コードの品質こそがビジネスを成功させる!
～ コードの品質を上げるために
命名編 ～

codeseek＋こみゅぷらす

コミュニティ紹介

http://www.codeseek.net/

- 「システムの品質はコードに宿る」
- 第13回codeseek勉強会 (昨日)

http://comuplus.net/

- 昨年12/1 発足
- INETA コミュニティ リーダーや MSMVP などからなる 11 名の運営スタッフ

自己紹介

- 原　敬一
- 衣川 朋宏
- 小島 富治雄

Microsoft Most Valuable Professional
Visual Developer –
Visual Basic(2005/07 - 2007/06)

Microsoft Most Valuable Professional
Visual Developer –
Visual Basic(2006/04 - 2007/04)

Microsoft Most Valuable Professional
Visual Developer –
Visual C#(2005/07 - 2007/06)

本日のテーマ

名前重要。

その前に...

コミュニティ イベントなので

- 盛り上げることが重要です。
  - 参加した方がどちらかというと得
  - Give and Give
    - 盛り上がった方が、お互いにどちらかというと得

拍手重要。

同意の場合は拍手してください。

※ 記念品付き。

豪華腰リールキット、メモ帳セット、福井銘菓

練習します。

アンケートやります。

アンケート

「codeseek＋こみゅぷらす の
勉強会のことを、ご存知ですか?」

1. はい。
2. いいえ。

アンケート

「美しいコードを書く方法について」

1. 正直どうでもいい。
2. 上司がうるさいのでしかたなしに学んでいる。
3. 積極的に[学んで|推奨して]いるほうだ。

アンケート

「変数名について: どちらかというと…」

1. **camel**派だ。
   - *typeName, backColor*
2. **Pascal**派だ。
   - *TypeName, BackColor*
3. **アンダーバー区切り**派だ。
   - *type_name, back_color*
4. **ハンガリアン**派だ。
   - *szTypeName, clrBack*

くソースの例: 命名編

- ***Dim i As Integer***

  意図がない。

- ***tmpWork***

  内容物を表していない。

- ***input***

  予約語と混同しやすい。

- ***lclusrdafName***

  読みにくい。

くソースの例: 命名編

- ***Dim sName as DataTable***
  Stringと勘違いしてしまう罠系。
- ***Dim ds as DataSet***
  サンプルソース丸写し系。
- ***Dim strBuffer as String***
  「だから何のバッファだ!」というツッコまれ系。
- **変数名が全部ジャニーズ**
  或る会社の新人研修で実在。

くソースの例: 命名編

- ***InitShori***

  途中から日本語のローマ字表記。

- ***GetNam***

  中途半端な省略。

- ***bool flg = false;***

  一体何のフラグか分らない。

- ***InisiariseEmproiiDeta***

  何語だよ!

名前重要。

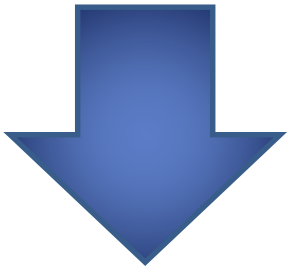
名前重要 アジェンダ

1. **Accountability** (説明責任)
2. **Name and Conquer** (定義攻略)
3. **SON : Service Oriented Naming** (サービス指向名前付け)

1.
Accountability
(説明責任)

プログラミングとは:

コンピュータにどうやったらいいかを
逐一教えてやること
(How)



パラダイム シフト

何をやりたいかという**意図**を
人がわかりやすいように表現すること
(What)

例.「日付チェック」

- 或る日付 (年・月・日) が、日付として正しいかどうかをチェック
  - ○ 2007/02/14
  - × 2007/13/32
  - × 2007/02/29
  - × 2100/02/29
  - ○ 2000/02/29

例 1.「日付チェック (1)」

例.「日付チェック (1)」

```
static void ChkFunc2(int y, int m, int d)
{
   string txt = "エラー: 日付が正しくありません。";
   if (y < 1)
      Console.WriteLine(txt);
   else if (m < 1 || m > 12)
      Console.WriteLine(txt);
   else if (m == 2) {
      if (y % 4 == 0 && y % 100 != 0 || y % 400 == 0)  {
         if (d < 1 || d > 29)
            Console.WriteLine(txt);
      } else {
         if (d < 1 || d > 28)
            Console.WriteLine(txt);
      }
   } else if (m == 4 || m == 6 || m == 9 || m == 11) {
      if (d < 1 || d > 30)
         Console.WriteLine(txt);
   } else {
      if (d < 1 || d > 31)
         Console.WriteLine(txt);
   }
}
```

```
int y, m, d;
GetDat(out y, out m, out d);
ChkFunc2(y, m, d);
```

意図がシンプルに
表現されているか?

例 2.「日付チェック (2)」

例.「日付チェック (2)」

```
public class 日付
{
   int 年 = 2000;
   int 月 = 1;
   int 日 = 1;

   ……

   public bool 日付として正しい
   {  get { return 年が正しい && 月が正しい &&
                   日が正しい; } }

   bool 年が正しい
   {  get { return 年 >= 1; } }

   bool 月が正しい
   {  get { return (月 >= 1 && 月 <= 12); } }

   bool 日が正しい
   { get { return (日 >= 1 && 日 <= 月の最終日); } }
```

```
   int 月の最終日
   {
      get {
         switch (月)
         {
            case 2:
               return 二月の最終日;
            case 4: case 6: case 9: case 11:
               return 30;
            default:
               return 31;
         }
      }
   }

   int 二月の最終日
   { get { return うるう年か ? 29 : 28; } }

   bool うるう年か
   { get { return 年 % 4 == 0 && 年 % 100 != 0 ||
                  年 % 400 == 0; } }
}
```

例.「日付チェック (2)」

```
if (!友人.誕生日.日付として正しい)
    エラー表示("日付が正しくありません。");
```

意図がシンプルに表現されているか

・「日付チェック (2)」

```
bool 日付として正しい
{
   get { return 年が正しい && 月が正しい &&
                日が正しい; }
}
```

意図:
「日付として正しい」というのは、
「年が正しくて、月が正しくて、日が正しいこと」

意図がシンプルに表現されているか

- 「日付チェック (2)」

```
if (!友人.誕生日.日付として正しい)
    エラー表示("日付が正しくありません。");
```

意図:
もし、友人の誕生日が日付として正しくないならば、
「日付が正しくありません」とエラー表示する

意図がシンプルに表現されているか

- 「日付チェック (2)」

```
int 二月の最終日
 { get { return うるう年か ? 29 : 28; } }
```

意図:
「二月の最終日」は、
「もし、うるう年なら29日で、うるう年でなければ28日」

名前

- 「日付として正しい」「うるう年か」
- 「年が正しい」「 月が正しい」「 日が正しい」
- 「年」「月」「日」「日付」
- 「友人」「誕生日」「エラー表示」
- … …

名前

- 意図が明確であること
  - 何がやりたいのか?
- 責務の範囲が明確であること
  - 何をする?
  - 何をしない?

名前付けは、
モデリング。

- 頭の中のモデルにもっとも近いもの
  →意図をもっとも自然に、頭の中で表現するとどうなる?
  →自分の設計モデル。
  →分かりやすさ。

- ソースコードは設計を語るべき。
- ソースコードは意図を語るべき。

- それ自身が語る。
  - 例. アフォーダンス

例えば、或るクラスに “Employee” という名前を付けるということは、

『暗黙知』
自分の中にしかなかったある関心の範囲の概念

形式知化

他人にも分かる概念

2.
Name and Conquer
(定義攻略)

ソフトウェア開発の二つの攻略法

- Divide and Couquer
(分割攻略)

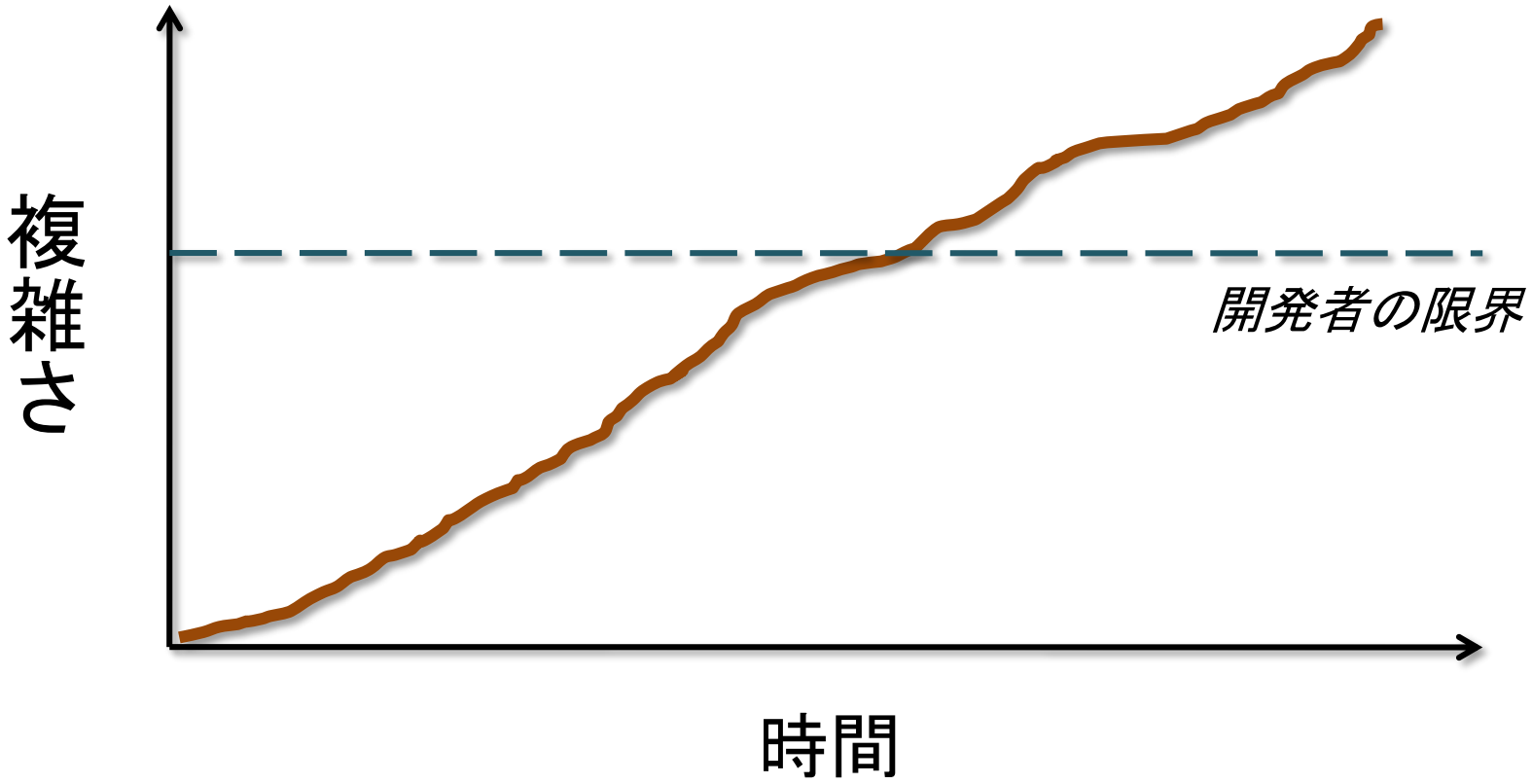
- Name and Conquer
(定義攻略)

ソフトウェア開発は
複雑さとの戦い。

時間とともに
ソフトウェアのエントロピーは
増大する傾向に

- 10年前 → 現在 → 10年後
- プロジェクトの初期 → プロジェクト後期 →プロジェクト末期

もう、どんどん複雑に。



ソフトウェア開発の複雑さが
ふつうの開発者の限界を超えたら
どうなる?

複雑さの解消を行う手段

- 先ずは、

Divide and Couquer (分割攻略)

- 複雑な問題を、シンプルな問題に分ける
  - 問題の切り分け
    - ここで扱う問題は何か?
  - 関心の分離

モデリング

- どう分けると、よりシンプルか?

  →腕の見せ所。

例.

関数で分割、クラスで分割、アスペクトを分割、レイヤで分割、M・V・Cを分割、コンポーネントとして分割、固定部と可変部を分割、まあとにかく関心を分離...

Name and Conquer

Name and Conquer

「ある注目すべきもの」を見つけ、
それに名前を付ける。

Name and Conquer

概念を切り出す。
ある概念を「他のものから」切り分ける。

名前を付けることは、
概念を確定させること。

例えば、

クラス/オブジェクト/メソッドを作り、

それに名前を付けるということは、

プログラムにおける
或る範囲の概念と
それ以外の間の
境界を決めること

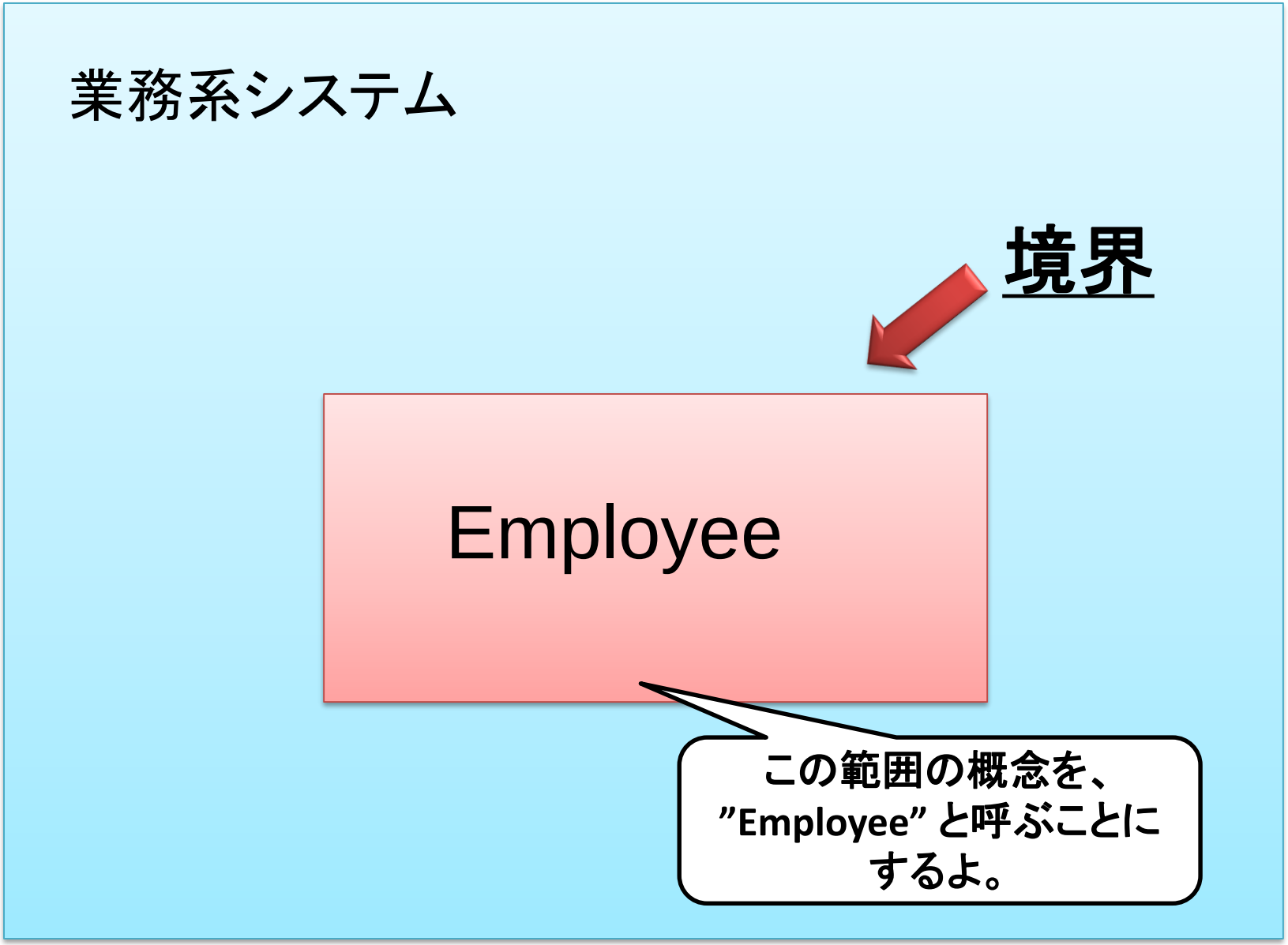
境界を決めるということは…

- それは何か?
- それは何でないか?

を決めるということ

例えば、或るクラスに “Employee” という名前を付けるということは、

- 「システムの中のこの範囲の概念を “Employee” と呼ぶことにするからね」ということ。
  - システム全体という混沌の中から “Employee” という概念を切り出す。
- “Employee” とそれ以外との間に境界を与え、“Employee” の概念の範囲を決めること。
  - **「Employee なもの」と「それ以外」を決定。**


業務系システム
境界
Employee
この範囲の概念を、
”Employee” と呼ぶことに
するよ。

クラスやメソッドを作るとき:

「どんな名前が良いかなー…
まあ、めんどくさいから、
適当に付けて、とにかく作っちゃえ」

というのは

何を作るか決めずに、作ること

こう行きたいところ:

何故作る? (Why?)

何を作る? (What?)

どう作る? (How?)

目的が手段を駆動する。

3.
SON :
Service Oriented Naming
(サービス指向名前付け)

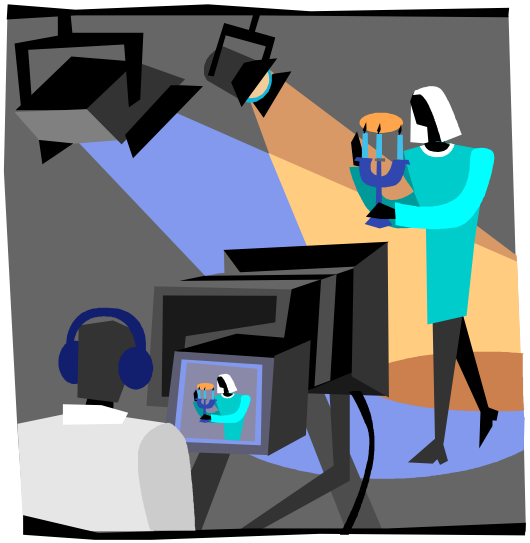
突然ですが…

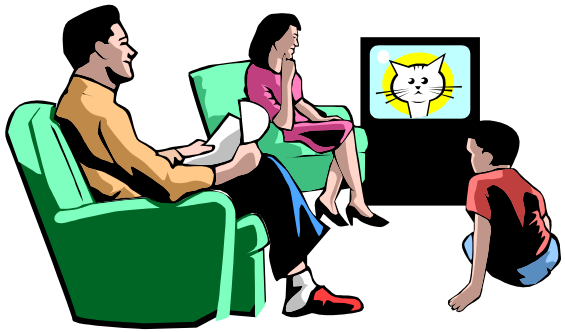
- 「テレビ」って何?
- 「電話」って何?

本来は

- tele-vision、tele-phone ⇒

  「遠くに**映像**や**音声**をとどけるシステム全体の名前」





じゃー　これは間違いなの?

- これがテレビ?

- これが電話?

「いいえ。」

- それこそがエンジニアの持つべき視点。

それって、システムがユーザーに提供する

インタフェイス

- テレビ

- 電話

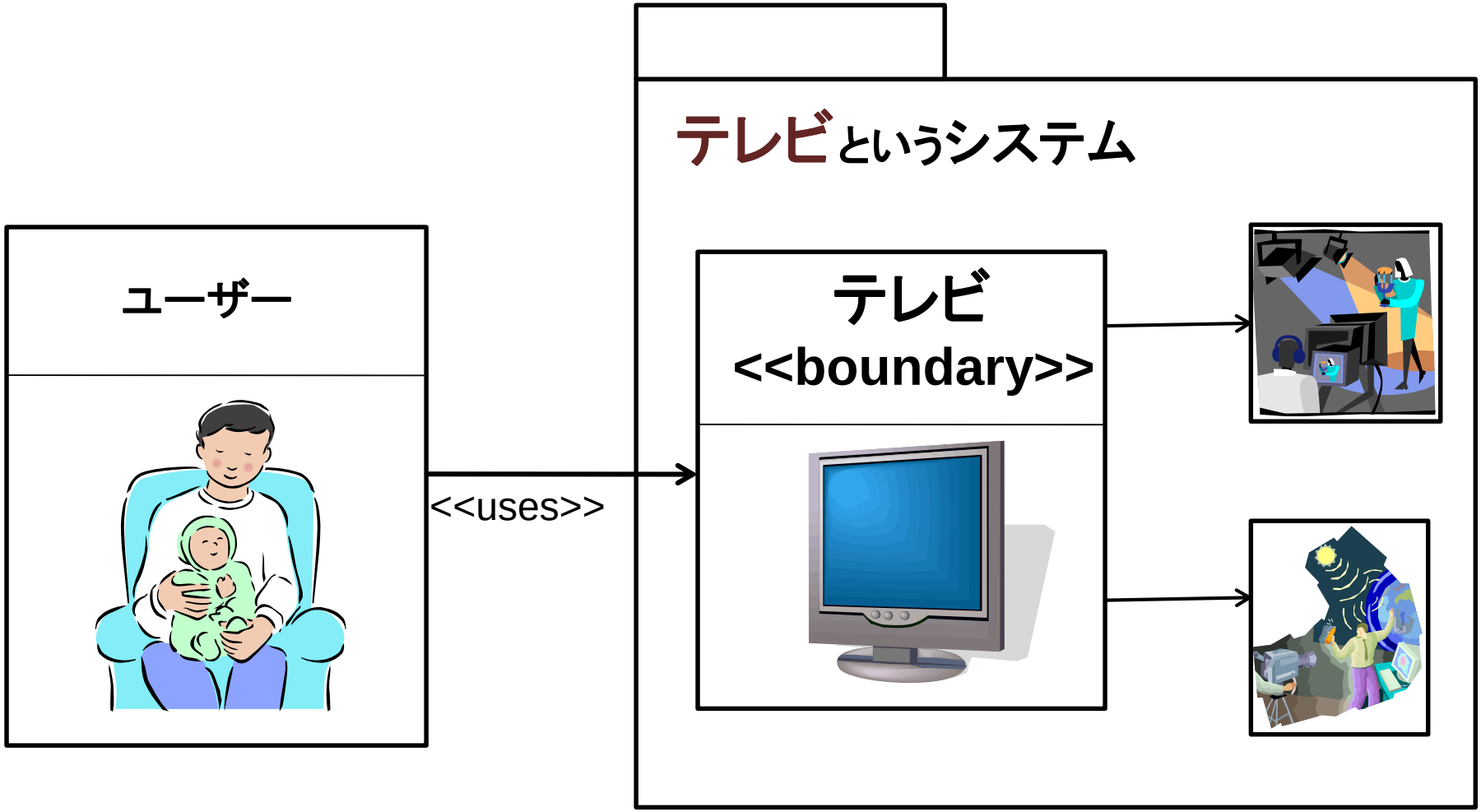
ユーザー インタフェイスが名前になる

ユーザー

テレビ
＝インタフェイス

テレビという名の
システム

UML によるモデル



ユーザーにとっては:

ユーザー インタフェイス

の名前が

そのものの名前。

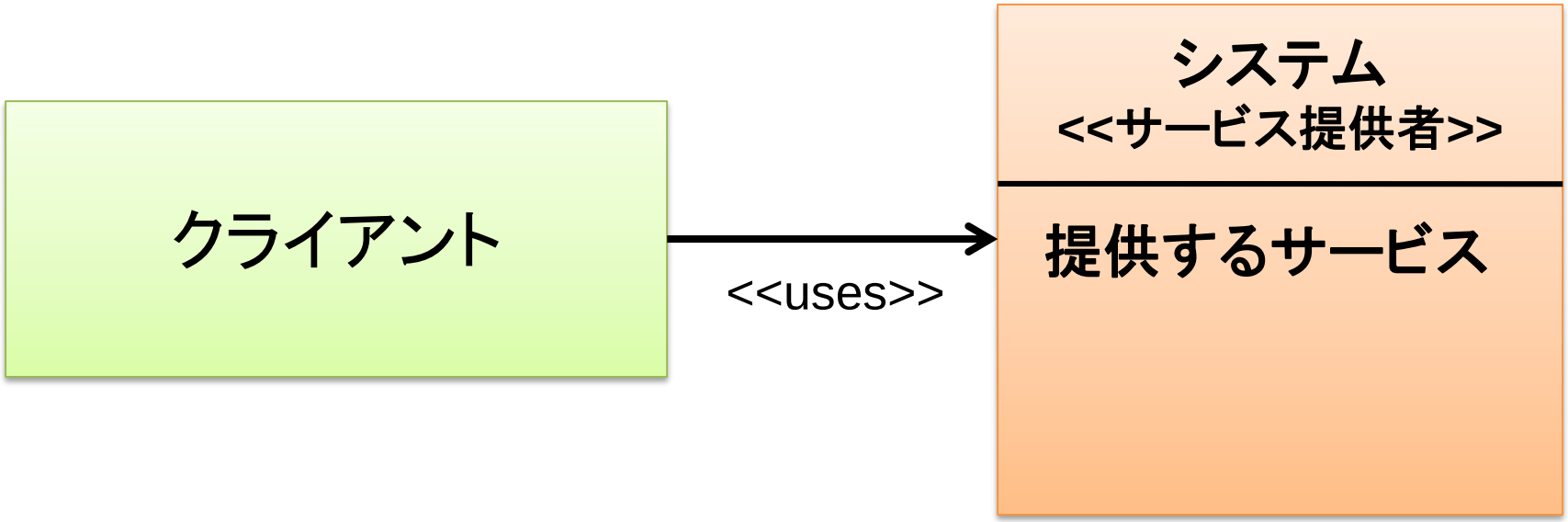
ところで…

システムを開発する目的は?

顧客の問題を
IT技術で解決すること。

目的 (＝顧客の問題解決) が
手段 (＝開発) を駆動するべき。

クライアントにサービスを提供する



クライアント視点重要。

クライアント視点でみると:

プログラムで使われている名前は、
プログラムがクライアントに提供する

インタフェイス

名前＝
インタフェイス

クライアント視点でみると:

プログラムで使われている名前は、
プログラムがクライアントに提供する

サービスの名称

名前＝
サービス

名前重要。

名前は
クライアント視点で。

粒度が異なっても同じ。

粒度が異なっても同じ。

- サービス側クラスの名前 →
  クライアント側クラスの視点で決定。
- サービス側メソッドの名前 →
  クライアント側メソッドの視点で決定。
- オブジェクトの名前 →
  オブジェクトをどう使うかで決まる。

使う側の視点で、使われる側の名前が決まる。

名前は顧客側の視点で決定

クライアント メソッド側の
モデル記述でサービスの
名前が決定

サービス提供側
クラス

```
if (!友人.誕生日.日付として正しい)
    エラー表示("日付が正しくありません。");
```

日付

日付として正しい：bool

サービス指向の名前付け

「クライアント側のモデルが
開発を駆動すべき。」

「クライアント側のモデル記述するのに必要な概念が、
サービス側の**名前を付けることで決定する**」

名前重要

Service Oriented Naming

開発者視点:
実装のための名前付け

パラダイム シフト!

クライアント視点:
クライアント側のモデルを記述するための名前付け
Service Oriented Naming