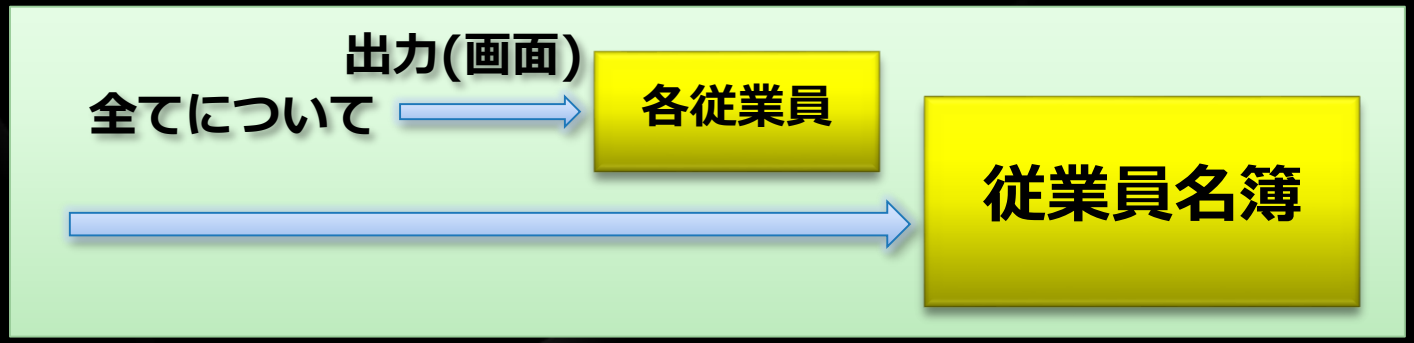
でも、

「従業員名簿内の全ての各従業員を画面に出力する」

という意図のモデルだったら、

```
従業員名簿.ForAll(各従業員 => 各従業員.出力(画面));
```

でなく、例えば、



のように書かれても良いはず。

図解型プログラミング

寧ろ、

C# のような汎用言語より、
DSL (ドメイン特化言語) の方が、
よりピュアにモデルを書くことが
できる可能性がある。

モデルがピュアである
ことが重要か?

そう。
モデルはピュアであることが重要。

なぜならば、

モデルとは

「関心事を抽出したもの」

だから。

「意図をモデリングした」
ものがソースコード。

意図以外のものが
書かれていないのが
ベター

かつ意図が
書き尽く
されている

例

例えば…

「従業員名簿内の 全ての各従業員を
画面に出力する」

が意図なら、

```
for (int i = 0; i < 従業員リスト.Count; i++)
        出力(従業員リスト[i]);
```

の場合、

「int」、「i」、「=」、「<」、「Count」、「i++」、「[i]」

は「意図以外」のもの。
(＝モデルにとっては「ノイズ」)

「この場合は」、

```
for (int i = 0; i < 従業員リスト.Count; i++)
        出力(従業員リスト[i]);
```

よりは、

命令型プログラミング

```
従業員リスト.ForAll(各従業員 => 各従業員.出力(画面));
```

の方がノイズが少ない。
(=SN比が高い)

宣言型プログラミング

例

例えば…

「或るコレクションに対して或る条件で絞り込みを行う」というアルゴリズムのモデルとしてメソッドを書く場合、

```
public static ArrayList 絞り込み(int[] collection)
{
    ArrayList resultList = new ArrayList();
      foreach (int item in collection) {
          if (item % 5 == 0)
              resultList.Add(item);
    }
    return resultList;
}
```

では、意図通りのプログラムになっていない。

「或るコレクションに対して或る条件で絞り込みを行う」
というアルゴリズムのモデルとしてメソッドを書く場合、

```
public static IEnumerable<T> 絞り込み<T>(
    IEnumerable<T> collection,
    Func<T, bool>    絞り込み条件)
{
    foreach (var item in collection) {
        if (絞り込み条件(item))
            yield return item;
    }
}
```

の方が、意図通りのプログラム。

ジェネリック・プログラミング

美しいソースコードを書くコツ

美しいソースコードを書くコツ:

単一責務の法則

ひとつのプログラム
(クラス、オブジェクト、メソッド、変数など)
には、
ひとつの関心事のみを書く。

かつ
「書き尽くす」

「プログラム＝モデル
＝或る関心事を抽出したもの」
だから。

その コンテキストでの
関心事以外を混ぜない

```
public static void 絞り込み(int[] collection)
{
    foreach (int item in collection) {
        if (item % 5 == 0)
            Console.WriteLine(item);
    }
}
```

※ 赤字は関心事以外。

状況に適したパラダイムの採用

- 手続き型プログラミング
  ⇔ オブジェクト指向型プログラミング
  ⇔ 関数型プログラミング
- 命令型プログラミング
  ⇔ 宣言型プログラミング
- テキスト型プログラミング
  ⇔ 図解型プログラミング
- ジェネリック・プログラミング
- 並列プログラミング

などの組み合わせ

美しいソースコードを書くコツ:
名前付け重要

名前付け重要。

「名前を付ける」行為が
モデリングの中心。

※ 名前付けはモデリング全体の73% を占める (主観による概算)

「名前を付ける」のは、

「概念の意味を特定し、概念の範囲（それが何であるか、何でないか）」を決める行為。

「名前を付ける」ときに

「関心事」が抽出される。

例えば、

プログラムの部品
(クラス、オブジェクト、メソッド、変数など)
には、

「それの唯一の関心
(それが何をするためのものか)
を一言で言ったもの」
が名前として付く。

それの単一責務＝それの名前

例.

```
public static IEnumerable<T> 名簿Func<T>(
    IEnumerable<T>    名簿,
    Func<T, bool>     デリゲート)
{
    foreach (T x in 名簿) {
        if (デリゲート(x))
            yield return x;
    }
}
```

```
public static IEnumerable<T> 絞り込み<T>(
    IEnumerable<T>    collection,
    Func<T, bool>     絞り込み条件)
{
    foreach (var item in collection) {
        if (絞り込み条件(item))
            yield return item;
    }
}
```

「このクラス（or オブジェクト, メソッド, 変数...）の、仕事を『一言』でいうと何?」

に即答できるかを注意。

「一言では言えない」

＝ **「単一責務になってない」。**

(とすると)

美しいソースコードを書くコツ:

説明責任

「 ソースコードについて
書いたプログラマが説明できるか? 」
に注意。

例.

「このメソッドは、何故このクラスに在る?」
「この変数は、どうしてこの名前?」
「このメソッドの、仕事をひとことでいうと何?」

「うまく説明できない」

＝
(とすると)

「意図（モデル）がない」
または、
「意図（モデル）通りに書いていない」。

美しいソースコードを書くコツ:

説明責任

先ずは、

「 ソースコードについて
書いたプログラマが説明できるか? 」

次に、

「 ソースコードについて
他のプログラマが説明できるか? 」

FAQ

Q. 美しいソースコードと言っても、ひとによって異なったコードになるのでは?

A. もちろん。
視点が異なれば、異なったモデルになる。

Q.コメントは不要?

A. 意図したモデルが、そのプログラミング言語で「書き尽くせる」なら不要。
「書き尽くせない」分があれば、
「必要悪」として書くべき。

例. “How” や “What” はプログラミング言語で記述しやすいが、”Why” は記述し切れないことが多い。

Q.ソースコードの美しさよりも
動くかどうかの方が重要では?

Q. ソースコードの美しさよりも動くかどうかの方が重要では?

A.

論点がずれている。

「美しいソースコード」と「動くプログラム」を比較する理由がない。

「美しいソースコード」と「汚いソースコード」の比較は、
「保守性が高い動くプログラムを速く書くこと」
と
「保守性が低い動くプログラムを遅く書くこと」
の比較。

Q.そうは言っても、実務では美しいソースコードより他のことが重視されることも有るように思う。

例えば、チーム開発を円滑に進めるために、オブジェクト指向的に美しくない分割をするべきこともあるのでは?

Q.美しくないモデルを抽出すべきときもあるのでは?

- A.プログラムは関心事に対して書かれている (＝モデル) べき。
- ホットスポットが最大の関心事。
- 関心を分離したものがモデル。
- 「チーム開発がホットスポット」なら、それが最大の関心事。
- それに合うように分割することが「美しい分割」。

- ※ 最大の関心事で分離しているのに「美しくないモデル」と考える方が不思議。

ご清聴ありがとう
ございました。

LEARNING
LIVES
HERE

こみゆぷらすBoF マルチパラダイム時代の美しいコード

亀川 和史 (めさいあ)

まず去年の振り返り

- BoF12『プログラミング! プログラミング! プログラミング! .NET 3.5 時代のコーディング ～これからの実装技術について考えよう～』

原水さんのBlogから借用
http://blogs.technet.com/shinhara/archive/2008/08/29/teched-2008-bof-12-report.aspx

最近よく聞くもの

- 関数型言語
- メタプログラミング
- HTML5(ま、一応…)
- JavaScript

でも…

- Visual BASIC 6.0で新規開発をするという人たちもいる
  - 慣れているから
  - 新しいものを覚えるコストがかかる
  - 「何ができるかに価値がある」から問題ない
- http://el.jibun.atmarkit.co.jp/g1sys/2009/06/vb6-c373.html
  ベンチャー社長で技術者で(VB6を使い続けること)

Windows 7時代で予想される質問

- Windows 7のXP ModeがあるからVB 6.0もサポート続くんだよね？(続きません)
- XP Modeの公開アプリケーションと、Windows 7でちょっと作ったマネージドコードを連携して動かしたい
  - どうやったら共有メモリやり取りできますか？
  - XP ModeのXPと通信する方法は？

さらに…

- 遠い昔のCOBOLソースを今のオープンシステム(死語)にマイグレーションしたい
- 元のソースは一行たりとも変えたくない
  - 作り直したらバグが入る
  - 再評価しなきゃならないじゃないか
  - 性能の問題

こんな感じ？

業務システム

新規

スタブ

レガシー
移行

作るはず…

役割いろいろ

- レガシーの保守
  伝統的なプログラムを保守？このまま？
- スタブを作る人
  和洋…新古折衷？
- 新規で作る人
  新しいものを導入するだけなので、他のことを考えられなくなる？

これもマルチパラダイム？

- COBOL,VB6.0,.NET Framework(C#/VB.NET/F#...)
- これに将来クラウドサービス(Azureなど)が入ると…？

どうやってレガシー時代のシステムと付き合い、美しいプログラミングをするか？

3.ディスカッション

ディスカッション

『マルチパラダイム時代のプログラミング』

- きれいなソースコードって?
- これからの実装はどう変わる?
- 期待する実装技術 (M言語など)

きれいなソースコードって?

- 変更が容易
- 分かりやすい
- テスト (検証) が容易
- … …

4.エピローグ

Microsoft®

Your potential. Our passion.™