C#コーディング標準

初版 2002 年 9 月 25 日

オリジナル：http://ObjectClub.esm.co.jp/eXtremeProgramming/CodingStd.doc
http://ObjectClub.esm.co.jp/eXtremeProgramming/CodingStdVB.doc

このドキュメントは Java コーディング標準(オブジェクト倶楽部バージョン)、VB.NET コーディング標準を C# 用に変更したもので、フリーかつ AS-IS ベースで提供しています。

コピー、修正、配布してかまいません。みなさんのプロジェクトでこれをカスタマイズして使用することを歓迎します。

フィードバックを歓迎します。ご意見などを以下のアドレスへ頂けると嬉しく思います。

フィードバック先：
extremeprogramming-jp@ObjectClub.esm.co.jp
bata@gold.ocn.ne.jp

1. 方針

このコーディング標準は、ソフトウェア開発プロジェクトにおいて C#でコーディングする際のルール、推奨、および迷った時の指針を提供するものである。

標準策定の方針は、読みやすくメンテナンスしやすいコードを書くことである。実際のコーディングにあたっては、プロジェクトメンバー全員がこのルールに合意していることが必要である。

実プロジェクトにおいては、このコーディング標準をカスタマイズして用いることを推奨する。

また、あわせて .NET Framework SDK ヘルプ「クラス ライブラリ開発者向けのデザイン ガイドライン」を参照することを推薦する。

※ 特に Extreme Programming プロジェクトでの利用を意識したものではなく、あらゆる C#を使ったプロジェクトでの利用を想定している。

2. ファイル構成

(1) ファイル名

public クラスはそのクラス名の 1 ファイルにする。

例：public class Customer は、Customer.cs に入れる。

パッケージ内の非パブリッククラスは、そのクラスが主に使われるパブリッククラスのファイルに含めて良い。

例外クラスは、1 ファイルに複数クラスをまとめてもよい。

(2) ファイルの位置

プロジェクトのルートディレクトリを決め、名前空間の“.”をディレクトリ階層に置き換えた位置に入れる。但し、ソリューション / プロジェクトに対応している名前空間の階層は、ソリューション名 / プロジェクト名をディレクトリ名に使用する。

例：
名前空間：
CompanyName.OrganizationName.TechnologyName.CoreFeatureName.SubFeatureName

ソリューション **SolutionName** に対応する名前空間：**TechnologyName**
プロジェクト **ProjectName** に対応する名前空間：**CoreFeatureName**

配置先：
C:\CompanyName\OrganizationName\SolutionName\ProjectName\SubFeatureName

(3) テストクラス名

クラス **ClassName** のユニットテストクラス名は、**ClassNameTest** とする。ソリューション毎テストは **SolutionNameTests** とする。

例：**Customer** クラスなら **CustomerTest.cs** を作成
例：**CsSample** ソリューションなら、**CsSampleTests.csproj** とする

理由：一貫性のある名前づけ。テストコードは使い方のサンプル、デモともなる。

(4) テストクラスの位置

テストクラスの位置は、被テストクラスと同じ階層のディレクトリまたはそのサブディレクトリに配置する。

例：
被テストクラスの位置：
C:\CompanyName\OrganizationName\SolutionName\ProjectName

テストクラスの位置：
C:\CompanyName\OrganizationName\SolutionName\SolutionNameTests
C:\CompanyName\OrganizationName\SolutionName\SolutionNameTests\ProjectName

理由:物理的に近い位置でないとメンテが忘れ去られる。製品コードとの分離については、別のツール(NAnt の build ファイルなど)で調整可能。

3．命名規則

(5)　名前空間

企業略式名.組織略式名.テクノロジー名.機能名を使用する。また、テクノロジー名にはソリューションが、機能名にはプロジェクトが対応していること。

```
using CompanyName.OrganizationName.TechnologyName.FeatureName
```

(6)　ファイル名

パブリックなクラス名は、ファイル名と同じでなくてはならない。(大文字小文字の区別を含めて)

(7)　クラス名

先頭大文字。あとは区切りを大文字。

```
class PascalCasing
```

(8) 例外クラス名

最後を **Exception** としたクラス名。

```
class ClassNameEndsWithException
```

(9) インターフェイス名

クラス名に同じ。また常に最初に **I** を付ける。

```
interface INameOfInterface
```

また、クラスにある能力を加える様な利用の場合、その能力を示す形容詞とし、**-able** を接尾にする。

例: **IEnumerable, ICloneable, IXmlSerializable, …**

(10) 実装クラス名

特に **Interface** と区別の必要があれば、最後に **Impl** を付ける。

```
class ClassNameEndsWithImpl
```

(11) 抽象クラス名

抽象クラス名に適当な名前が無いとき、 **Abstract** から始まりサブクラス名を連想させる名前を付ける。

```
abstract class AbstractBeforeSubClassName
```

(12) 定数(Const)

大文字もしくは大文字を “ _ ” でつないだもの。

```
const int UPPERCASE = 0;
const int UPPERCASE_WITH_UNDERSCORES = 0;
```

(13) 列挙型(enum)

先頭大文字。あとは区切りを大文字。

```
enum PascalCasing
```

列挙型がビットフィールドを表すときは複数形にし、**FlagsAttribute** を付加する。

```
[Flags] enum PascalCasings
```

(14) 列挙値

先頭大文字。あとは区切りを大文字。

```
PascalCasing
```

(15) イベント名

先頭大文字。あとは区切りを大文字。

```
event PascalCasing()
```

(16) メソッド名

先頭大文字。あとは区切りを大文字。

```
void PascalCasing()
object PascalCasing()
```

(17) ファクトリメソッド(オブジェクトを new するもの)

```
X NewX()
X CreateX()
```

(18) コンバータメソッド(オブジェクトを別のオブジェクトに変換するもの)

```
X ToX()
```

(19) プロパティ名

先頭大文字。あとは区切りを大文字。

```
object PascalCasing()
```

(20) Boolean 変数を返すメソッド

Is + 形容詞、**Can +** 動詞、**Has +** 過去分詞、三単元動詞、三単元動詞 **+** 名詞。

良い例：

```
bool IsEmpty()
bool CanGet()
bool HasChanged()
bool Contains(object x)
bool ContainsKey(string key)
```

悪い例：

```
bool Empty()     //「空にする」という動詞的な意味に取れる。
bool CheckXXX() // true がどちらの意味か分かりづらい。
```

理由： **if, while** 文等の条件が読みやすくなる。また **true** がどちらの意味か分かりやすい。

(21) bool 変数

形容詞、is + 形容詞、can + 動詞、has + 過去分詞、三単元動詞、三単元動詞 + 名詞。

```
bool isEmpty;
bool dirty;
bool containsMoreElements;
```

(22) 英語と日本語

すべての識別子の名前は英語を基本とし、別に、日英対応用語辞書を作成してプロジェクトの全ライフサイクルでメンテナンスすること。

(23) 名前の対称性

クラス名、メソッド名を付ける際は、以下の英語の対称性に気を付ける。

```
Add / Remove
Insert / Delete
Get / Set
Start / Stop
Begin / End
Send / Receive
First / Last
Get / Release
Put / Get
Up / Down
Show / Hide
Source / Target
Open / Close
Source / Destination
Increment / Decrement
Lock / Unlock
Old / New
Next / Previous
```

(24) ループカウンタ

スコープ（通用範囲）が狭いループカウンタに i, j, k という名前をこの順に使う。

(25) スコープが狭い名前

スコープが狭い変数名は，型名を略したものを使って良い。

例： `DataSet ds = new DataSet();`

(26) 意味がとれる名前

変数名から役割が読み取れる名前を好め。

悪い例： `Copy(string s1, string s2)`
良い例： `Copy(string source, string destination)`

(27) 無意味な名前

`Info, Data, Temp, Str, Buf` という名前は再考を要する。

悪い例：`double temp = Math.Sqrt(b*b - 4*a*c);`
良い例：`double determinant = Math.Sqrt(b*b - 4*a*c);`

(28) private / protected / internal / protected internal スコープのインスタンス変数

最初小文字で、あとは区切りを大文字。プリフィクス / サフィクスを適用する場合は、変数の読みやすさを考慮すること。ハンガリアン表記法は使用しない。

```
private object camelCasing;
protected object camelCasing;
internal object camelCasing;
```

```
protected internal object camelCasing;
```

(29) Public スコープのインスタンス変数

先頭大文字。あとは区切りを大文字。極力使用しないようにすること。

```
public object PascalCasing;
```

(30) private / protected / internal / protected internal スコープの共有変数

最初小文字で、あとは区切りを大文字。プリフィクス / サフィクスを適用する場合は、変数の読みやすさを考慮すること。

```
private static object camelCasing;
protected static object camelCasing;
internal static object camelCasing;
protected internal static object camelCasing;
```

(31) public スコープの共有変数

先頭大文字。あとは区切りを大文字。極力使用しないようにすること。

```
public static object PascalCasing;
```

(32) ローカル変数

最初小文字で、あとは区切りを大文字。ハンガリアン表記法は使用しない。

```
object camelCasing;
```

(33) 大文字小文字

大文字と小文字は別な文字として扱われるが、それのみで区別される名前をつけてはならない。

4. ガイドライン

(34) #Region / #End Region ディレクティブ

コード領域は＃**Region / #End Region** ディレクティブで宣言し、その領域についての説明を含める。

例：

```
#Region "インスタンス変数"

    private string name;

#End Region

#Region "コンストラクタ"

    public MyClass(string name)
    {
        this.name = name;
    }

#End Region
```

(35) メソッド/プロパティの宣言

メソッド/プロパティの宣言ではスコープを明示的に指定する。

(36) 長い行

一行は最大 100 桁とし、それを超える場合は行を分割する。分割の指針は、(1) ローカル変数を利用、(2)算術演算子/連結演算子で改行、(3)カンマで改行、(4)優先度の低い演算子の前で改行　とする。

例：

```
double length = Math.Sqrt(Math.Pow(new Random().NextDouble(), 2.0) +
Math.Pow(new Random().NextDouble(), 2.0));

' 方針(1)
double xSquared = Math.Pow(new Random().NextDouble(), 2.0);
double ySquared = Math.Pow(new Random().NextDouble(), 2.0);
double length = Math.Sqrt(xSquared + ySquared);

' 方針(2)
double length = Math.Sqrt(Math.Pow(new Random().NextDouble(), 2.0),
                          Math.Pow(new Random().NextDouble(), 2.0));

' 方針(3)
LongMethodSignature(value[0], value[1], value[2],
                    value[3], value[4], value[5]);

' 方針(4)
return (this is obj) _
    || (obj is Class1 _
        and this.Field == obj.Field);
```

(37) 長いクラス宣言行

クラスの宣言が長い場合はカンマ、コロンで改行とする。

例:

```
public class LongNameClassImplemenation : AbstractImplementation,
IXmlSerializable, Icloneable
```

ならば、

```
public class LongNameClassImplemenation :
    AbstractImplementation,
    IXmlSerializable, ICloneable
```

(38) 長いメソッド宣言行

メソッドの宣言が長い場合はカンマで改行とする。

例：

```
public void LongMethodSignature(int a, int b, int c
                                , int d, int e, int f)
```

(39) abstract Class vs. interface

抽象クラス(abstract Class)はなるべく使わず、interface を多用せよ。abstract Class は、一部実装があり、一部抽象メソッドであるような場合にのみ使う。

理由：interface は幾つでも継承できるが、Class は 1 つだけ。1 つから継承してしまうと、もう継承できずもったいない。

(40) public Variable

インスタンス変数は、極力 public にせず、妥当なプロパティを設ける。

理由：オブジェクト指向の標準。クラスの内部状態に勝手にアクセスさせるのはよくない。ただし、以下の条件をすべて満たす場合、インスタンス変数を public にし、直接アクセスさせてもよい。

- そのインスタンス変数が他のインスタンス変数と独立であり、単独で変更されても内部の整合性をくずさない。

- どちらにしても，**get / set** アクセサを書く。
- インスタンス変数の実装が将来に渡って変更されないことが根拠付けられる。

また、上記に当てはまらない場合でも、極度に速度を気にする場合は、この限りではない。
(ただし、慎重にコメントすること**)**

(41) 初期化

初期化をあてにしない**(**参照が **null** に初期化されているとか**)**。また、２度初期化しない。

悪い例：

```
public class PoorInitialization
{
    private string name = "initial_name";

    public PoorInitialization()
    {
        name = "initial_name";
    }
}
```

理由：初期化に関するバグを最小化する。

(42) static 変数を避ける

static 変数**(**クラス変数**)**は極力避ける。

理由：**static** 変数は、セミグローバルと言って良い。より文脈依存なコードを招き、副作用を覆いかくしてしまう。

(43) private vs. protected

private よりは、**protected** を使用する。

理由：**private** は確実にそのクラス外からの使用をシャットアウトできるが、クライアントが、より細かいチューニングを **SubClass** 化によって行うことを出来なくしてしまう。

別法**: private** をより好んで使え。**protected** にしてしまうと以降、変更が起ったときにそれを継承している全クラスに影響が出てしまう。

(44) get / set アクセサ

無闇にインスタンス変数へのプロパティ**(get / set** アクセサ**)**を作成して **public** にすることは避ける。その必要性を検討し、もっと意味のあるプロパティ **/** メソッドにする。

理由：インスタンス変数は、他のインスタンス変数に依存していることが多い。クラス内部の整合性を崩してはならない。

(45) 変数隠し

基本クラスの変数名と、同じ変数名を使う事は避けよ。

理由：一般的にはこれはバグである。もし意図があるならコメントせよ。

(46) public メソッド

クラスの **public** メソッドは、「自動販売機のインターフェイス」を目標に。分かりやすく、使いかたを間違っても内部の整合性はこわれないように設計する。また、可能ならば契約による設計**(Design by Contract)**を行い、クラスの不変条件と共にメソッドの事前･事後条件をコードで表現せよ。

(47) 状態取得と状態変更の分離

メソッドは,「**1** つの事」を行うように設計せよ。特に、状態変更と状態取得の２つのサービスを１つのメソッドで行わない。状態を変更するメソッドの **return** 値は **void** にせよ.

理由 1：１つの事を行うメソッドの方が分かりやすい。

理由 2：並行性の制御、例外の安全保証がしやすい。

理由 3：サブクラス化による拡張がしやすい。

(48) this の return

クライアントの便宜を考えたつもりでも、this を return することはなるべく避ける。

理由：A.Meth1().Meth2().Meth3() というような連鎖は、一般的に Synchronization 上の問題の元になる。

(49) メソッドの多重定義

引数のタイプによるメソッドのオーバーロードはなるべく避ける(引数の数が違うものは OK である)。特に、継承と絡むと厄介である。

例：

×：
```
override void Draw(Rectangle rectangle)
override void Draw(Point point)
```

○：
```
void DrawRectangle(Rectangle rectangle)
void DrawPoint(Point point)
```

(50) Equals()と GetHashCode()

Object.Equals()メソッドをオーバーライドするなら、同時に GetHashCode()メソッドもオーバーライドせよ。逆も同じ。

理由：コンテナクラス(Hashtable)などに対応するため。

(51) Clone()

もし、Clone() メソッドが使われるようなら、ICloneable を実装し明示的にそれを書く。

例：

```
using System;

public class Foo : ICloneable
{
    public object ICloneable.Clone()
    {
        Foo myFoo = (Foo) this.Clone();

        // Foo クラスの属性のクローン
        //...
    }
}
```

理由：簡易コピーではよくないケースがほとんどである。

(52) デフォルトコンストラクタ

可能ならいつでもデフォルトのコンストラクタ(引数がないもの)を用意せよ。

理由：リフレクションを使用して、Assembly.CreateInstance(TypeName)で型名文字列からダイナミックにそのクラスを作成可能。

(53) abstract Method in abstract classes

abstract クラスでは、no-op のメソッドを書くより、明示的に abstract メソッドと宣言せよ。また、共有可能なデフォルトの実装を用意できるなら、それを protected とし、サブクラスが 1 行で処理を書けるようにせよ。

理由：.NET の IDE は、実装されていない abstract メソッドを検出できるため、単に実装

を忘れていただけ、というバグを回避できる。

(54) オブジェクトの同値比較

オブジェクトの比較では **Equals()** メソッドを使い、**=** を使うな。特に、**String** の比較では **=** を使用してはならない。

理由：もし実装者が **Equals()**を用意しているなら、それを使ってほしくて実装したはず。また、**String** の比較では **Option Compare Text** に設定されていると大文字/小文字が区別されない。

理由：ユニットテストでは、**AssertEquals** が **Equals()** を利用しているため、簡単に同値テストが書ける。

(55) 宣言と初期化

ローカル変数は、初期値と共に宣言せよ。

理由：変数の値に関する仮定を最小化する。

悪い例：

```
void f(int start)
{
    int i, j; // 初期値なしの宣言
    // 多くのコード
    // ...
    i = start + 1;
    j = i + 1;
    // i, jを使う
    // ...
}
```

良い例：

```
void f(int start)
```

```
{
    // 多くのコード
    // ...
    int i = start + 1;
    int j = i + 1;
    // i, jを使う
    // ...
}
```

(56) ローカル変数の再利用は悪

ローカル変数を使い回しするより、新しいものを宣言して初期化せよ。

理由：変数の値に関する仮定を最小化する。
理由：コンパイラの最適化を助ける。

悪い例：

```
void f(int n, int delta)
{
    int i; // 初期値なしの宣言
    for (i = 0; i < n; i++)
    {
        // iを使う
    }

    for (i = 0; i < n; i++) // またiを使う
    {
        if (...)
        {
            break;
        }
    }

    if (i != n - 1)     // 最後まで回ったかの判定にiを使っている
    {
```

```
        // ...
    }

    i = n - delta * 2;   // またまた再利用
    //...
}
```

良い例：

```
void f(int n, int delta)
{
    for (int i = 0; i < n; i++)
    {
        // iを使う
    }

    bool found = false;

    for (int j = 0; i < n; i++)
    {
        // jを使う
        if (...)
        {
            found = true;
            break;
        }
    }

    if (found)
    {
         // ...
    }

    int total = n - delta * 2;    // 別の意味ある変数
    //...
}
```

(57) 大小比較演算子

“<”、 “<=” を好んで使い、”>”、 “>=” はなるべく避けよ。

理由：大小の方向を統一し、右側を大きい方にすることで、混乱を避ける。

(58) if/while 条件中の "="

if, while の条件には、代入 **"="** を使ってはならない。

理由**:** ほとんどの場合、バグである。

(59) キャスト

キャストは、できる限り **is class** の条件文で囲め。

例：

```
C cx = null;

if (x is C)
{
    cx = (C) x;
}
else
{
    evasiveAction();
}
```

理由：これで、「オブジェクトがそのインスタンスじゃなかったら**?**」とういことを常に考える癖がつく。ただし、キャスト出来ない場合がバグである、と判断できる場合は、この限りではない。

(60) 例外クラス

例外クラスは大域的な性格をもち、多用するとプログラムの流れを読みにくくしてしまうことを認識する。

例外クラスは、新たに作成するよりも、.NET クラスライブラリに含まれているものを利用できれば利用する。

例: IOException, FileNotFoundException, ArgumentException などは利用しやすい標準例外。

新たな例外の作成は、そのパッケージ全体のインターフェイスとして検討すること。

(61) メソッド引数の変更は悪

原則としてメソッドの引数は入力であり、出力としては使わないこと。すなわちメソッド内部で引数の状態を変更するメソッドを呼ばないこと。出力引数に新たなオブジェクトを代入しないこと。

悪い例:

```
void MoveX(Point p, int dx)
{
    p.X = p.X() + dx; // 引数を変更している(なるべく避ける)
}

void MoveX(Point p, int dx)
{
    Point p = New Point(p.X + dx, p.Y);
    // これは呼び出し側に伝わらない
}
```

例外：パフォーマンスを気にする場合。

(62) メソッド引数の名前

メソッドの引数は、読みやすいものにすること。特に、インスタンス変数と重なった場合、**this** を活用し、引数の読みやすさを犠牲にしないこと。

悪い例：

```
void Reset(int x_, int y_)
{
    x = x_;
    y = y_;
}
```

良い例：

```
// 引数名を x_, y_ などとしない
void Reset(int x, int y)
{
    this.x = x;
    this.y = y;
}

void Reset(int x, int y)
{
    _x = x;
    _y = y;
}
```

(63) ToString()

ToString() メソッドは可能ならいつでも実装すること。

理由 **1**：**Console.WriteLine(Object)** でいつでもプリントできる。
理由 **2**：ユニットテスト等で失敗した場合の表示が分かりやすくなる。

(64) switch, if/else の繰り返しは悪

switch 文で分岐する処理が現れた時には、よくない設計の兆候だと考え、ポリモーフィズムで実現できないか再考する。特に同じような switch が 2 箇所以上現れたら、必ずポリモーフィズム、FactoryMethod, Prototype パターン等でリファクタリングすること。if / else の連続も同様。さらに、null チェックを行う同様の If が多くの場所に現れたら、NullObject パターンの利用を検討せよ。

5. コメント

(65) ドキュメントコメントの活用

/// <tag>コメント</tag> を多いに活用すること。このコメントは、VS.NET や csc によって、XML、HTML 形式でのドキュメントに変換することができる。

public クラス、メソッド、プロパティには、必ず /// コメントをつける。

(66) 長いコメント

コメントが複数行に渡る場合は、最初の短い一文で何が言いたいかを書き、その後に長いコメントを付けること。またそのような長いコメントの必要性を感じたときには常に設計をもっとシンプルにできないかを再考し、積極的にリファクタリングすること。

(67) Design by Contract (契約による設計)

契約による設計を行うため、Debug オブジェクトの Assert メソッドを使用せよ。Assert メソッドを使って、契約を表現せよ。

例：

```
using System.Collections;
```

```
public class MyStack : Stack
{
    private int capacity;
    private int itemCount;

    public overrid void Push(Object obj)
    {
        Debug.Assert(!(obj is null));    // 事前条件
        // ...
        // ...
        Debug.Assert(this.Contains(obj));  // 事後条件
    }

    public bool Invariant()  // 不変条件
    {
        Debug.Assert(0 <= capacity);
        Debug.Assert(0 <= itemCount);
        Debug.Assert(itemCount <= capacity);
        return true;
    }
}
```

注意：ユーザ入力チェックなどを **Assert** してはいけない。バグを捕まえるために **Assert** せよ。

6. パフォーマンス

(68) まず計測

パフォーマンス改善はまず計測から始めよ。当てずっぽうではだめ。

(69) new

.NET では、new は時間が掛かる。ヘビーなループの中で new が呼ばれる場合、必要ならば出力引数を用いる。

```
X GetX()
{
    return (new X(this.value));
}
```

が遅い場合、呼び出し側に new を任せ、

```
void GetX(X x)
{
    x.Value = this.value;
}
```

とせよ。

(70) 変数への null の代入

使われない変数が大量に発生する場合、積極的に null を代入せよ。特に、配列の要素(パフォーマンス要求が厳しい場合)。

理由：ガベージコレクションを助ける。

7. その他

(71) 自分で新しく作る前に相談

他人が作成したクラスに対するある操作が新たに必要となるとき、自分でそのクラスを継承して新たなクラスを作成したり、そのクラスをインスタンス変数として持つクラスを作

成したりするより、まずそのクラスの作成者に相談すること。汎用的な形でその要望を満たしてくれれば、全体をコンパクトにできる。

(72) 複雑な設計は悪

設計で迷った場合、多くのケースは ‘Simplicity’ を重視した方がよい。また、後のメンテナビリティにも ‘Simplicity’ は重要である。

(73) パフォーマンス調整は測定後

最初からパフォーマンスを気にしたコーディングをするべきではない。読みやすさ、保守のしやすさを優先する。パフォーマンスは測定してから改善する。

(74) トリッキーなコードは悪

.NET の平均プログラマに分かるようなコードを書く。演算子の順序、初期化に関する規則など、誰もが必ずしも自信をもって答えられないような仮定を持ち込まず、() を使って演算順序を明確にしたり、明示的な初期化をしたりする方が読みやすい。

悪い例: `return cond == 0 ? a < b && b < c : d == 1;`
良い例: `return (cond == 0) ? ((a < b) && (b < c)) : (d == 1);`

悪い例：

```
// 単位行列を作るが、時間もかかるし誰も読めない。
for (int i = 1; i <= n; i++)
{
    for (int j = 1; j <= n; j++)
    {
        m[i-1, j-1] = (i / j) * (j / i);
    }
}
```

(75) 100%正しいことはない

ここに書かれていることに、100% 準拠する必要はない。迷ったら考えを整理し、相談すること。十分な理由があってルールから外れることはよくある。コミュニケーションができるチームの助けとなることが、このコーディング標準の目的である。

8. 謝辞

このコーディング標準を Java オリジナル版、VB.NET 版に変更するにあたって、**さんから貴重なご意見を頂きました。ありがとうございました。また、オリジナルからの変更を快く了承して頂いた、**** さんに感謝致します。

9. 参考資料

Kenji Hiranabe, Java コーディング標準(オリジナル)
http://objectclub.esm.co.jp/eXtremeProgramming/CodingStd.doc

Writing Robust Java Code 高橋さんによる日本語訳版
http://www.alles.or.jp/~torutk/oojava/codingStandard/writingrobustjavacode.html

Writing Robust Java Code オリジナル(英語)
http://www.ambysoft.com/javaCodingStandards.html

太陽システム(株)　中西庸文　VB.NET コーディング標準
http://ObjectClub.esm.co.jp/eXtremeProgramming/CodingStdVB.doc

以上