

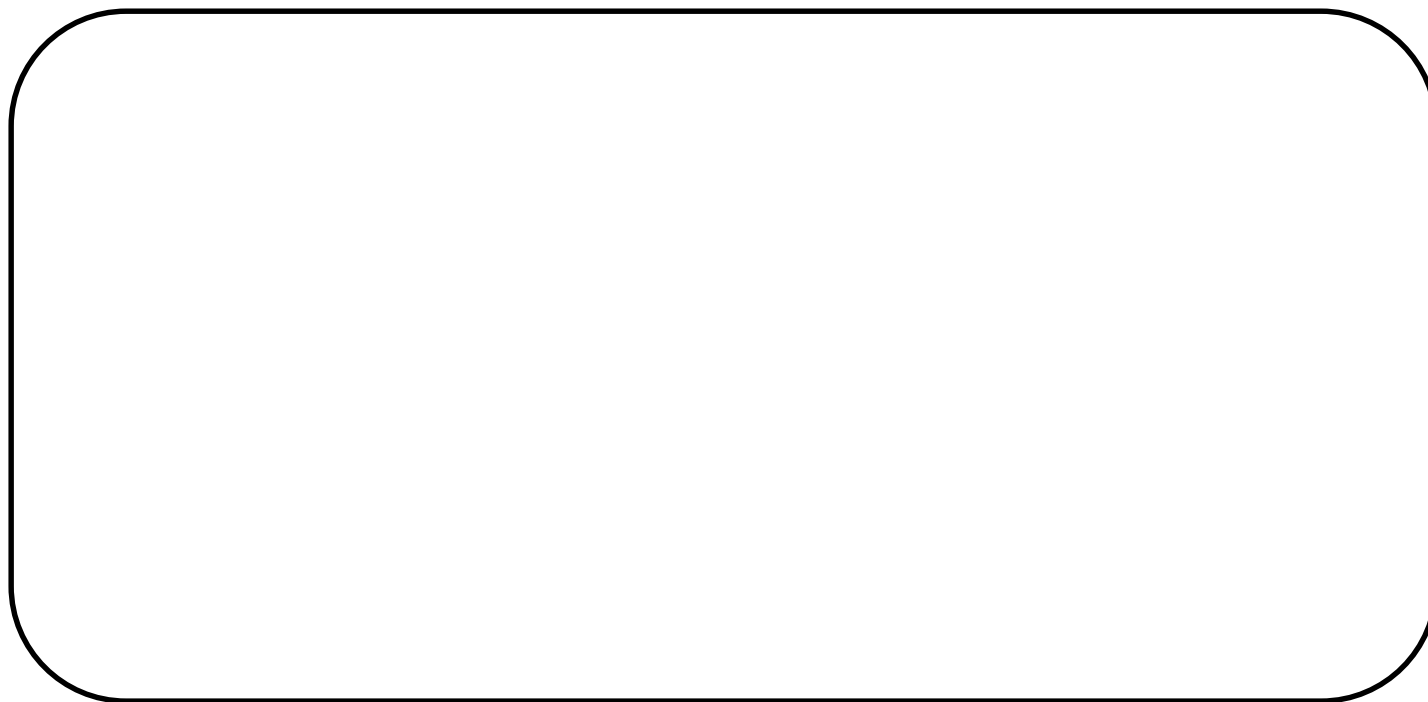
オブジェクト指向による ソフトウェア最適設計手法

2007/02/21(水) 10:00 ~ 17:00

22(木) 9:30 ~ 16:30

セミナーの目的

- オブジェクト指向の考え方のコツをつかみ、効果的な設計／実装を行う方法を習得する。

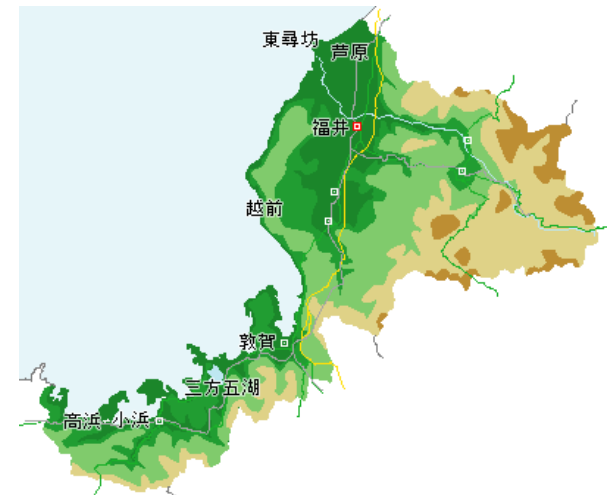


受講対象/前提条件

- C++、Java、C#、Visual Basic、Delphi などのオブジェクト指向プログラミング言語環境を使ったことがある
- 使用する言語
 - C#

自己紹介

- 小島 富治雄
- 福井コンピュータ株式会社
企画開発部 シニア エキスパート
- 福井県在住
－ 福井県？



自己紹介 — 所属コミュニティ

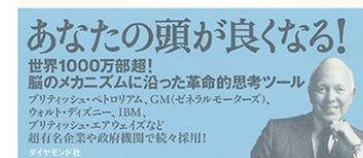
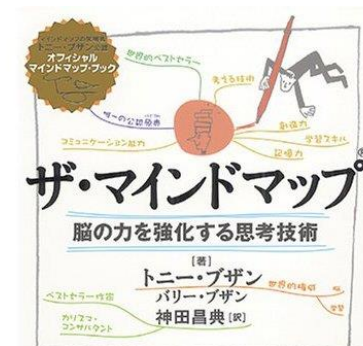
- Microsoft MVP — Visual Developer - Visual C#
- VSUG (Visual Studio Users Group) — 「開発プロセス」ボードリーダー
- INETA — コミュニティリーダー
- Culminis — コミュニティリーダー
- こみゅふらす (COMU+) — 代表
- FITEA - 福井情報技術者協会 — 代表
- 日本XPユーザーズグループ
- NAgile
- (社)情報処理学会 ソフトウェア工学研究会 パターンワーキンググループ

自己紹介 — 興味など

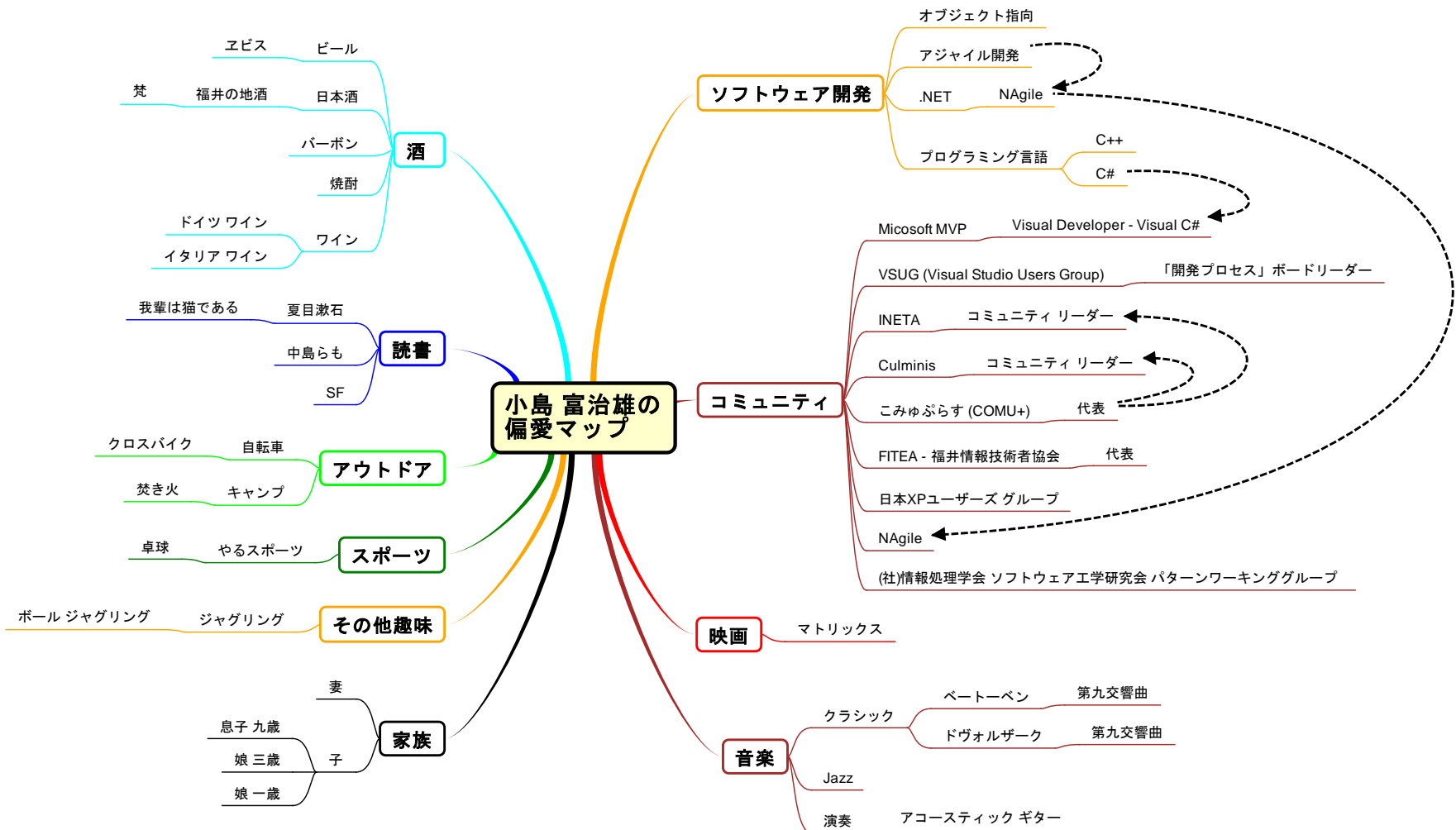
- .NET
- Agile
- .NET + Agile = NAgile
 - @IT — 『開発をもっと楽にするNAgileの基本思想 (<http://www.atmarkit.co.jp/fdotnet/nagilemind/index/>)』連載中。
- オブジェクト指向
- C#

自己紹介のやり方

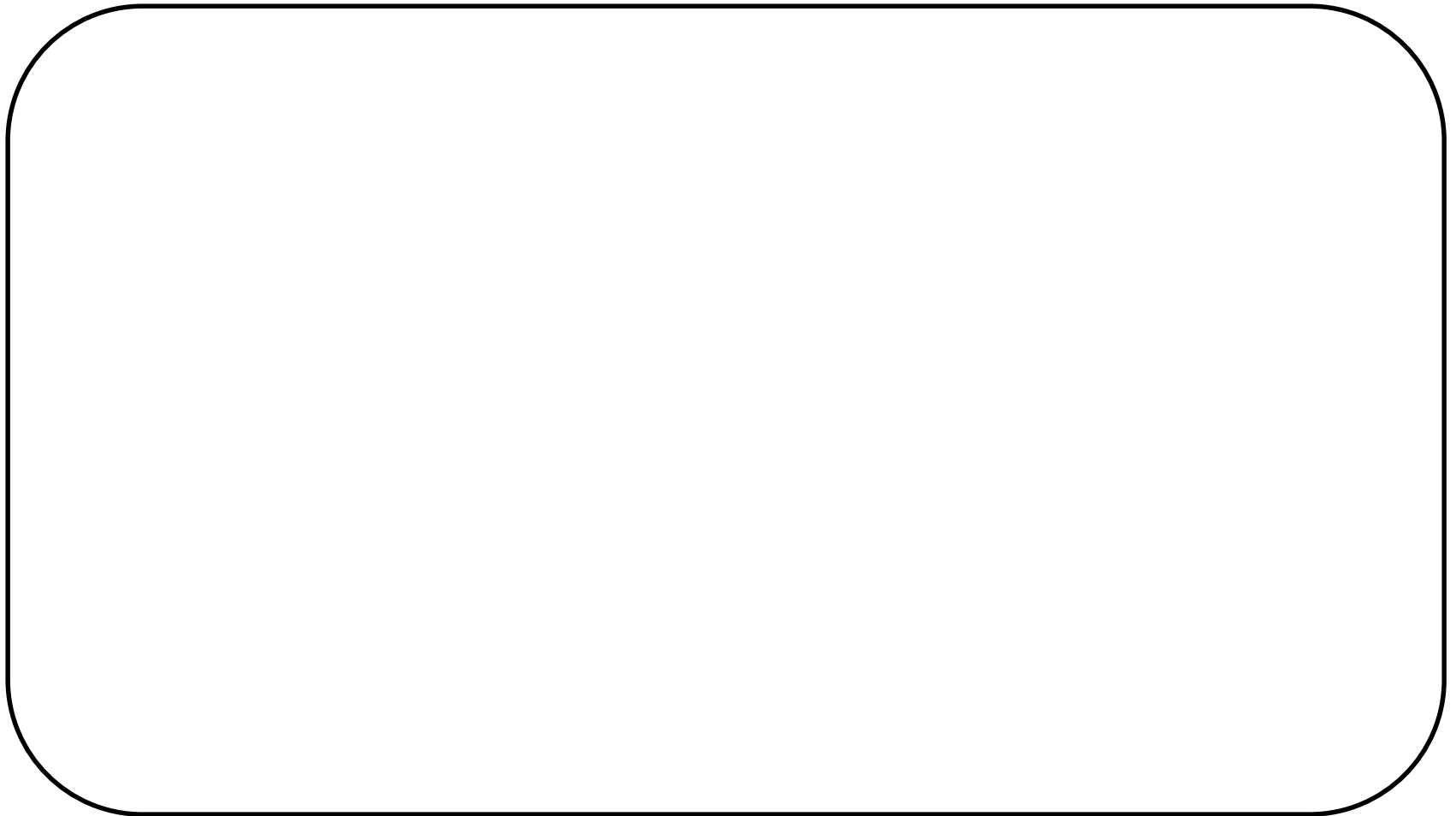
- 偏愛マップ
 - 『偏愛マップーキラいな人がいなくなる コミュニケーション・メソッド』 斎藤 孝 著
 - 『ザ・マインドマップ』 トニー・ブザン 著



自己紹介の例



自己紹介



アジェンダ 1

I. ソフトウェア開発の難しさ

1. ソフトウェア開発はなぜうまく行かないか?
 - a. ソフトウェア開発はうまくいかない
 - b. 開発がうまくいかないのは何故か?
 - c. ソフトウェア開発の複雑さ
2. 良いソフトウェアとは?
 - a. ITソリューションとは?
 - b. ソフトウェア開発の三つの要素
 - c. ソフトウェアの品質とその種類
3. どうやれば良いソフトウェアを楽に作れるのか?
 - a. ソフトウェア開発を成功させるための思考方法
 - b. ソフトウェア開発は複雑さとの戦い
 - c. アジャイル開発の五つの価値
 - d. シンプルに考えよう
 - e. シンプルに考えるコツ
 - f. ソフトウェア開発を成功させるための基本的な原則

アジェンダ 2

Ⅱ. オブジェクト指向設計

1. モデリングとは?
2. モデリングには視点が重要
3. プログラミングにおける設計の重要性
4. オブジェクト指向入門
 - a. オブジェクト指向以前のやり方
 - b. オブジェクト指向のやり方
5. 設計のコツ
 - a. わかりやすく綺麗なプログラムを書くコツ
 - b. なぜ名前付けが重要か?
 - c. 重要なのは、「どう作るか」ではなく「何を作るか」に視点を変えること
 - e. UML を使ってみよう
 - f. テストを行う意味とは?
6. 設計の実習

Ⅲ. エンジニアとして能力を伸ばすためには

1. 問題解決能力を高めるには
2. エンジニアとして過ごす「人生の時間の質」

I . ソフトウェア開発の難しさ

a. ソフトウェア開発は
うまくいかない

ソフトウェア地獄からの脱出

■ Software Hell



ソフトウェア危機

- ハードウェア・ソフトウェアの進化によりユーザーの要求が高度化しシステムが複雑化
 - マルティメディア＋ネットワーク＋リアルタイム
- 従来の手法では、開発要員・開発期間が掛かりすぎる
 - 開発コストの増大
 - 1000人以上・1000万ステップ以上
- 開発規模の拡大にともなうソフトウェアの品質低下
 - 中華航空機・もんじゅ・Y2K
- 保守のコストが増大している
 - 大手では80%以上の要員が保守



ソフトウェア危機の背景

- 変化するコンピュータ環境
 - メモリ容量・ハードディスク容量が千倍に
- 変化するソフトウェア環境
 - ユーザーの多様化・適用業務の多様化



プロジェクトは失敗する



プロジェクトの失敗の現状

開発コスト

| | |
|------------|-------|
| 20% 以下 | 15.5% |
| 21 - 50% | 31.5% |
| 51 - 100% | 29.6% |
| 101 - 200% | 10.2% |
| 201 - 400% | 8.8% |
| 400% 以上 | 4.4% |

平均 189%

納期

| | |
|------------|-------|
| 20% 以下 | 13.9% |
| 21 - 50% | 18.3% |
| 51 - 100% | 20.0% |
| 101 - 200% | 35.5% |
| 201 - 400% | 11.2% |
| 400% 以上 | 1.1% |

平均 222%

仕様充足率

| | |
|----------|-------|
| 25% 以下 | 4.6% |
| 25 - 49% | 27.2% |
| 50 - 74% | 21.8% |
| 75 - 99% | 39.1% |
| 100% | 7.3% |

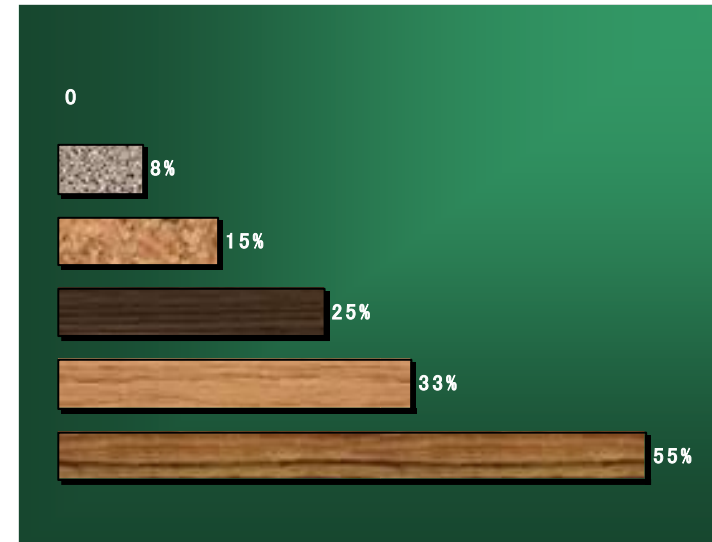
平均 61%

多くのプロジェクトは失敗している

プロジェクトの失敗の現状

プロジェクトの規模と成功率

| | | |
|--------|-------|-----|
| 500人以上 | 36月以上 | 0% |
| 250 | 24 | 8% |
| 40 | 18 | 15% |
| 25 | 12 | 25% |
| 12 | 18 | 33% |
| 6 | 6 | 55% |



大規模プロジェクトほど失敗している

うまく行かなかった例

c. ソフトウェア開発の複雑さ

b. 開発がうまくいかないのは何故か？

ソフトウェア開発は何故難しい？

- 「顧客と開発者の双方がビジネスを理解している」ということ
 - 何故顧客のいう通りに作ったのに満足してもらえないか
- 「他人が作りたいものを代わりに作る」という難しさ
 - 顧客は何にお金を払いたいのか

プログラミングの難しさ

- 人間の思考方法
 - 人間の思考方法に合わない
 - 人間にものを頼むときと異なる頼み方
 - プログラムは思った通りには動かない
- 言語が特殊
 - まだまだ言語のレベルが低い
- そもそも難しい
 - プログラムでは解けない問題がある
 - 数学基礎論・コンピュータ科学などで「実用的な大きさのプログラムではプログラムが正しいことを証明できない」ことが証明されている
- 大きなプログラムは難しい
 - 100 × 10 行よりも 1000 行

リスク ～何が開発を妨げるのか～

- なぜ成功しないのか?
 - － コミュニケーション エラー
 - ユーザーの関与不足/不明確な要求
 - 開発管理に問題
 - 作った人にしかわからない
 - 頭の中にあるデザイン
 - － 部品化 (再利用化) のプロセスがない
 - 納期内に仕様を満たすことへの強い要求



2. 良いソフトウェアとは?

a. ITソリューションとは?

ソフトウェア開発の目的は何か？

ITソリューションとは

- 顧客の問題をITで解決
 - 本当にITで解決すべき問題か
- 「問題」とは
 - 望まれる状態と現状とのギャップ



b. ソフトウェア開発の三つの要素

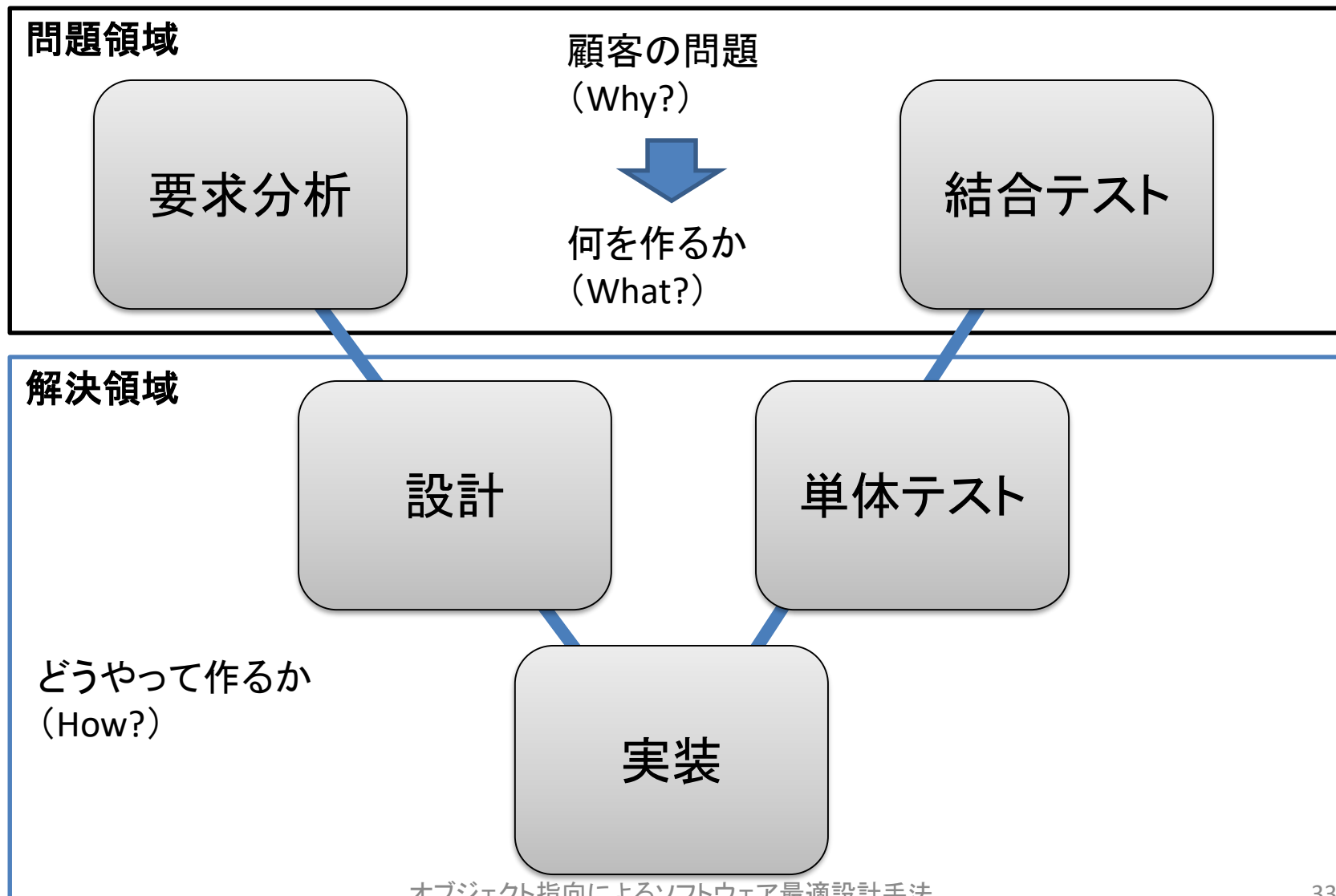
三大要素

- ビジョンと現状と問題
 - Problem = Tobe - AsIs
 - → ToDo (プラクティス)
- 「解決策が分らないのではない。問題が分っていないのだ」
 - チェスタートン

三大要素

- 顧客の問題にソリューションを与える
 - 「顧客の問題が、開発を駆動する」
- Why → What → How と考える

V字モデル



c. ソフトウェアの品質とその種類

ソフトウェアの品質

- 「ソフトウェアの品質とは、優れたプログラムが備えるべき七つの属性の集合である」
- 七つの属性を平均的なプロジェクトでの優先順にあげると以下の通り
 - 信頼性 (reliability)
 - 使用性 (usability)
 - 理解容易性 (understandability)
 - 変更容易性 (modifiability)
 - 効率 (efficiency)
 - 検証性 (testability)
 - 移植性 (portability)

外的品質要因

- 正確さ
 - 要求されたとおりに仕事を行えるか
 - ワープロが要求仕様通りに動く
 - バグが少ない
- 頑丈さ
 - 異常な状態においても機能するか
 - メモリが少ない状態でも動作
 - 2000 年になっても動作する (Y2K)
 - この辺まではどの現場でも行われている
- 拡張性
 - 仕様の変更に容易に対応できるか
 - 要求仕様は変化する
 - ユーザーも当初は要求仕様がはっきりしていない



外的品質要因

- 再利用性
 - 新しい応用にどの程度再利用できるか
- 互換性
 - 別のソフトウェアとの組み合わせが容易か
 - OS の変更. MS-DOS -> Windows, Linux 等
 - 開発言語・環境の変更



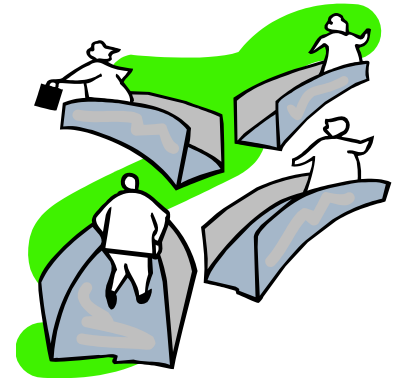
3. どうやれば良いソフトウェアを 楽に作れるのか？

a. ソフトウェア開発を成功させるための思考方法

b. ソフトウェア開発は複雑さとの戦い

システムの問題点 ～ 複雑さ(complexity) ～

- 要求の複雑さ
- ソフトウェア自身の複雑さ
- オブジェクト指向の複雑さ
 - オブジェクトは適切か
 - オブジェクトの責務は適切か



工学的アプローチによる複雑さへの対処

- 大人数をうまく分業化する
- 問題を分割し、適切に割り振る
- それぞれの分割単位を独立させる
- 複雑な現実世界のものをモデル化する
- 現実世界の問題をITでの解決領域にマッピングする
- 予測し、なるべく正確に計画する
- 変化する部分と変化しない部分を分離できるようにする

アジャイルのアプローチによる複雑さへの対処

- プロジェクトを大人数にしない
- チームに利害関係者を入れてしまう
- 「見える化」や「フェイス・トゥ・フェイスなコミュニケーション」で、チーム内のコミュニケーションの帯域を広げ(＝ブロードバンド・コミュニケーション)、問題を素早く単純なうちに解決する
- モデルやコードが複雑にならないようにする(＝YAGNI: You Aren't Going to Need Itの原則)
- 常にフィードバックを行い、ズレを俊敏に補正する

c. アジャイル開発の五つの価値

ていへんたいへんたいへん

シンパチカル

スーパーバミ

戦気

激怒

アジャイル開発の五つの価値

- 重要なのは価値にコミットすること
- 実践が必要

d. シンプルに考えよう

シンプルに考えよう

- 「キリンを冷蔵庫に入れるにはどうする？」
（“How do you put a giraffe into a refrigerator？”）

シンプルに考えよう

- 「キリンを冷蔵庫に入れるにはどうする？」
（“How do you put a giraffe into a refrigerator？”）
- 「冷蔵庫のドアを開けて、キリンを入れ、ドアを閉める」
（“Open the refrigerator, put in the giraffe and close the door.”）

シンプルに考えよう

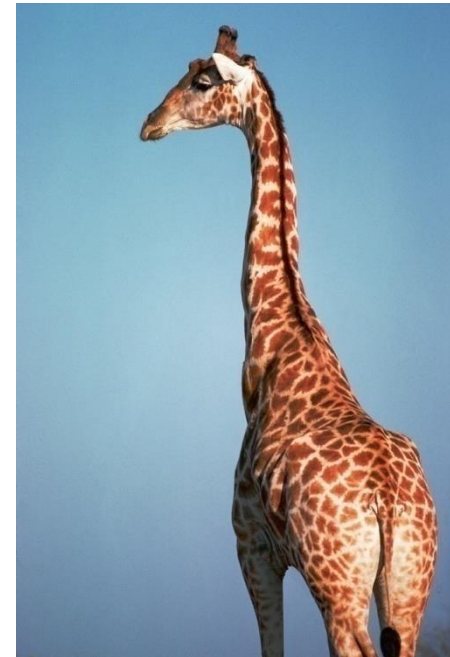
- 「ゾウを冷蔵庫に入れるにはどうする？」
（“How do you put an elephant into a refrigerator?”）
- 「冷蔵庫のドアを開けて、キリンを取り出し、ゾウを入れ、ドアを閉める」
（“Open the refrigerator, take out the giraffe, put in the elephant and close the door.”）

問題を複雑にしない

- 問題を複雑にする例:
 - 開発者A:「オブジェクト指向開発を現場に導入していきましょう」
 - 開発者B:「でもノウハウを持った経験者がいませんからね」
 - 開発者A:「最初は試行錯誤が続くかもしれませんが、これから少しずつ勉強をしながら経験を積んでいくべきかと」
 - 開発者B:「でもいまは忙しい時期ですし。それに上の人間はきっと反対しますよ」
 - 開発者A:「そうって先延ばししてはいいつまでたっても現状のままですから、徐々にやっていきましょうよ。実績を積んでいけば、上も説得できますし」
 - 開発者B:「徐々にじゃ、なかなか効果が見えてこないですね」
- 前もってリスクを適切に評価するのは重要

「冷蔵庫にキリンの原則」 (Giraffe-Refrigerator Principle)

- 問題は複雑にしないで単純なまま解く。



サイバラの法則

- $1 + 1 = 2$
 $123 \times 456 = \text{たくさん}$
 $12 \div 34 + 56 \div 78 = \text{そんな問題を解かなければならない状況に自分を持っていけない}$

複雑さへの対処

- 仕様がたびたび追加になり、プログラムがどんどん複雑化
- バグがいっぱいあってどうにも手が付けられない
- ものすごい工数が掛かってしまうので何とかしたい

「そんな複雑な状況に
自分を持っていかない」

「そんな複雑な状況に自分を持っていかない」

- そもそも複雑にしない
- 顧客レビューを常に行う
 - 早いうちに変化を受け入れる
- 設計や実装も常にレビュー
 - ペア・プログラミング、コードの共同所有
- 常にテスト
 - テスト駆動開発、常時結合
- リファクタリング

「サイバラの原則」 (Saibara's Principle)

- 「複雑な問題を扱わなければならないような状況に自分を持っていかないようにする」

e. シンプルに考えるコツ

シンプルでない考え方の例:

- この複雑な問題をどうやって解こうか？
- 将来の複雑な問題にどうやって立ち向かっていくべきだろうか？
- どうすればよいのだろうか？
- 考え得る解決策をすべて列挙して、そのすべてをよく検討しよう！
- 障害になるものをすべて列挙し、そのすべてについて対策を練ろう！

シンプルに考えるには:

- 複雑な問題をどうやって解こうか?
→ もっと問題を単純にするにはどうすればよいか?
- 将来の複雑な問題にどうやって立ち向かっていくべきだろうか?
→ 今後どうやれば問題を複雑にしないで済むだろうか?
- どうすればよいのだろうか?
→ 何をやろうか?
- 考え得る解決策をすべて列挙して、そのすべてをよく検討しよう!
→ 現在ある解決策を少しずつ実際に試してみよう。実践結果からのフィードバックによって次の手を調整しよう!
- 障害になるものをすべて列挙し、そのすべてについて対策を練ろう!
→ できない・やらない理由ばかり列挙せずに、まずはやり始めて、その結果からフィードバックを得よう!

例題. 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

C#

```
class Figure // クラス
{
    int data = 0; // フィールド

    public void Draw() // メソッド
    {
        // 描画処理
    }
}
```

```
class DrawToolProgram
{
    static void Main() // メイン メソッド
    {
        // クラスのインスタンス (=オブジェクト) の生成
        Figure aFigure = new Figure();
        aFigure.Draw();
    }
}
```

C# 補足 (C++ との相違)

• C# のプロパティ

```
class Movie
{
    private string title;

    // プロパティ (getter 及び setter)
    public string Title
    {
        get { return title; }
        set { title = value; }
    }
}

class MainClass
{
    static void Main(string[] args)
    {
        Movie movie = new Movie();
        movie.Title = "マトリックス レボリューションズ";
        string title = movie.Title;
    }
}
```

■ C++ の場合

```
class Movie
{
private:
    std::string title;
public:
    // getter
    std::string GetTitle() const
    { return title; }

    // setter
    void SetTitle(std::string text)
    { title = text; }
};

int main()
{
    Movie        movie;
    movie.SetTitle("マトリックス レボリューションズ");
    std::string title = movie.GetTitle();
    return 0;
}
```

f. ソフトウェア開発を成功させるための基本的な原則

ソフトウェア開発の二つの攻略法

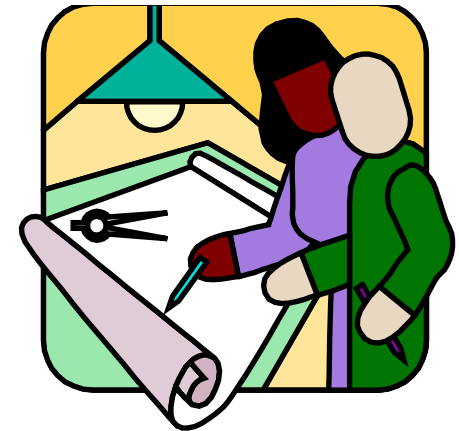
- Divide and Couquer
(分割攻略)
- Name and Conquer
(定義攻略)

Ⅱ．オブジェクト指向設計

1. モデリングとは?

モデリングとは

- 現実世界のモノ・現象から必要な事項を抽出すること
 - 何か目的があって、その目的に沿った部分だけを抽出すること
- 複雑なものを単純に
 - プログラム
 - ビジネス - 業務内容
- 設計図
- 抽象化のプロセス



モデルとは

- 物理学などでいうモデルと同じ
- 例. 物体の運動を考える場合:
 - 「リンゴを上向きに放ったら、どういう動きをするか？」
- 問題をシンプルにする
 - リンゴの色、味、買った場所などは排除
 - 運動に関係のある、リンゴの質量とか放ったときの速さなどだけを考える
 - 関係のないことを省いて思い切りシンプルにした「物体が運動するときに起こること」を考える
- それがモデル

モデルとは

- 社員名簿アプリケーションの例:
- 社員クラス
 - 持たせる属性は、現実の社員が持つとる情報の一部
 - 現実の社員が、「好きな食べ物」「好みの音楽」「好きなスポーツ」のような情報を持っていたとしても、そのシステムのユーザーの視点からそれらの情報が不要なら入れない

モデルとは

- 「関心の外のものを取り去ってシンプルにしたもの」
- 「関心の分離」
- 関心事だけを考える
 - 複雑さの排除
- 関心事だけを伝える
 - S/N 比の向上
 - 「詳細に伝えるのと正確に伝えることは違う」

モデリング

- 図で描くとしたら? もっとも意図が伝わるシンプルな図は?
- 頭の中のモデルにもっとも近いものは?
 - → 分かりやすさ。
- C# で書いてもモデル
 - C#でモデル駆動開発 (Model Driven Development)
- 設計と実装を対応させる
 - 「自分の設計モデル」と「自分の実装コード」が乖離?
- シンプルであること → 意図以外のことを排除 → モデル化

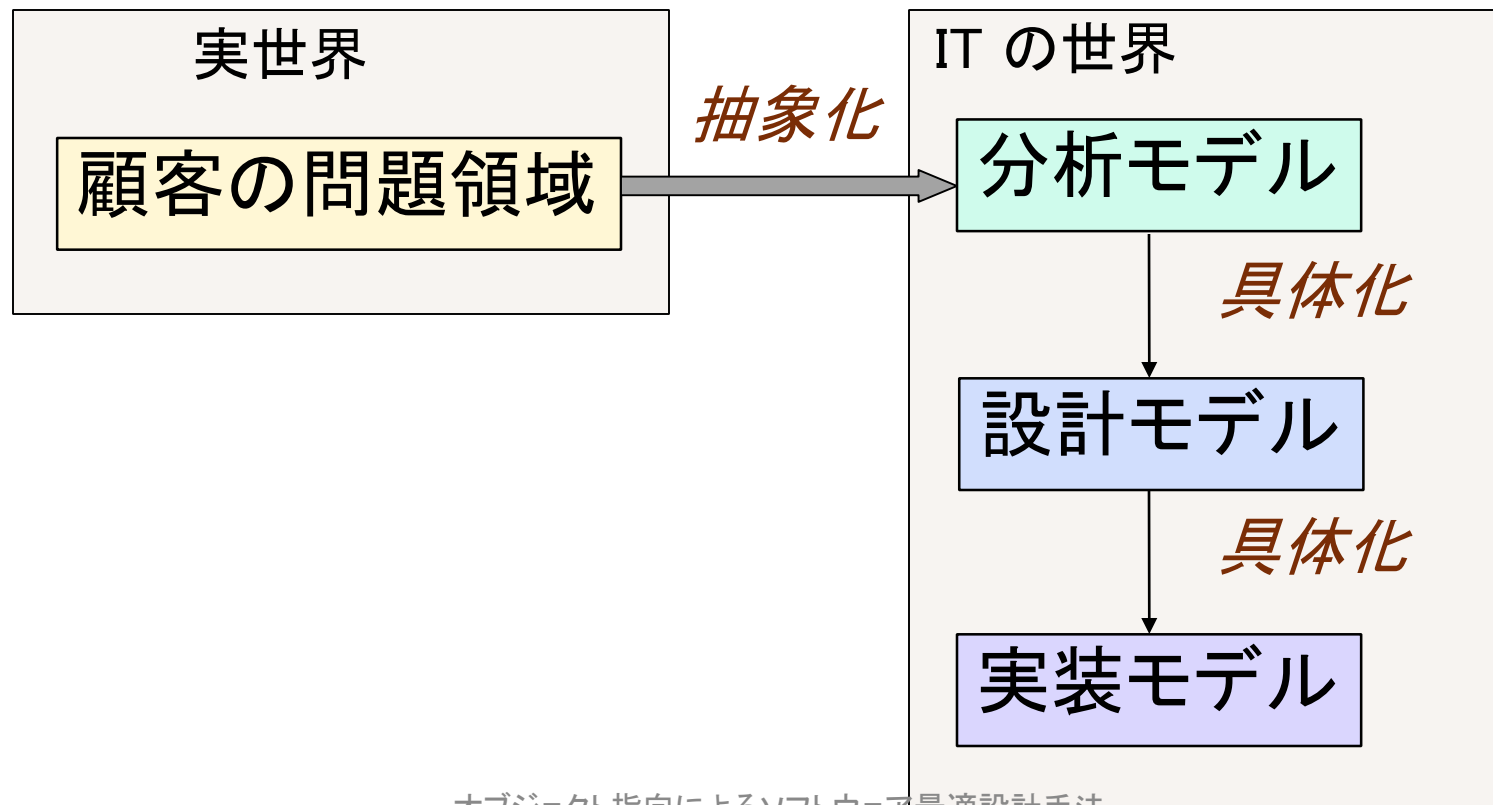
キー概念

- 抽象化の概念
 - 最も基本的な概念で、大規模なプログラムをより単純に記述できるように
 - プロシージャの抽象化
 - データの抽象化
 - クラス
- カプセル化の概念
 - プログラムの変更とメンテナンスを簡単に
- クラス階層の概念
 - プログラムを簡単に拡張できるように

2. モデリングには視点が重要

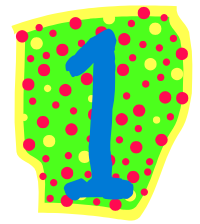
モデル化とは

■「IT ソリューション」の場合

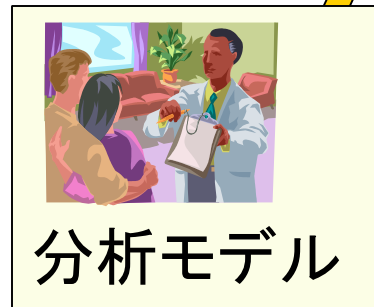
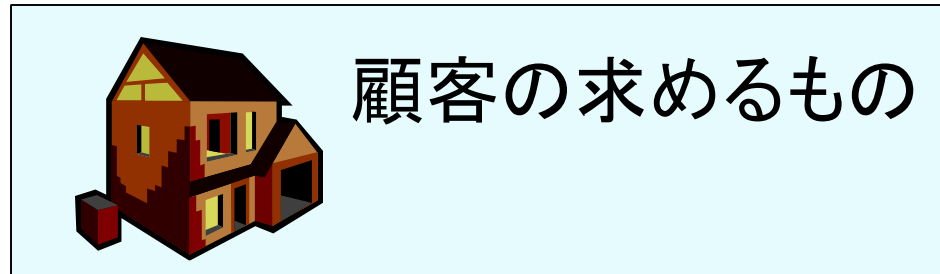


モデルはひとつ

- 顧客の問題への解決方法をモデル化
 - 分析モデル
 - 設計モデル
 - 実装モデル
- 視点 (ビュー) の違い
 - 「モデルはひとつ」



モデルはひとつ



分析者の視点



設計者の視点



実装者の視点

オブジェクト指向によるソフトウェア最適設計手法

視点の高さ

- 鳥の視点、虫の視点
 - 例.
 - ソフトウェア工学は鳥の視点
 - テーラリングは虫の視点

モデリングの上達のためには

- 良いモデルを見る



3. プログラミングにおける 設計の重要性

初期設計 V.S. リファクタリング

- 初期設定の必要性
 - アーキテクチャドリブン
- アジャイル モデリング
 - アジャイルな設計
- リファクタリング
 - テスト→実装→設計



リファクタリング

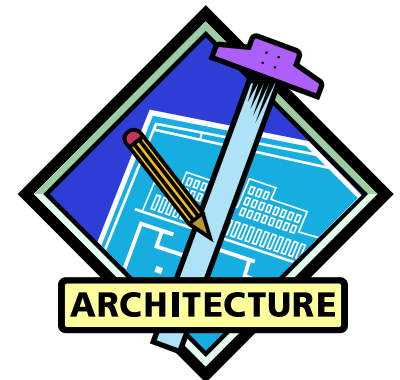
- テキスト「リファクタリング編」へ

ソフトウェア アーキテクチャ

アーキテクチャの重要性

コア アーキテクチャ

アーキテクチャ パターン



ソフトウェア・アーキテクチャ

- ソフトウェア・アーキテクチャとは
 - ソフトウェアシステムの設計思想を明確に反映したソフトウェアの構造を表現するもの
- もっとオブジェクト指向的にいうと
 - ある要求をソフトウェアで実現する際に、重要となる設計思想に基づいて、オブジェクトを配置する場(field)を定義し、その場の責務、および場を形成しうるメカニズム、そして、場と場の協調関係を定義する
 - この場の責務をコンポーネント・フレームワークの構造として表現される
 - フレームワークを使う場合も、そのフレームワークの持つアーキテクチャを理解する必要がある

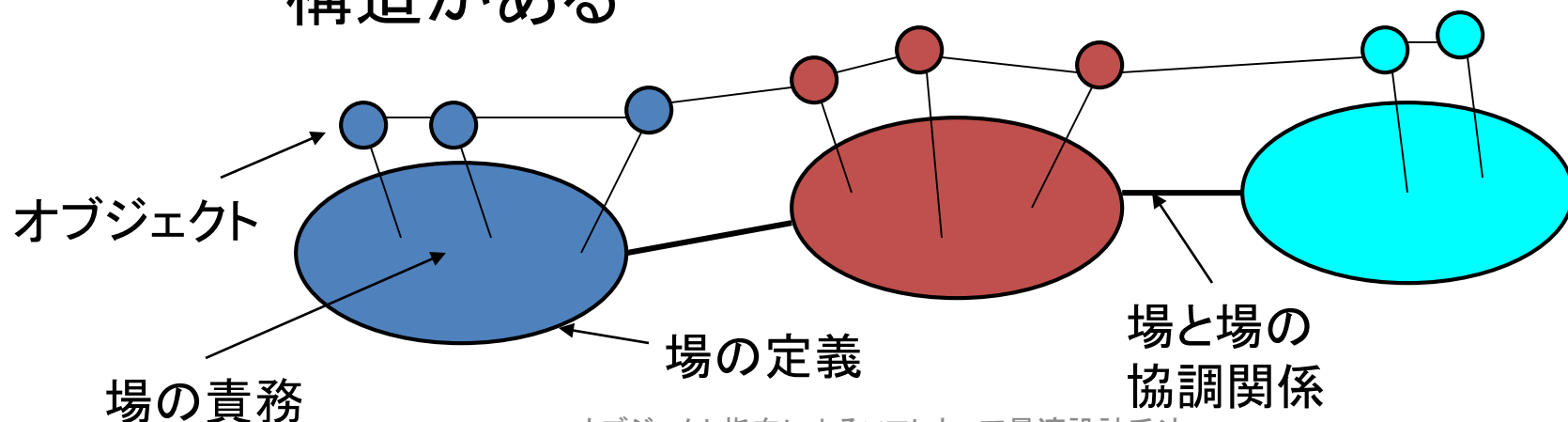
ソフトウェア・アーキテクチャ不在

- 問題点

- 設計の初期段階から細かなクラスの構造に着目しすぎるのでは?
- 個々の設計はうまくいっているが全体的な設計に一定のポリシーが見当たらないような気がする
- システム全体の見通しが悪い感じがする
- システムの重要なアーキテクチャは何か誰も語れない

ソフトウェア・アーキテクチャの重要性

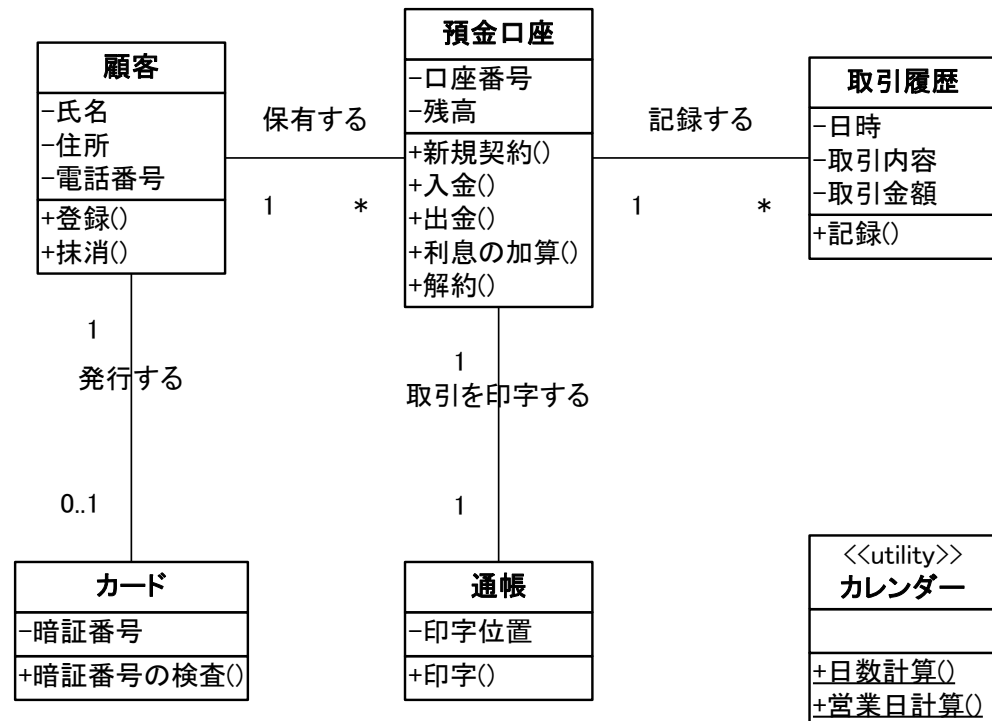
- システムの大局的視点を重視する構造とメカニズムの把握
- 場の定義があつてこそオブジェクトの存在価値が評価できる
- コンポーネントやオブジェクトが存在する場にも構造がある



オブジェクトの存在する場

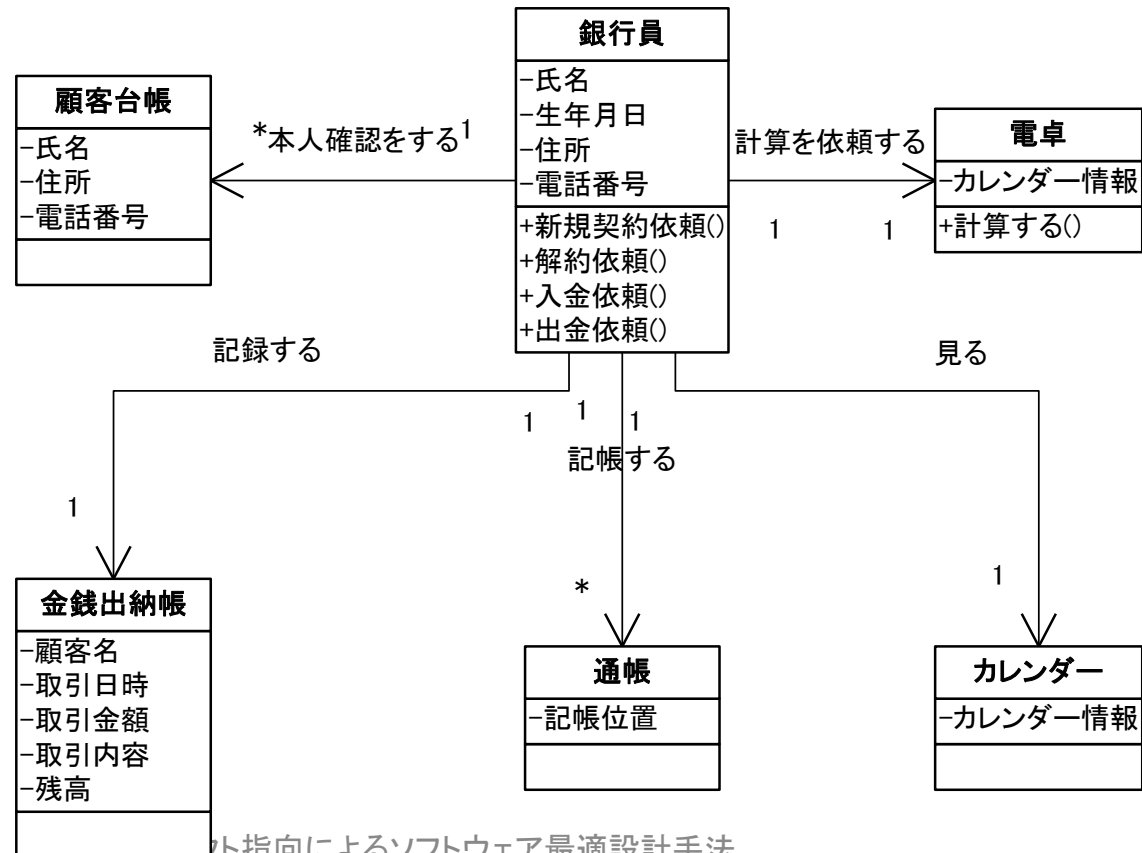
- オブジェクト指向モデリングでは実世界をそのままオブジェクトにマッピングするというが

よくあるモデリング



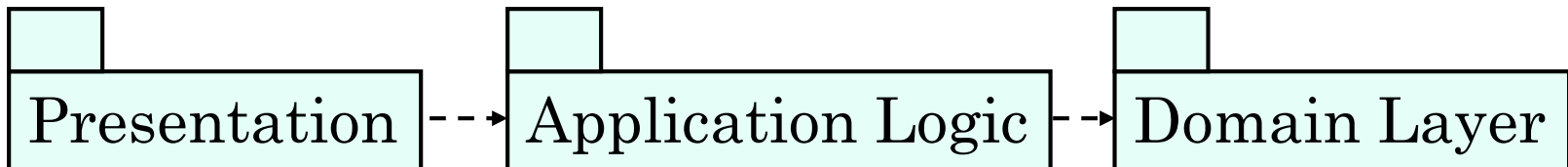
オブジェクトの存在する場

実世界をそのままマッピング?



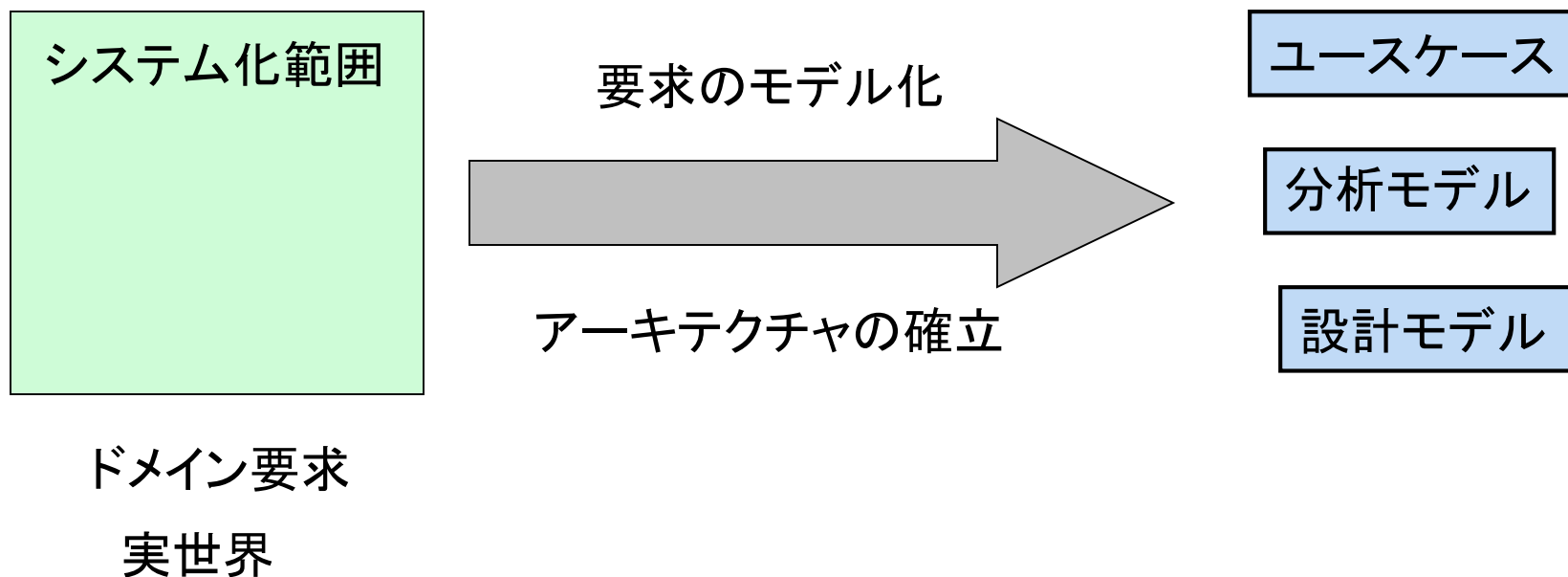
ソフトウェアアーキテクチャの表記法

- 場の定義・構造
 - パッケージ図を使う
 - パッケージは概念的なグループを示す



ソフトウェア開発の初期段階

- 要求のモデル化
- アーキテクチャモデルの確立



コア アーキテクチャ

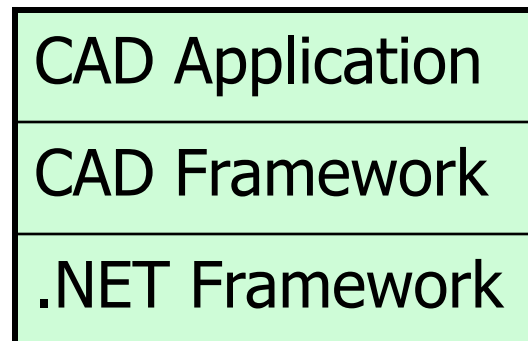
- レイヤ分割
 - ある要求をソフトウェアで実現する際に、最も重要となる設計課題に基づき、オブジェクトを配置する「場」を定義し、その「場」の責務、「場」を形成するメカニズム、「場」と「場」の協調関係を定義する.
- コンポーネント
 - 各「場」に配置されるコンポーネントの責務と相互作用を定義.

設計課題

- 機能要求
 - ユースケース モデルによって表現
 - ユーザーから見えるシステムの提供するサービス
- 非機能要求
 - システム化に対する機能要求でない要求
 - ここは再利用可能にして欲しい
 - この部分は将来拡張可能にして欲しい
 - 既存システムと同じミドルウェアを使って欲しい
- 非機能要求が重要
 - 非機能要求によってアーキテクチャ ベースラインが確定
 - アーキテクチャ ベースラインを確定しておかないと, システム境界が判らなくなる

場の定義（横分割）

- レイヤ分割
 - 抽象化のレベルで分割
 - 再利用性の高いのどのレイヤ？
 - 抽象化による再利用性
 - クラス: オブジェクトの抽象化によるオブジェクトの再利用
 - パターン: モデルの抽象化によるモデルの再利用



場の定義（縦分割）

- 論理ティア
 - 論理的なオブジェクトの配置空間
 - MVC (View, Controller, Model)
 - View: 表示を制御するクラス
 - Controller: 要求に応じて Model を制御するクラス
 - Model: ドメイン (ビジネス ロジックとエンティティ) のクラス
 - BCE (Boundary, Control, Entity)
 - Boundary: システムの境界線上に位置するクラス
 - Control: 制御に関わるビジネス ロジックを持つクラス
 - Entity: システムの実体 (永続データ) を持つクラス

場の定義（縦分割）

- 物理ティア
 - Web 層
 - ビジネス ロジック層
 - データベース層

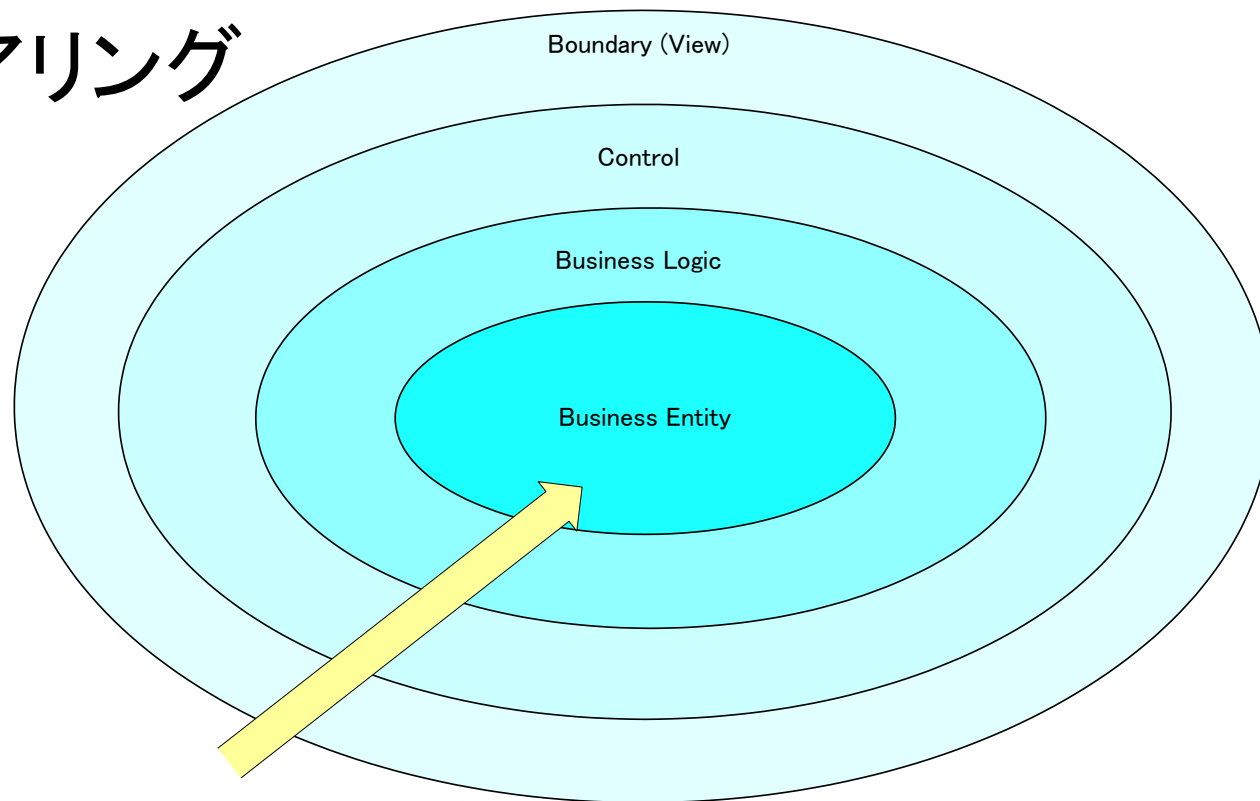


場の定義（縦分割）

- 特性によって開発担当者をティア毎に分けることが可能
 - Boundary: 画面デザイナー
 - Control: ビジネス ロジックに詳しい開発者
 - Entity: データベースに詳しい開発者
- 変更箇所の局所化
 - Boundary — Control — Entity
 - 多 <-----> 少（変更頻度）

場の定義

- レイアリング



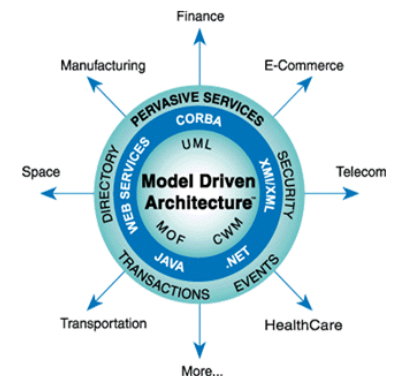
内側に向かう程

- ・分析対象
- ・業務レベルの再利用性を重視

オブジェクト指向によるソフトウェア最適設計手法

モデル駆動開発

- MDA (Model Driven Architecture: モデル駆動開発)
 - PIM (Platform Independent Model: プラットフォームに依存しないモデル)
 - PSM (Platform Specific Model: プラットフォームに特化したモデル)



実装における場の表現

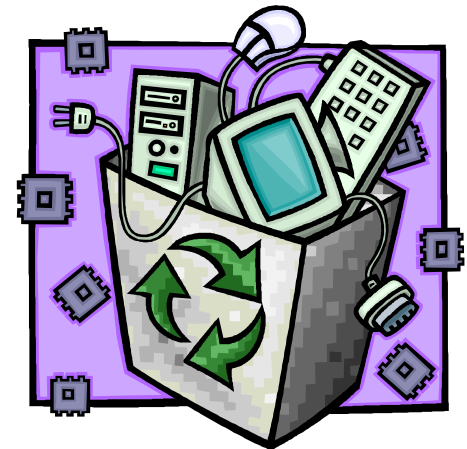
- C++/C#
 - namespace を使う
- Java
 - パッケージを使う

```
namespace NUnit.Framework;  
public class TestCase {  
}
```

```
package junit.framework;  
public class TestCase {  
}
```

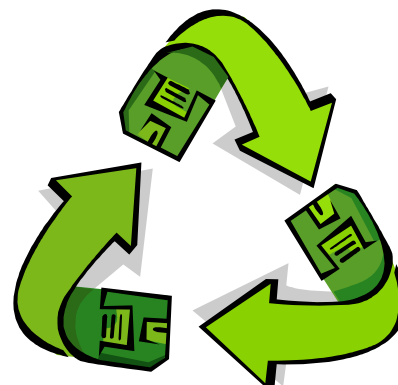
ソフトウェアの再利用

- ソフトウェアの再利用性の重要性
 - 短納期化
- ソフトウェアの再利用はどのようにすれば可能か？



再利用化

- 再利用は何故必要か
- 「オブジェクト指向」に「パターン」,
「フレームワーク」, 「コンポーネント」
 - どう違う?



ソフトウェアの再利用

- ライブラリ
 - フレームワーク
 - 横分割
 - コンポーネント
 - 縦分割
- 知識の再利用
 - 暗黙知→形式知



アーキテクチャパターン

- 特定の問題領域に現れるソフトウェア・アーキテクチャのデザインをパターン化したもの

アーキテクチャパターンの種類

| 分類 | パターン名 | 説明 |
|----------|--|---|
| システムの構造化 | Layers Pipe and Filters Blackboard | サブシステムを特定の抽象レベルに属するようにグループ化 データをストリームとして扱う 独立したプログラムを共通のデータ構造上で協調動作 |
| 分散システム | Broker | 分散システム上でクライアント・サーバー間の結合度を弱める |
| 対話型システム | MVC PAC | GUIとモデルを分離する 複雑な意味的概念を捉えやすくする |
| 適合化システム | Micro Kernel Reflection | 中核となるサービスを顧客依存部分から独立 言語の型に依存せずにシステムの構造と振る舞いを動的に変更 |

アーキテクチャパターン比較

| | 可視化 | 制御 | データ管理 | 効果的なドメイン |
|---------------|-----|----|-------|------------|
| Smalltalk MVC | | | | 対話型・制御系ツール |
| PAC | | | | 複雑なシステム |
| BCE | | | | データ処理系 |
| PADD | | | | 大規模 |
| J2EE MVC | | | | 大規模データ処理 |

MVCアーキテクチャパターン

- 設計方針

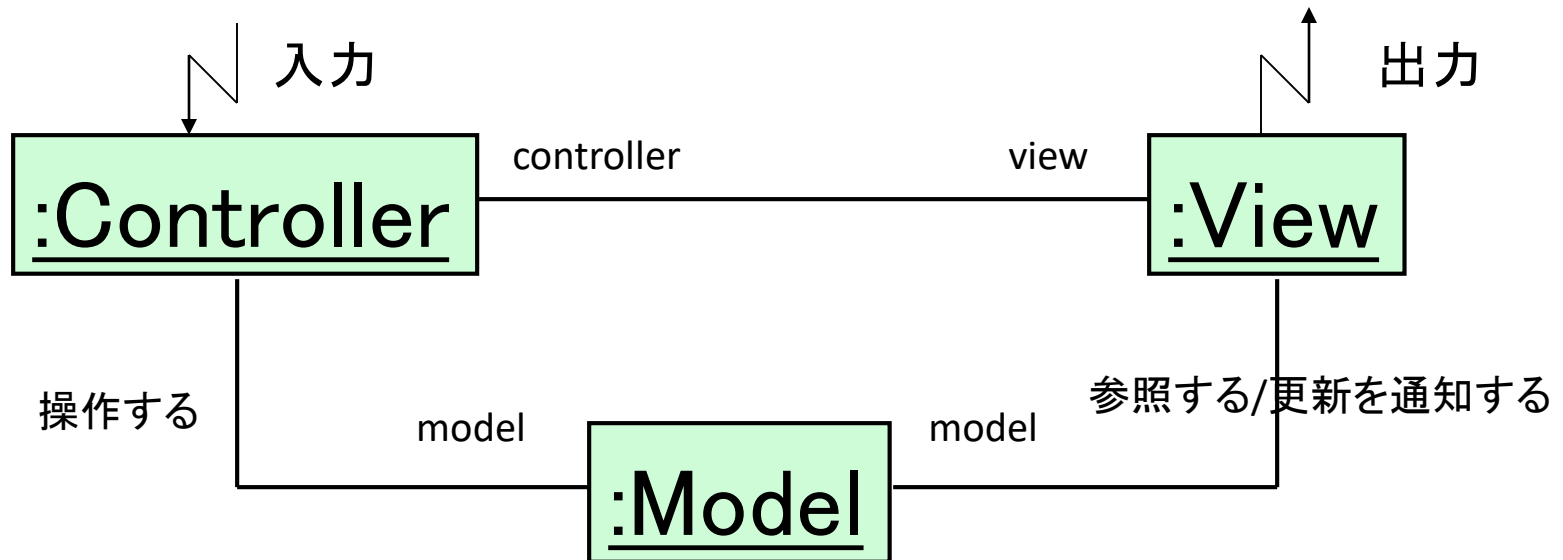
- 対話型システムを、アーキテクチャ上、「Model」と「View」、「Controller」の3つの領域に分ける
- Modelを見せ方(View)と操作(Controller)から分離
- Smalltalk-80によりモデルが確立

MVCアーキテクチャパターン

- **Model**
 - アプリケーションロジックをあらわす
- **View**
 - Modelを外部に表示する
- **Controller**
 - マウスやキーボードの入力を受け取りViewやModelに適切に伝達

MVCのメカニズム

- MVCでは、ViewとControllerが1対1に対応しており、1つのModelを覆っている



MVCパターンの利点

1. 一つのModel(アプリケーションロジック)に対して複数のGUIを提供することが可能
2. 複数のGUIは選択的に提供が可能であるだけでなく、同時に提供することが可能
3. GUI部分がアプリケーションロジックから切り離されているため、異なるプラットフォームへのアプリケーションロジックの移植が容易

MVCパターンの欠点

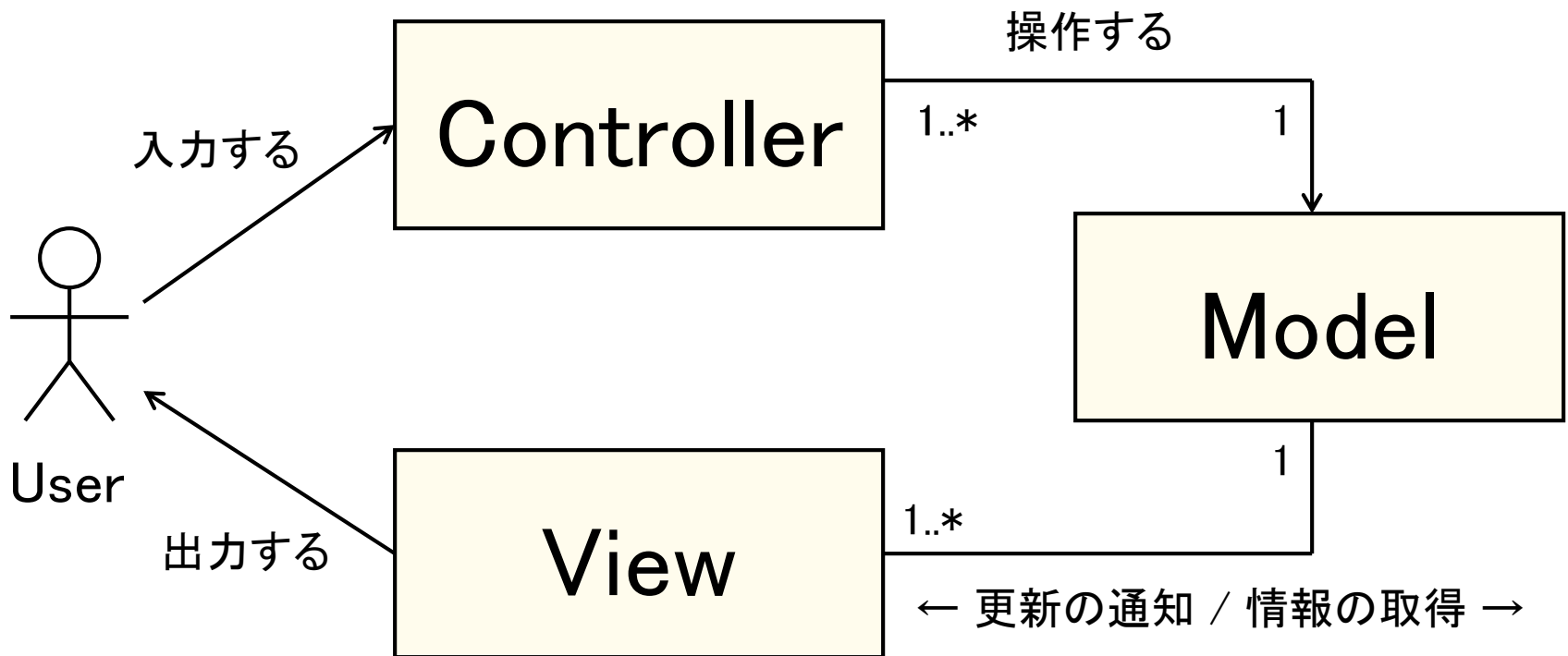
1. MVC構造を常に構築するため、簡易なGUIの場合には必要以上に複雑な実装になることがある
2. Modelが内部状態の更新伝播を熱心に行うと、GUIの書き換えが頻繁に発生する
3. ViewとControllerの依存関係が高く、個別の再利用は困難
4. ViewとControllerのModelへの依存が高く、Modelのインタフェース変更がViewとControllerに影響を及ぼしてしまう

MVCからの発展

- 現在のソフトウェア環境ではMVCはGUIフレームワークに隠蔽
- MVCは現在ではM-VCとして様々なシステムやツールの基礎的なアーキテクチャとして応用されている [Swing MVC やJ2EE MVC]
- これからは、もっとネットワークシステムを考慮したものや大局的な見地からシステム全体にアーキテクチャを見出せるアーキテクチャモデルが必要とされる
 - PAC パターン
 - Layers パターン
 - BCE,PADD

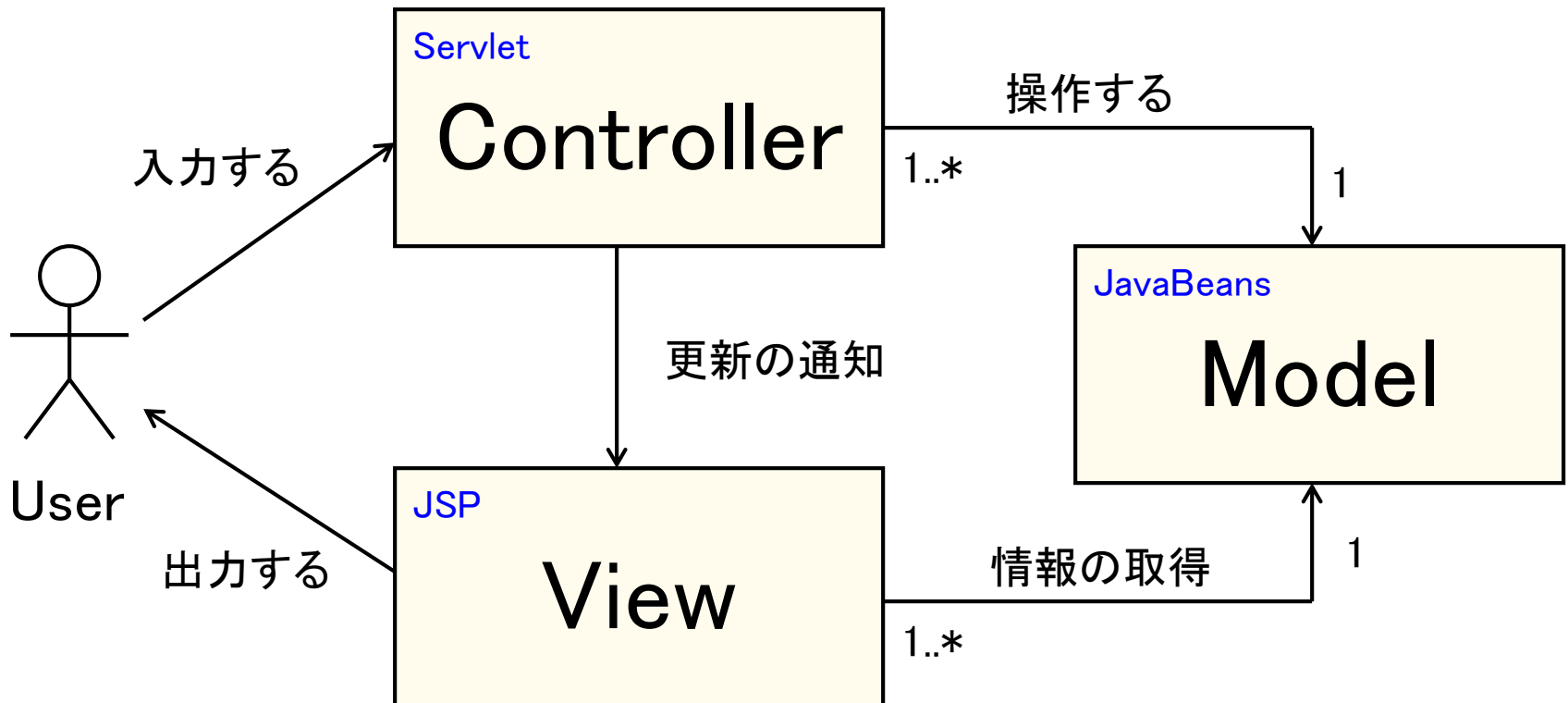
MVC0 と MVC2

- 古典的な Smalltalk MVC

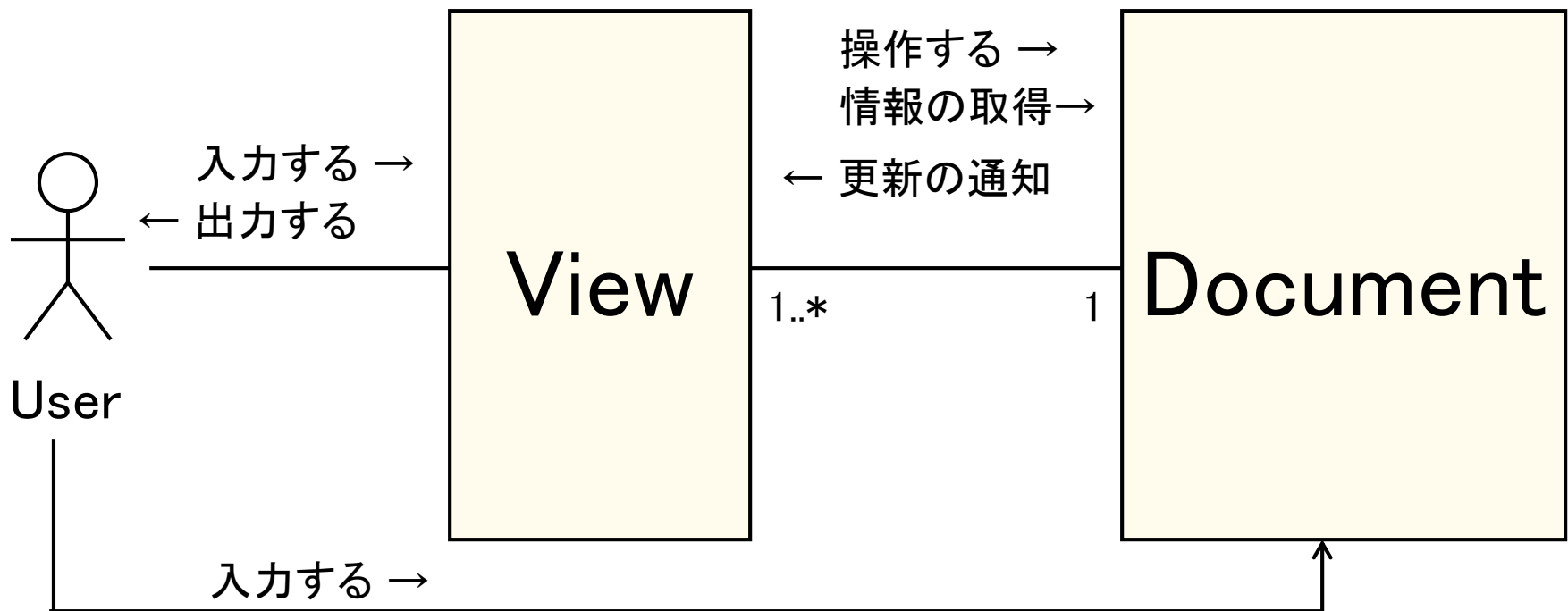


MVC2

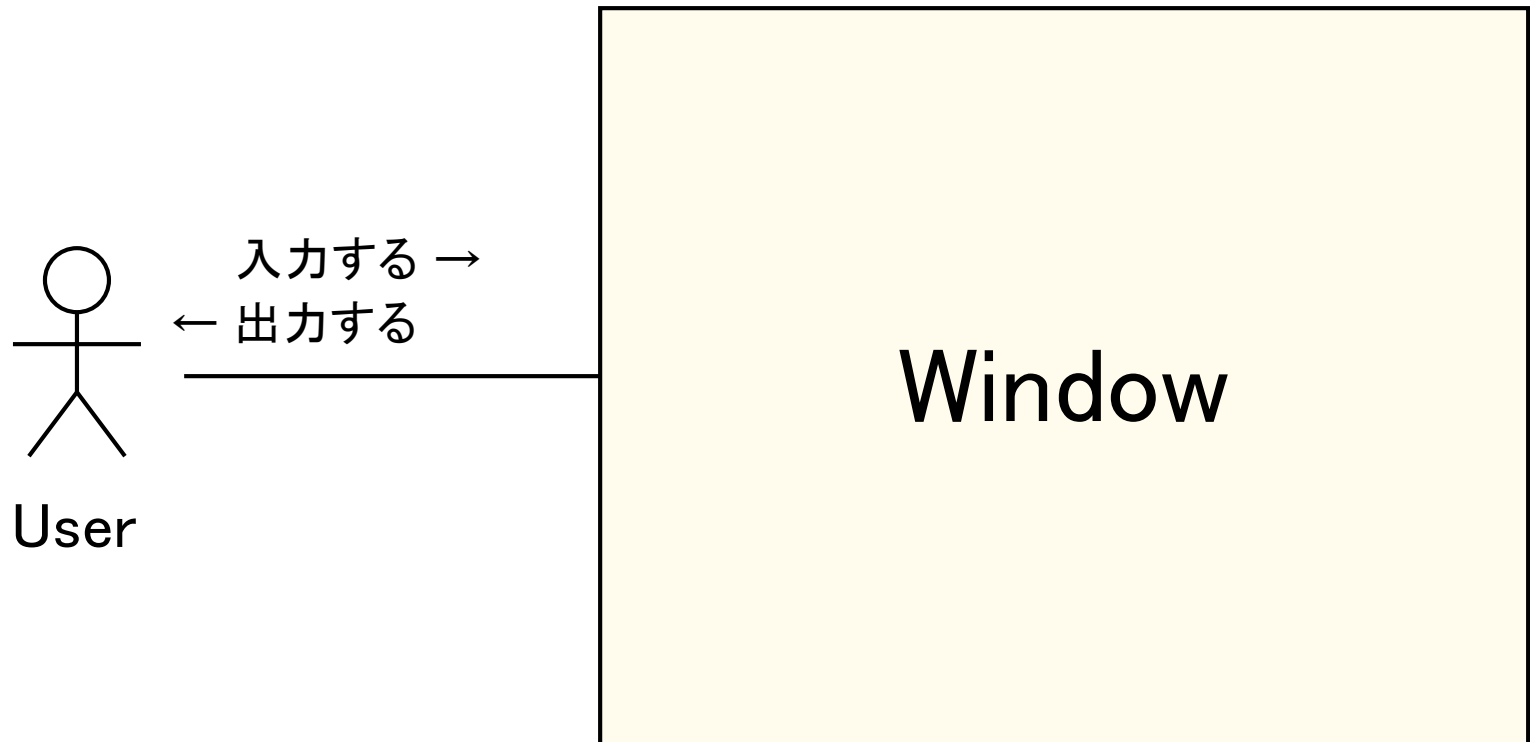
- J2EE の MVC



MFC (Document–View) の場合



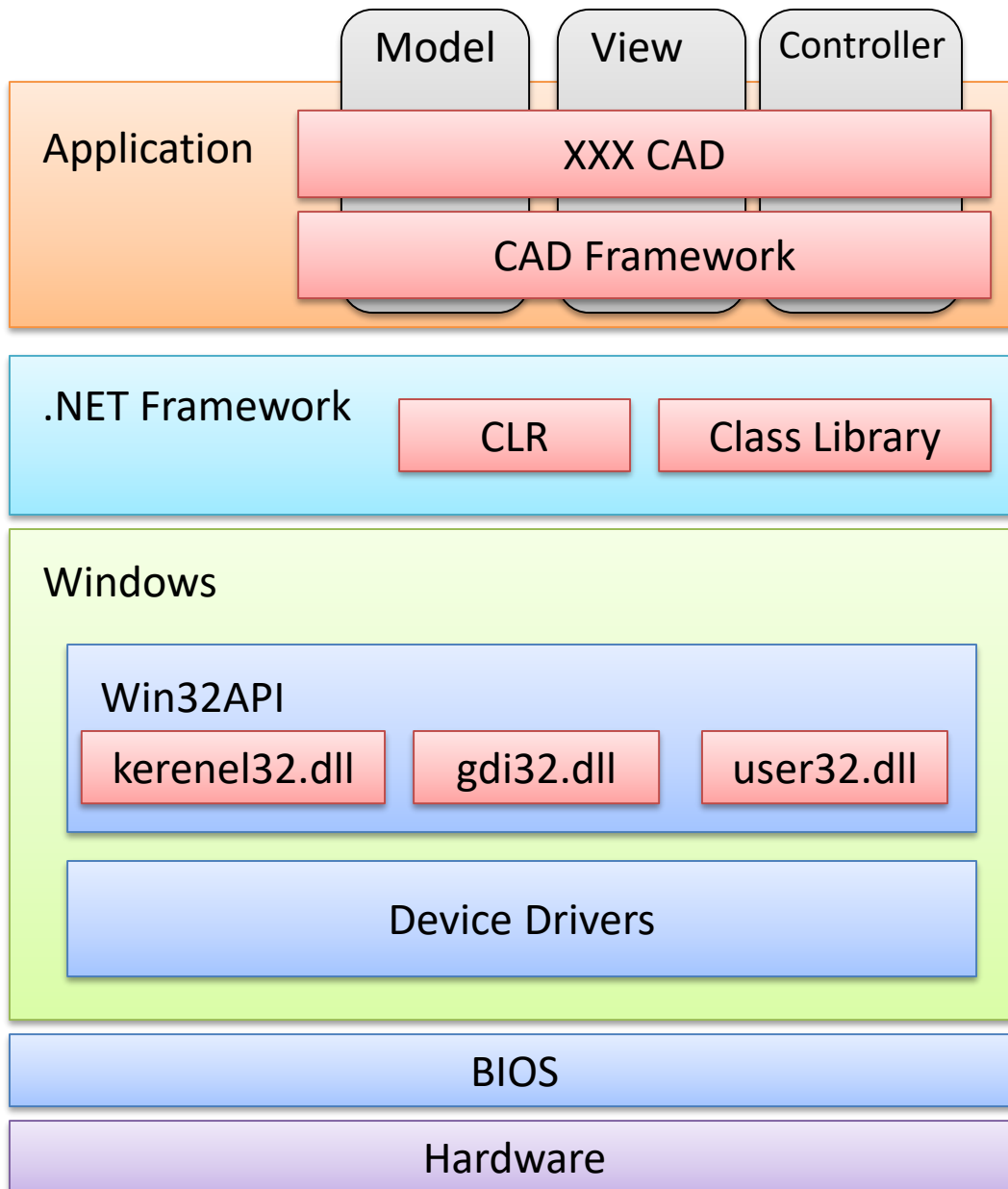
MFC (Non Document–View)



Layers パターン

- 設計方針
 - ソフトウェア・システムを特定の抽象レベルに属するサブタスクのグループに分割する
- 典型例
 - OSI 7層ネットワーク・モデル
- 基本メカニズム
 - 重要な点は、隣接する下位レイヤ(抽象度が低い)のサービスのみしか利用できない
 - 各レイヤが下位(抽象度の低い)にあるすべてのレイヤのサービスを使用できるRelaxed Layered System

| OSI参照モデル | 例 |
|--|--------------------|
| <p>第7層 - アプリケーション層</p> <p>具体的な通信サービス(例えばファイル転送、メール転送、遠隔データベースアクセスなど)を提供。</p> | HTTP、FTP、SMTP、SOAP |
| <p>第6層 - プレゼンテーション層</p> <p>データの表現方法。</p> | |
| <p>第5層 - セッション層</p> <p>通信プログラム間の通信の開始から終了までの手順。</p> | |
| <p>第4層 - トランスポート層</p> <p>ネットワークにおける通信管理(エラー訂正、再送制御等)。</p> | |
| <p>第3層 - ネットワーク層</p> <p>ネットワークにおいて通信経路の選択。</p> | IP |
| <p>第2層 - データリンク層</p> <p>直接的に接続されている通信機器間の信号の受け渡し。</p> | イーサネット、xDSL、無線LAN |
| <p>第1層 - 物理層</p> <p>電気信号の変換、コネクタ形状等。</p> | |



Layers/パターンの応用例(1)

- ヤコブソンのBCE (Boundary- Control- Entity)
- Control (制御) はBoundary (システム境界) とEntity に分析ステレオタイプとして分類

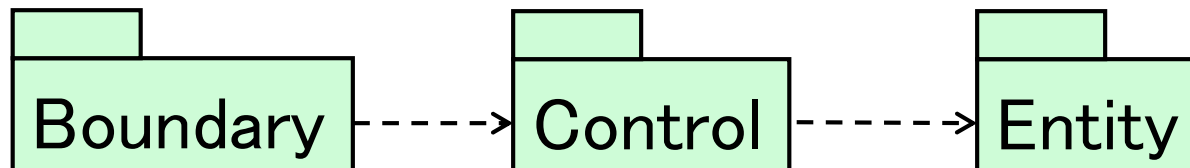
BCE

- **Boundary**
 - システム外部との間の対話を受け持つ
- **Control**
 - ユースケースで規定された振る舞いを実行するためにBoundaryから呼び出され、いくつかのEntity をアクセスしてデータを加工しBoundary に送り返す
- **Entity**
 - システムに長時間存続するオブジェクト
 - 問題領域をシステム化するうえで最も本質的なサービスだけに着目して切り出されたオブジェクト(データ処理が中心のアプリケーションの場合、永続オブジェクトが典型的な例)

BCEの特徴

- BCEの利点

- システムを問題領域に対して最も変化の激しいBoundary最も安定しているEntityそしてその中間に位置するControl に分類することによりソフトウェアの変更／拡張範囲を局所化
- 対話型システム以外にも適用できるよう配慮されている
 - Boundary は、MVCのView とは違い、システムの可視化を担当するのではなく、システムとシステム外部とのインタフェースを担当

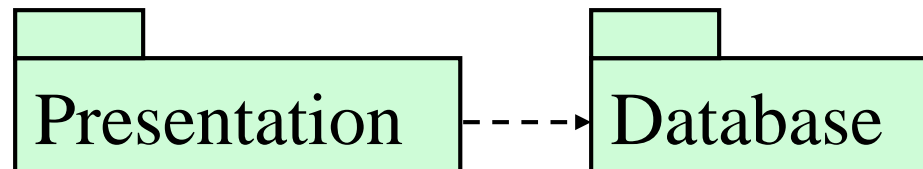


BCEとMVCの違い

- MVCでは問題領域のオブジェクトをModelで表す。BCEでは、MVCのModel部分を、Entity(永続データ管理)とControl(制御)に分類する。
- MVCのViewとControllerが、BCEのBoundaryに相当
- BCEのControl(制御)的な役割を持つコンポーネントの分類は、MVCに存在しない

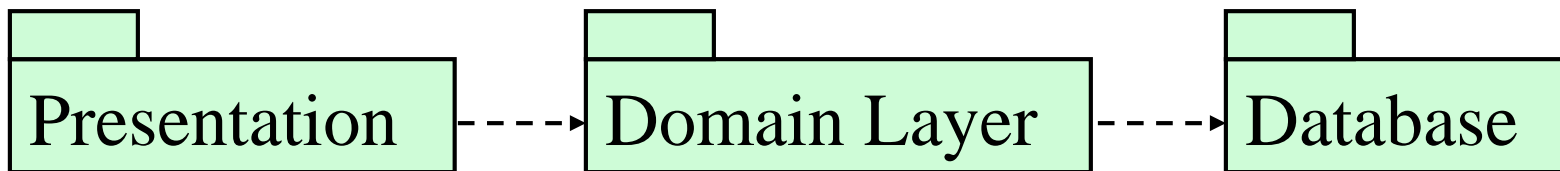
Layers/パターンの応用例(2)

- ネットワークシステムにおけるレイア分割
 - その1(2層アーキテクチャ)
 - プレゼンテーション層と、データベース層を分類した2層からなるレイア・アーキテクチャを採用したクライアント・サーバーシステム
 - 2層アーキテクチャでは、複数の異なるアプリケーションがプレゼンテーション層となり、下位のデータベース・サービスを受けながら並列に動作
 - 問題
 - ユーザインタフェースとデータ表現が直接的に結合しあう



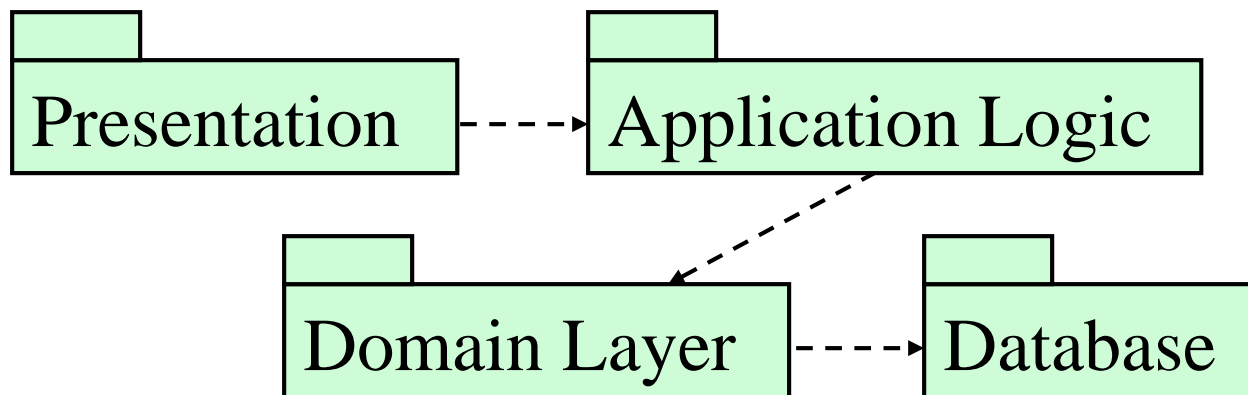
Layers/パターンの応用例(2)

- ネットワークシステムにおけるレイア分割
 - その2(3層アーキテクチャ)
 - プレゼンテーション層と、データベース層の間に問題領域の概念構造をモデル化するためのレイア(Domain Layer)を設ける
 - 問題
 - 新たにレイア化されたDomain Layerとアプリケーションの依存度が高く再利用がきかない



Layers/パターンの応用例(2)

- ネットワークシステムにおけるレイア分割
 - その3(4層アーキテクチャ)
 - アプリケーション固有の処理を行うApplication Logicを置いてドメインレイアの他のアプリケーションでの再利用を高める

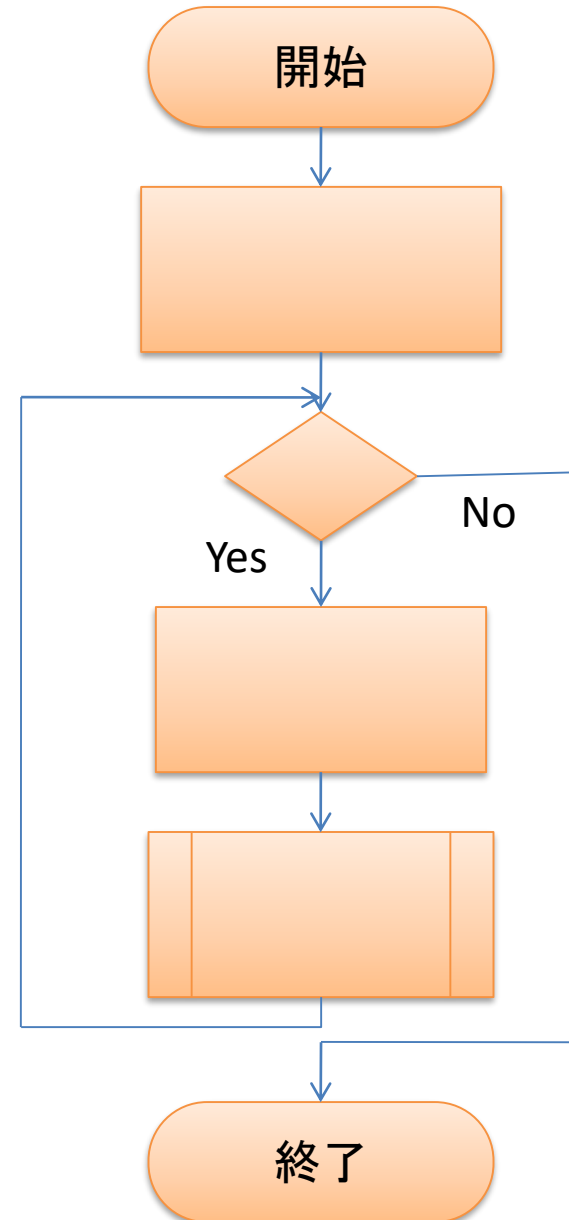


4. オブジェクト指向入門

a. オブジェクト指向以前のやり方

手続き型

- 構造化手法
 - 順次実行
 - 条件分岐
 - 繰り返し
- 機能の粒度
 - ブレークダウン

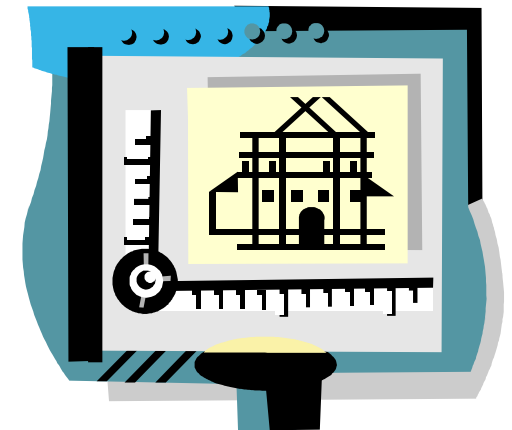


b. オブジェクト指向のやり方

- オブジェクト単位
 - システム上意味がある「もの」「こと」
 - 属性と操作を持つ
 - 他のオブジェクトと関係を持つ
- 関係のあるオブジェクト同士がメッセージを投げ合う
 - コラボレーション

オブジェクト指向分析・設計

- OOA (Object Oriented Analysis)
 - 要求モデルを開発
 - ユーザーの視点
 - What
- OOD (Object Oriented Design)
 - 詳細および設計判断を追加
 - 開発者の視点
 - How
- 視点・情報量の違い



『オブジェクト脳の作り方』より

- オブジェクト指向のキー概念
 - オブジェクト
 - クラス
 - 継承
 - カプセル化
 - ポリモーフィズム

参考文献:『オブジェクト脳の作り方』牛尾剛著

『オブジェクト脳の作り方』より

- 先の五つのキー概念それぞれについて
 1. コンピュータの動作イメージやコードに置き換えずに説明してください
 2. コードに置き換えて説明してください
 3. メリットを説明してください

参考文献:『オブジェクト脳の作り方』牛尾剛著

オブジェクト指向理解度チェック

- 変数とオブジェクトの違いは?
- オブジェクトとインスタンスの違いは?
- オブジェクト指向のクラスと C# の class の違いは?
- オブジェクト指向の継承と C# の派生 の違いは?
- C でオブジェクト指向プログラミングをする場合、カプセル化、継承、ポリモーフィズムはどう実装する?

オブジェクト

- 非公式には、物理上・概念上・ソフトウェア上の対象物
 - 物理上の対象物「あるトラック」
 - 概念上の対象物「ある化学反応」
 - ソフトウェア上の対象物「あるリンクリスト」
 - 設計時に出現



オブジェクト

- アプリケーションにとって意味があり, 明確な境界をともなう, 概念, 抽象, あるいはもの (対象) のこと
- 次の性質を保持したもの
 - 状態
 - 振舞い
 - 識別性 (アイデンティティ)



オブジェクトの状態

- そのオブジェクトがとりうる状況を一つ指定したもの
- 普通, 時間経過とともに変化
- 具体的には, 各属性の値および他のオブジェクトとのリンクの種別とその有無で実装

名前: Joyce Clark

従業員ID: 567138

入社日: 1987年3月21日

オブジェクトの振舞い

- オブジェクトがどのように行動したり反応したりするか
- 他のオブジェクトからの要求にどのように反応するかを定義
- 外部から見えるオブジェクトの振舞いは、反応できるメッセージの集合としてモデル化

オブジェクトの識別性

- オブジェクトの状態が他のオブジェクトと同一でも別のものとして識別

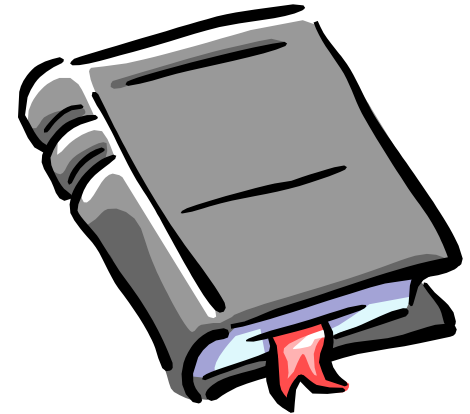
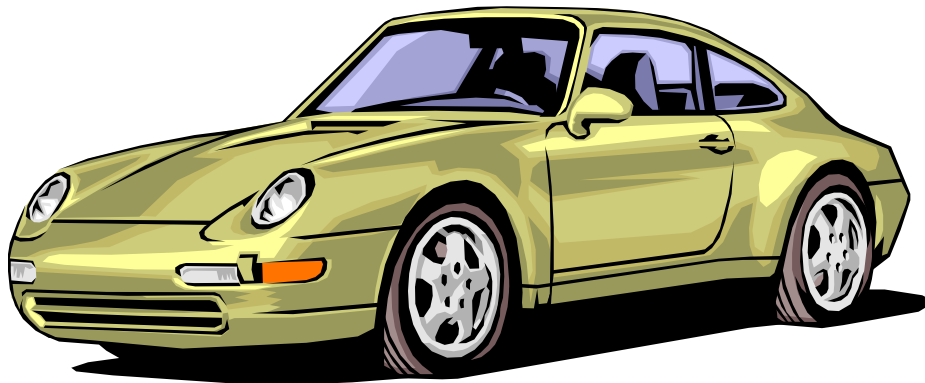


クラス

- 共通の性質 (属性), 共通の振舞い (操作), 他のオブジェクトとの共通の関係 (関連・集約), 共通の意味を持ったオブジェクトの集合を記述したもの
 - オブジェクトはクラスのインスタンス (実例)
- 次のような点で抽象的
 - 適切な特徴を強調
 - 他の特徴を抑制
- 抽象化により複雑なものが扱いやすくなる

クラスとオブジェクトの例

- 本とある本
- 車とある車
- . . .

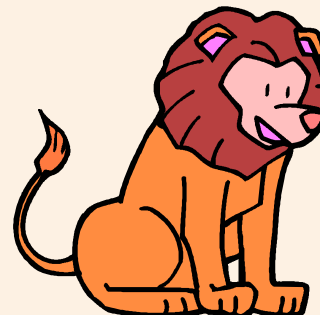
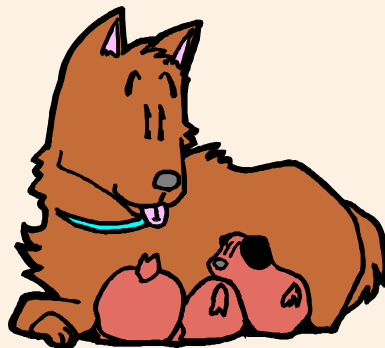
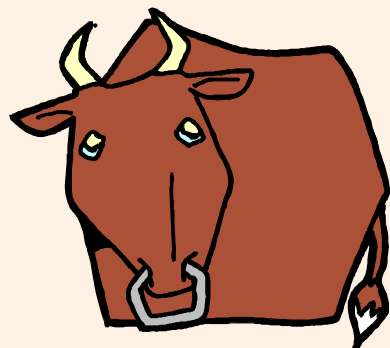


クラスの例

- コース
 - 属性
 - 名前, 場所, 実施日数, 単位数, 開始時刻, 終了時刻
 - 振舞い
 - 学生の追加, 学生の削除, コース名簿の出力, 満員かどうかの確認

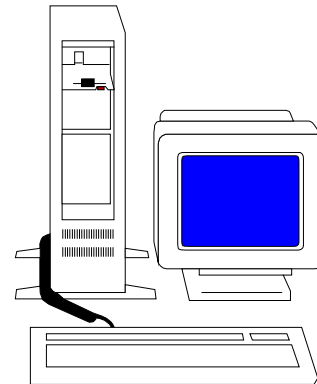
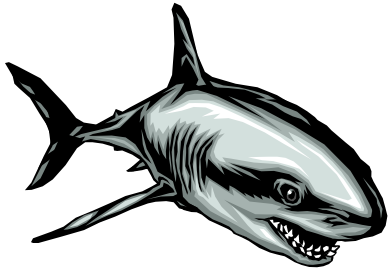
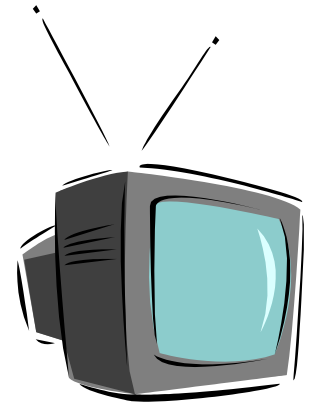
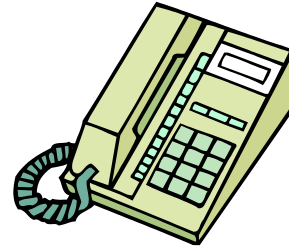
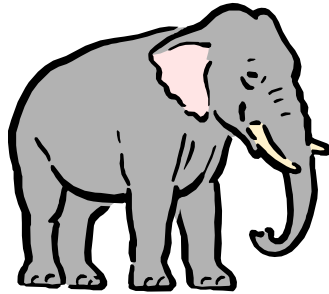
クラスの識別

仲間外れはどれ?



クラスの識別

どんなクラスが見えますか？



クラスとオブジェクトの関係

- クラスはオブジェクトを抽象的に定義したもの
 - クラスの中で各オブジェクトの構造・振舞いを定義
 - クラスはオブジェクト生成のためのテンプレート(型)
- オブジェクトをクラス単位にグループ分け

Smith教授

Mellon教授

Jones教授



バウンダリ クラス

- 他システムとのインタフェースのモデル化
 - 他システムに情報を渡す
 - 通信プロトコルを使って他システムと会話



エンティティ クラス

- 情報や関連する振舞いをモデル化
 - 通常永続的なクラス
 - 現実世界の事柄を表す
 - 内部的な作業のために必要な場合も
 - 属性の値はアクターから与えられることも
 - 環境とは独立した振舞いを持つ



コントロール クラス

- 一つまたは複数のユースケースに関する制御に関する振舞いをモデル化
 - 制御対象のオブジェクトの生成, 初期化, 削除
 - 処理のシーケンスや相互作用を制御
 - 並行性を制御
 - ほとんどの場合実体のないオブジェクト



C# の場合

- クラス

```
class Calculate
{
    double d1, d2;

    public void displayAverage()
    {
        Console.WriteLine((d1 +
            d2) * 0.5);
    }
}
```

```
// Calculate 型のオブジェクト
Calculate calculate =new
    Calculate();
```

```
// calc の関数を呼び出す
// (オブジェクト calc へ
//   メッセージ送信)
calc.displayAverage();
```

5. 設計のコツ

最も重要な原則

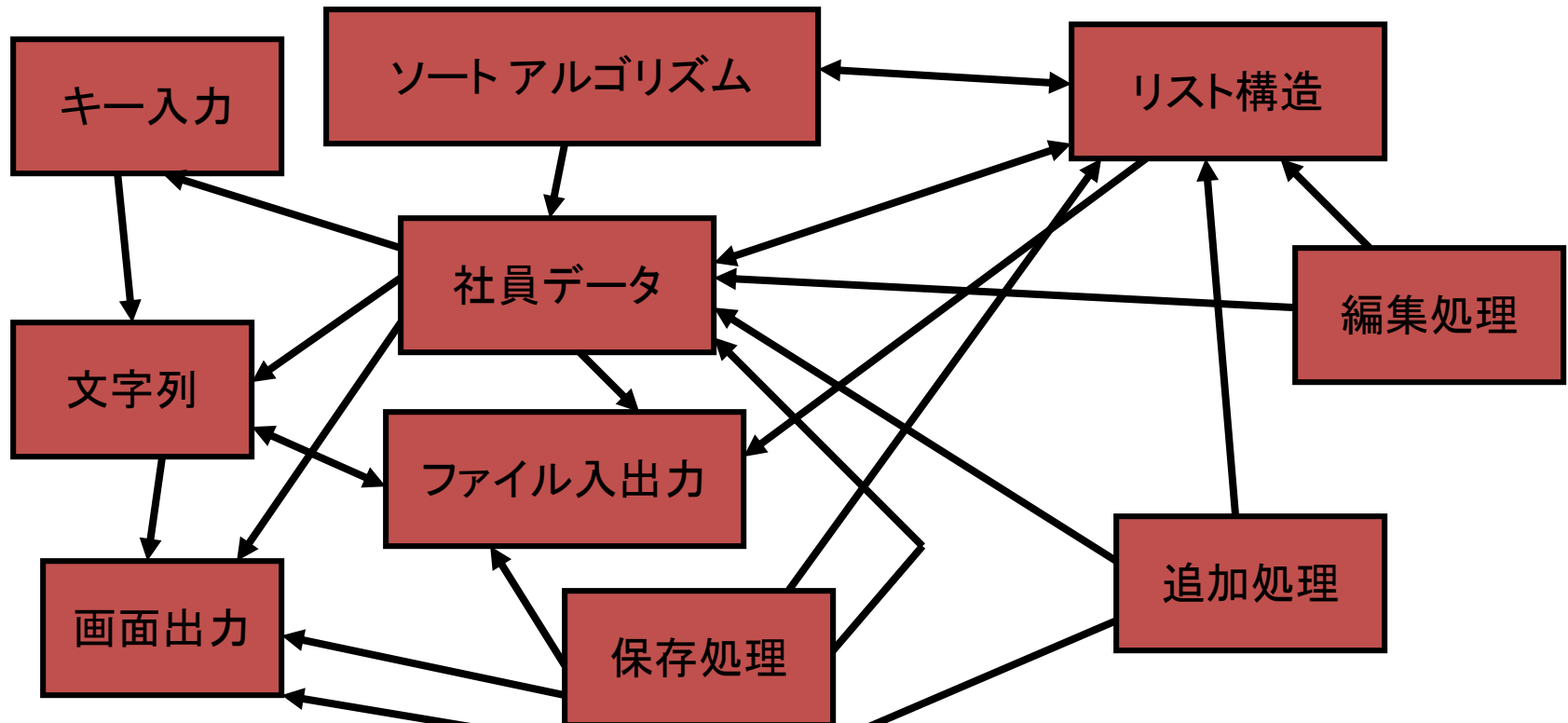
- 高凝集 (high cohesion) 且つ疎結合 (low coupling)
- 関心事の分離 (separation of concerns)

モジュール概論

- モジュール
 - プログラム構造の基本単位
 - 実行可能なプログラムの命令の集合
 - 閉じたサブルーチンであること
 - プログラム内の他のどんなモジュールからも呼び出すことができること
 - 独立してコンパイルできる可能性をもっていること

モジュール性の原則

- この状態から機能追加



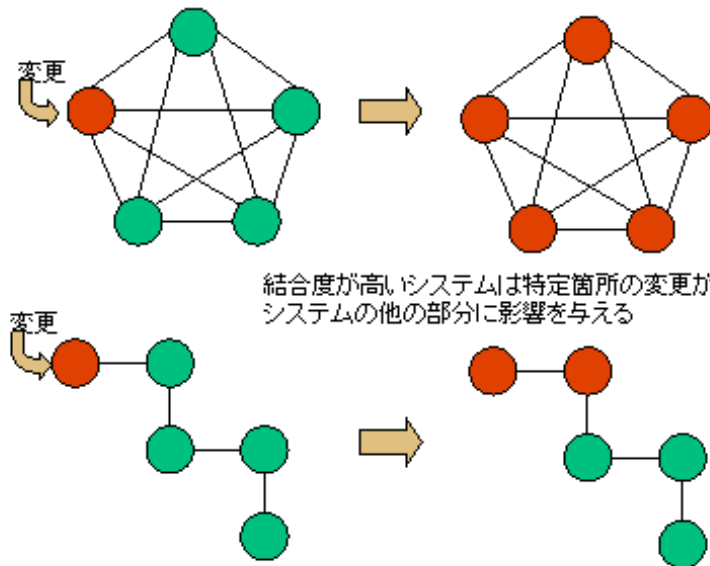
モジュールの凝集度と結合度

- 凝集度 (cohesion) : 強度(strength)
 - 各モジュール内の関連性
- 結合度 (coupling)
 - モジュール間の関連性

モジュール性の原則 (参考)

- 高凝集度 (high cohesion)
- 低結合度 (low coupling)

結合度が高い設計と低い設計の違い



結合度が高いシステムは特定箇所の変更がシステムの他の部分に影響を与える

オブジェクト指向によるソフトウェア設計
結合度が低いシステムは、全体として見通しがよく変更が局所化される

高凝集 (high cohesion) かつ 疎結合 (low coupling)

- 高凝集 (high cohesion)
 - クラスは一つの明確な目的と名前を持ち、不整合のない首尾一貫した中身を持っているべき。
- 疎結合 (low coupling)
 - クラスは単純なインタフェースを持ち、関係のないクラスとは出来るだけ話さないようにすべき。

凝集度

1. 暗号的凝集度
 - － 機能を定義することすらできない
2. 論理的凝集度
 - － 呼び出しモジュールによって選択され実行
3. 時間的凝集度
 - － 関連の少ない幾つかの機能を逐次的に実行
4. 手順的凝集度
 - － 仕様によって定められた関連の少ない複数の機能を逐次的に実行
5. 連絡的凝集度
 - － データに関連のある複数の機能を逐次的に実行
6. 機能的凝集度
 - － 固有の機能を実行
7. 情動的凝集度
 - － 概念・データ構造・資源などを隔離

結合度

1. 内容結合
 - 1つが他の内部を「直接」参照
2. 共通結合
 - グローバル データ構造を参照するグループ
3. 外部結合
 - 単一のグローバルデータを参照
4. 制御結合
 - 他のモジュールの論理を制御
5. スタンプ結合
 - モジュール間でやり取りするデータが不必要なデータを含んでいる
6. データ結合
 - モジュール間のインタフェースデータが単一のデータ
7. 非直接結合 (分析において使用する)

関心事の分離 (separation of concerns)

- 複数の関心事が一つのクラスの中に混在しないようにする。或る関心事を局所的にする
他の関心事とは分けてカプセル化しておく
- 違うものは分ける \Leftrightarrow 同じものはかためる
- 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
 - どの関心で分けるか？

良いモデルとは

- “Once only Once”
 - 重複しないこと
- 一つの理由による変更が一箇所の修正で済む

良いモデルとは

- シンプル
 - 複雑さの度合いが人が理解できる範囲を越えない
 - 自然に表現されたモデル
 - 図で描いてみる → 図とソースコードが同等のモデル
- Ease To Understand
 - わかりやすい
- 名前が的確

良いモデルとは

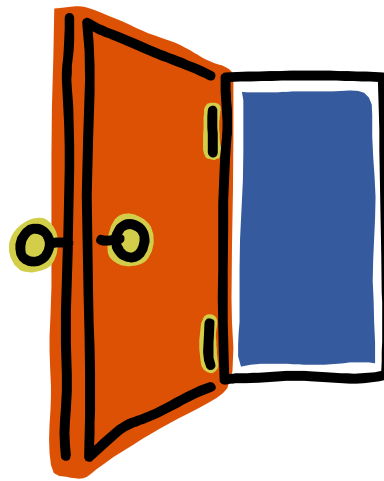
- Ease To Change
 - 変更しやすい
 - 拡張しやすい
- Ease To Test
 - 検証しやすい
 - 検証しやすいように分離

良いモデルとは

- 保守性重要
 - 保守しやすい
 - 拡張しやすい
 - 再利用しやすい
 - 視認性が高い
- 機能、使いやすさ、パフォーマンス、は関心の外

The Open-Closed Principle (開いて閉じての法則)

- 「ソフトウェアモジュールは、変更に対して閉じており、拡張に対して開いているべき」



C の場合

```
typedef struct { int x, y;      } Point ;
typedef struct { Point p1, p2;  } Line  ;
typedef struct { Point o; int r; } Circle;
typedef enum   { LINE, CIRCLE } Kind;
typedef union  { Line line; Circle circle;  } ShapeUnion;
typedef struct { Kind kind; ShapeUnion shape; } Figure;

void draw(Figure* figure) {
    switch(figure->kind) {
        case LINE :
            drawLine (figure->shape.line.p1, figure->shape.line.p2);
            break;
        case CIRCLE:
            drawCircle(figure->shape.circle.o, figure->shape.circle.r);
            break;
    }
}
```

C++ の場合

```
class Point { int x, y; };
```

```
class Figure {  
public:  
    virtual void draw() = 0;  
};
```

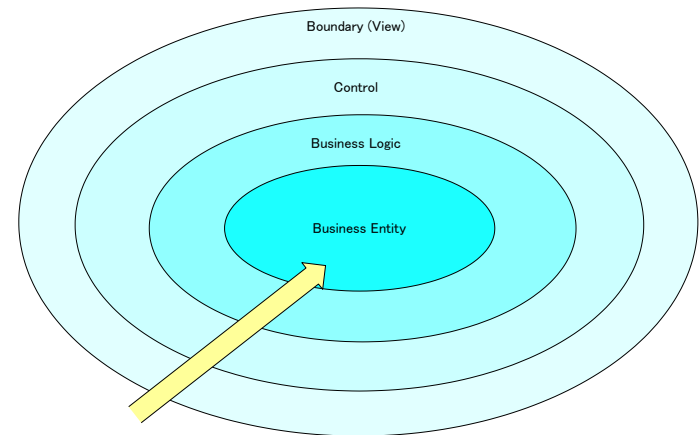
```
class Line : public Figure {  
    Point p1, p2;  
public:  
    void draw() { drawLine(p1, p2); }  
};
```

```
class Circle : public Figure {  
    Point o; int r;  
public:  
    void draw() { drawCircle(o, r); }  
};
```

IOP (Inside-Out Principle)

内側から外側への原則

- 中から外へ向って設計せよ。
- モデルを先に設計し、ユーザーインタフェースは後で設計せよ。



内側に向かう程
・分析対象
・業務レベルの再利用性を重視

単一責務の原則

- SRP (Single Responsibility Principle)
 - 一つのクラスは一つの責務を持つべき。
 - クラスに変更が起こる理由は一つであるべき。
 - 良い抽象には良い名前がつく。

「Once and Only Once」

- 「一度、たった一度だけ」
 - 同じものを重複して書かない、ということ。これを守らないと、変更によって同じ修正を複数箇所で行うことになる。

Design by Contract

契約による設計

- 事前条件と事後条件
- C/C++/C# 等ではAssert で実装
- 形式手法

その他の原則

- LSP (The Liskov Substitution Principle) — リスコフの置換原則。
 - スーパークラス (継承元のクラス) をサブクラス (継承したクラス) で置き換えることが可能であるべき。
 - スーパークラスへの参照 (やポインタ) を使うメソッド (関数) は、そのサブクラスのオブジェクトを、それとは知らずに使えなければならない。
- DIP (The Dependency Inversion Principle) — 依存関係逆転の原則。
 - 上位モジュールが下位モジュールに依存すべきではない。上位モジュールも下位モジュールも抽象概念に依存すべき。
 - 抽象が詳細に依存すべきではない。詳細が抽象に依存すべきである。
- ISP (The Interface Segregation Principle) — インタフェース分離の原則。
 - クライアントは自分が使わないメソッドに依存することを強制されない。

その他の原則

- REP (The Reuse/Release Equivalency Principle) — 再利用・リリースの粒度の原則
 - 再利用の粒度はリリースの粒度であるべき。
- CCP (The Common Closure Principle) — 共通閉鎖の原則
 - パッケージ内のクラス群は或る修正に対して閉じている方が良い。従って、将来の変更に対して同じような修正が予想されるクラスは同一パッケージに入れた方が良い。
- CRP (The Common Reuse Principle) — パッケージ再利用の原則
 - パッケージ内のクラスは一緒に再利用されるべき。
- ADP (The Acyclic Dependencies Principle) — 循環依存禁止の原則
 - パッケージ間の依存関係は依存してはならない。
- SDP (The Stable Dependencies Principle) — 安定依存の原則
 - パッケージの依存関係は、依存元が依存先より安定している方が良い。
- SAP (The Stable Abstraction Principle) — 安定抽象の原則
 - 抽象的な方がより安定しているべき。

その他の原則

- The Law of Demeter — デメテルの法則
 - オブジェクト中の全てのメソッド (関数) は、以下のいずれかの種類のオブジェクトのメソッドのみを呼び出すべきである。
 - それ自身
 - メソッドに渡されたパラメータ (引数)
 - それが生成したオブジェクト
 - 直接保持しているコンポーネント オブジェクト
- Edelman's Law — エーデルマンの法則
 - 他人と話すな。 (Don't talk to strangers.)
 - クラスは関係のないクラスには話しかけないようにせよ。
- Hollywood Principle — ハリウッドの法則
 - "Don't call us. We'll call you." (貴方から呼ばないで下さい。必要な時に私が呼びますから)
 - ソフトウェア開発におけるライブラリとしての「フレームワーク」の性格を表す言葉。ユーザーは、フレームワークから呼ばれる必要な部分だけの個別のプログラムを書くことでアプリケーションを作成する。ユーザー プログラムは、フレームワークから制御されることとなる。

- a. わかりやすく綺麗なプログラムを書くコツ
- b. なぜ名前付けが重要か?
- c. 重要なのは、「どう作るか」ではなく「何を作るか」に視点を変えること

- テキスト「名前編」へ

d. UML を使ってみよう

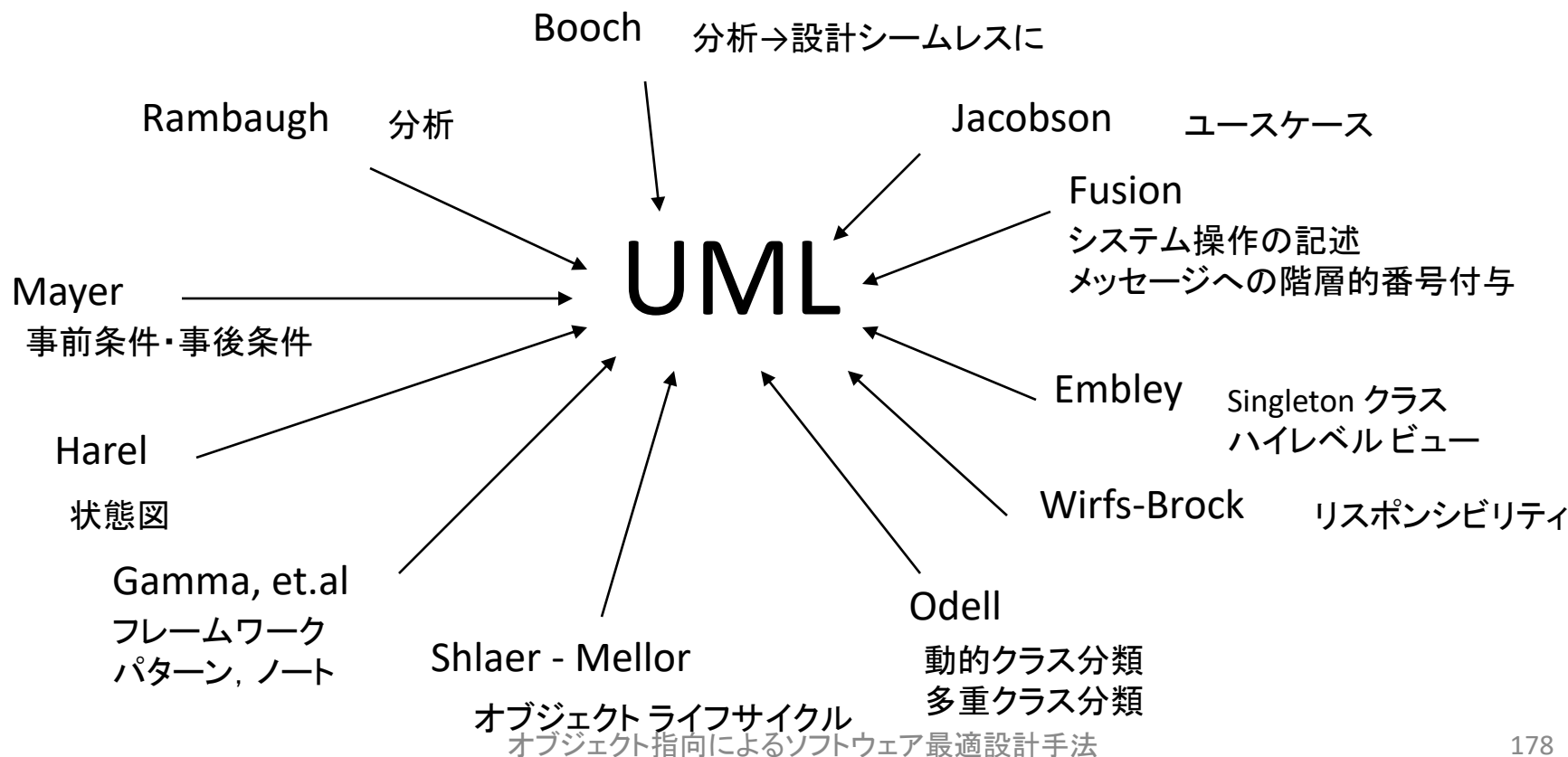
統一モデリング言語 (UML)

- Unified Modeling Language
- Grady Booch (Booch 法), Jim Rumbaugh (OMT 法), Ivar Jacobson (OOSE 法)
 - Three Amigos
- Rational Software, Microsoft, Hewlett-Packard, Oracle, Texas Instruments, MCI Systemhouse, IBM... から OMG へ

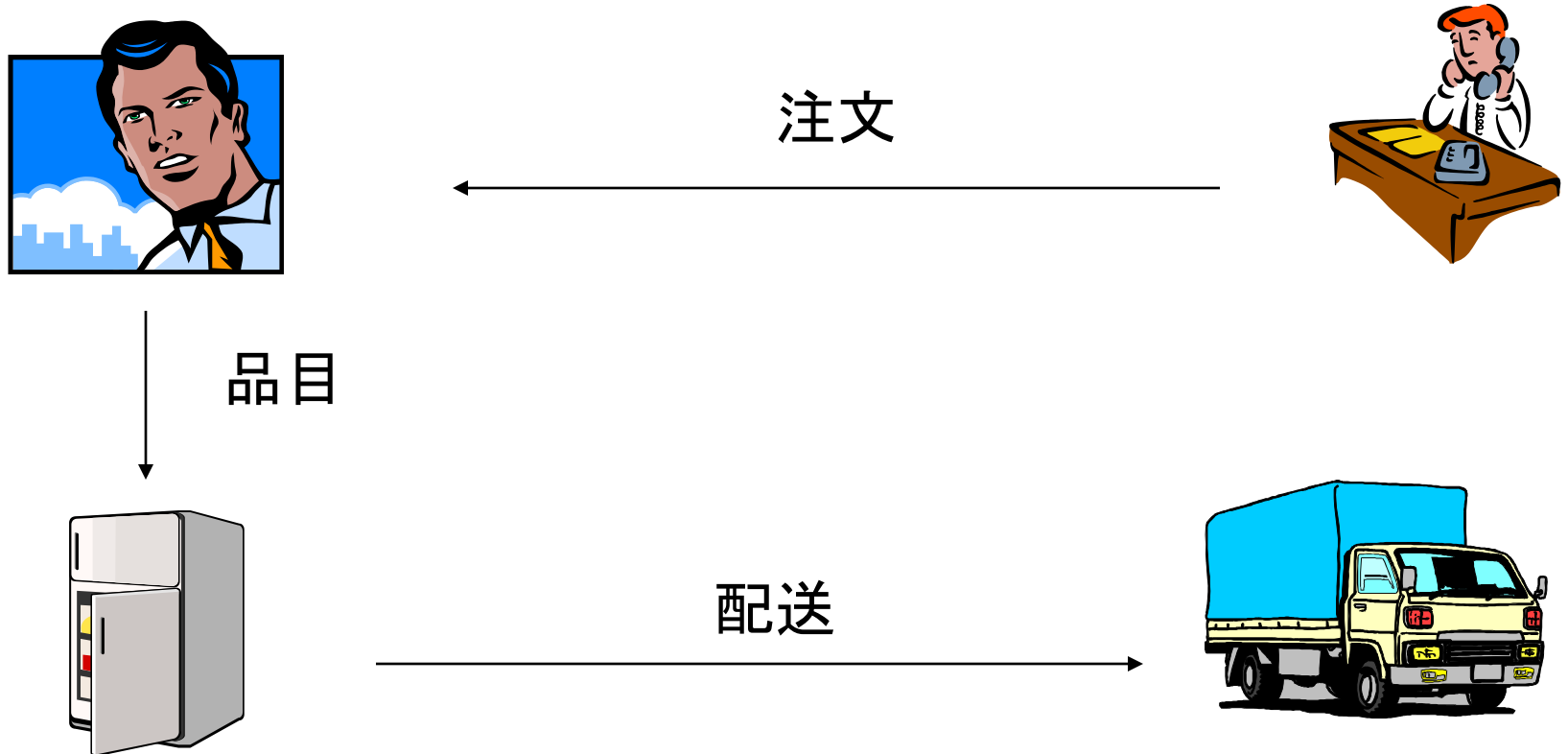


統一モデリング言語 (UML)

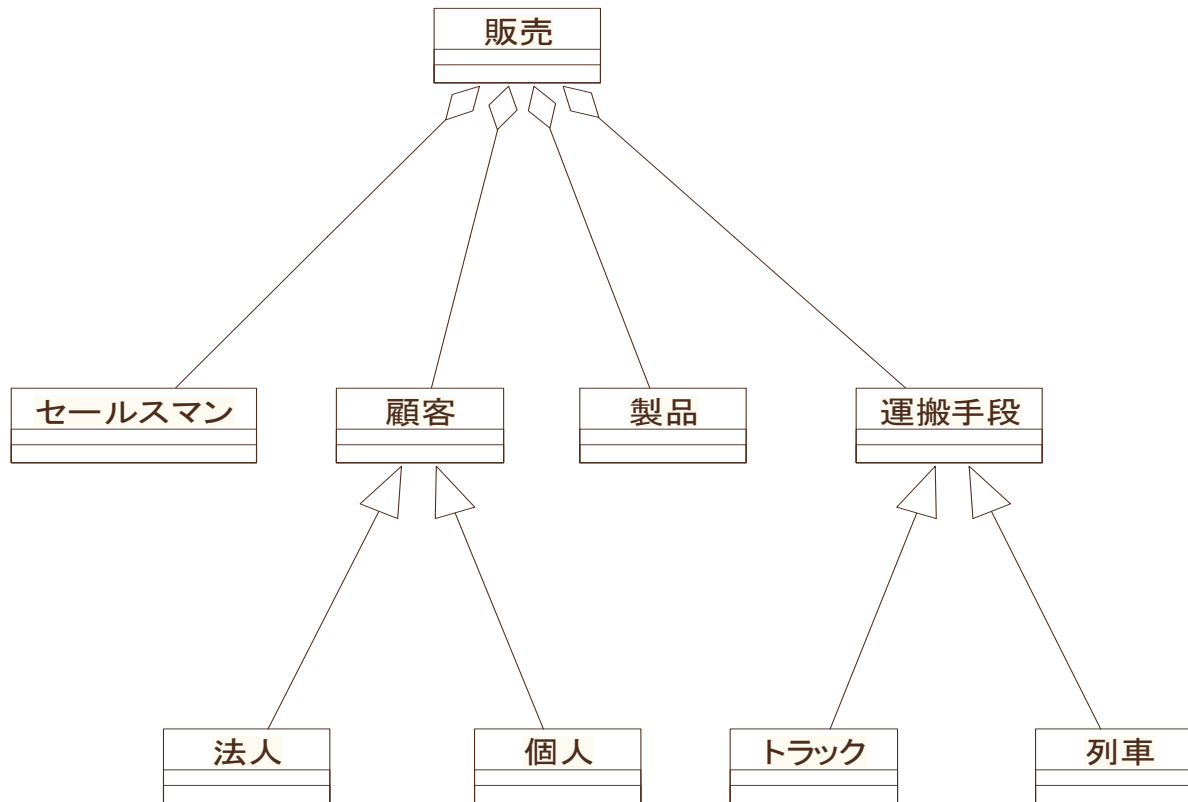
• 他の著者からの影響



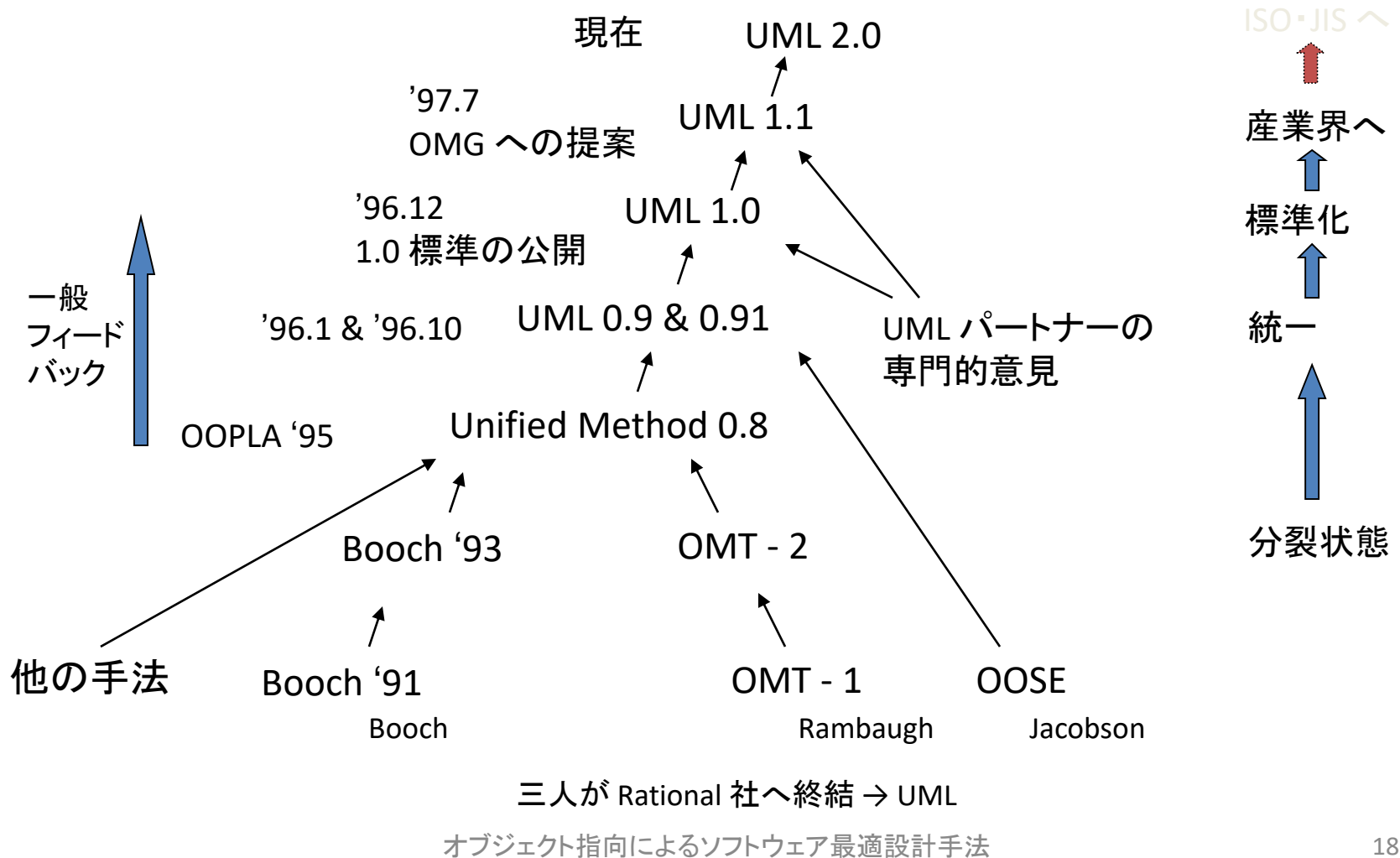
簡単な販売受注の例



販売システムのクラス図の例



UML の発展

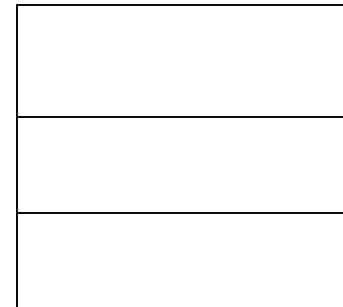
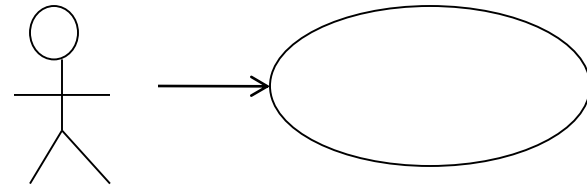


統一モデリング言語の利点

- 分析・設計，実装間のシームレスな対応関係の定義
- 表現力があり一貫性のある表記法
 - コミュニケーションの手段
 - 見落としや不整合の発見の容易さ
 - 小規模なものから大規模なものまでサポート

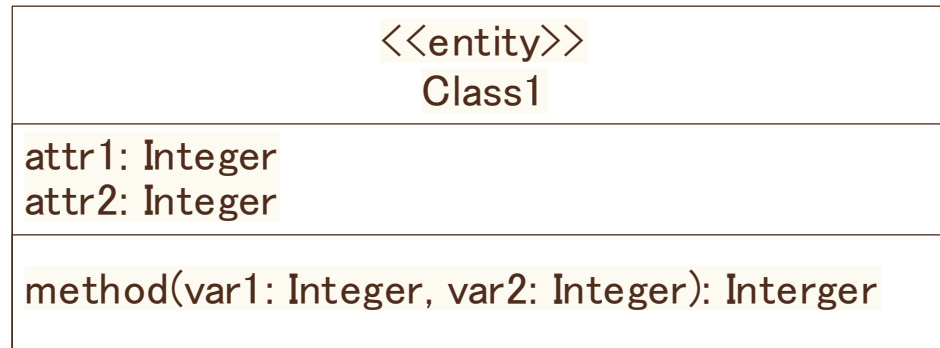
UML の図

- ユースケース図
- アクティビティ図
- クラス図
- コラボレーション図
- シーケンス図
- 状態図
- 配置図
- コンポーネント図



クラス

▪UML クラス図



▪C# の例

```
class Class1
{
    private int attr1, attr2;

    public int method(int var1, int var2)
    { return var1 + var2; }
}
```


クラスの関係－関連

- UML クラス図



・C# の例

```
class Class1
{
}
```

```
class Class2
{
    private Class1 role1;
}
```

クラスの関係 – 集約/コンポジション

・UML クラス図



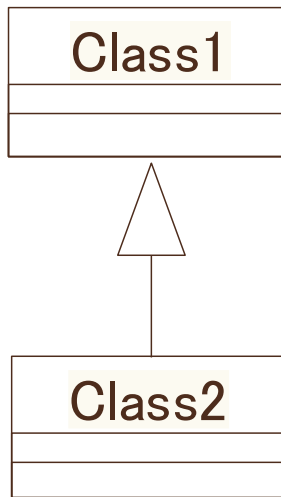
・C# の例

```
class Class
{
}
```

```
class Class2
{
    private Class1 role;
}
```

クラスの関係 – 継承

・UML クラス図

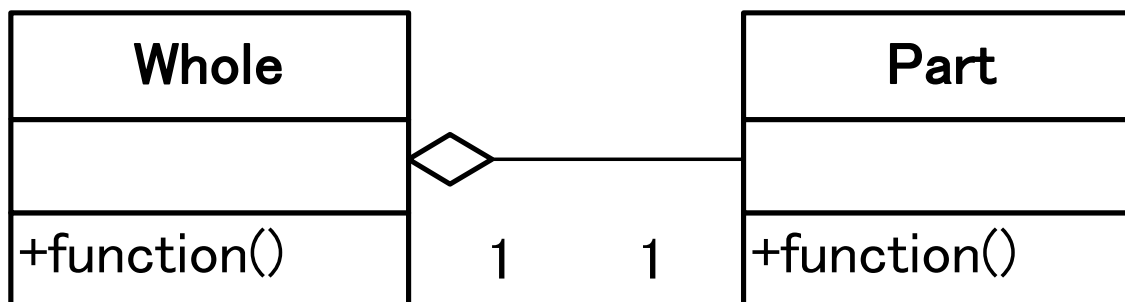


・C# の例

```
class Class1
{
}
```

```
class Class2 : Class1
{
}
```

委譲 (Delegation)



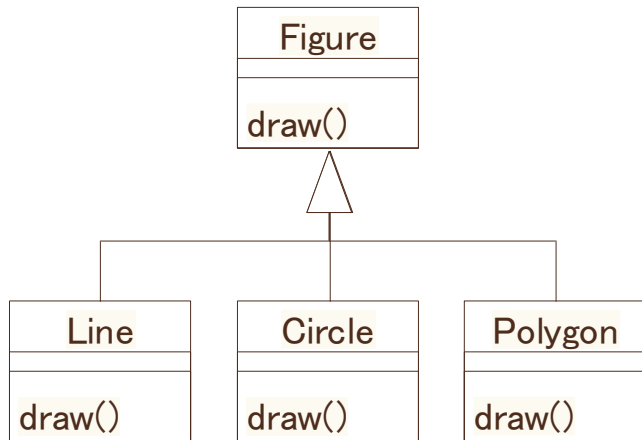
・C# の例

```
class Whole
{
    Part part;
    public void function()
    { part->function(); }
}
```

```
class Part
{
    public void function()
    { /* 処理 */ }
}
```

ポリモーフィズム (多態)

・UML クラス図



・C# の例

```
abstract class Figure {
    public abstract void draw();
}

class Line : Figure {
    public override void draw()
    { /* 線の描画 */ }
}

class Circle : Figure {
    public override void draw()
    { /* 円の描画 */ }
}

class Polygon : Figure {
    public override void draw()
    { /* 多角形の描画 */ }
};
```

```
Figure[] figures = new Figure[3];
```

```
figures[0] = new Line();
```

```
figures[1] = new Circle();
```

```
figures[2] = new Polygon();
```

```
foreach (Figure figure in figures)
    figure.draw();
```

属性

- クラスのインスタンスが持つデータの定義
- 名前は単純な名詞か名詞句でつける
 - クラス内でユニークなもの
- 明確で簡潔な定義を持たなければならない
 - 学生クラスの適切な属性
 - 氏名 – ファースト ネームとラスト ネーム
 - 専攻 – 分野
 - 学生クラスの不適切な属性
 - 選択したコース → 関係であり, 属性ではない

属性値

- 特定のオブジェクトに対する属性の値
- 各オブジェクトはクラスに定義されたすべての属性に対してそれぞれ値を持つ
 - 教授クラスのオブジェクト

属性

氏名

職員ID番号

担当教科



Sue Smith

567892

数学



George Jones

578391

生物

属性の表示

- 属性はクラスの二つ目のコンパートメントに表示

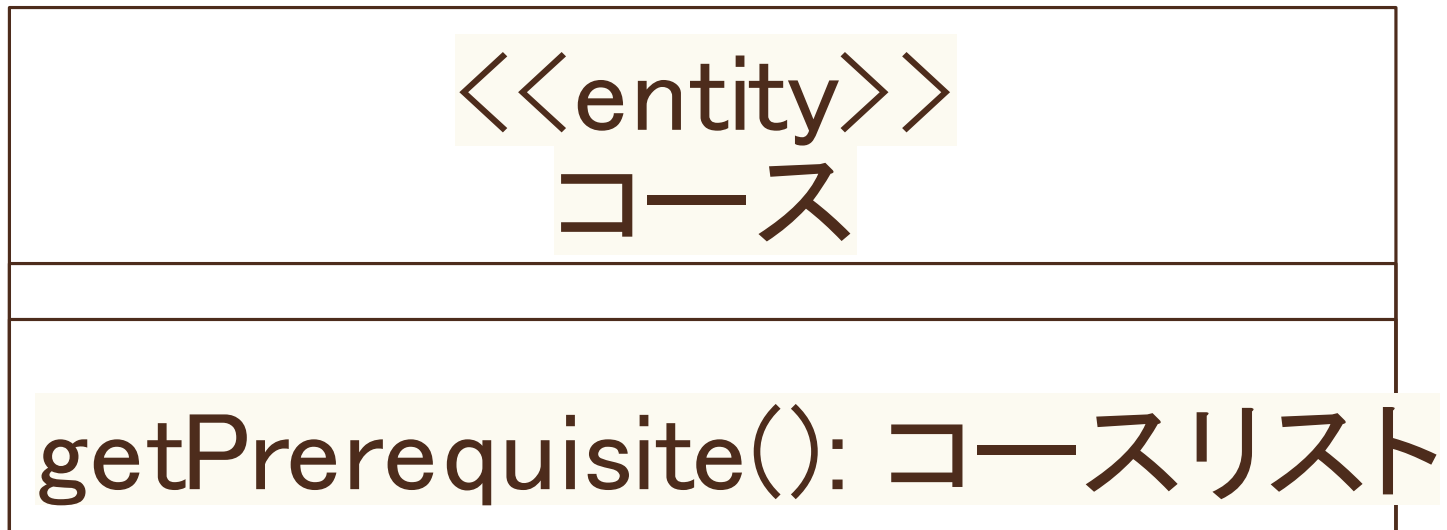


操作

- クラスに属するオブジェクトの振舞いを定義
 - 責務 (responsibility) の集合
- クラスの持つ責務は, 操作 (群) によって実行される
 - 責務と操作は必ずしも一対一のマッピングではない
 - 「製品」クラスの責務 – 「価格を提示すること」
 - 操作 – データベースで必要な情報を検索, 価格を計算する
- 操作は, あるオブジェクトからの要求で起動するサービス

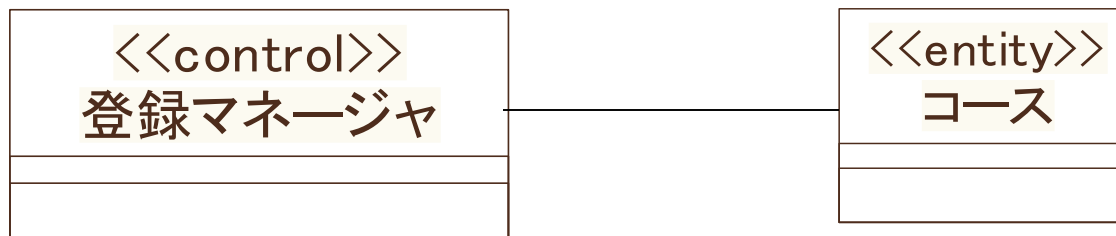
操作の表示

- クラスの三つ目のコンパートメントに表示



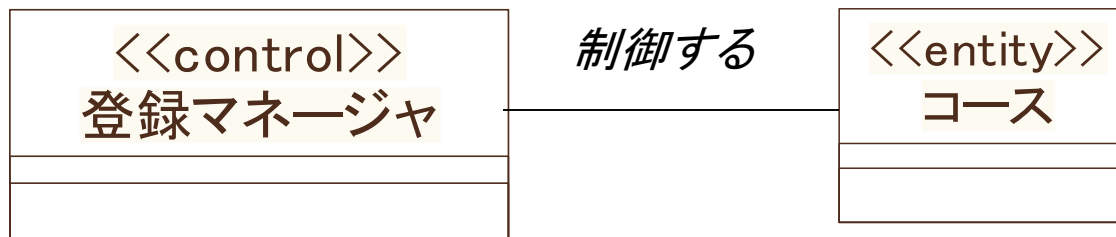
関連

- クラスとクラスの間の方方向の意味的な接続
 - オブジェクト同士の間リンクがあることを意味する
- クラス図では、関連したクラスを接続する線分で表す
- データはリンクを通して片方向または双方向に流れる



関連のネーミング

- 意味を明らかにするため，関連に名前が付けられる
- 関連の名前は，関連の線に沿ったラベルで表す
- 通常，動詞か動詞句



役割

- あるクラスが他のクラスと関連する目的, 立場または能力を示す
- 名詞または名詞句
- 関連の線に沿って, 修飾されるクラスの近くに
 - 関連の一方または両方の端に付けられる
 - 役割名を付ける場合は関連名は付けないことも多い



関連の多重度

- あるクラスの一つのインスタンスが、もう一つのクラスのいくつかのインスタンスと関係するか
- 関連ごとの両端に多重度が必要
 - 人の各インスタンスは、複数の学課を教えることがある (0 以上)
 - 学課の各インスタンスに対し、必ず一人の人が教える



多重度表記

- 関連は, それぞれの端に多重どの情報を持つ
 - 関係に参加しているオブジェクトの数

| | |
|----------|------------------|
| 多 (0 以上) | <hr/> * |
| 厳密に 1 | <hr/> 1 |
| 0 以上 | <hr/> 0..* |
| 1 以上 | <hr/> 1..* |
| 0 か 1 | <hr/> 0..1 |
| 範囲指定 | <hr/> 2..4 |
| | <hr/> 1..3, 6..7 |

集約

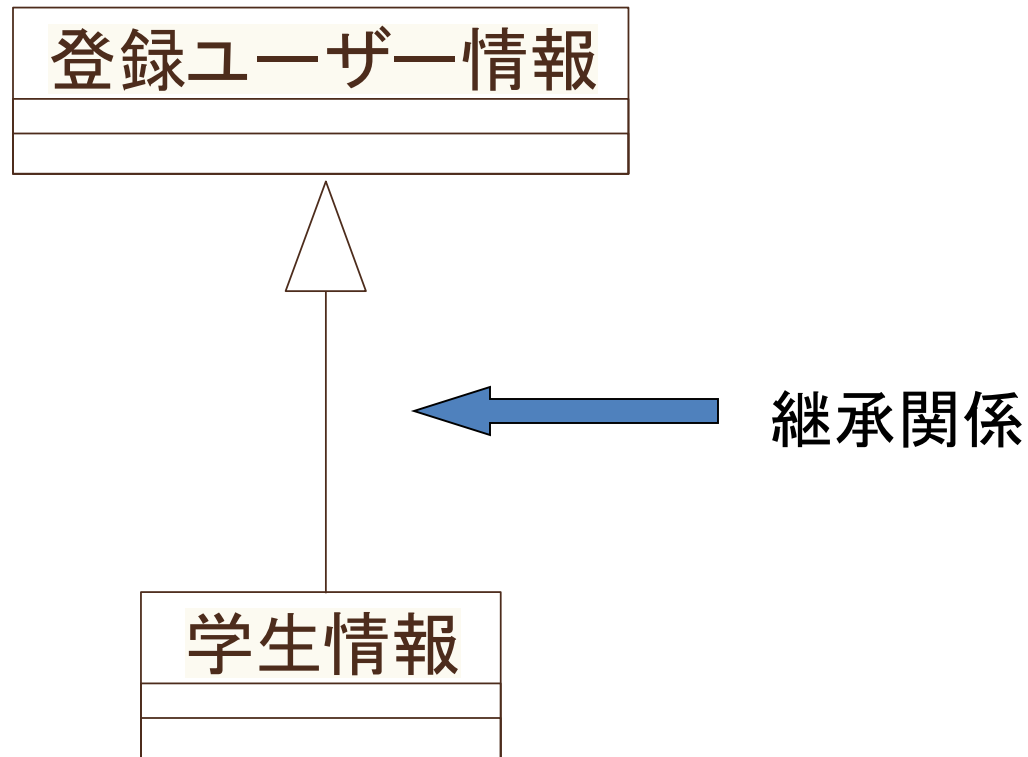
- 関連の一種で，全体とその部分との関係
 - 「has-a 関係」「part-of 関係」「保有関係」
- 全体を表すクラス側にダイヤモンド型の付いた関連として表現
- 多重度は他の関連と同様



集約の識別

- 「の一部である」「part of」という言葉で関係を表せるか
 - ドアは車の一部 (part of) である
- 全体オブジェクトに対する操作のいくつかは, その部分オブジェクトにも自動的に適用されるか
 - 車を移動するとドアも移動する
- いくつかの属性値は, 全体からその (全ての, あるいは一部の) 部分オブジェクトに伝播するか
 - 車は青色, ドアも青色
- 一方のクラスが他方のクラスに従属する, というような本質的な非対称性があるか
 - ドアは車の一部だが, 車はドアの一部でない

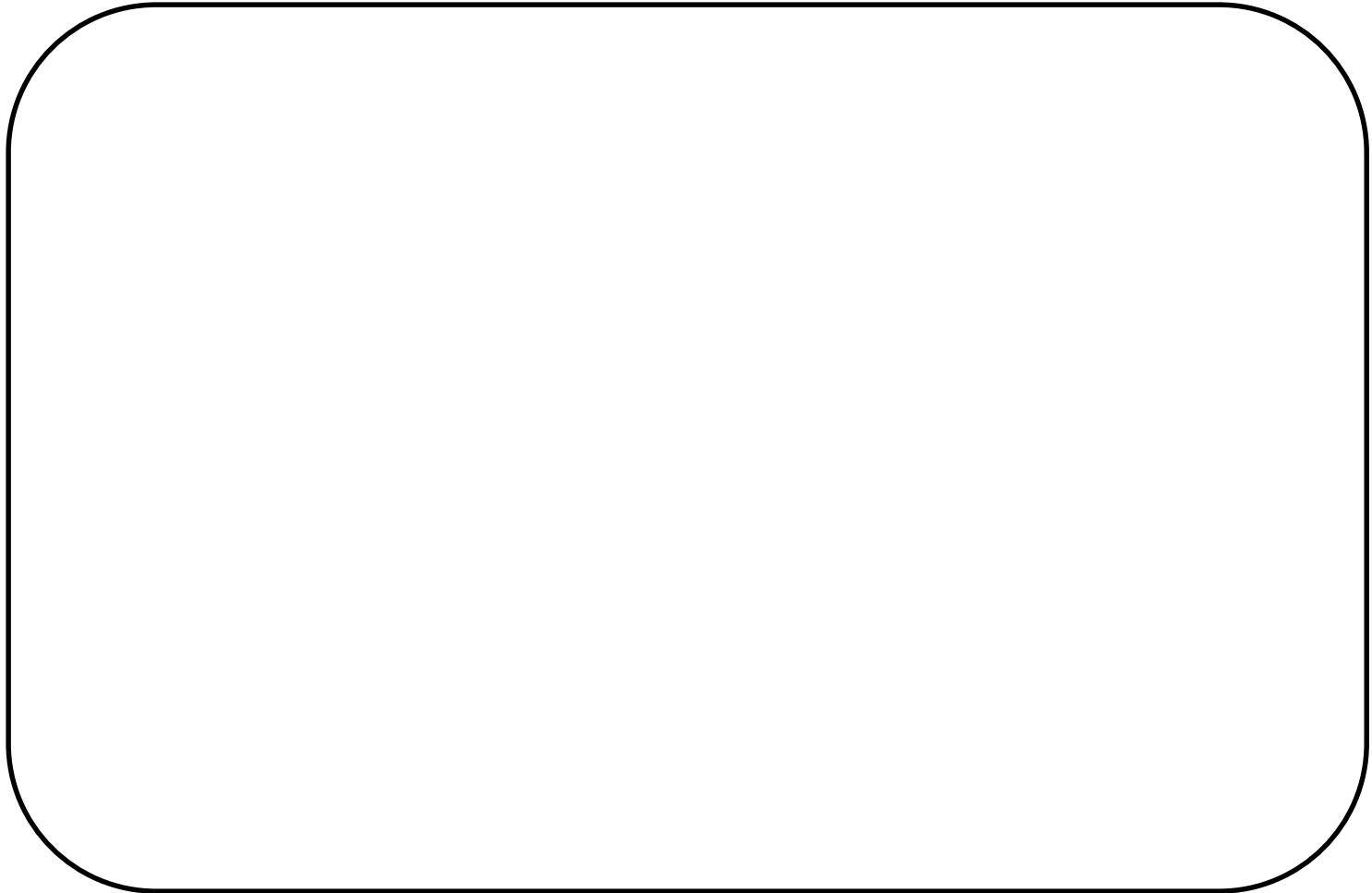
継承の表記



UML の目的

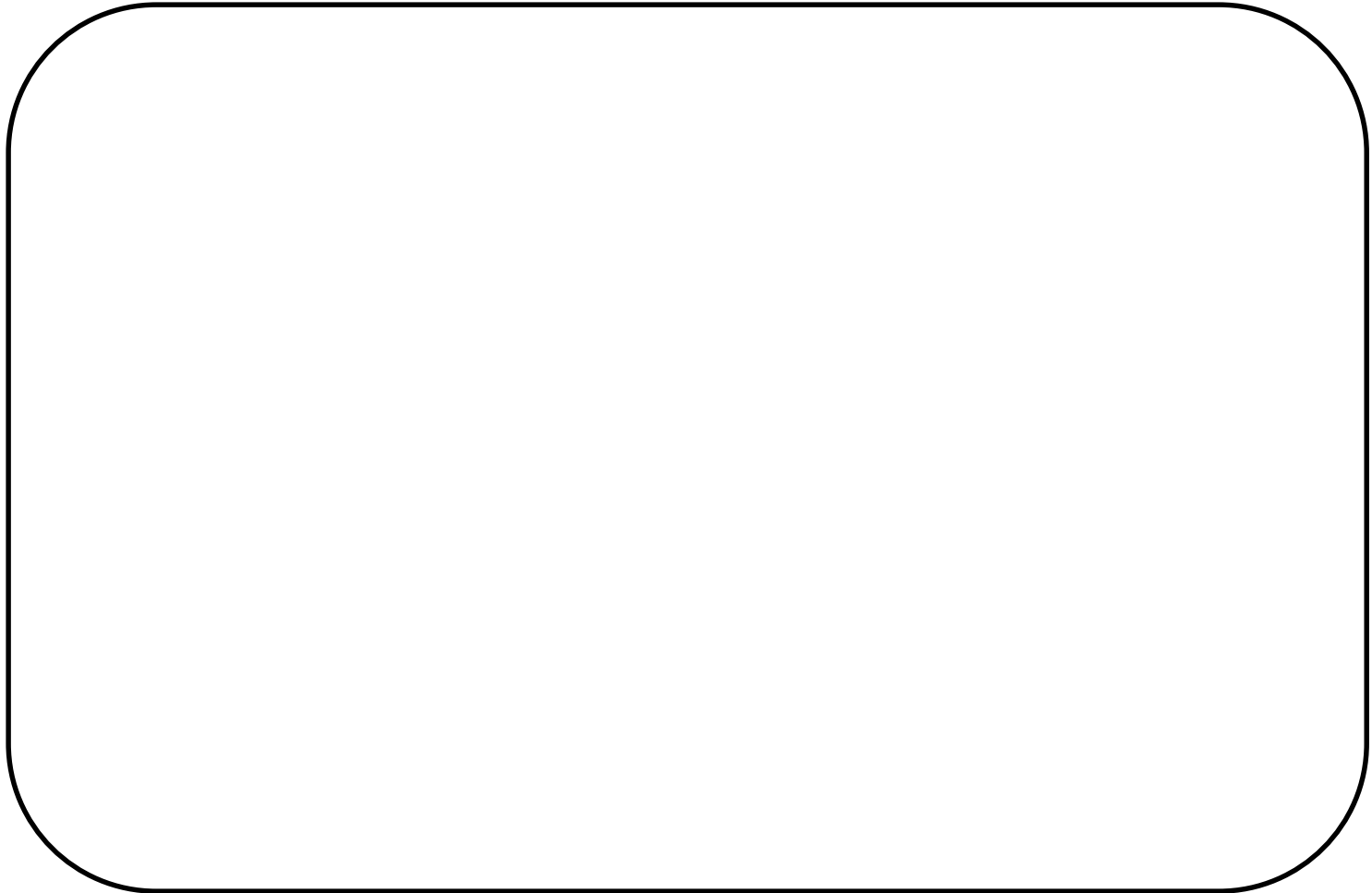
- スケッチとしての UML
 - UML As Sketch
- 設計図としての UML
 - UML As Blueprint
- プログラミング言語としての UML
 - Uml As Programming Language
- コミュニケーション ツールとしての UML
 - シンプルに伝える (S/N比の高いコミュニケーション)
- 思考のフレームワークとしての UML
 - シンプルに考える

UMLを描いてみよう



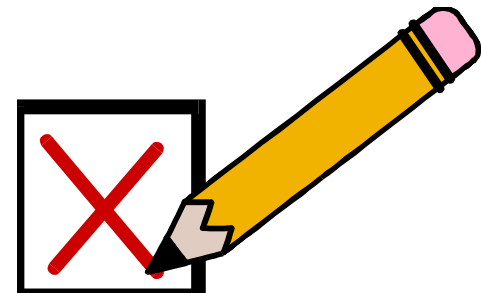
e. テストを行う意味とは？

どんなテストをしていますか？



テスト効率

- テストは大変
- 下流工程をいくら改良しても...
 - 障害の大きさ = エラー × エラーの滞在時間
 - 上流でのデバッグ

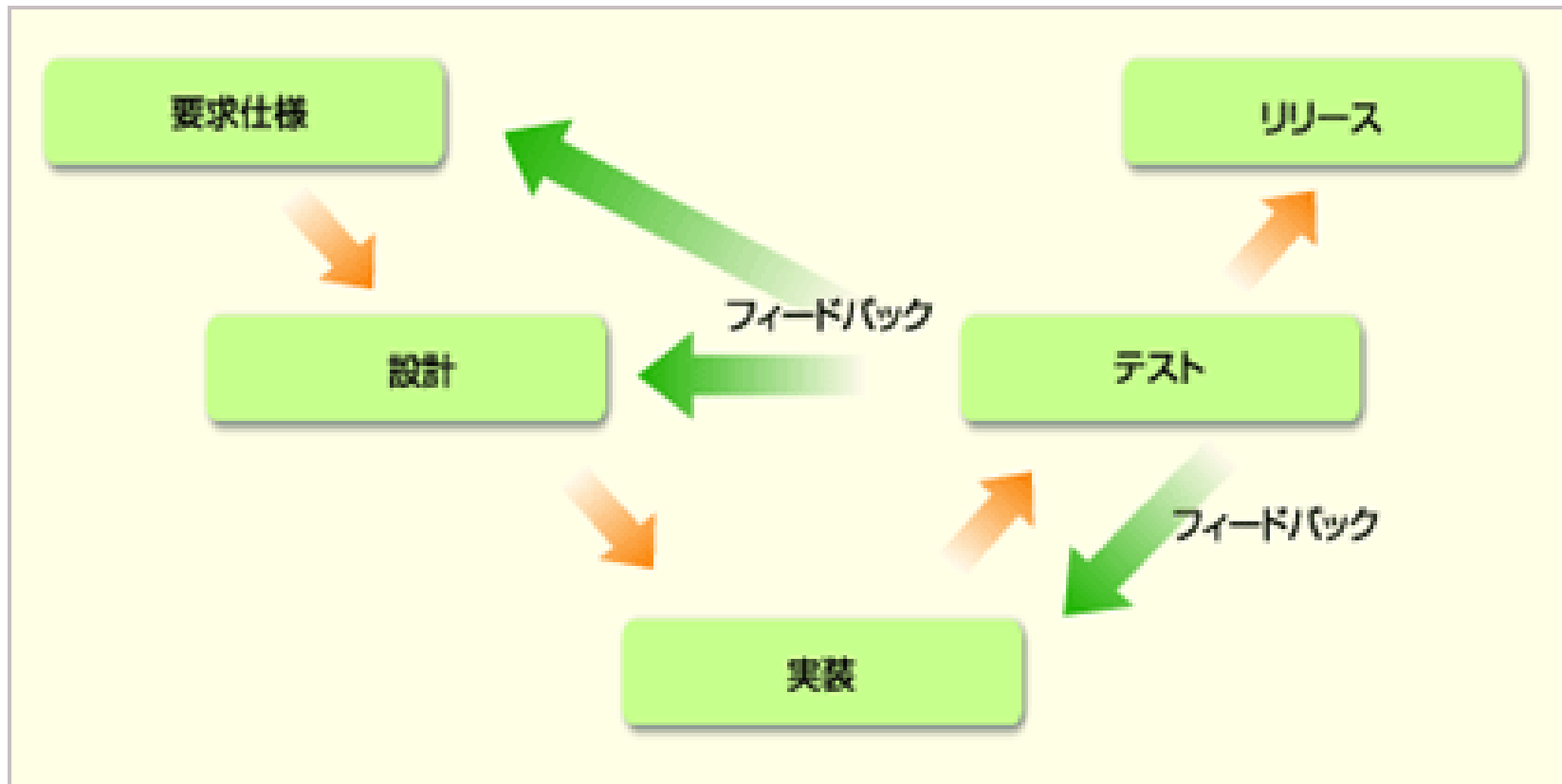


カイゼン

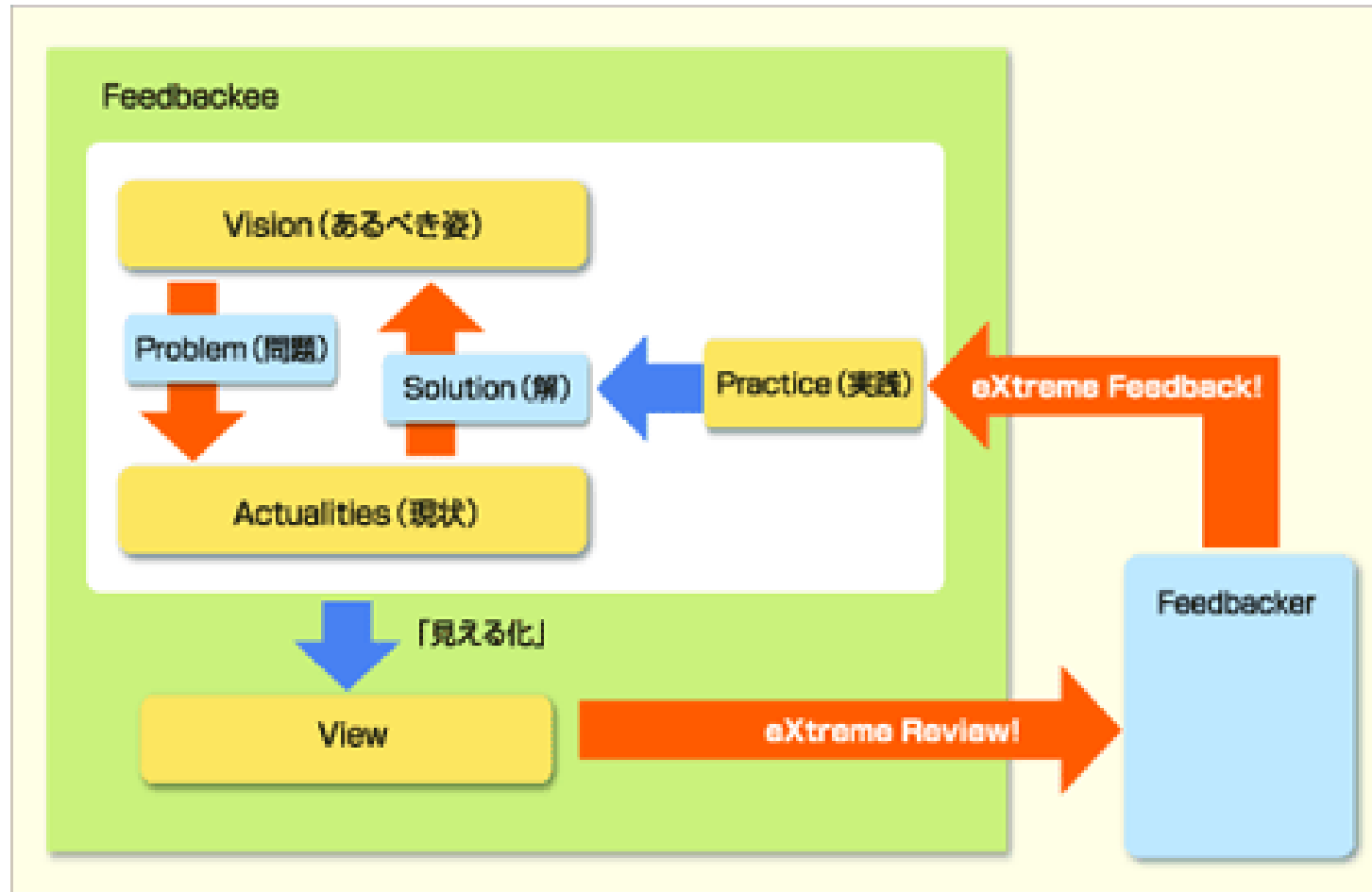
- カイゼンのループ
 - PDCA
- 検証 (=フィードバック)
 - 設計は要求分析へのフィードバック
 - 実装が設計へのフィードバック
 - ペア・プログラミングやコンパイラ、静的コード分析、単体テスト・ツールは、フィードバック・ツール



フィードバック重要



カイゼンのループ



テスト

- 不完全なテストは、実行できない完璧なテストよりもまし。

6. 設計の実習

デザインパターン – 生成パターン

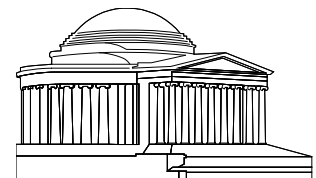
- Abstract Factory （抽象工場: Kit）
 - 互いに関連するオブジェクト群を、その具象クラスを明確にしないまま生成するインターフェースを提供する
- Builder （建築業）
 - 複合オブジェクトの作成過程と表現形式を分離することにより、同じ作成過程で異なる表現形式のオブジェクトを生成する
- Factory Method （工場操作: Virtual Constructor）
 - オブジェクトを生成するためのインターフェースだけを規定して、実際のインスタンス生成をサブクラスにまかせる
- Prototype （模型）
 - 模型(Prototype)からクローンとして新しいインスタンスを作る
- Singleton （1枚札）
 - クラスにインスタンスが一つしかないことを保証する

デザインパターン – 構造パターン

- Adapter（翻案者：Wrapper）
 - あるクラスのインターフェースを、他のインターフェースへ変換する
- Bridge（橋：Handle/Body）
 - 論理クラスと実装クラスを分離し、それぞれの独立性を保つ
- Composite（混成）
 - 部分-全体構造を表現するために、オブジェクトを木構造で構成する

デザインパターン – 構造パターン 続き

- Decorator (装飾者 : Wrapper)
 - オブジェクトに動的に責任を追加する
- Façade (見せかけ)
 - サブシステムのインタフェースを単純化する
- Flyweight (フライ級選手)
 - 多数の小さなオブジェクトを共有し、空間効率を高める
- Proxy (代理人 : Surrogate)
 - オブジェクトの代理を提供する



デザインパターン – 振る舞いパターン

- Chain of Responsibility (責任の連鎖)
 - 責任のあるサービス提供者へ要求を「連鎖的に」委任する
- Command (命令: Action, Transaction)
 - パラメータ化した要求をオブジェクトとしてカプセル化する
- Interpreter (通訳)
 - データを言語と考え、文法・意味を与え、それを解釈する
- Iterator (繰り返し)
 - オブジェクトの数を指定せず、順番に処理する
- Mediator (調停者)
 - オブジェクト群の相互作用をカプセル化するオブジェクトを定義する
- Memento (形見: Token)
 - オブジェクト個別の内部状態を捉え、後でその状態に戻す

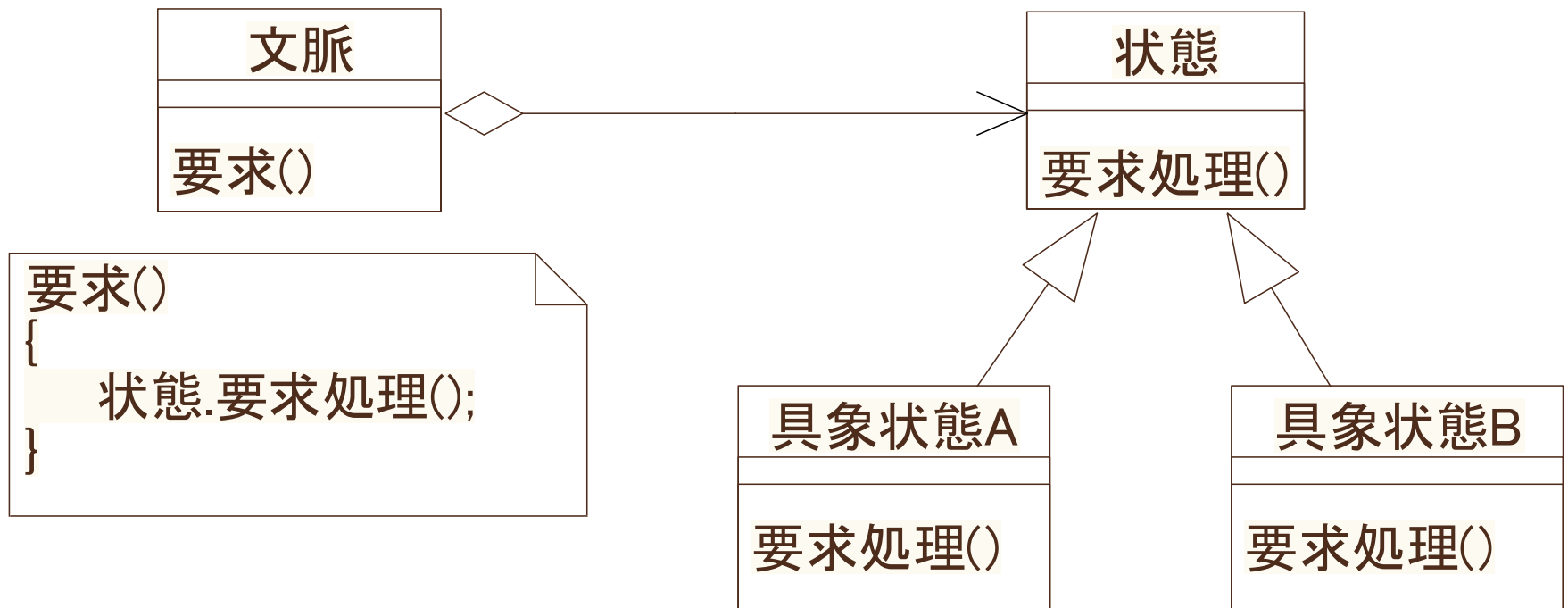
デザインパターン – 振る舞いパターン 続き

- Observer（観察者：Dependents, Publish/Subscribe）
 - オブジェクトが状態を変えたとき、それに依存したオブジェクトが自動的に更新される
- State（状態）
 - オブジェクトの内部状態が変わったとき、振る舞いを変える
- Strategy（戦略：Policy）
 - アルゴリズム群の各々をカプセル化し、交換可能にする
- Template Method（型紙方式）
 - 一部をサブクラスで実装するアルゴリズム
- Visitor（訪問者）
 - オブジェクト構造上の要素で実行される操作を表現する

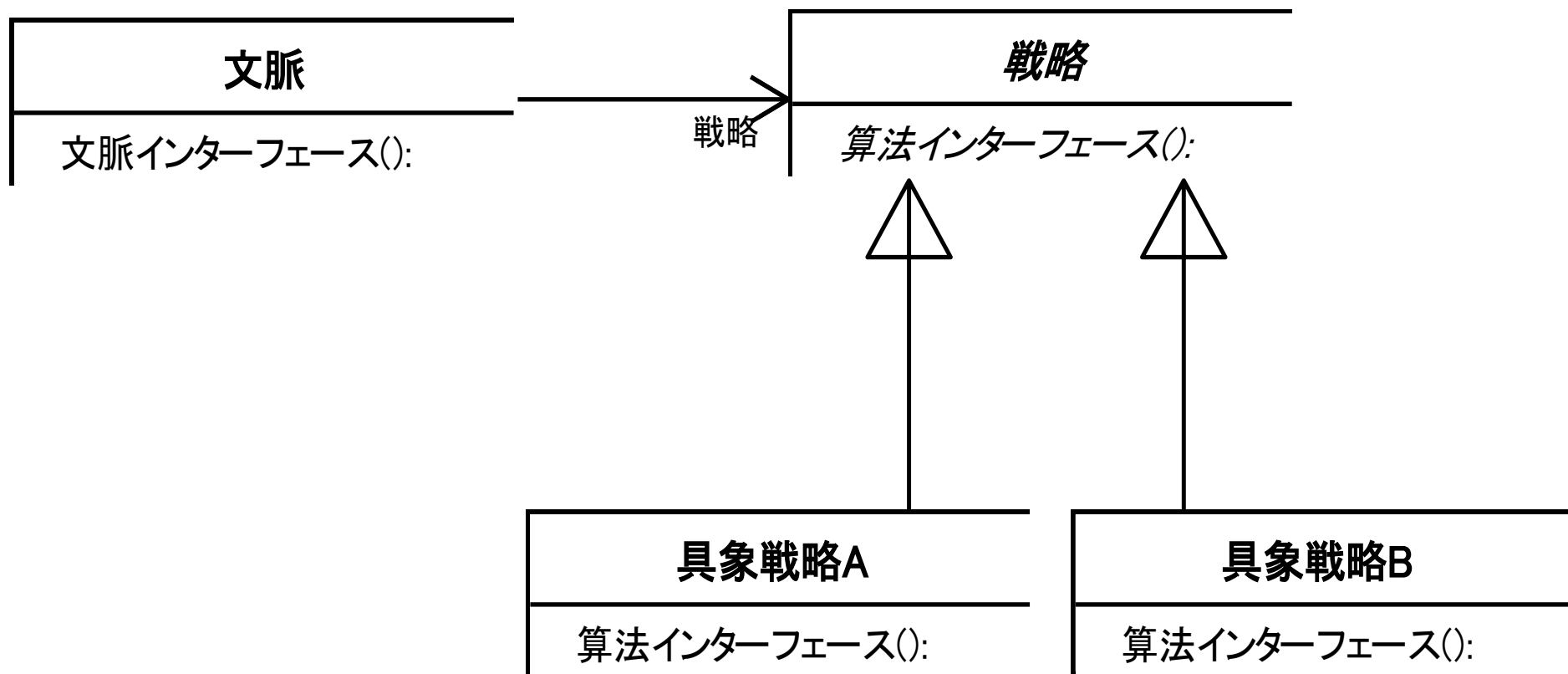
デザインパターン – 振る舞いパターンより

- State (状態)
 - オブジェクトの内部状態が変わったとき、振る舞いを変える
- Strategy (戦略)
 - アルゴリズム群の各々をカプセル化し、交換可能にする
- Template Method (型紙方式)
 - 一部をサブクラスで実装するアルゴリズム

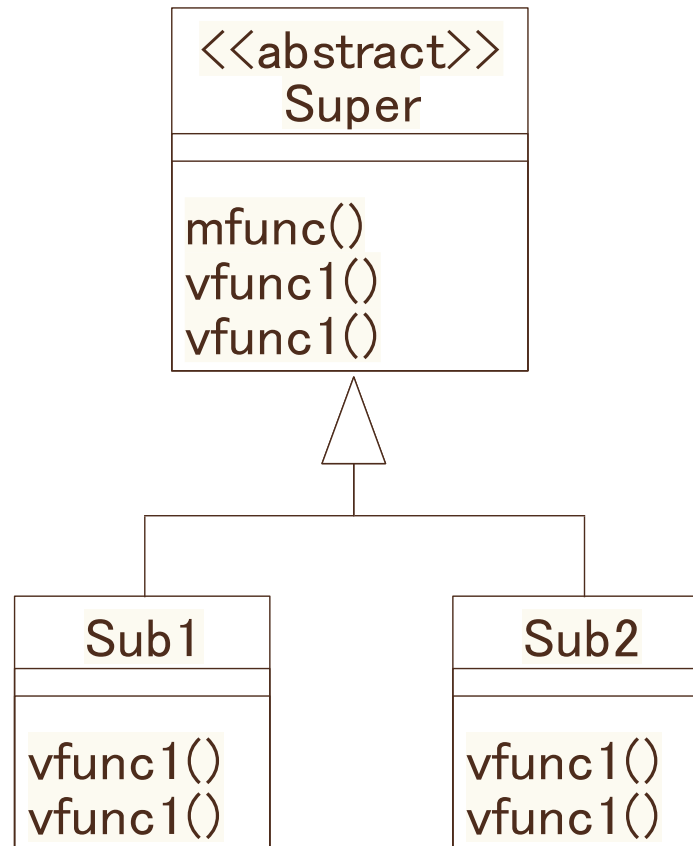
State (状態) パターンのクラス



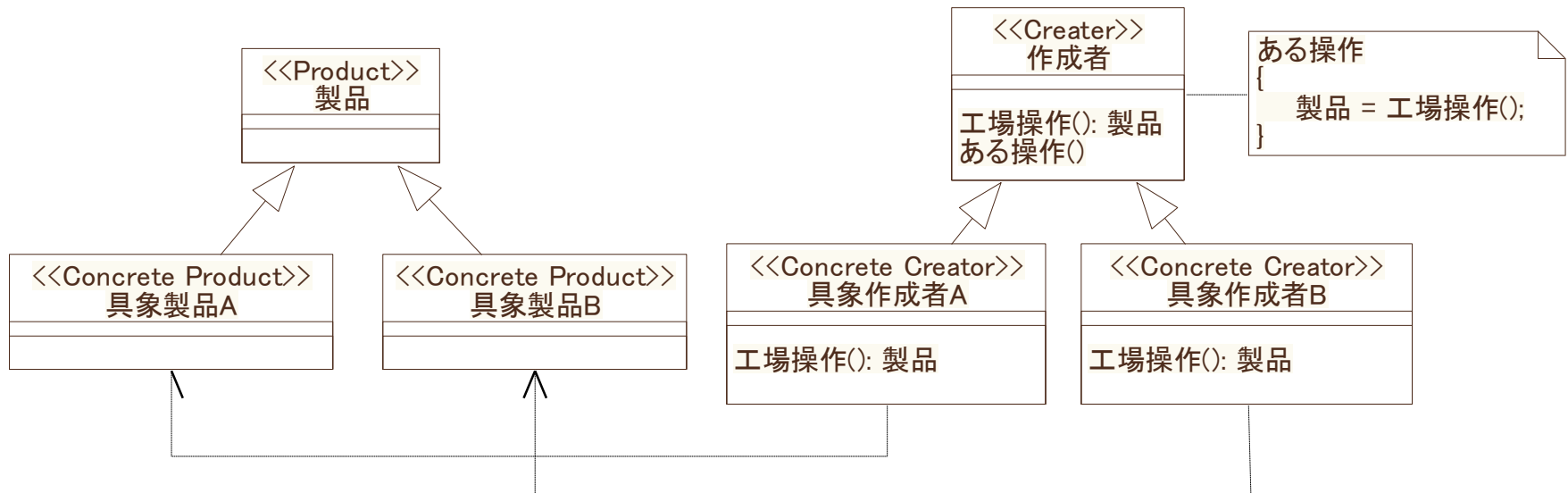
Strategy (戦略) パターンのクラス



Template Method（型紙方式）パターンのクラス



Factory Method (工場操作) のクラス



Ⅲ. エンジニアとして能力を伸ばす ためには

1. 問題解決能力を高めるには

掲示板などでの初心者の典型的な質問:

- Subject: 教えてください!!! (至急)
 - 「× × したいのですが、いまいちうまくいきません。どなたか分かる方、私にもわかるように教えてください。できれば、具体的なソースコード付きがいいです」
- どこが問題?

技術習得のレベル

- 段階 1 : 無知の段階 — その技術について聞いたこともない
- 段階 2 : 気がかりな段階 — その技術について文献を読んだことがある
- 段階 3 : 見習いの段階 — その技術について3日間のセミナーに通った
- 段階 4 : 実践しようとする段階 — その技術を実際のプロジェクトに適用しようとしている
- 段階 5 : 職人の段階 — その技術を仕事の上で自然に自動的に使っている
- 段階 6 : 名人の段階 — その技術を完全に消化していて、いつルールを破るべきかを知っている
- 段階 7 : エキスパート — 専門書を著作し、講演し、その技術を拡張する方法を探究する

「守・破・離」

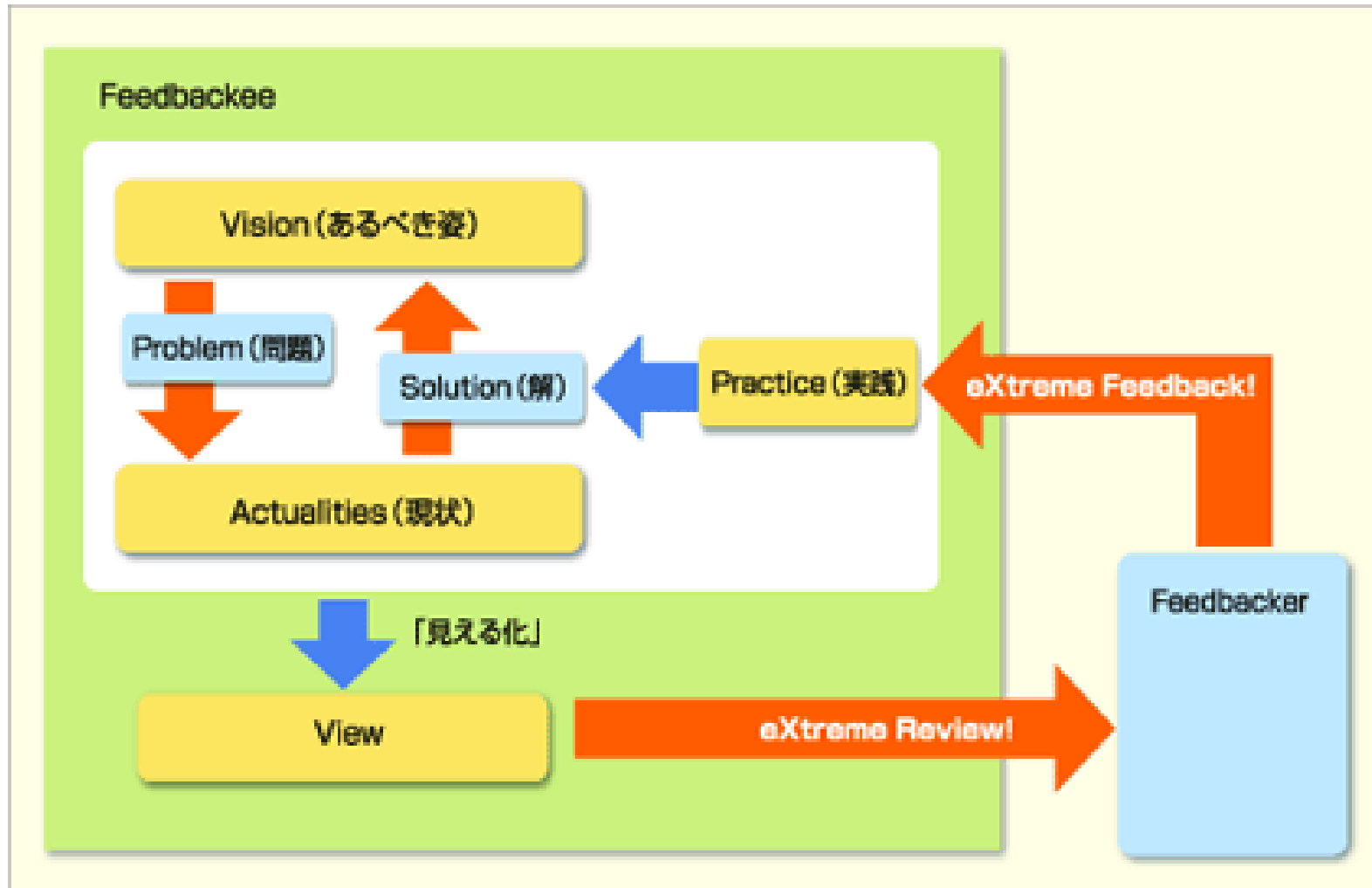
- 人や本などから何かを学び、ひとり立ちしていくまでに、『守』・『破』・『離』という段階を進んでいくと言われている
 - 「守」: 最初は教わった型を守り、型の通りにやる
 - 「破」: 教わった型通りでなく、自分なりに変化させていく
 - 「離」: 最後には型を離れて独自のやり方を作り出し、行う

何故開発がうまく出来ないか

- 「それは識らないことが在るから」
 - 何を作れば良いのか識らない
 - どうやって作れば良いのか識らない
 - 何が使えるのか識らない
 - 情報が何処にあるのか識らない
- 「解決策が分らないのではない。問題が分っていないのだ」 – チェスタートン

カイゼン

フィードバック・ループ



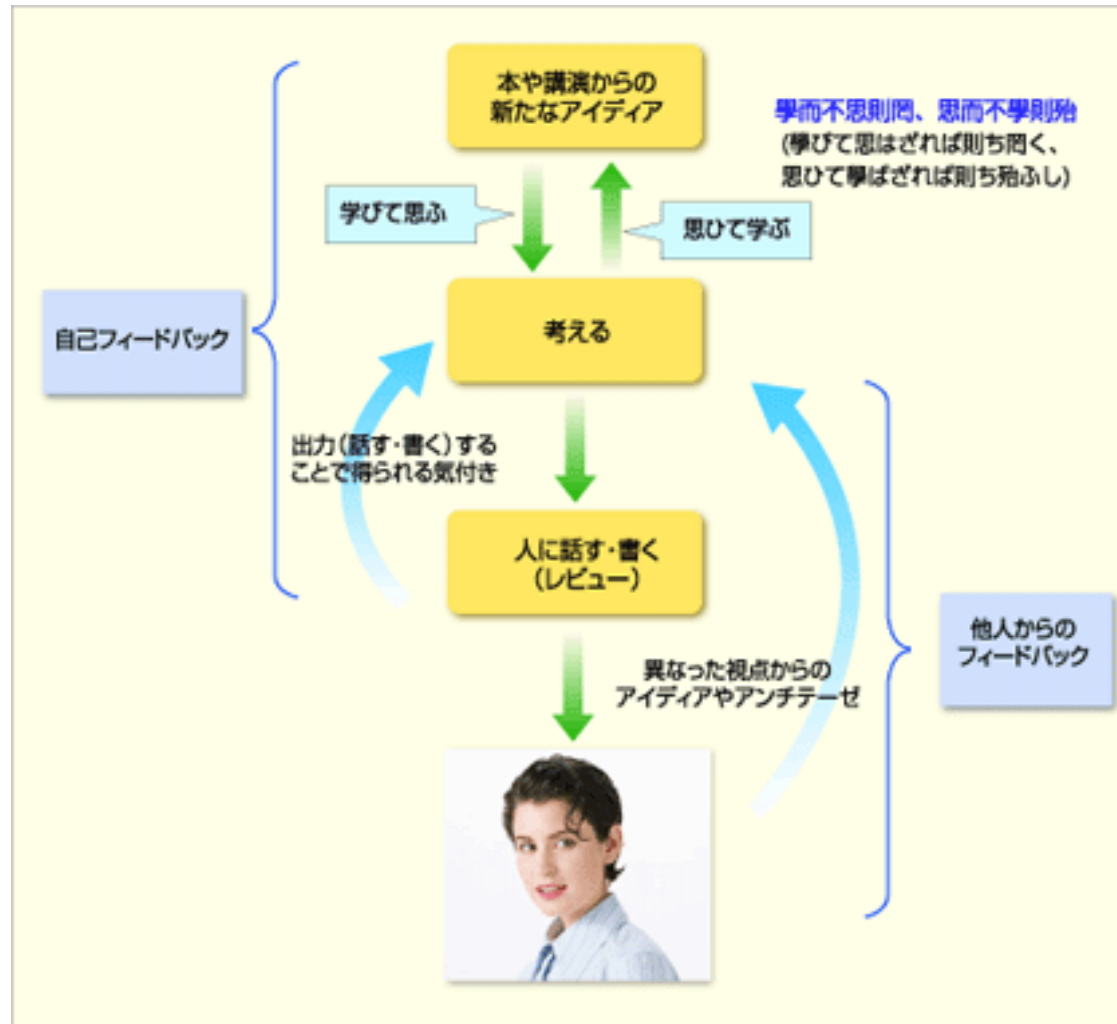
ビジョンと現状

- Problem = ToBe – AsIs
 - 「見える化」によるフィードバック
 - カイゼンの実践が重要
 - PDCA
- 手段に縛られずに、目的に縛られるべき
 - 「カメがウサギに勝てたのは、ウサギを目指したからではなく、ゴールを目指したから」
 - 「カメがウサギに勝てたのは、ウサギがカメを目標にしたのに対し、カメはゴールを目標にしたから」

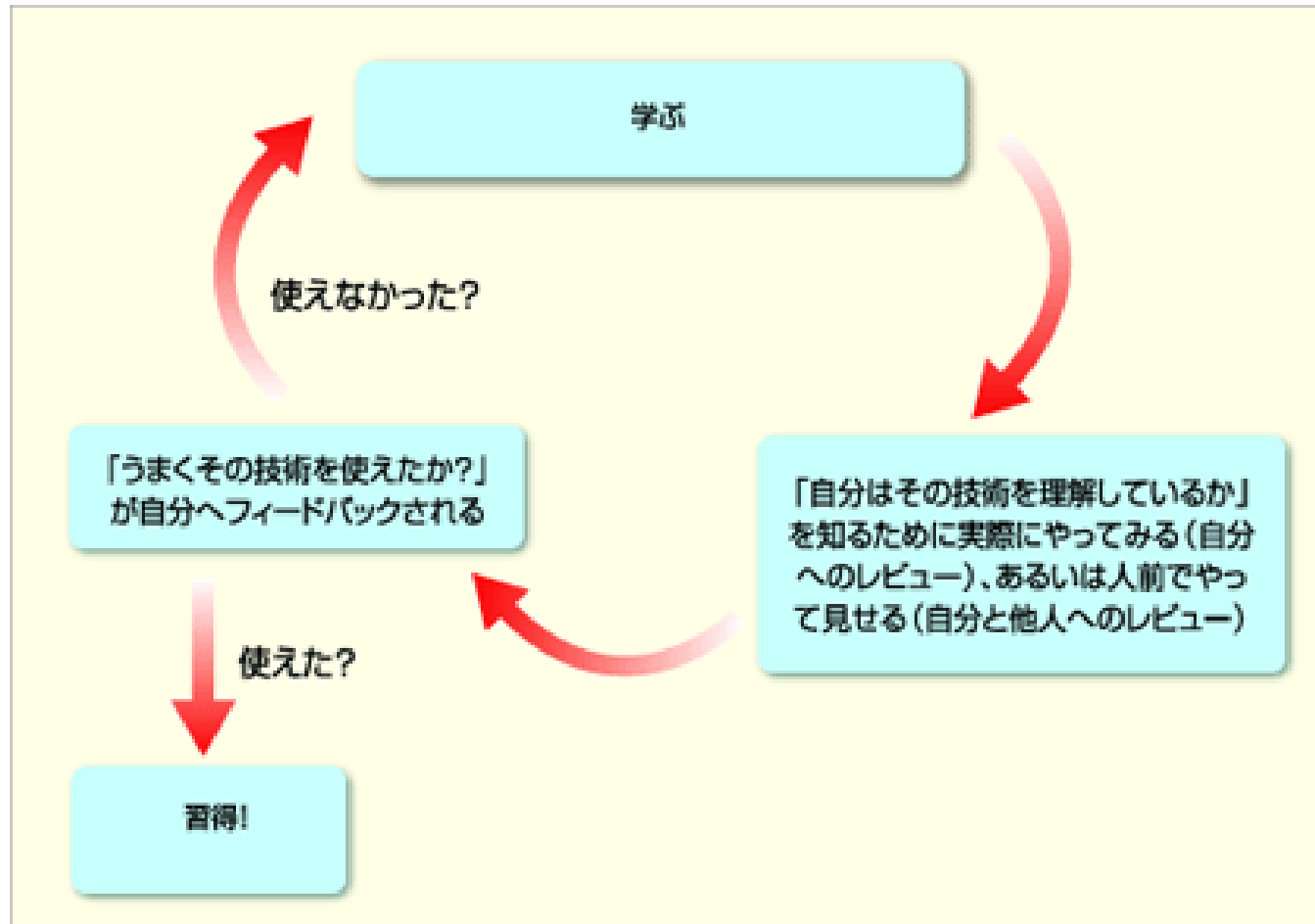
素直さ重要

- 「本当の気付きは、既知であると思ったことを見つめ直す過程で得られるもの」
- 「答えをもって聴かない。無批判に聴いてみる」
- 「実践なくしてフィードバックなし」
- 「完全を求めてはいけない。十分であれば良い」

問題解決能力を改善するループ



カイゼンとフィードバック



カイゼン

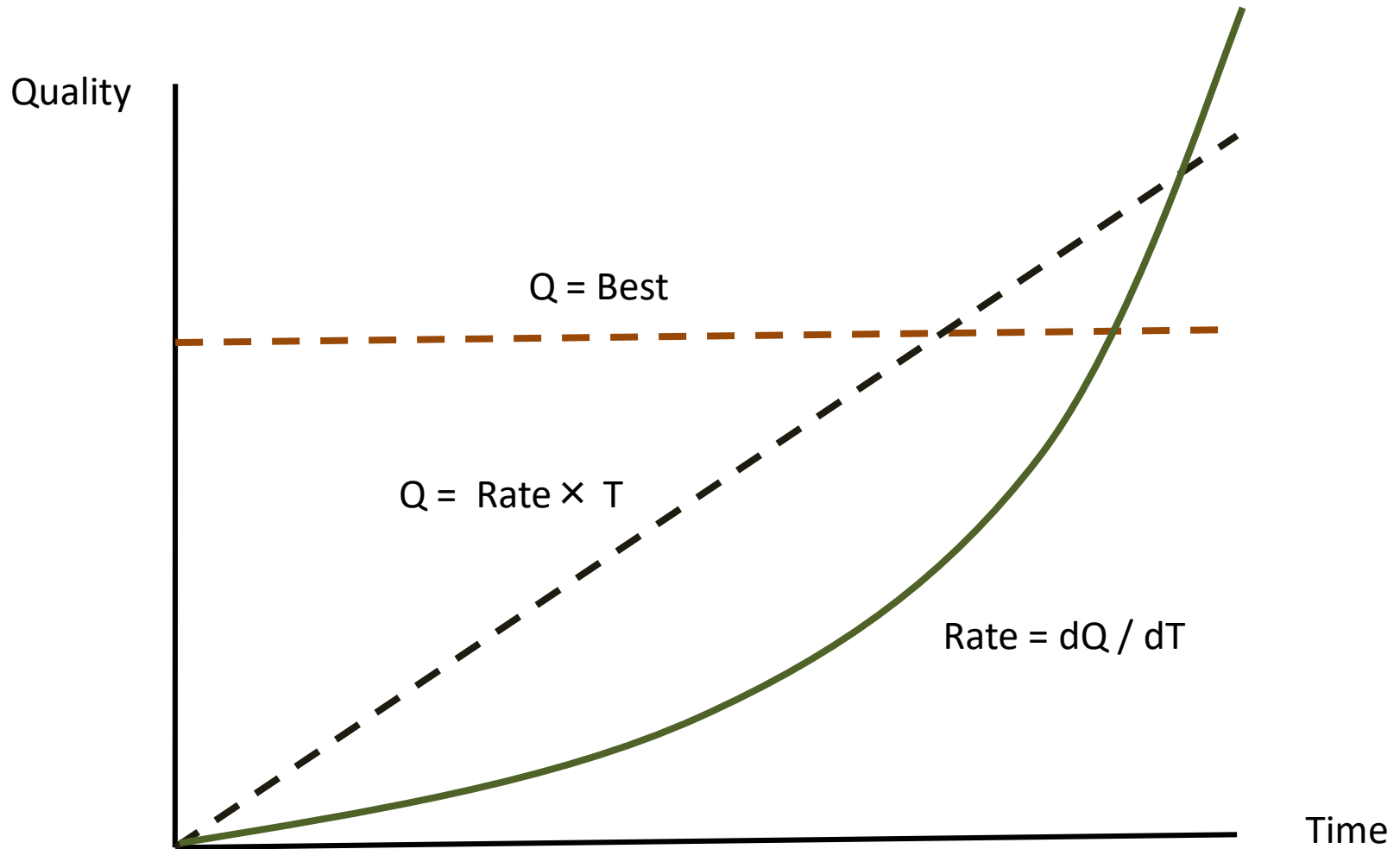
- カイゼン:
 - 「良いことを行うこと」→「良いと思うことを試し続けること」
- フィードバックがキモ
 - 角度が重要 or 角度の変化が重要
- 行動を変化させること
 - “XP is about social change. Social change is changing yourself.”

XP とは人と人とのつながりを変えることである。人と人とのつながりを変えるとは、あなた自身が変わること。

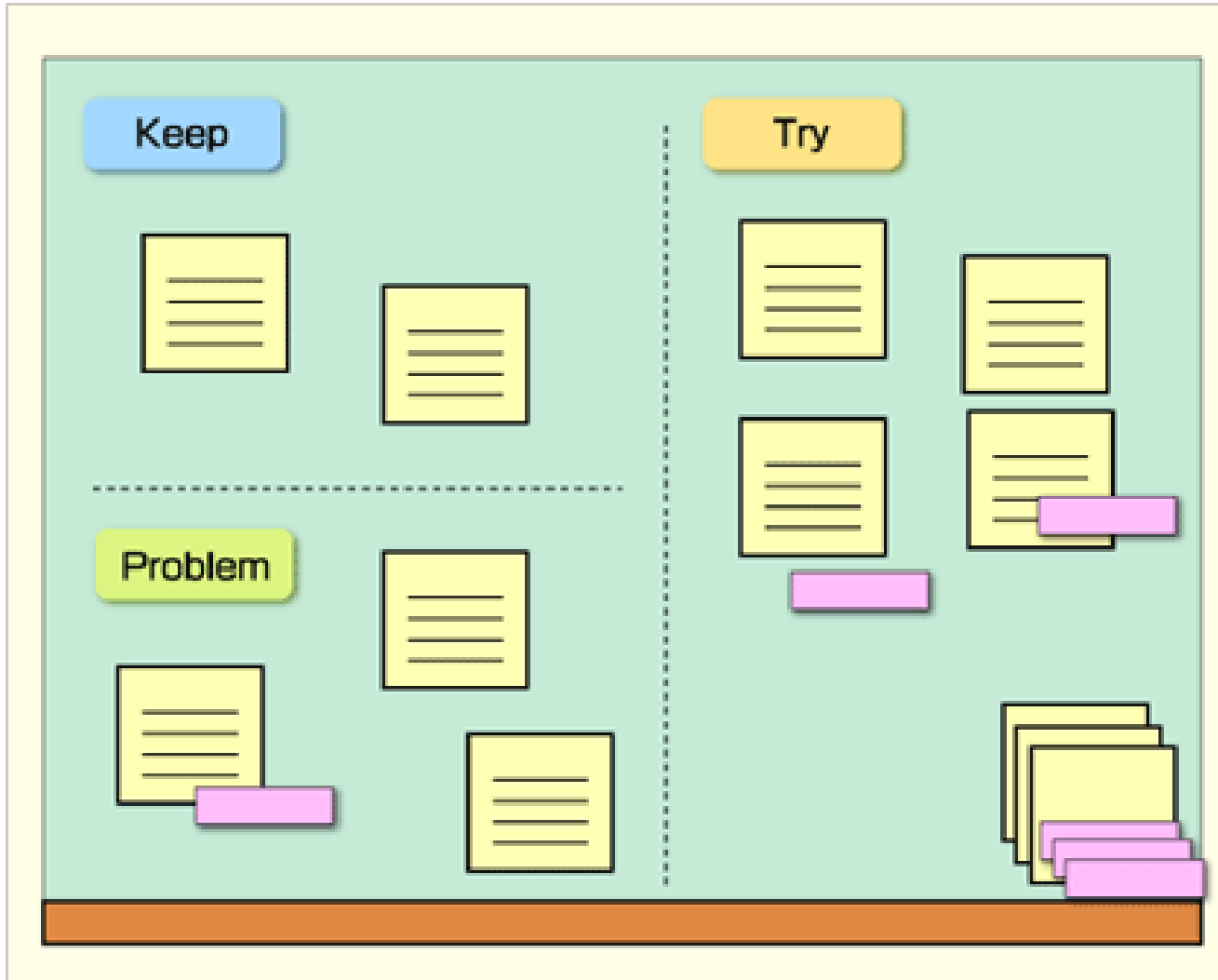
※ Kent Beck の「Extreme Programming Explained: Embrace Change, 2nd Edition」(邦訳: XPエクストリーム・プログラミング入門ー変化を受け入れる 第二版) より。

カイゼン

角度が重要 or 角度の変化が重要



KPT法



創造的でないコミュニケーション

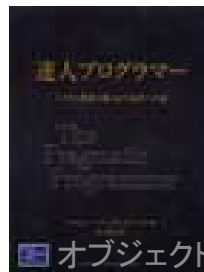
- ずっと愚痴ってるだけ
- どこが悪いかをいうだけで、解決策についてコミュニケーションしない
- 「誰のせいでこんな羽目に陥ったか？」と犯人探しをするだけ
- うまくいっていない点や失敗については、コミュニケーションしない

「創造的コミュニケーション」

- プッシュ型のコミュニケーション
- コミュニケーションにコミットする
- 相手の立場で伝える
- オープンかつ正直なコミュニケーション
(Open, honest Communication)
- 互いに成長する

良書

- 『達人プログラマー システム開発の職人から名匠への道』
 - アンドリュー・ハント、デビッド・トーマス
- 『リファクタリング プログラムの体質改善テクニック』
 - マーチン・ファウラー
- 『アジャイルソフトウェア開発の奥義』
 - ロバート・C・マーチン
- 『UMLモデリングの本質』
 - 児玉公信
- 『オブジェクト指向における再利用のためのデザインパターン』
 - エリック・ガンマ、ラルフ・ジョンソン、リチャード・ヘルム、ジョン・ブリッヂ



2. エンジニアとして過ごす 「人生の時間の質」

QoEL (Quality of Engineering Life)

エンジニアという仕事

- 3K
- 子供にエンジニアを勧められる?

スキルについて

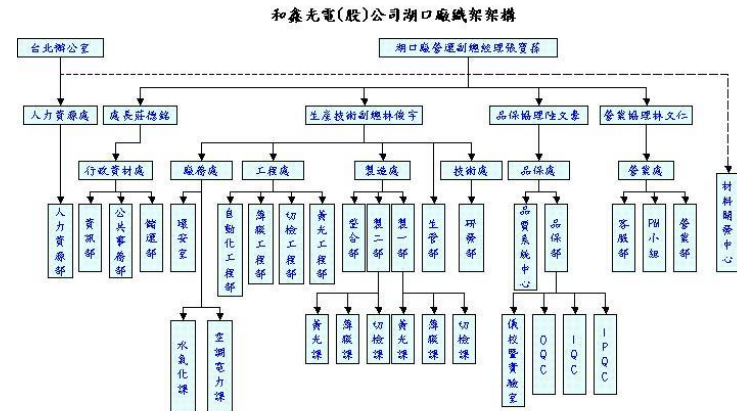
- 企業の考える5年後のスキルと個人の考える5年後のスキル
- Developer 2.0
- スキルの多様化
- ドッグイヤー
 - 周りの人が前に進んでいるのにじっとしているのなら、それは下がっているのと同じ
- 素直さ重要

素直さ重要

- 「本当の気付きは、既知であると思ったことを見つめ直す過程で得られるもの」
- 「答えをもって聴かない。無批判に聴いてみる」
- 「完全を求めてはいけない。十分であれば良い」
- 「手段に縛られずに、目的に縛られるべき」
 - 「カメがウサギに勝てたのは、ウサギを目指したからではなく、ゴールを目指したからである」

人と組織の Win-Win

- 組織と技術者が共に成長し Win-Winの関係を獲得するには



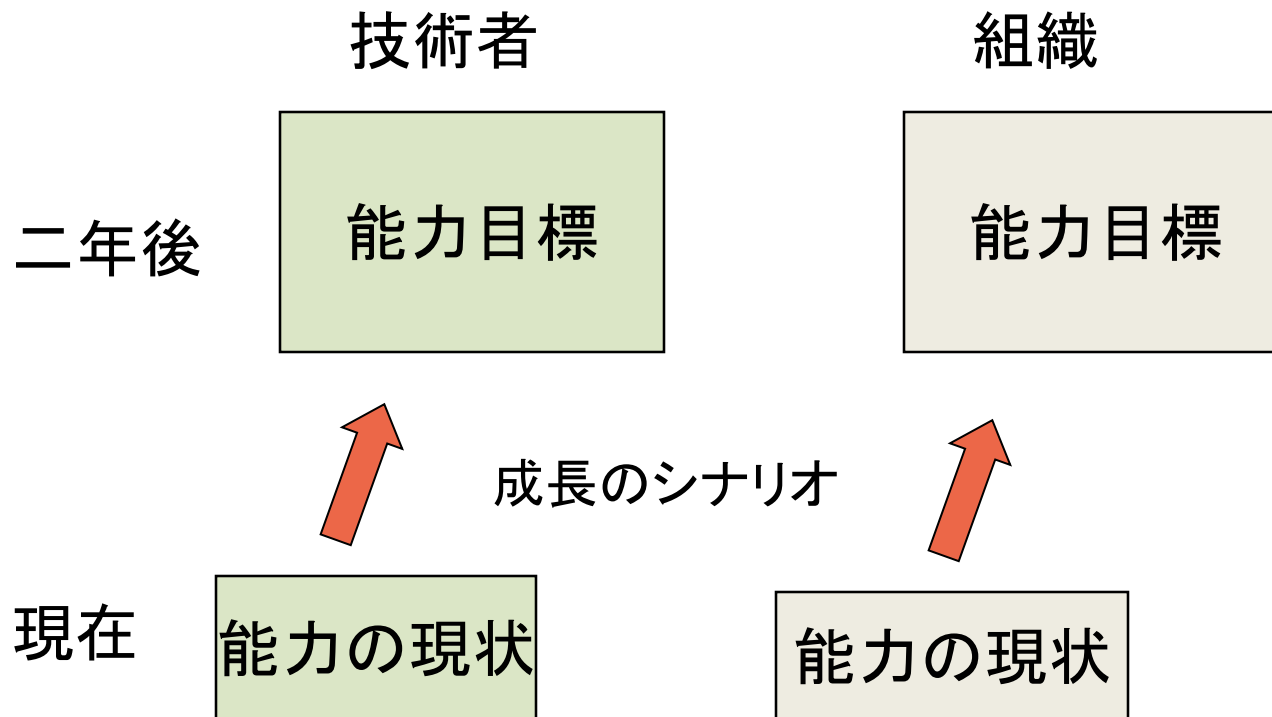
人と組織の Win-Win

- 現状への疑問
 - こんなプロジェクトに属していて大丈夫か？
- 自分と組織の接点
- 成長へのシナリオ
- 自分と組織の成長のモデル



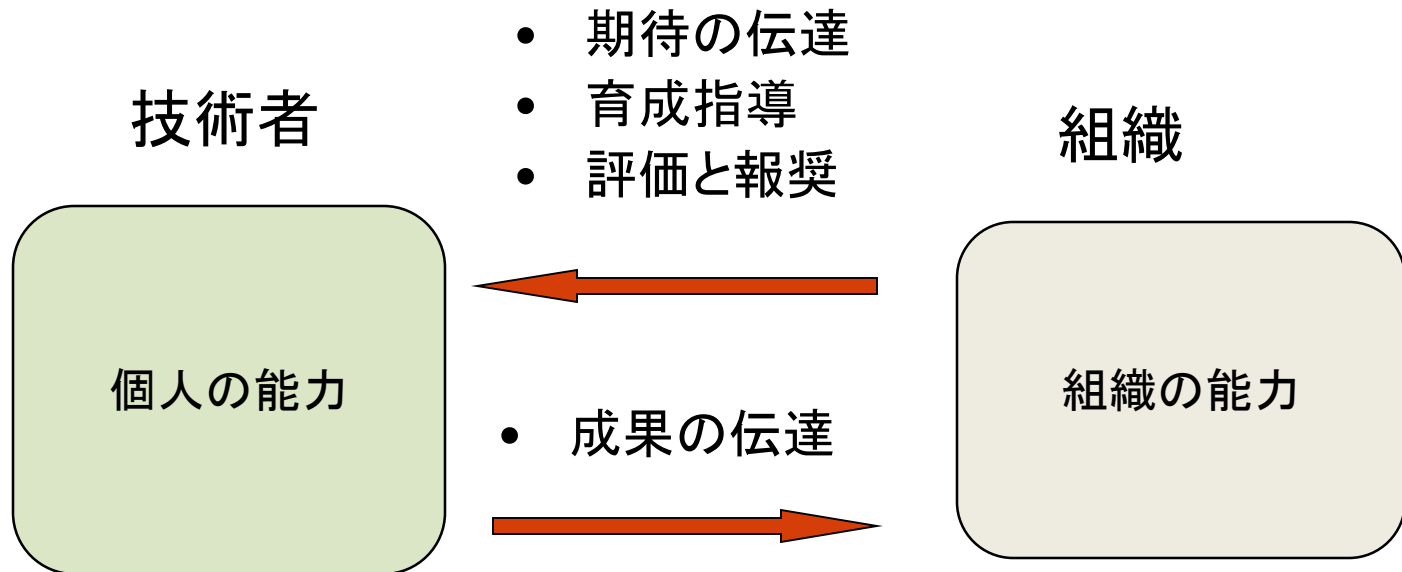
人と組織の Win-Win

- 技術者個人の成長と組織の成長



人と組織の Win-Win

- 技術者個人と組織の成長のための仕組み



そこに足りないのは “Respect”

- 顧客と開発者のゼロサム ゲーム
- マネージャーは開発者の陰口ばかり言い、開発者はマネージャーの陰口ばかり言う

ともに成長していることが
信じられる