

## 01-01. システムの開発手法の種類

### ◇ システム開発の要素

- 設計
- 実装
- テスト

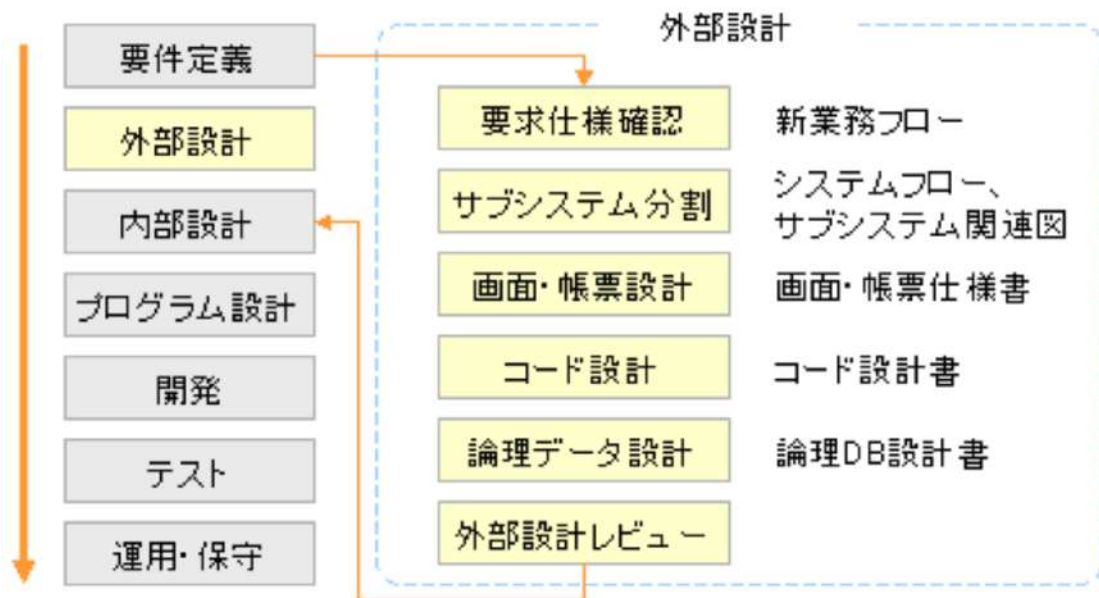
⇒ 15章と16章で解説していく。

### ◇ ウォーターフォール型開発



- 外部設計の詳細

外部設計では、ユーザ向けのシステム設計が行われる。



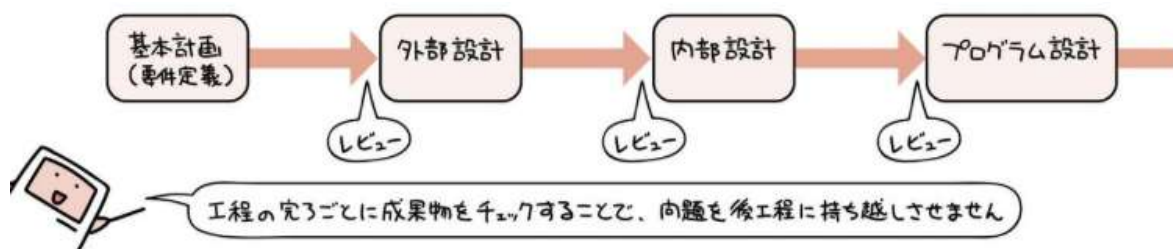
## ◇ プロトタイプ型開発

システム設計に入るまでに試作品を作り、要件定義をより正確にする開発方法。



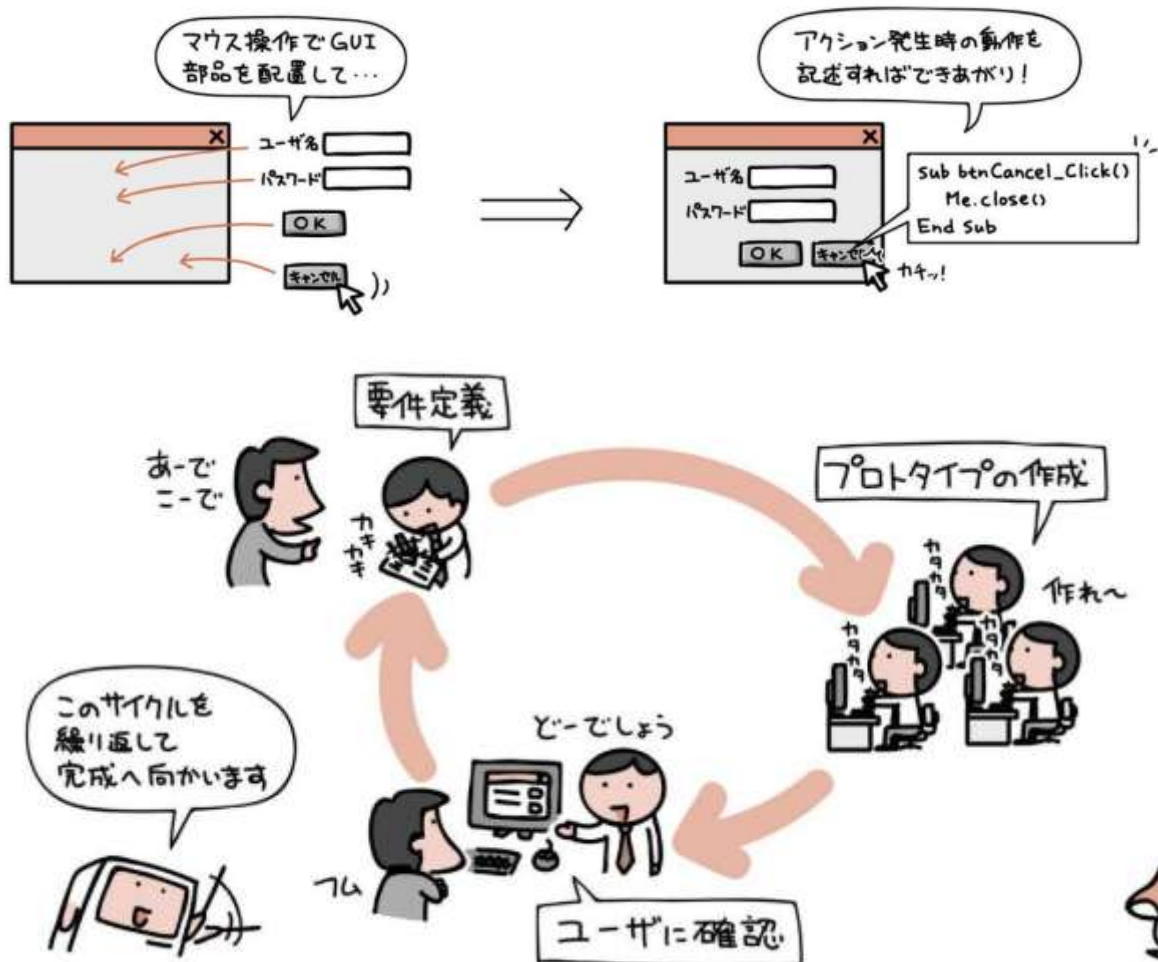
## ◇ レビュー

各工程が完了した段階で、レビューを行う開発方法。



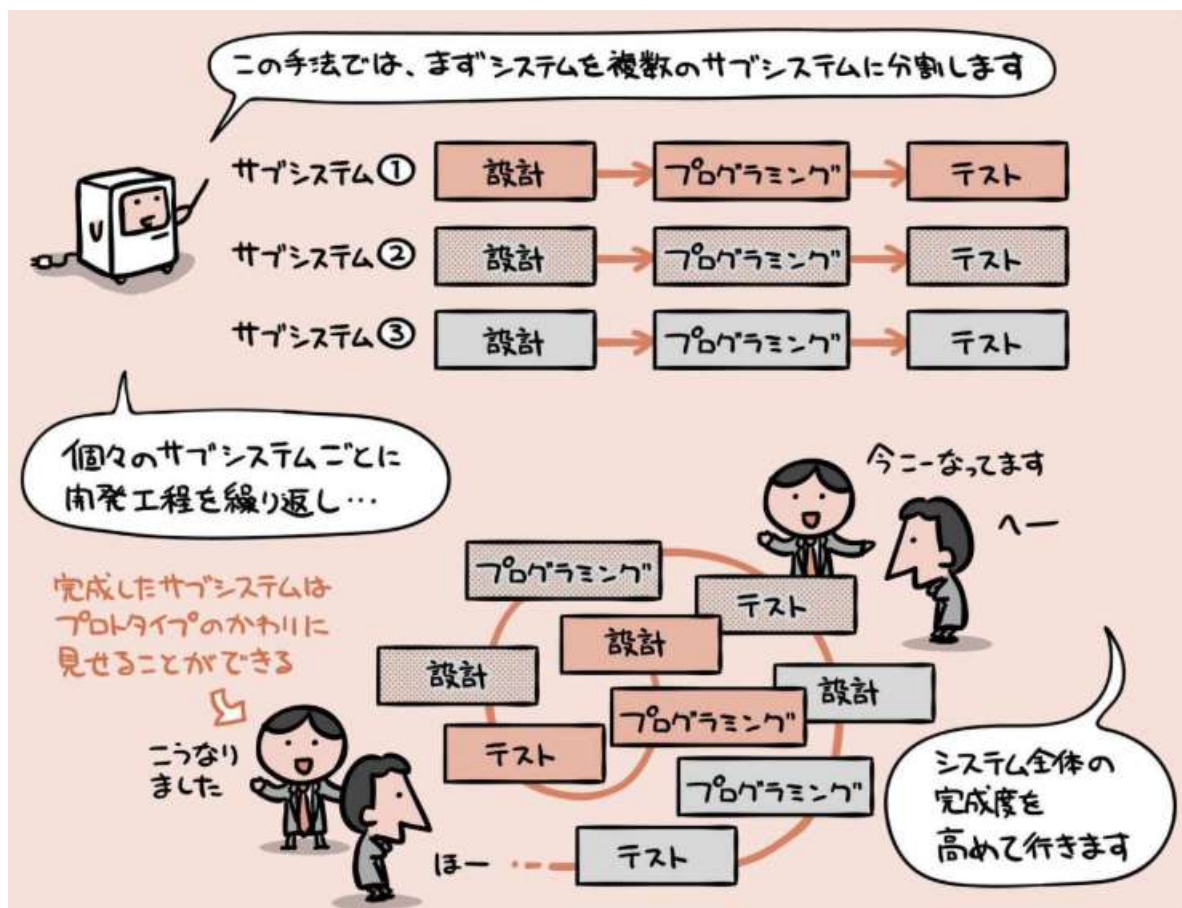
## ◇ RAD (Rapid Application Development)

Visual Basicなどの開発支援ツールを用いて、短期間で設計～テストまでを繰り返す開発方法。



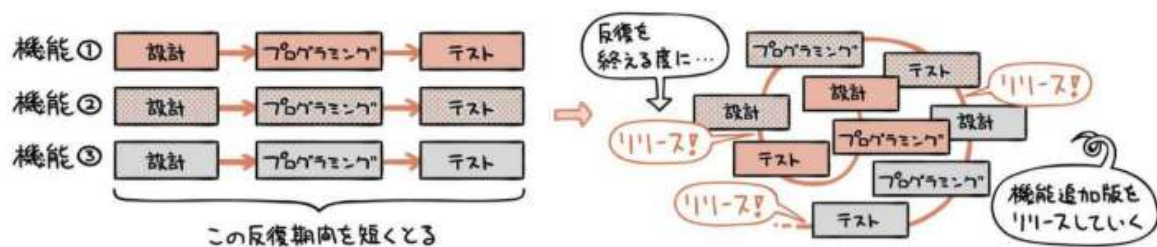
## ◇ スパイラル型開発

システムをいくつかのサブシステムに分割し、ウォーターフォール型開発で各サブシステムを開発していく方法。



## ◇ アジャイル型開発

スパイラルモデルの派生型。スパイラルモデルよりも短い期間で、設計～テストまでを繰り返す開発方法。

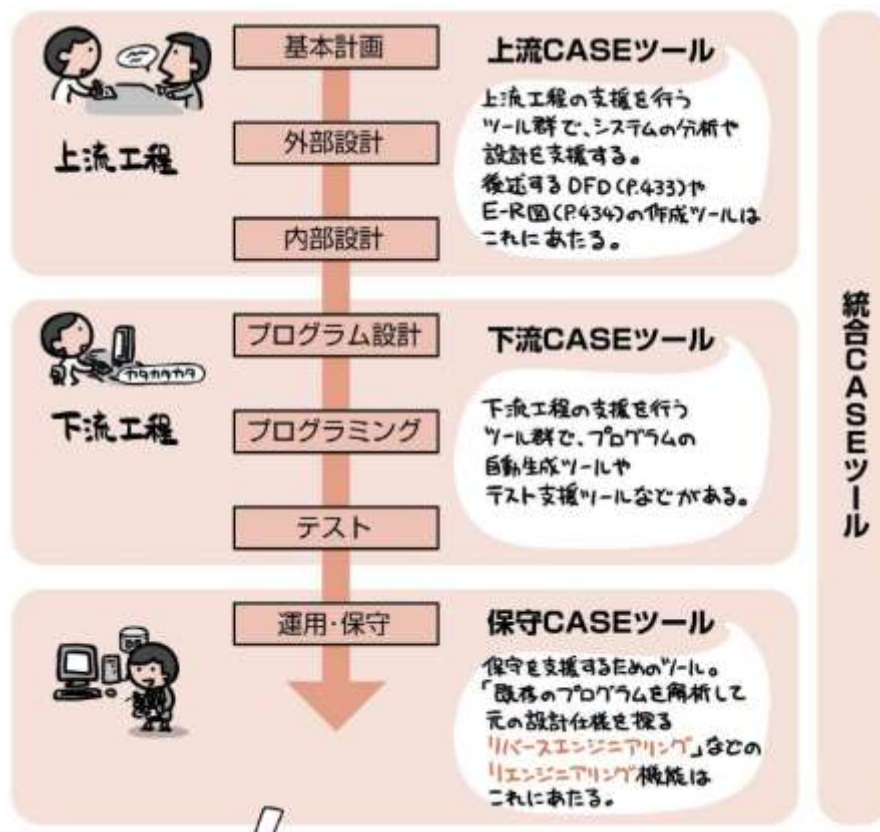


## ◇ CASEツール : Computer Aided Software Engineering

システム開発をサポートするツール群のこと。

- 上流CASEツール  
データフロー図、ER図
- 下流CASEツール  
テスト支援ツール
- 保守CASEツール  
リバースエンジニアリング





## 01-02. サーバーサイドのテスト

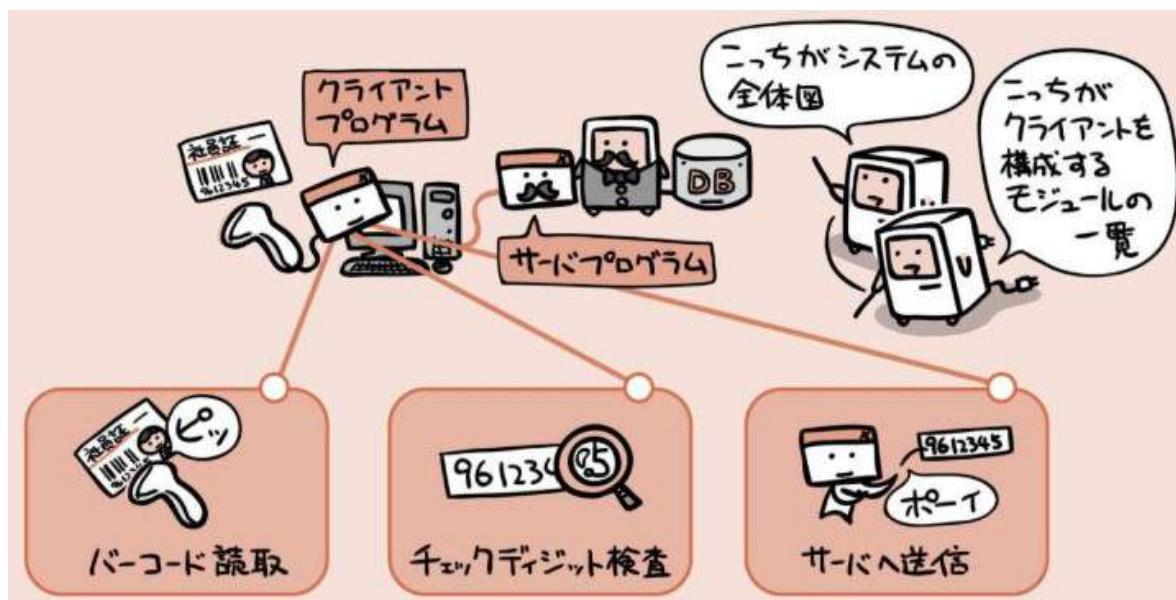
### ◇ テストの流れ

テストは、小さい範囲から大きい範囲へと移行していく。



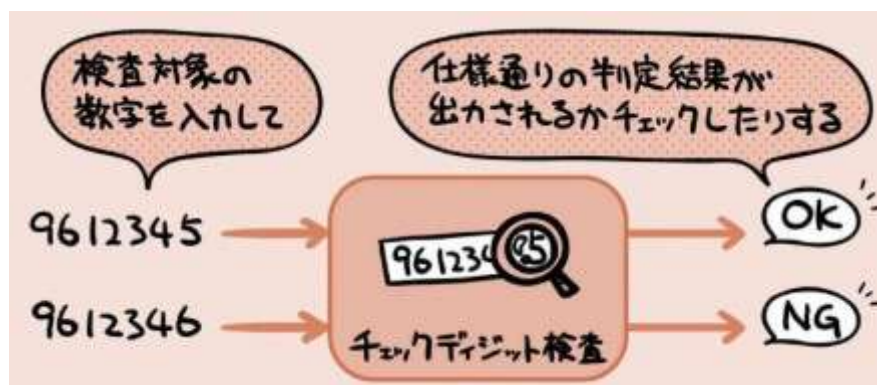
### ◇ 単体テスト

モジュール別に適切に動いているかを検証。ブラックボックステストやホワイトボックステストが用いられる。



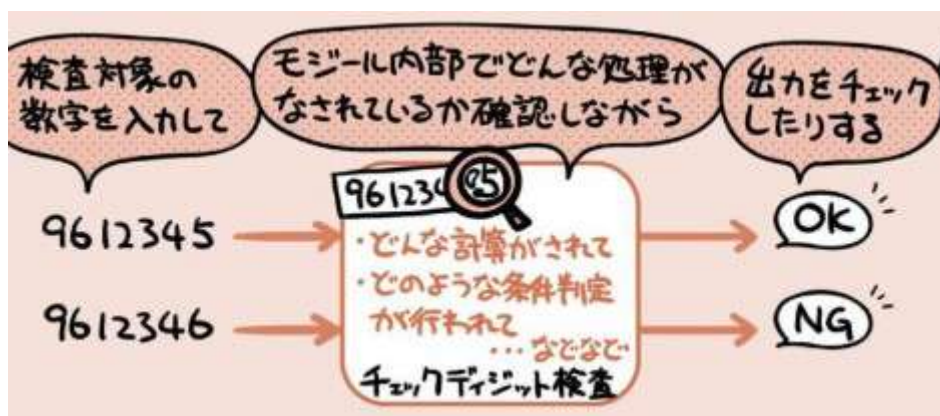
### • ブラックボックステスト

モジュール内の実装内容は気にせず、入力に対して、適切な出力が行われているかを検証。



### • ホワイトボックステスト

モジュール内の実装内容が適切かを確認しながら、入力に対して、適切な出力が行われているかを検証。



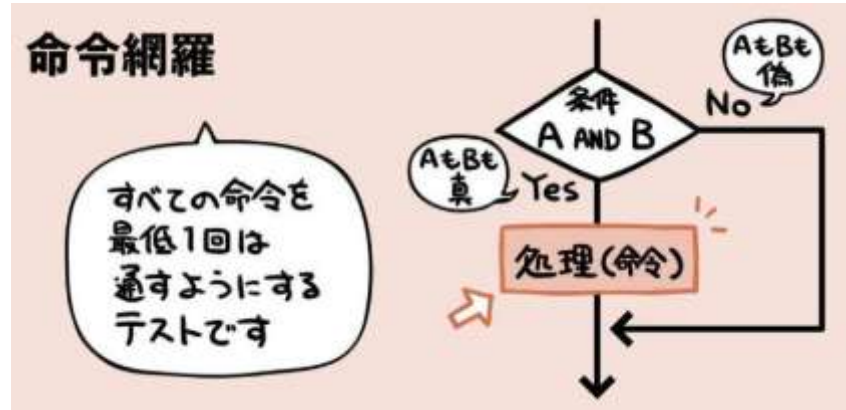
◇ ホワイトボックステストの方法（何をテストするかに着目すれば、思い出しやすい）

【実装例】

```
if (A = 1 && B = 1) {
    return X;
}
```

上記のif文におけるテストとして、以下の4つの方法が考えられる。基本的には、複数条件網羅が用いられる。

- 命令網羅（『全ての処理』が実行されるかをテスト）

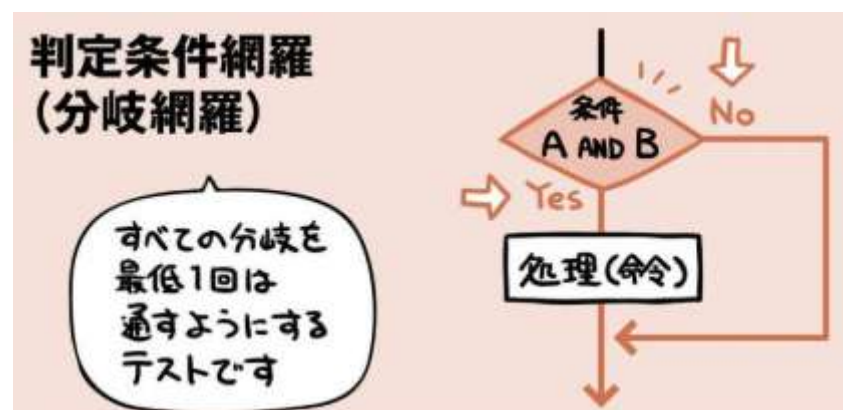


全ての命令が実行されるかをテスト（ここでは処理は1つ）。

すなわち...

A = 1、B = 1 の時、return X が実行されること。

- 判定条件網羅（『全ての分岐』が実行されるかをテスト）



全ての分岐が実行されるかをテスト（ここでは分岐は2つ）。

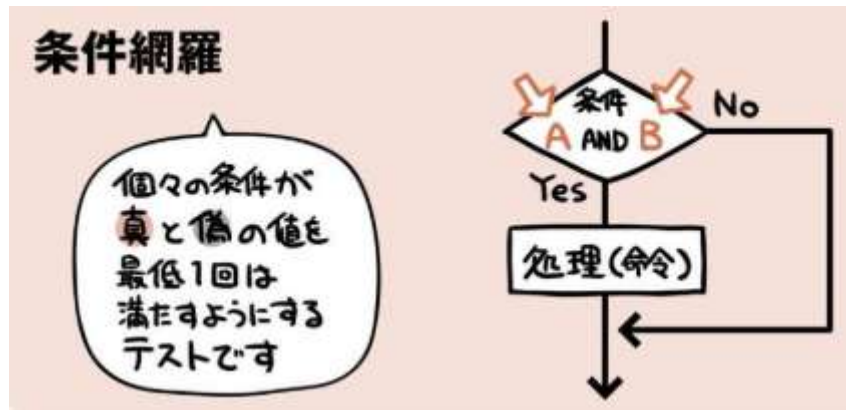
すなわち...

A = 1、B = 1 の時、return X が実行されること。

A = 1、B = 0 の時、return X が実行されないこと。

- 条件網羅（『各条件の取り得る全ての値』が実行されるかをテスト）

## 条件網羅



各条件が、取り得る全ての値で実行されるかをテスト（ここでは、Aが0と1、Bが0と1になる組み合わせなので、2つ）

すなわち...

A = 1、B = 0 の時、return X が実行されないこと。

A = 0、B = 1 の時、return X が実行されないこと。

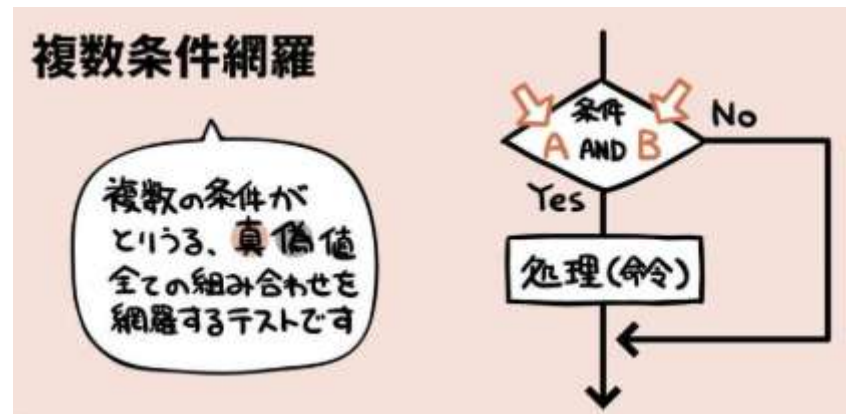
または、次の組み合わせでもよい。

A = 1、B = 1 の時、return X が実行されること。

A = 0、B = 0 の時、return X が実行されないこと。

- 複数条件網羅（『各条件が取り得る全ての値』、かつ『全ての組み合わせ』が実行されるかをテスト）

## 複数条件網羅



各条件が、取り得る全ての値で、かつ全ての組み合わせが実行されるかをテスト（ここでは4つ）

すなわち...

A = 1、B = 1 の時、return X が実行されること。

A = 1、B = 0 の時、return X が実行されないこと。

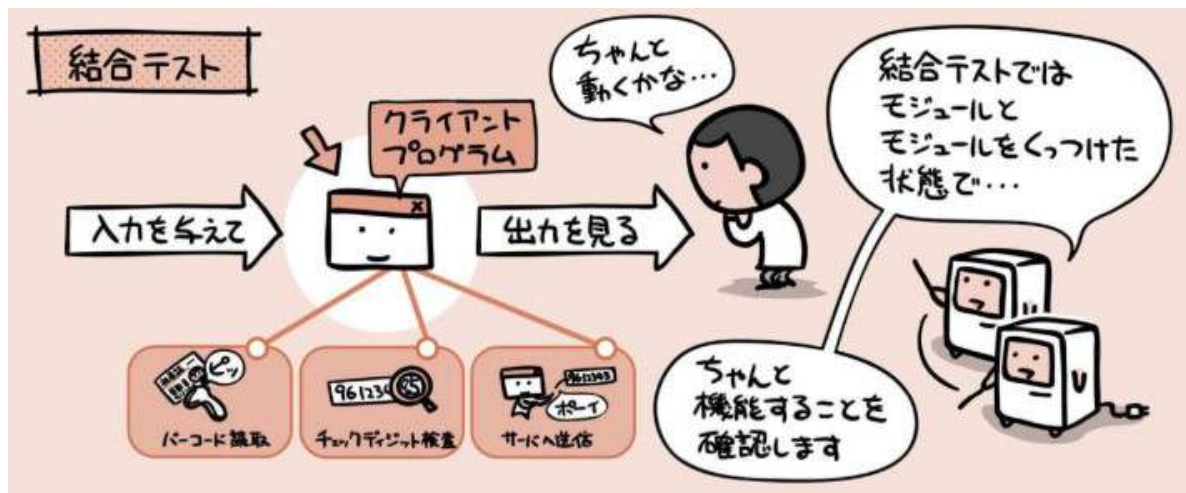
A = 0、B = 1 の時、return X が実行されないこと。

A = 0、B = 0 の時、return X が実行されないこと。

## ◇ 結合テスト

単体テストの次に行うテスト。複数のモジュールを繋げ、モジュール間のインターフェイスが適切に動いているかを検証。

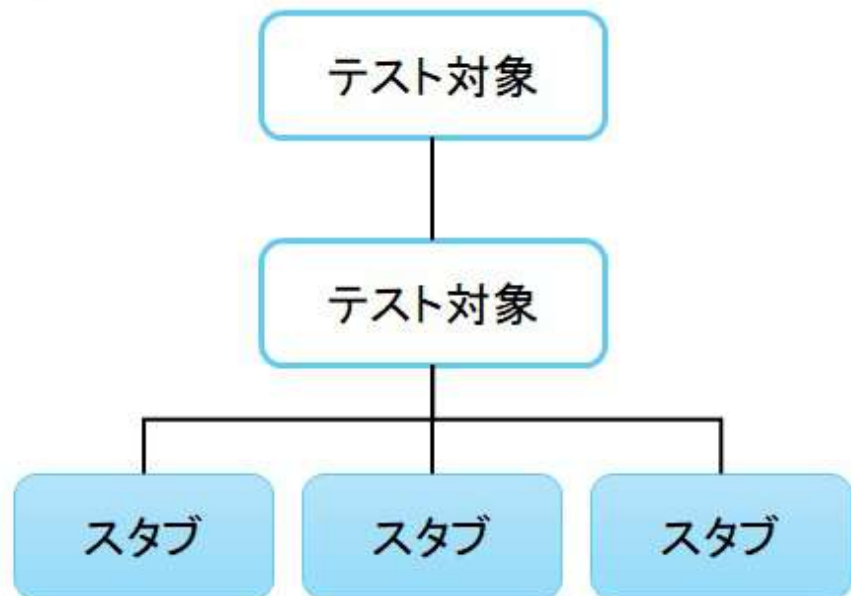




- Top-down テスト

上位のモジュールから下位のモジュールに向かって、結合テストを行う場合、下位には Stub と呼ばれるダミーモジュールを作成する。

上位モジュール

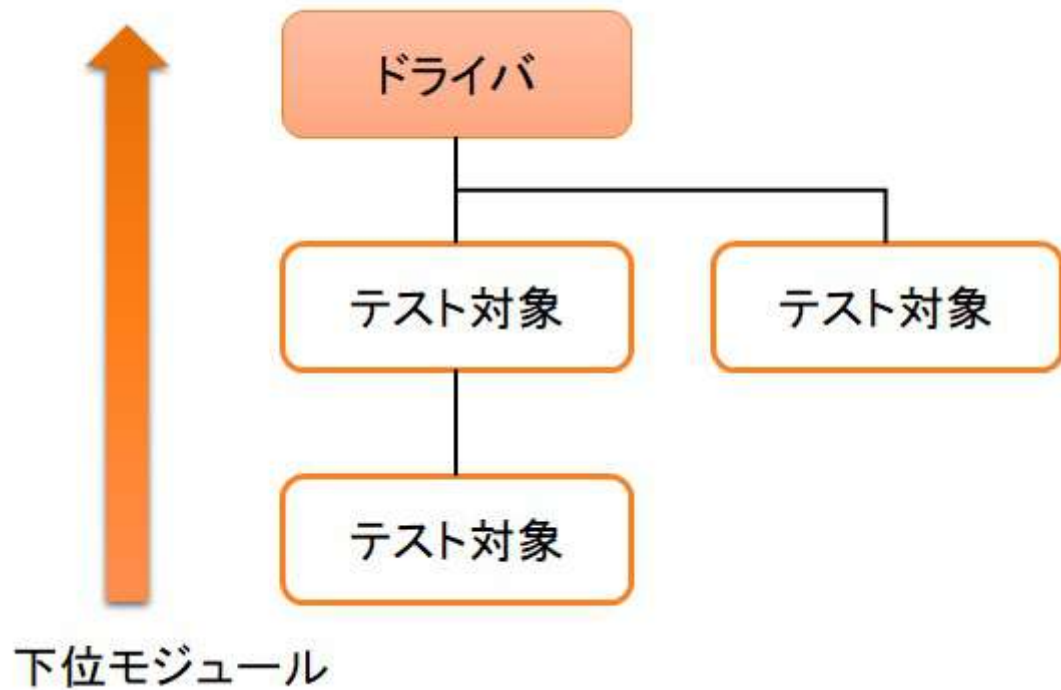


下位モジュール

- Bottom-up テスト

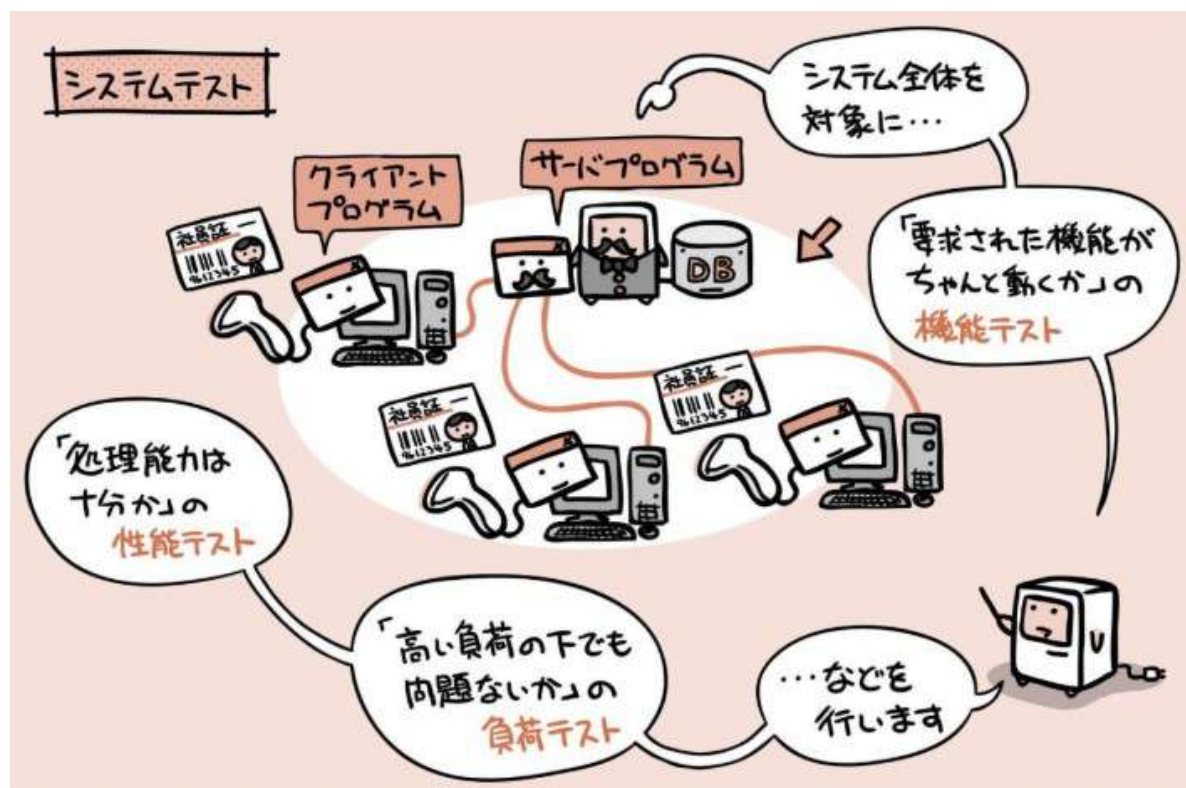
下位のモジュールから上位のモジュールに向かって、結合テストを行う場合、上位には Driver と呼ばれるダミーモジュールを作成する。

上位モジュール



## ◇ システムテスト

結合テストの次に行うテスト。システム全体が適切に動いているかを検証。



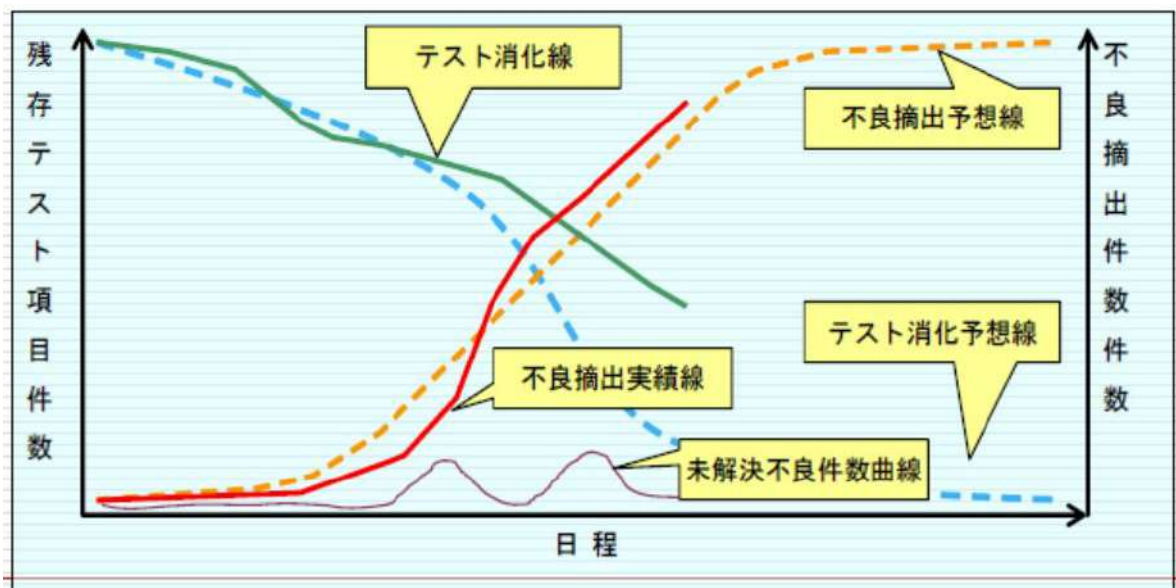
## ◇ Regression テスト

システムを変更した後、他のプログラムに悪影響を与えていないかを検証。



## ◇ バグ管理図

プロジェクトの時、残存テスト数と不良摘出数（バグ発見数）を縦軸にとり、時間を横軸にとること  
で、バグ管理図を作成する。それぞれの曲線の状態から、プロジェクトの進捗状況を読み取ることが  
できる。



不良摘出実績線（信頼度成長曲線）は、プログラムの品質の状態を表し、S字型でないものはプログラ  
ムの品質が良くないことを表す。



## 01-03. システム開発におけるプロジェクト管理

### ◇ 開発規模、工数、生産性の求め方

- 開発規模

(プログラム本数による開発規模) = (プログラム本数)

(プログラム行数による開発規模) = (kステップ行数)

- **工数**

(人月による工数) = (人数・月) = (人数 × 月数)

(人時による工数) = (人数・時) = (人数 × 時間)

- **生産性**

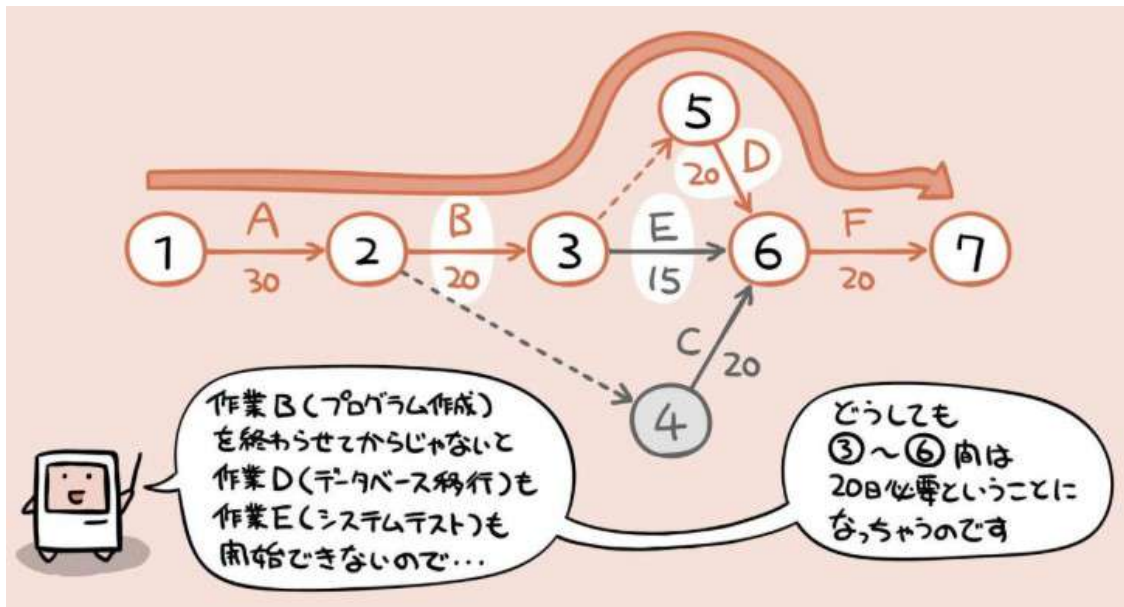
(プログラム本数の生産性) = (プログラム本数 / 人時) = (プログラム本数) ÷ (人数 × 時間)

(kステップ行数の生産性) = (kステップ行数 / 人月) = (kステップ行数) ÷ (人数 × 月数)

## ◇ Arrow ダイアグラム

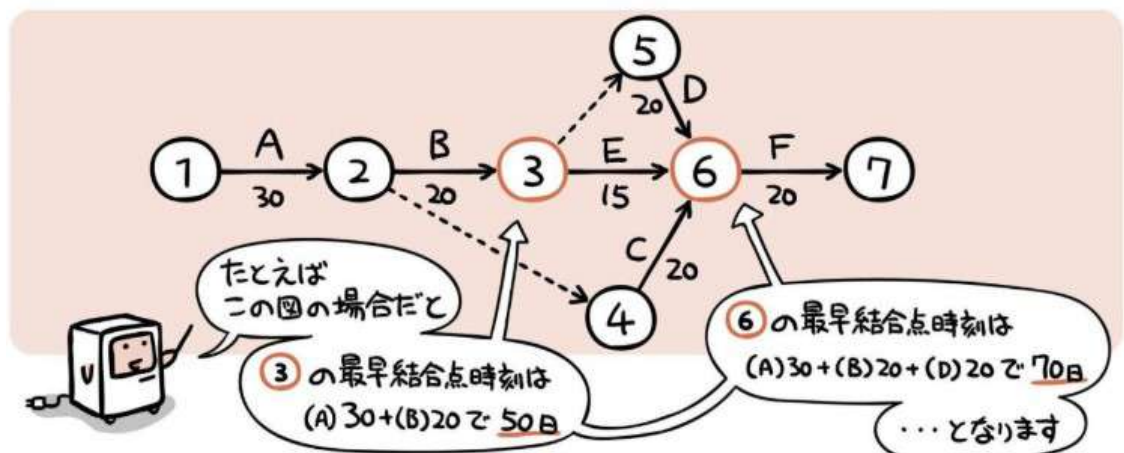
- **プロジェクトに必要な日数**

全体的な工程に必要な日数は、所要日数が最も多い経路に影響される。この経路を、Critical Path という。



- **最早結合点時刻**

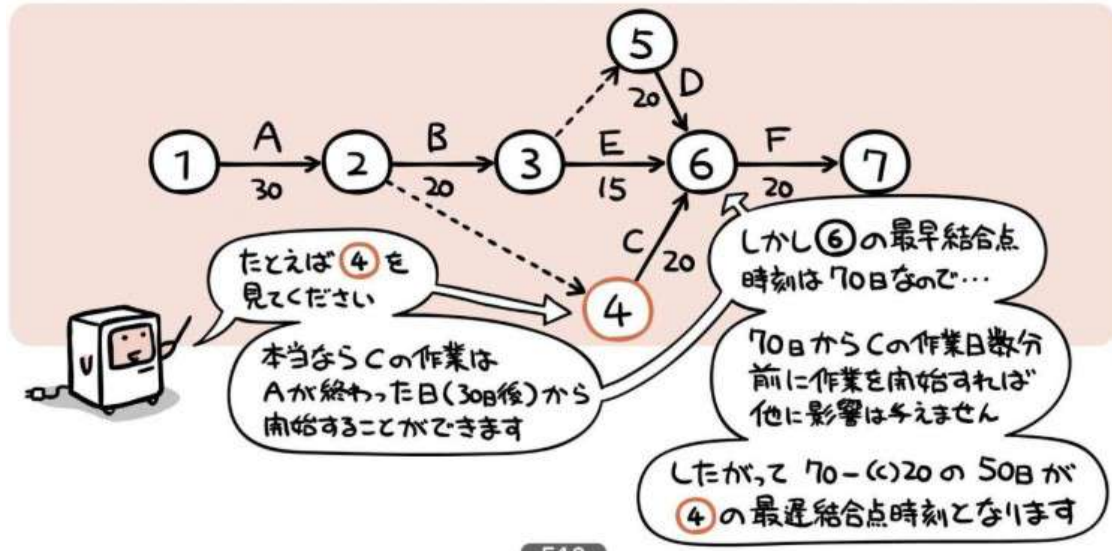
全体的な工程の中で、任意の結合点に取り掛かるために必要な最少日数のこと。Critical Path に影響されるので、注意。





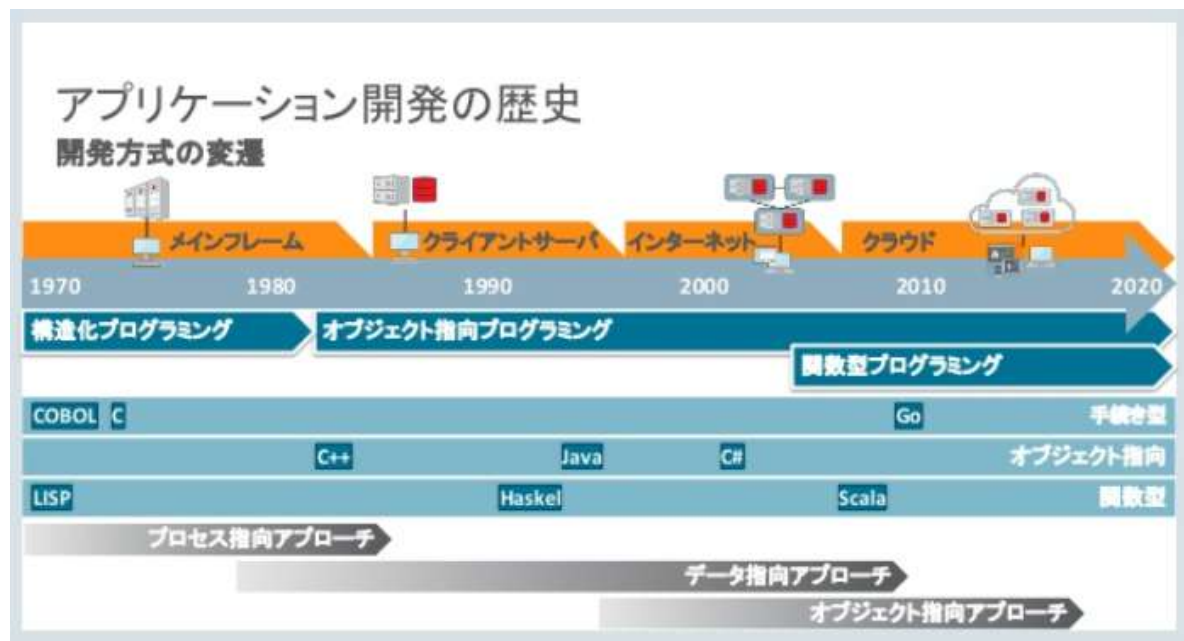
- 最遅結合点時刻

全体的な工程の中で、任意の結合点に取り掛かるために必要な最大日数のこと。



## 02-01. システム開発における設計方法

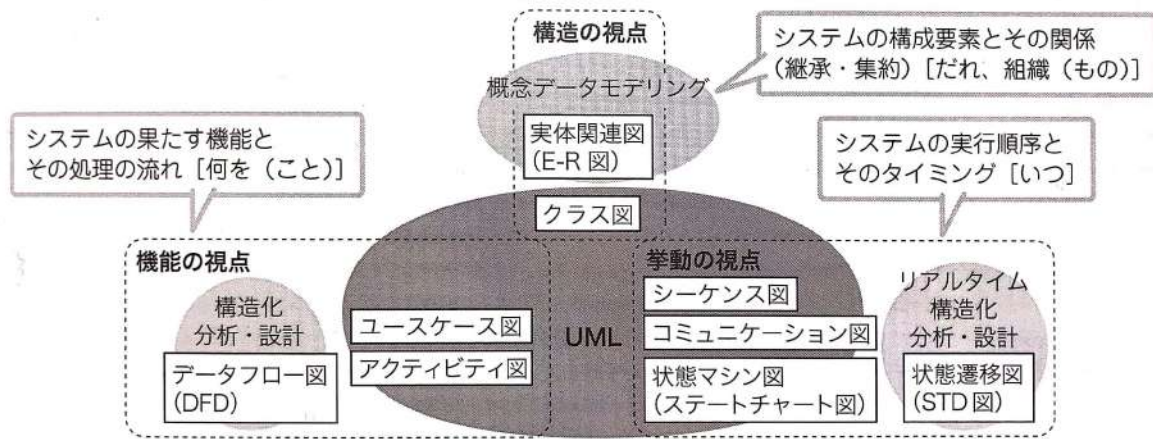
### ◇ システムの設計方法の種類と歴史



## 02-02. システム設計における設計図の種類

### ◇ オブジェクトモデリングの図式化方法の種類

設計図においては、オブジェクトモデリングや、オブジェクト間の関係を図式化することが必要になる。



## ◇ UML : Unified Modeling Language (統一モデリング言語)

オブジェクト指向プログラミング型のシステム設計を行うには、まず、オブジェクトモデリングを行う必要がある。オブジェクトモデリングをダイアグラム図を用いて解りやすく視覚化する表記方法を Unified Modeling Language という。UMLにおけるダイアグラム図は、構造の視点に基づく構造図と、振舞いの視点に基づく振舞図に分類される。

(※ちなみに、UMLは、システム設計だけでなく、データベース設計にも使える)



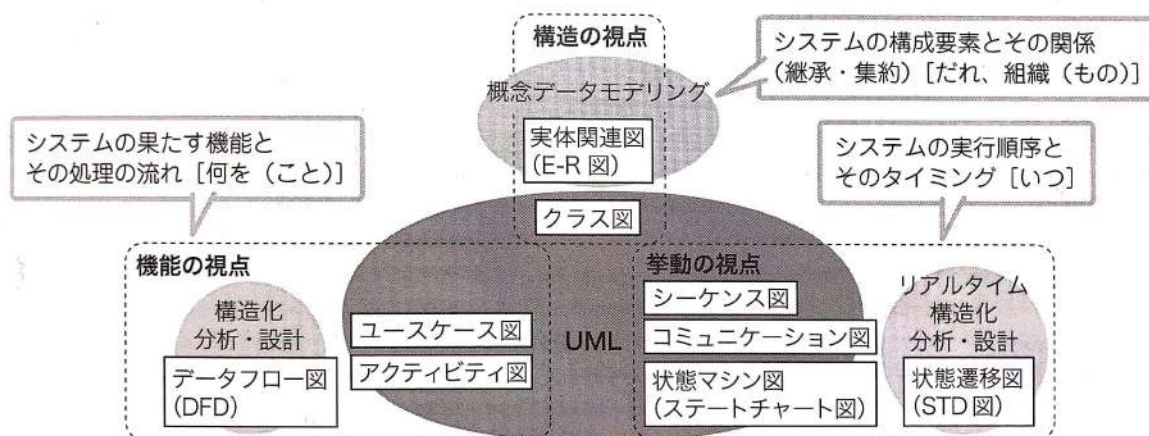
## ◇ UMLのダイアグラム一覧

構造図	クラス図 (class diagram)	クラスの定義や、関連付けなど、クラス構造をあらわす。
	オブジェクト図 (object diagram)	クラスを実体化させるインスタンス (オブジェクト) の、具体的な関係をあらわす。
	パッケージ図 (package diagram)	クラスなどがどのようにグループ化されているかをあらわす。
	コンポーネント図 (component diagram)	処理を構成する複数のクラスを1つのコンポーネントと見なし、その内部構造と相互の関係をあらわす。
	複合構造図 (composite structure diagram)	複数クラスを内包するクラスやコンポーネントの内部構造をあらわす。
	配置図 (deployment diagram)	システムを構成する物理的な構造をあらわす。

振る舞い図	ユースケース図 (use case diagram)	利用者や外部システムからの要求に対して、システムがどのような振る舞いをするかをあらわす。
	アクティビティ図 (activity diagram)	システム実行時における、一連の処理の流れや状態遷移をあらわす。フローチャートのなもの。
	状態マシン図 (state machine diagram)	イベントによって起こる、オブジェクトの状態遷移をあらわす。
	シーケンス図 (sequence diagram)	オブジェクト間のやり取りを、時系列にそってあらわす。
	コミュニケーション図 (communication diagram)	オブジェクト間の関連と、そこで行われるメッセージのやり取りをあらわす。
	相互作用概要図 (interaction overview diagram)	ユースケース図やシーケンス図などを構成要素として、より大枠の処理の流れをあらわす。アクティビティ図の変形。
	タイミング図 (timing diagram)	オブジェクトの状態遷移を時系列であらわす。

## 02-03. UMLの構造図：『構造』の視点

### ◇ オブジェクトモデリングの図式化方法の種類（再掲）









構造図	クラス図 (class diagram)	クラスの定義や、関連付けなど、クラス構造をあらわす。
	オブジェクト図 (object diagram)	クラスを実体化させるインスタンス (オブジェクト) の、具体的な関係をあらわす。
	パッケージ図 (package diagram)	クラスなどがどのようにグループ化されているかをあらわす。
	コンポーネント図 (component diagram)	処理を構成する複数のクラスを1つのコンポーネントと見なし、その内部構造と相互の関係をあらわす。
	複合構造図 (composite structure diagram)	複数クラスを内包するクラスやコンポーネントの内部構造をあらわす。
	配置図 (deployment diagram)	システムを構成する物理的な構造をあらわす。

### ◇ クラス図



クラスの構造、クラス間の関係、役割を表記する方法。

関連 (association)		基本的なつながりをあらわす。
集約 (aggregation)		クラスBは、クラスAの一部である。ただし、両者にライフサイクルの依存関係はない。
コンポジション (composition)		クラスBはクラスAの一部であり、クラスAが削除されるとクラスBもあわせて削除される。
依存 (dependency)		クラスAが変更された時、クラスBも変更が生じる依存関係にある。
汎化 (generalization)		クラスBはクラスAの性質を継承している。クラスAがスーパークラスであり、クラスBはサブクラスの関係にある。
実現 (realization)		抽象的な定義にとどまるクラスAの振る舞いを、具体的の実装したものがクラスBである。

- Class

1つのクラスを、クラス区画、属性区画、操作区画の3要素で表記する方法。



- Association (関連)

2つのクラスを関連させる場合、クラスを線で繋ぐことで関連性を表記する方法。クラス図の実装の章を参照せよ。



- Aggregation (集約)

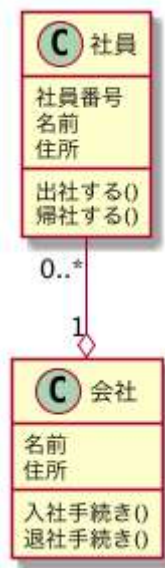
クラスと別のクラスが、全体と部分の関係であることを表記する方法。クラス図の実装の章を参照せよ。



### 【具体例】

社員は1つの会社に所属する場合

「社員」から見た「会社」は1つである。逆に、「会社」からみた「社員」は0人以上であることを表現。



- **Composition (合成)**

クラス図の実装の章を参照せよ。

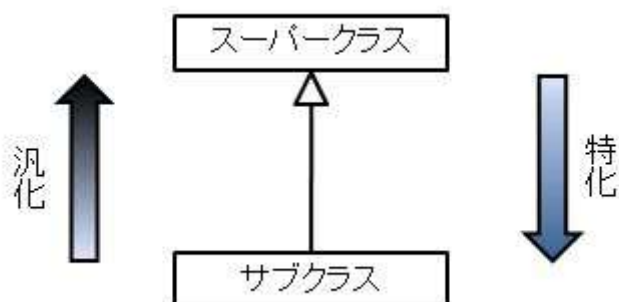
- **Dependency (依存)**

- **Realization (実現) = 抽象オブジェクト**

クラス図の実装の章を参照せよ。

- **Generalization (汎化)**

クラス間で属性、操作、関連を引継ぐことを表記する方法。サブクラスから見たスーパークラスとの関係を『汎化』、逆にスーパークラスから見たサブクラスとの関係を『特化』という。プログラミングにおける『継承』は、特化を実装する方法の一つ。



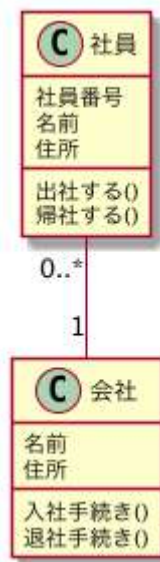
- **Cardinality (多重度)**

クラスと別のクラスが、何個と何個で関係しているかを表記する方法。

### 【具体例】

社員は1つの会社にしか所属できない場合







「社員」から見た「会社」は1つである。逆に、「会社」からみた「社員」は0人以上であるという表記。



多重度の表記	説明
0.. 1	0 または 1
1	1
*	0 以上
0.. *	0 以上
1.. *	1 以上
m.. n	m以上n以下

## 02-04. UMLの構造図の実装方法

---

関連 (association)		基本的なつながりをあらわす。
集約 (aggregation)		クラスBは、クラスAの一部である。ただし、両者にライフサイクルの依存関係はない。
コンポジション (composition)		クラスBはクラスAの一部であり、クラスAが削除されるとクラスBもあわせて削除される。
依存 (dependency)		クラスAが変更された時、クラスBも変更が生じる依存関係にある。
汎化 (generalization)		クラスBはクラスAの性質を継承している。クラスAがスーパークラスであり、クラスBはサブクラスの関係にある。
実現 (realization)		抽象的な定義にとどまるクラスAの振る舞いを、具体的に実装したものがクラスBである。

## ◇ Association（関連）

## ◇ Aggregation（集約）

【Tireクラス】

```
class Tire {}
```

【CarXクラス】

```
//CarXクラス定義
class CarX
{
    //CarXクラスがタイヤクラスを引数として扱えるように設定
    public function __construct(Tire $t1, Tire $t2, Tire $t3, Tire $t4)
    {
        $this->tire1 = $t1;
        $this->tire2 = $t2;
        $this->tire3 = $t3;
        $this->tire4 = $t4;
    }
}
```

【CarYクラス】

```
//CarYクラス定義
class CarY
{
    //CarYクラスがタイヤクラスを引数として扱えるように設定
    public function __construct(Tire $t1, Tire $t2, Tire $t3, Tire $t4)
    {
        //引数のTireクラスからプロパティにアクセス
        $this->tire1 = $t1;
        $this->tire2 = $t2;
        $this->tire3 = $t3;
        $this->tire4 = $t4;
    }
}
```

以下の様に、Tireクラスのインスタンスを、CarXクラスとCarYクラスの引数として用いている。  
Tireクラスの各インスタンスと、2つのCarクラスの双方向で、依存関係はない。

```
//Tireクラスをインスタンス化
$tire1 = new Tire();
$tire2 = new Tire();
$tire3 = new Tire();
$tire4 = new Tire();
$tire5 = new Tire();
$tire6 = new Tire();

//Tireクラスのインスタンスを引数として扱う
$suv = new CarX($tire1, $tire2, $tire3, $tire4);

//Tireクラスのインスタンスを引数として扱う
$suv = new CarY($tire1, $tire2, $tire5, $tire6);
```

## ◇ Composition（合成）

【Lockクラス】

```
//Lockクラス定義
class Lock {}
```

【Keyクラス】

```
//Keyクラス定義
class Key {

    public function __construct()
    {

    }

}
```

【Carクラス】



```
//Carクラスを定義
class Car
{

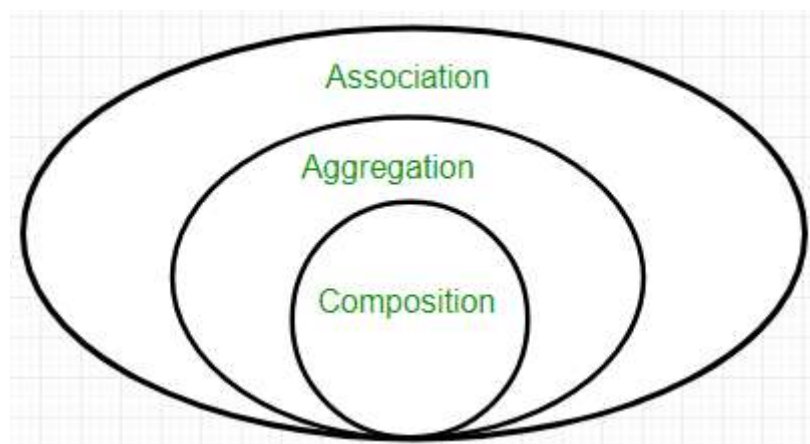
    public function __construct()
    {
        //引数Lockクラスをインスタンス化
        $lock = new Lock();
    }
}
```

以下の様に、LockクラスのLockインスタンスは、Carクラスの中で定義されているため、Lockインスタンスにはアクセスできない。また、Carクラスが起動しなければ、Lockインスタンスは起動できない。このように、LockインスタンスからCarクラスの方には、強い依存関係がある。

```
//エラーになる。$lockには直接アクセスできない。
$key = new Key($lock);
```

## ◇ Dependency（依存）※関連、集約、合成の依存性の違い

『Association > Aggregation > Composition』の順で、依存性が低い。



## ◇ Generalization（汎化）

- 汎化におけるOverride

汎化の時、子クラスでメソッドの処理内容を再び実装すると、処理内容は上書きされる。

【親クラス】

```
<?php
class Goods
{
    // 商品名プロパティ
    private $name = "";

    // 商品価格プロパティ
    private $price = 0;

    // コンストラクタ。商品名と商品価格を設定する
```

```

public function __construct(string $name, int $price)
{
    $this->name = $name;
    $this->price = $price;
}

// ★★★★★★注目★★★★★★
// 商品名と価格を表示するメソッド
public function printPrice(): void
{
    print($this->name."の価格: ¥".$this->price."<br>");
}

// 商品名のゲッター
public function getName(): string
{
    return $this->name;
}

// 商品価格のゲッター
public function getPrice(): int
{
    return $this->price;
}
}

```

## 【子クラス】

```

<?php
class GoodswithTax extends Goods
{
    // ★★★★★★注目★★★★★★
    // printPriceメソッドをOverride
    public function printPrice(): void
    {
        // 商品価格の税込み価格を計算し、表示
        $pricewithTax = round($this->getPrice() * 1.08); // (1)
        print($this->getName()."の税込み価格: ¥".$pricewithTax."<br>"); // (2)
    }
}

```

## • 抽象クラス

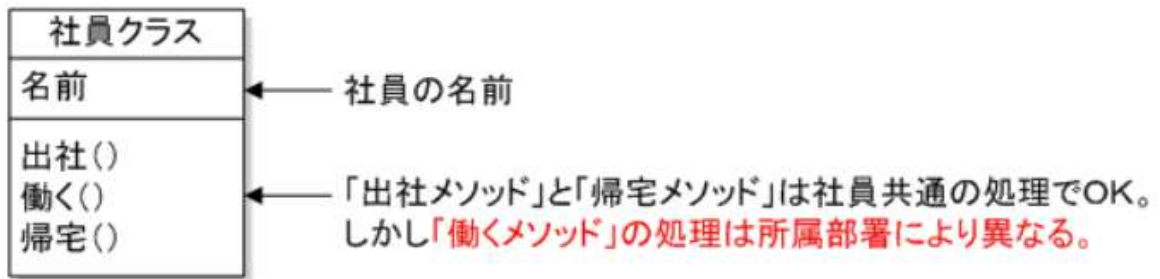
クラスのモデルを作るイメージ。抽象クラスで、メソッドの定義や処理内容を記述し、それをモデルとする。子クラスでは、処理内容をOverrideしなければならない。多重継承はできず、単一継承しかできない。

## 【具体例1】

以下の条件の社員オブジェクトを実装したいとする。

1. 午前9時に出社
2. 営業部・開発部・総務部があり、それぞれが異なる仕事を行う
3. 午後6時に退社

この時、『働くメソッド』は部署ごとに異なってしまうが、どう実装したら良いのか...



これを解決するために、例えば、次の2つが実装方法が考えられる。

1. 営業部社員オブジェクト、開発部社員オブジェクト、総務部社員オブジェクトを別々に実装

⇒メリット：同じ部署の他のオブジェクトに影響を与えられる。

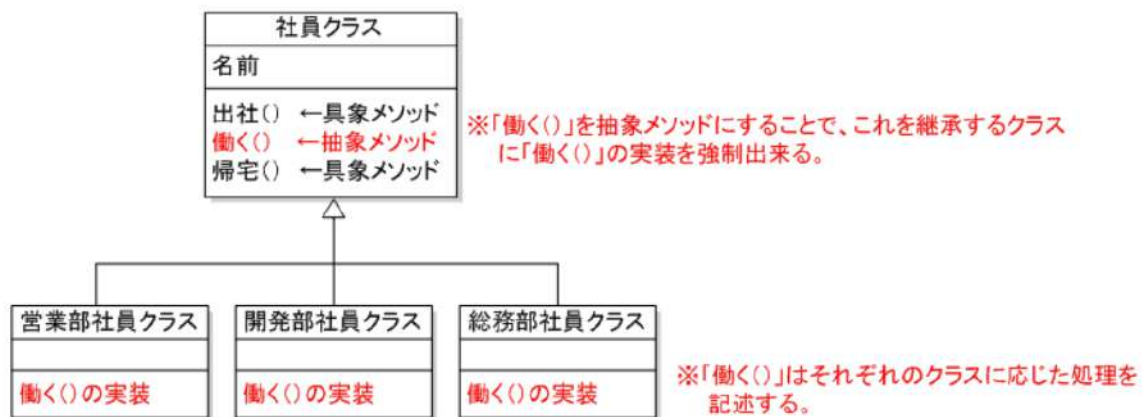
⇒デメリット：各社員オブジェクトで共通の処理を個別に実装しなければならない。共通の処理が同じコードで書かれる保証がない。

2. 一つの社員オブジェクトの中で、働くメソッドに部署ごとに変化する引数を設定

⇒メリット：全部署の社員を一つのオブジェクトで呼び出せる。

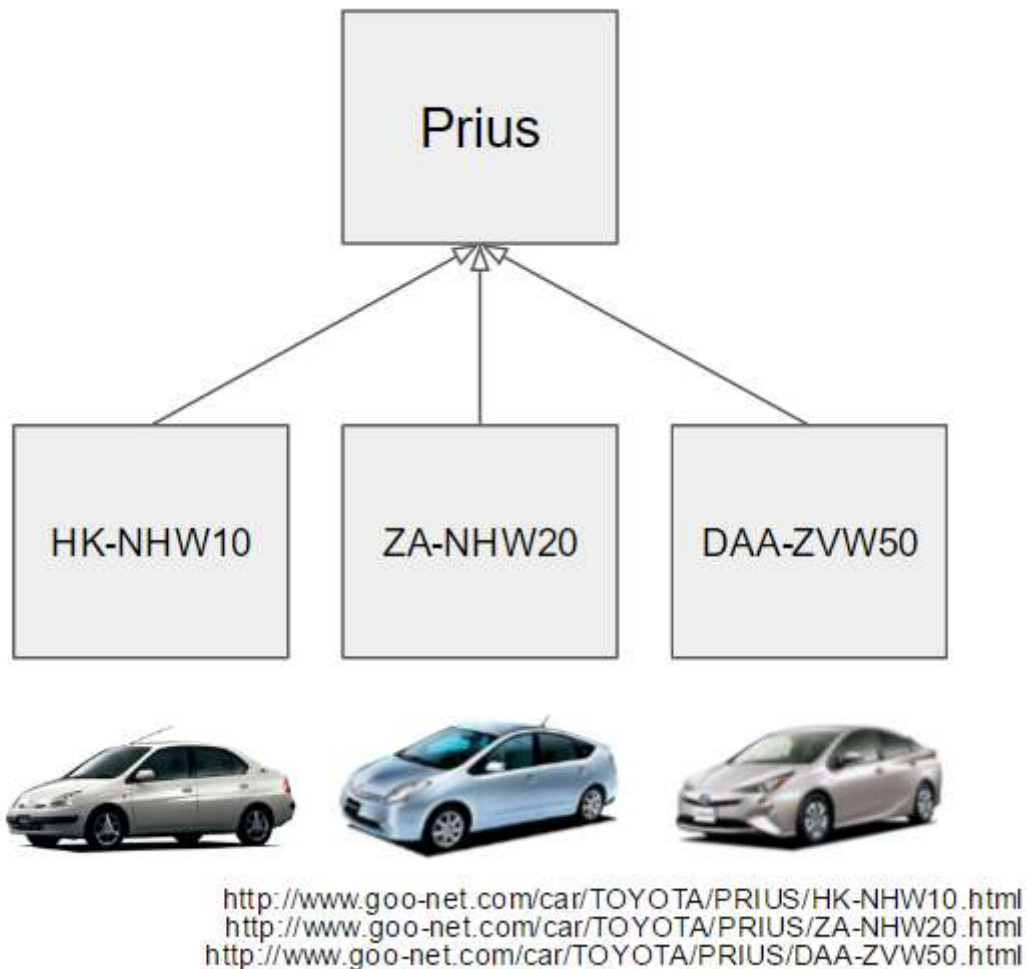
⇒デメリット：一つの修正が、全部署の社員の処理に影響を与えてしまう。

抽象オブジェクトと抽象メソッドを用いると、2つのメリットを生かしつつ、デメリットを解消可能。



## 【具体例2】

プリウスと各世代プリウスが、抽象クラスと subclasses の関係にある。



- **parent::**

オーバーライドによって上書きされる前のメソッドを呼び出せる。

```
<?php
class GoodswithTax2 extends Goods
{
    //商品名と価格を表示するメソッド。税込みで表示するように変更
    public function printPrice(): void
    {

        //親オブジェクトの同名メソッドの呼び出し
        parent::printPrice(); // (1)

        //商品価格の税込み価格を計算し、表示
        $pricewithTax = round($this->getPrice() * 1.08);
        print($this->getName()."の税込み価格: ¥".$pricewithTax."<br>");
    }
}
```

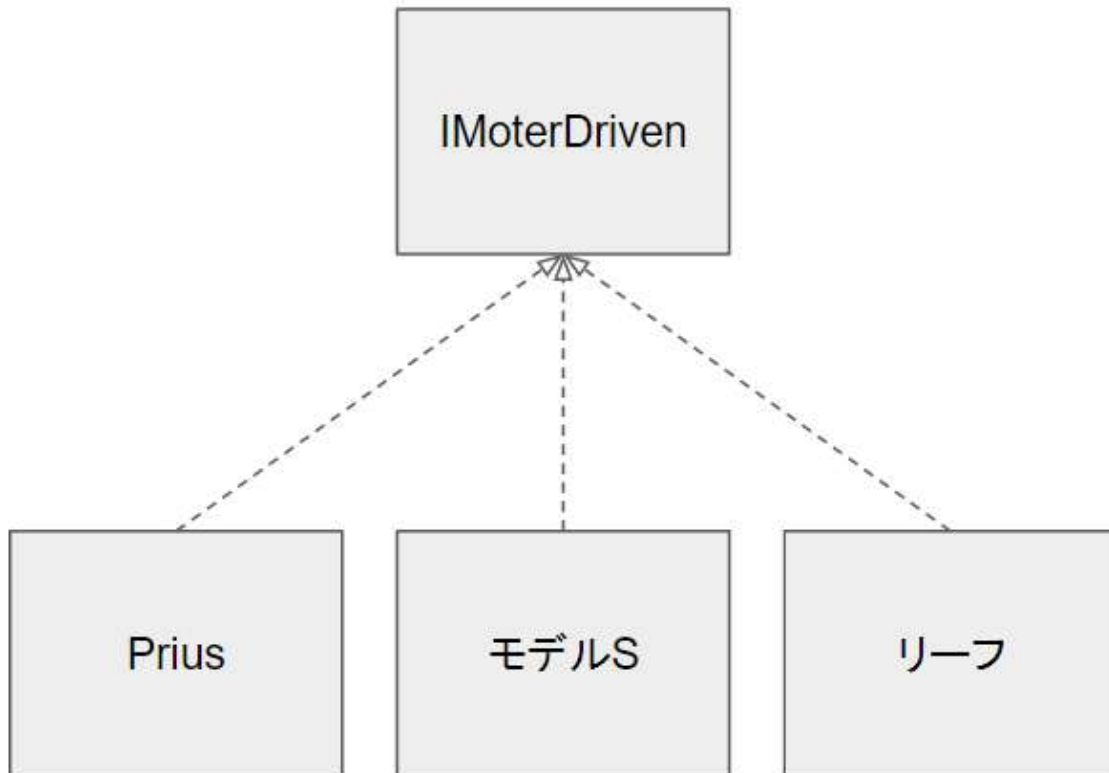
## ◇ Realization (実現)

インターフェースから実装クラスへ機能を追加するイメージ。または、実装クラスに共通して持たせたいメソッドの型を定義しておくイメージ。

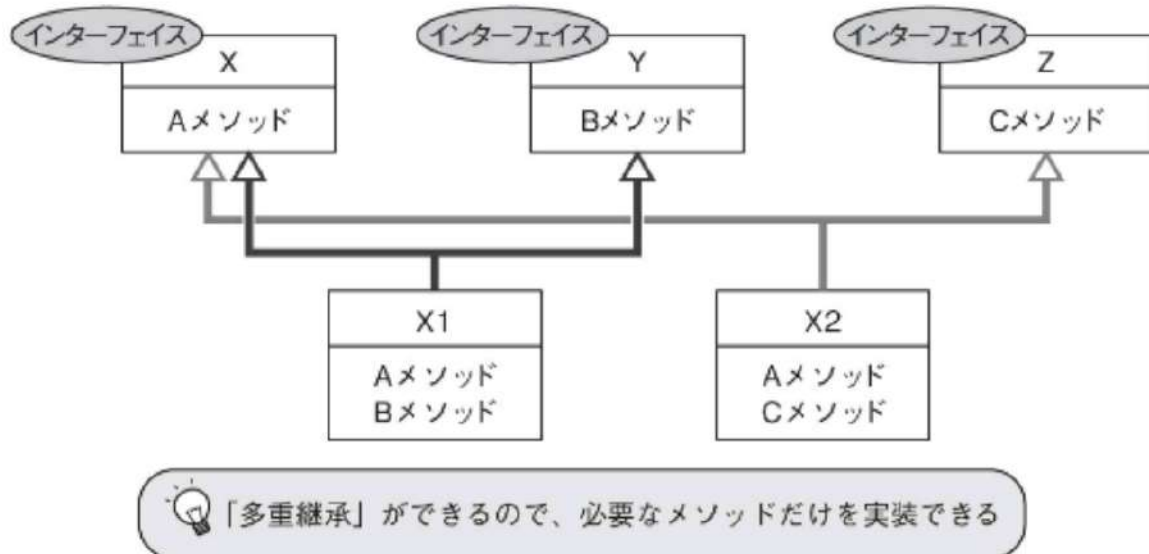
### 【具体例】



各車にはモーター機能を追加したい。



実装クラスに処理内容を記述しなければならない。すなわち、抽象クラスにメソッドの型のみ定義した場合と同じである。多重継承できる。



#### 【実装例】

```
# コミュニケーション機能を持つインターフェース
interface Communication
{
    // インターフェイスでは、実装を伴うメソッドやプロパティの宣言はできない
    public function talk();
    public function touch();
    public function gesture();
}
```

```
# コミュニケーションの機能を追加したい実装クラス
class Human implements Communication
{
    public function talk()
    {
        // 話す
    }

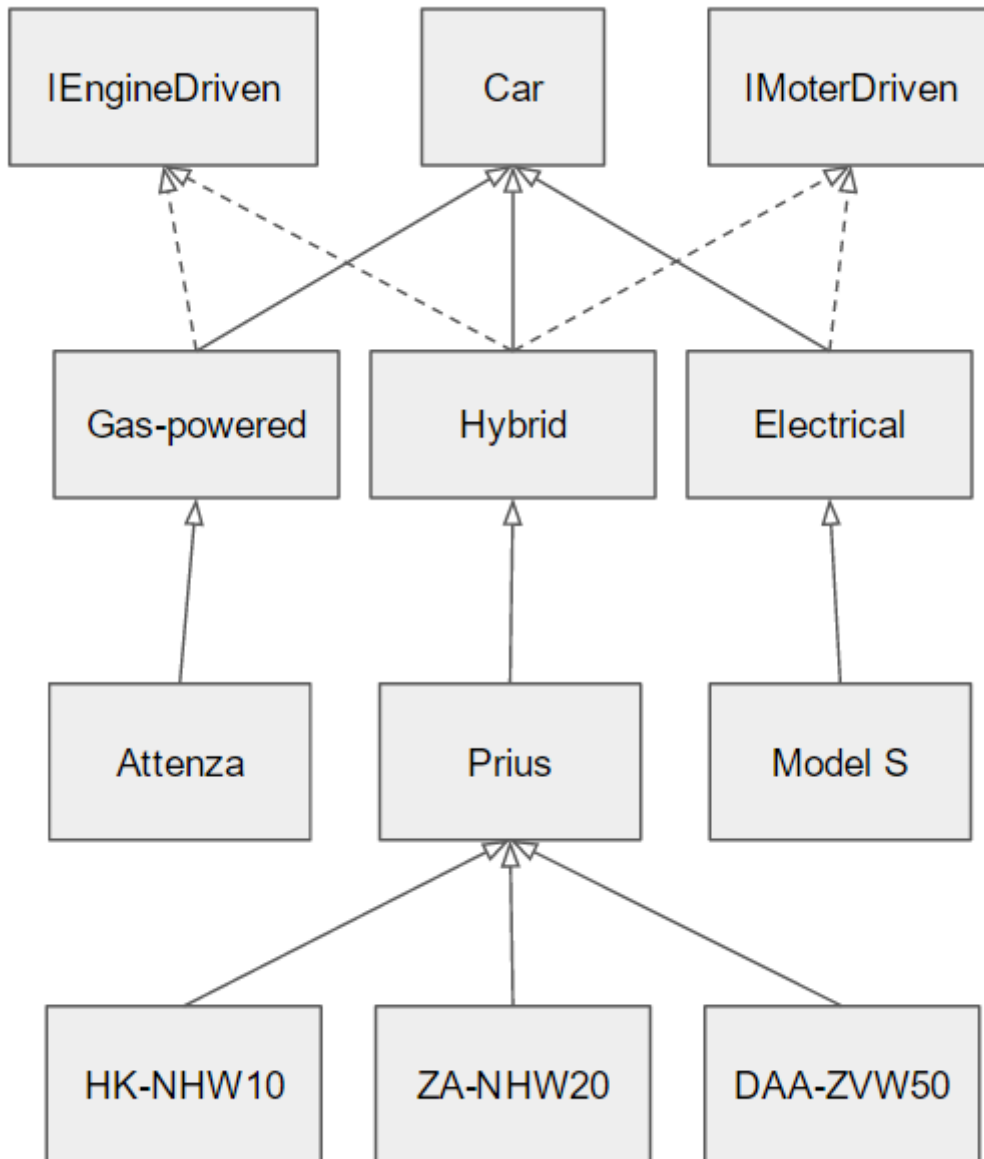
    public function touch()
    {
        // 触る
    }

    public function gesture()
    {
        // 身振り手振り
    }
}
```

## ◇ 汎化の抽象クラスと実現のインターフェースの違い

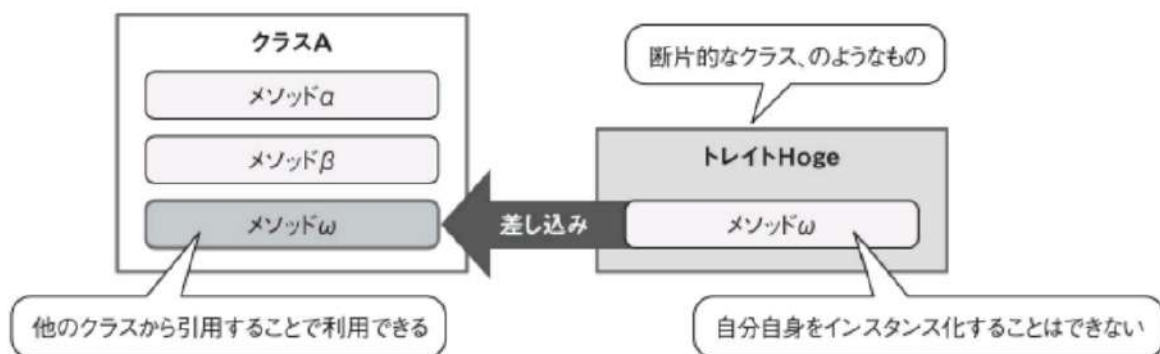
★Carクラスのモデルと、そのの上書きクラスを作りたい場合、抽象Carクラスと子クラスを作成する。

★Driven機能をもつインターフェースと、それを追加したい実装クラス関係を作りたい場合、Drivenインターフェースと実装クラスを作成する。



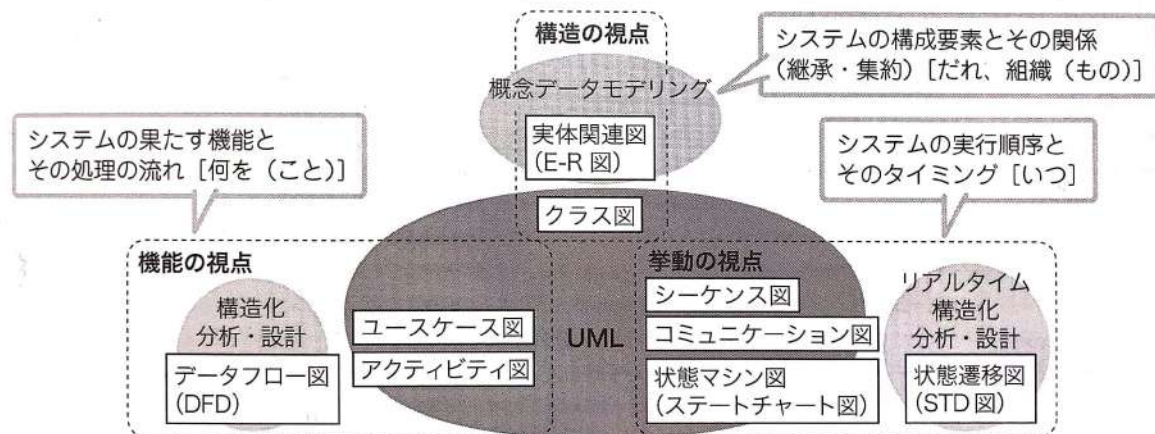
### ◇ Trait（※php独自の機能）

再利用したいメソッドやプロパティを部品化し、利用したい時にクラスに取り込む。Traitを用いるときは、クラス内でTraitをuse宣言する。Trait自体は不完全なクラスであり、インスタンス化できない。



## 02-05. 概念データモデリング

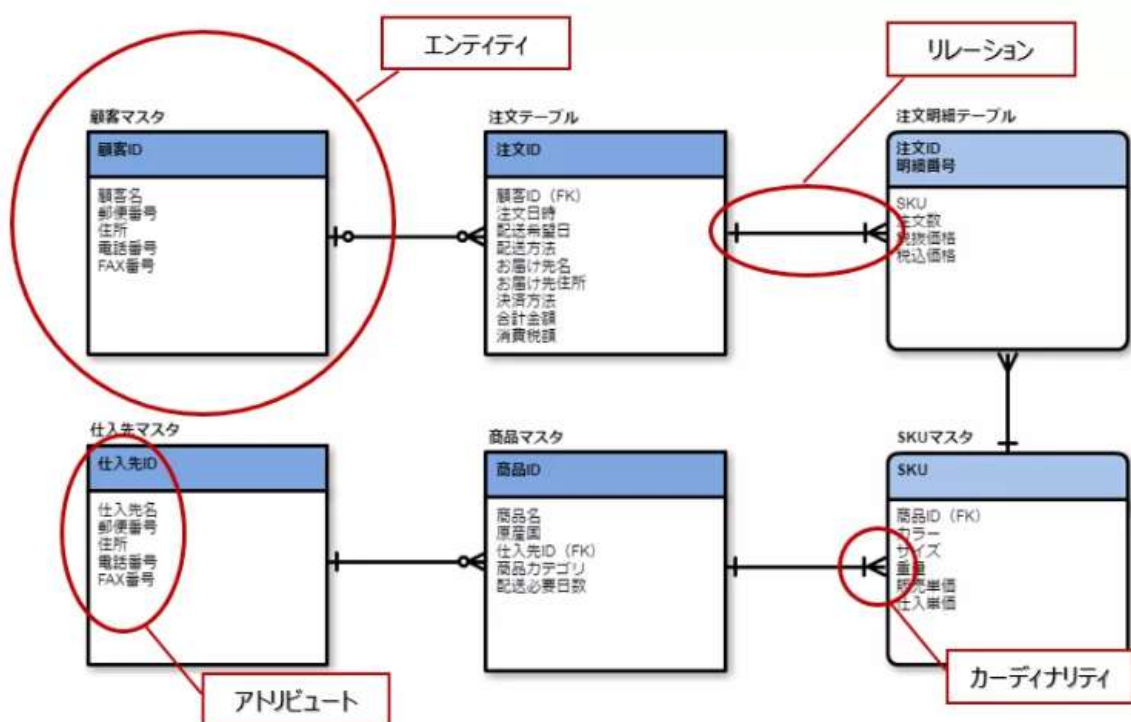
### ◇ オブジェクトモデリングの図式化方法の種類（再掲）



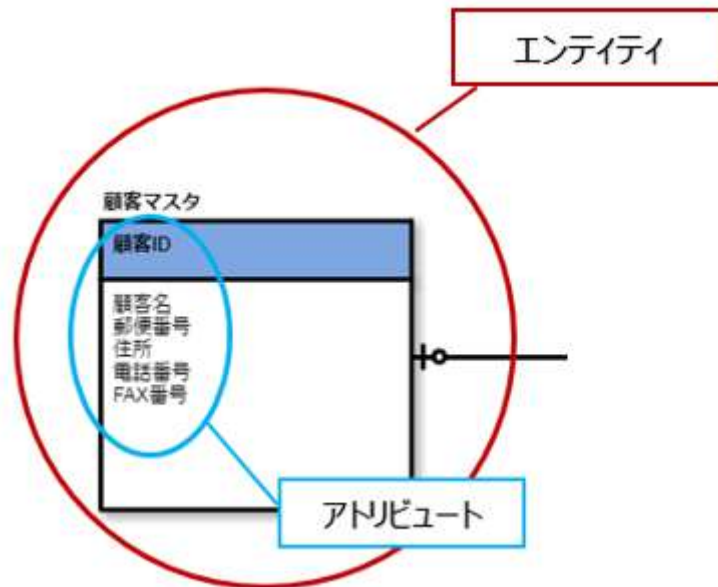
### ◇ ER図 : Entity Relation Diagram

データベースの設計において、エンティティ間の関係を表すために用いられるダイアグラム図。『IE 記法』と『IDEF1X 記法』が一般的に用いられる。

#### ER図の例 (IE記法)

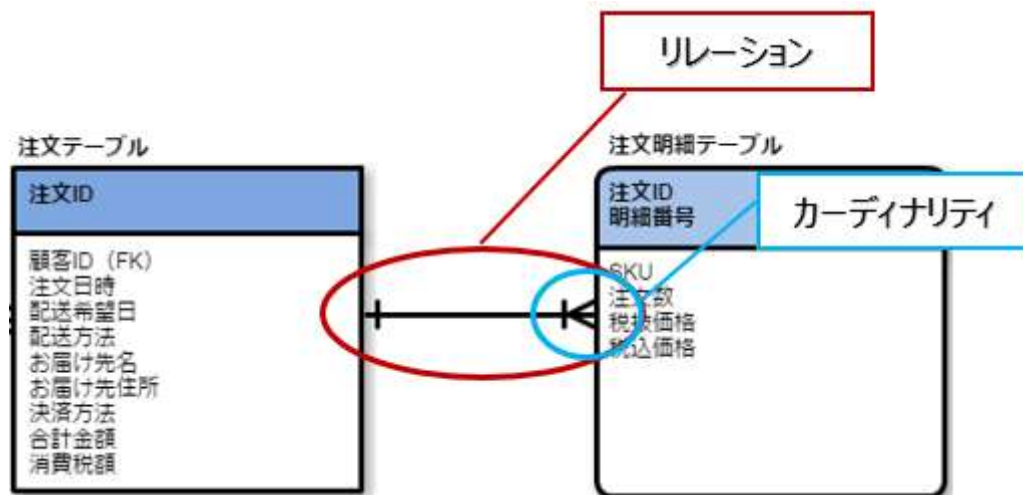


- Entity と Attribute

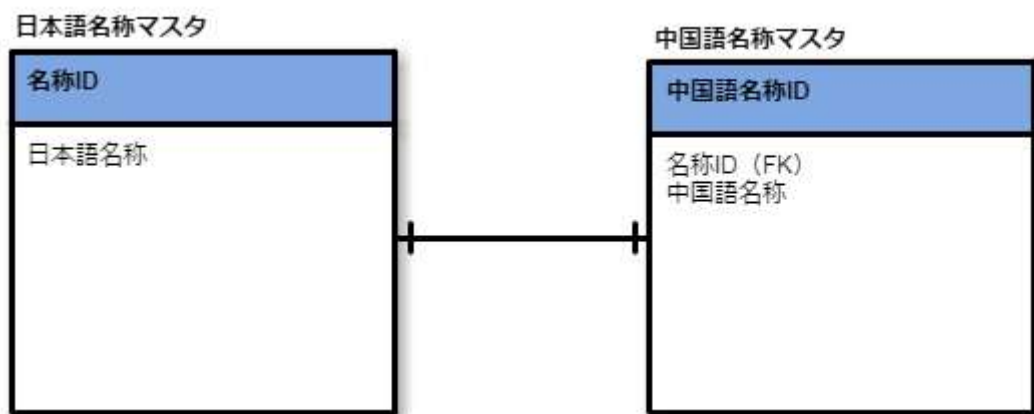


- Relation と Cardinality (多重度)

エンティティ間の関係を表す。



- 1 : 1



- 1 : 多 (Relation が曖昧な状態)

設計が進むにつれ、「1 : 0 以上の関係」「1 : 1 以上の関係」のように具体化しく。





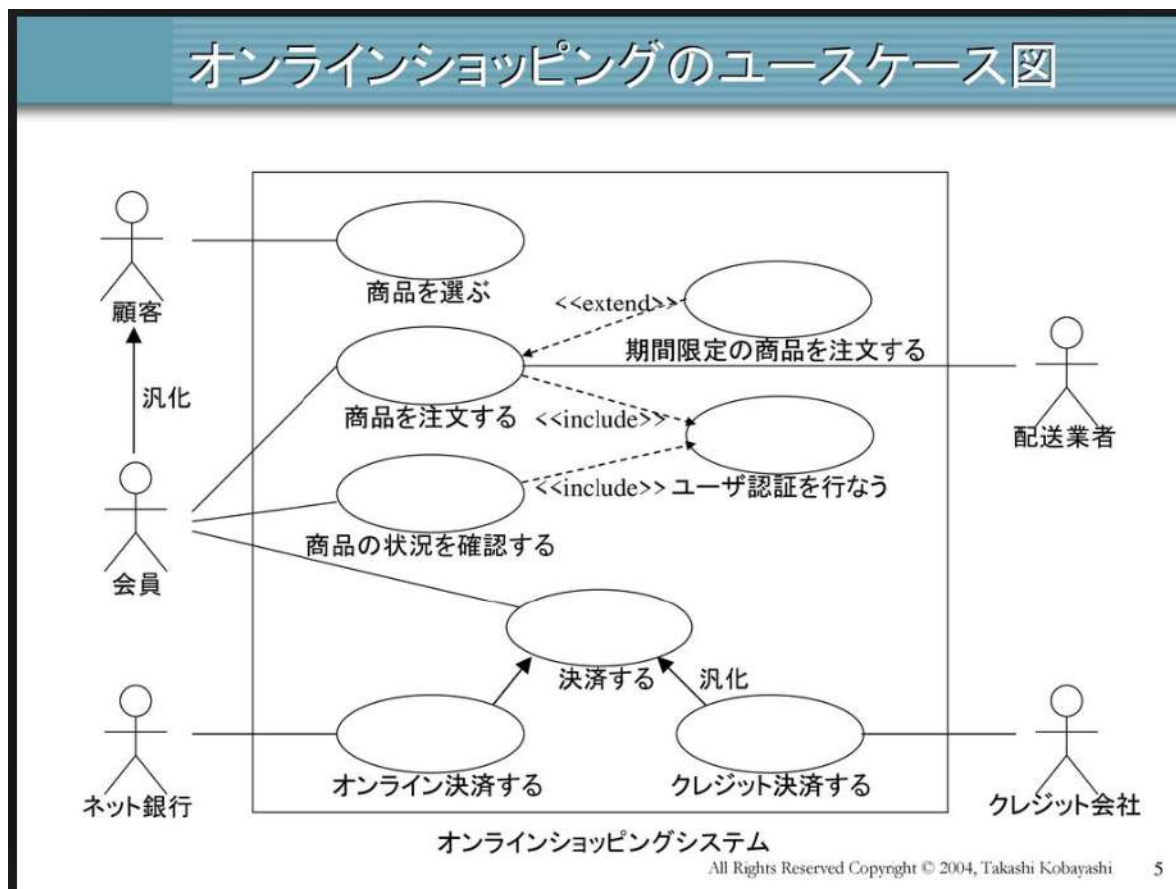
振る舞い図	ユースケース図 (use case diagram)	利用者や外部システムからの要求に対して、システムがどのような振る舞いをするかをあらわす。
	アクティビティ図 (activity diagram)	システム実行時における、一連の処理の流れや状態遷移をあらわす。フローチャートのなもの。
	状態マシン図 (state machine diagram)	イベントによって起こる、オブジェクトの状態遷移をあらわす。
	シーケンス図 (sequence diagram)	オブジェクト間のやり取りを、時系列にそってあらわす。
	コミュニケーション図 (communication diagram)	オブジェクト間の関連と、そこで行われるメッセージのやり取りをあらわす。
	相互作用概要図 (interaction overview diagram)	ユースケース図やシーケンス図などを構成要素として、より大枠の処理の流れをあらわす。アクティビティ図の変形。
	タイミング図 (timing diagram)	オブジェクトの状態遷移を時系列であらわす。

## ◇ Use case 図（使用事例図）

ユーザーの視点で、システムの利用例を表記する方法。

### 【具体例】

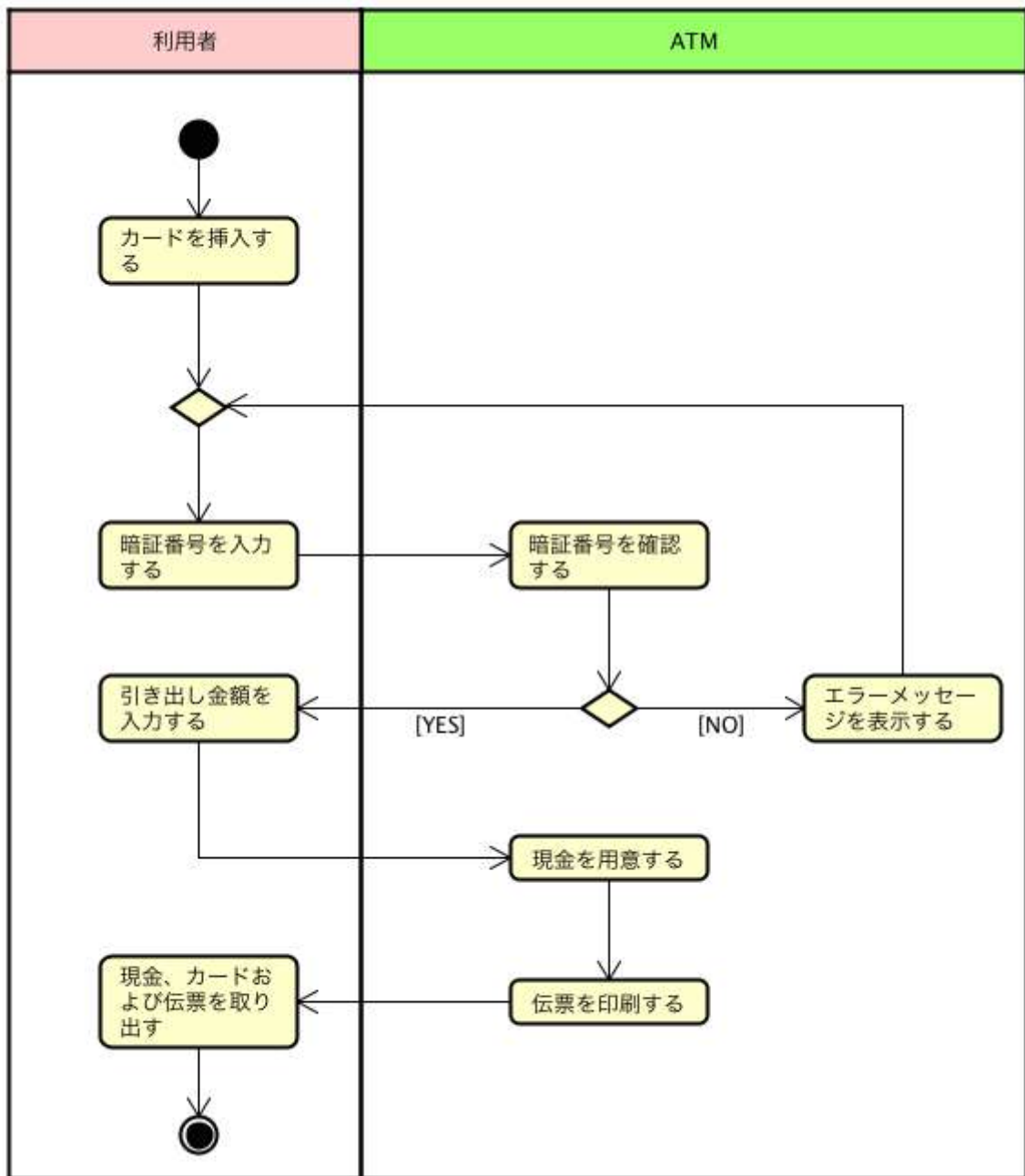
オンラインショッピングにおけるUse case



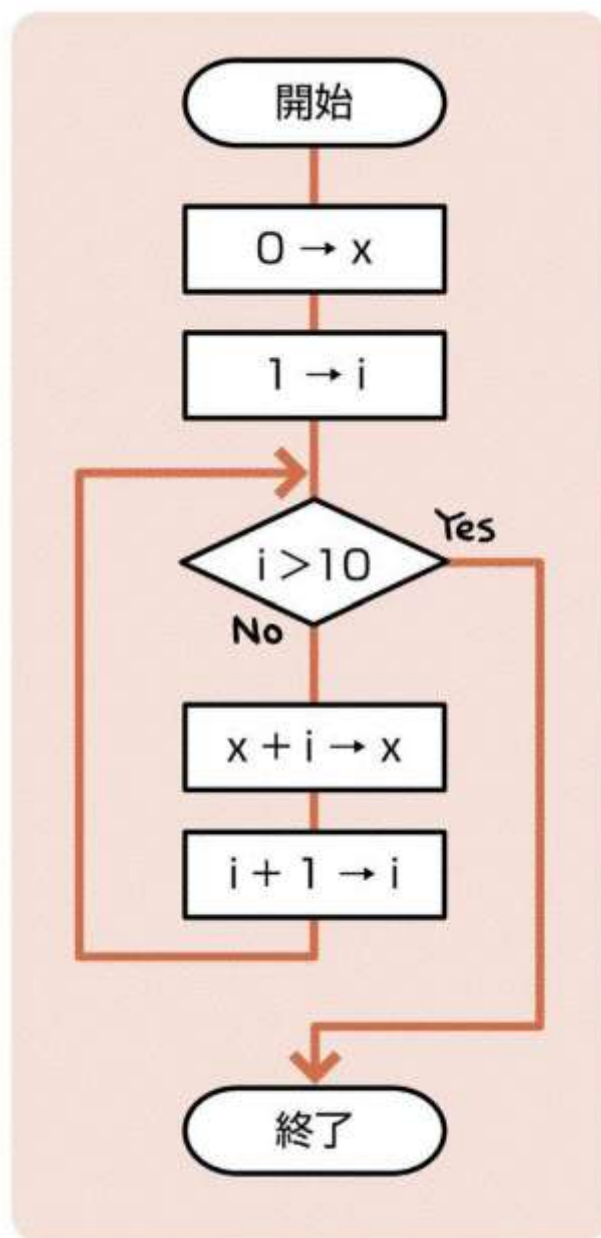
## ◇ アクティビティ図

ビジネスロジックや業務フローを手続き的に表記する方法。

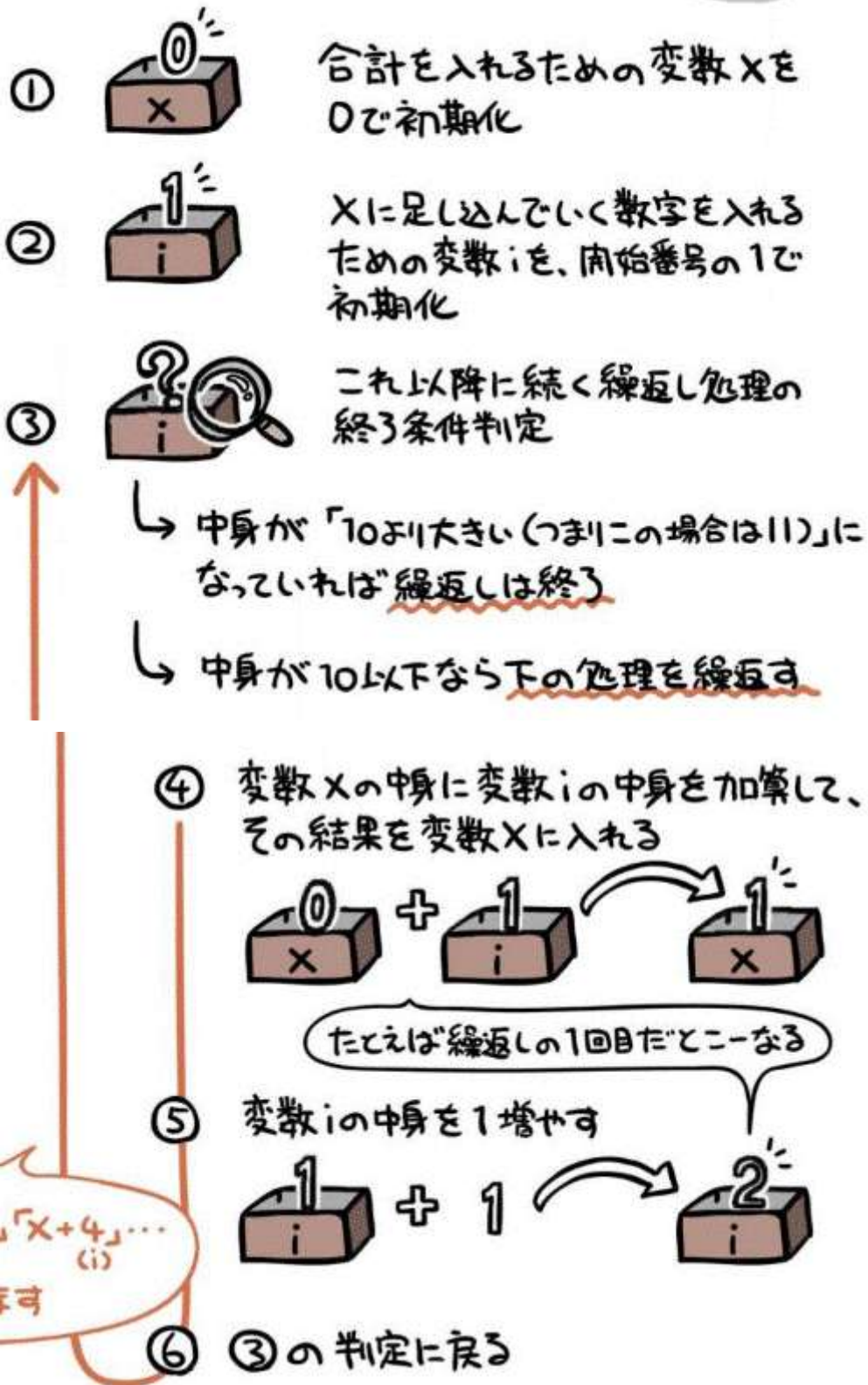
【設計例】



- アルゴリズムとフローチャート

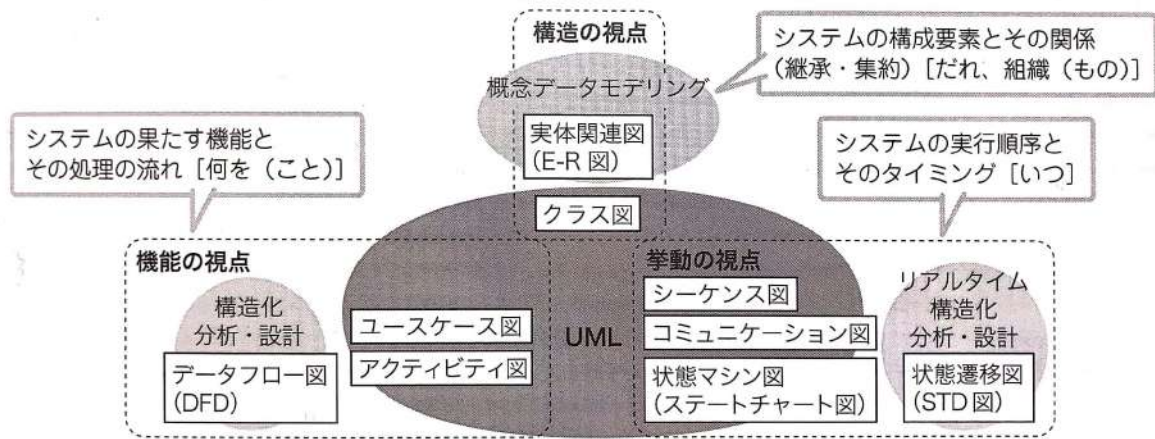




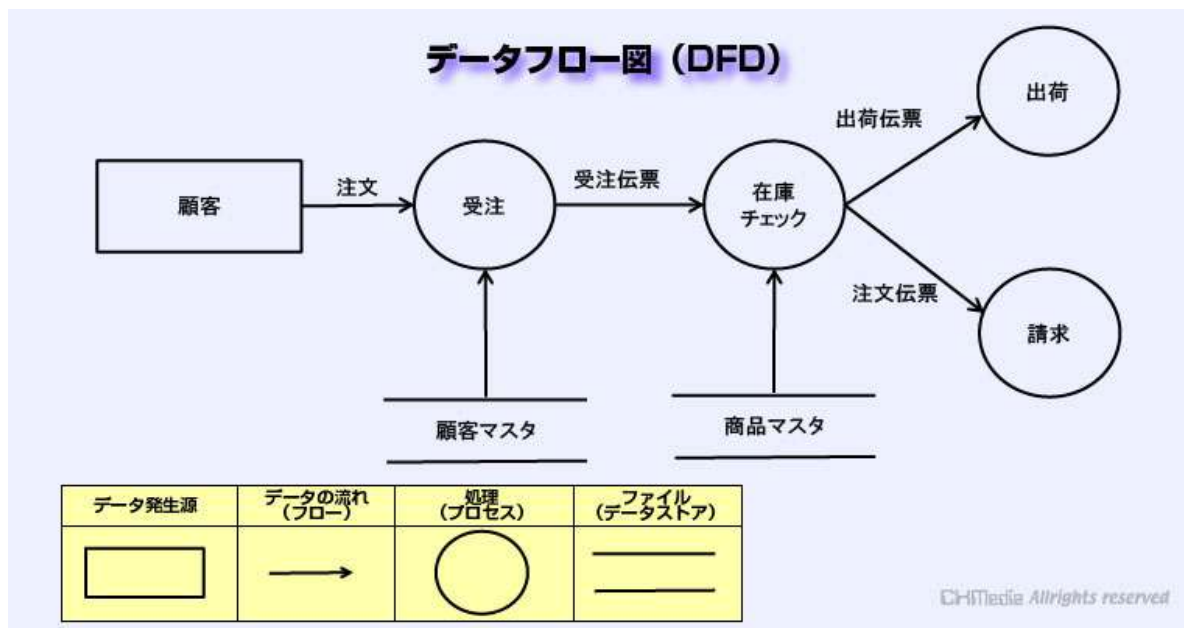


## 02-07. 構造化分析・設計

◇ オブジェクトモデリングの図式化方法の種類 (再掲)



## ◇ DFD : Data Flow Diagram (データフロー図)



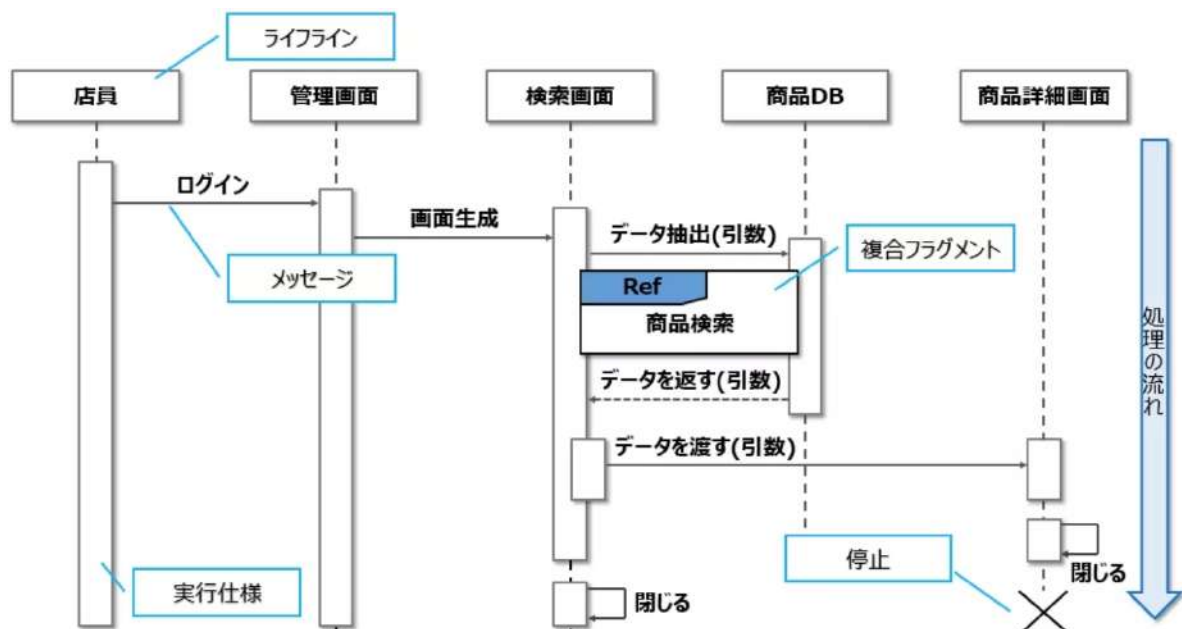
## 02-08. UMLの振舞図 : 『振舞』の視点

### ◇ シーケンス図

オブジェクトからオブジェクトへの振舞の流れを、時間軸に沿って表記する方法。Alfortの設計ではこれが用いられた。

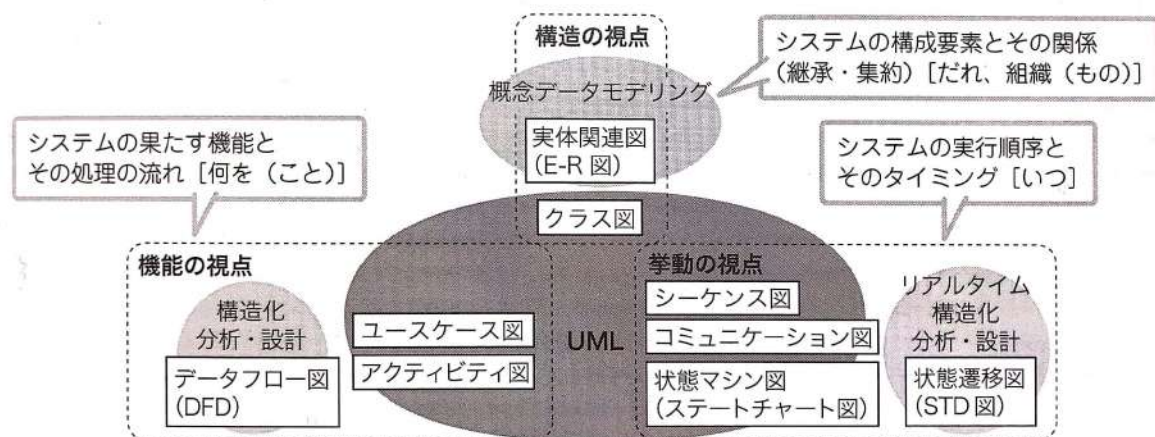
#### 【設計例1】

1. 5つのライフライン (店員オブジェクト、管理画面オブジェクト、検索画面オブジェクト、商品DBオブジェクト、商品詳細画面オブジェクト) を設定する。
2. 各ライフラインで実行される実行仕様間の命令内容を、メッセージや複合フラグメントで示す。



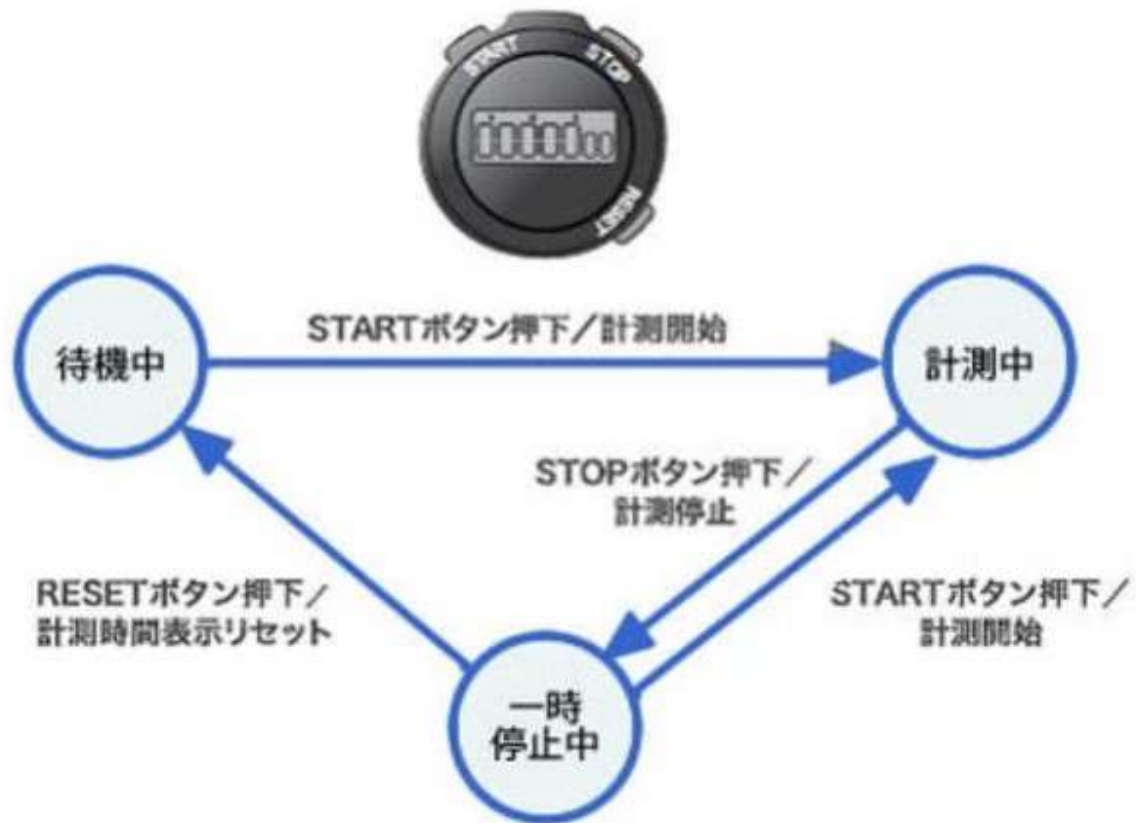
## 02-09. リアルタイム構造化・分析

### ◇ オブジェクトモデリングの図式化方法の種類（再掲）



### ◇ 状態遷移図

「状態」を丸、「遷移」を矢印で表す。矢印の横の説明は、遷移のきっかけとなる「イベント（入力）／アクション（出力）」を示す。



## ◇ 状態遷移表

状態遷移表を作成してみると、状態遷移図では、9つあるセルのうち4つのセルしか表現できておらず、残り5つのセルは表現されていないことに気づくことができる。

		イベント		
		STARTボタン押下	STOPボタン押下	RESETボタン押下
遷移前の状態	待機中	計測中	—	—
	計測中	—	一時停止中	—
	一時停止中	計測中	—	待機中

↑  
遷移後の状態

### 【例題】 12.2 という状態

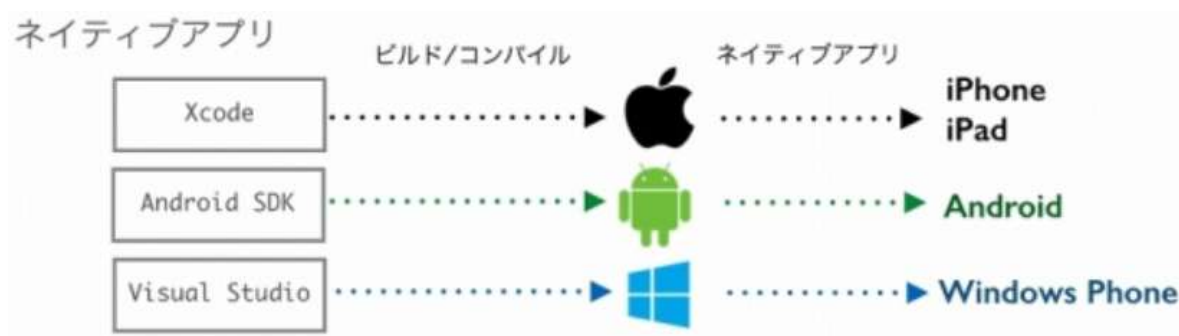
1. 初期の状態を『a』として、最初が数字なので、a行の『b』へ移動。
2. 現在の状態『b』から、次は数字なので、b行の『b』へ移動。
3. 現在の状態『b』から、次は小数点なので、b行の『d』へ移動
4. 現在の状態『d』から、次は数字なので、b行の『e』へ移動



		文字				
		空白	数字	符号	小数点	その他
現在の状態	a	a	b	c	d	e
	b	a	b	e	d	e
	c	e	b	e	d	e
	d	a	e	e	e	e

## 02-10. アプリケーションの種類

### ◇ ネイティブアプリケーション



端末上のプログラムによって稼働するアプリのこと。一度ダウンロードしてしまえば、インターネットに繋がっていなくとも、使用できる。

【アプリ例】

Office

### ◇ Webアプリケーションとクラウドアプリケーション



- Webアプリケーション

Webサーバ上のプログラムによって稼働するアプリのこと。

【Webアプリ例】

Googleアプリ、Amazon

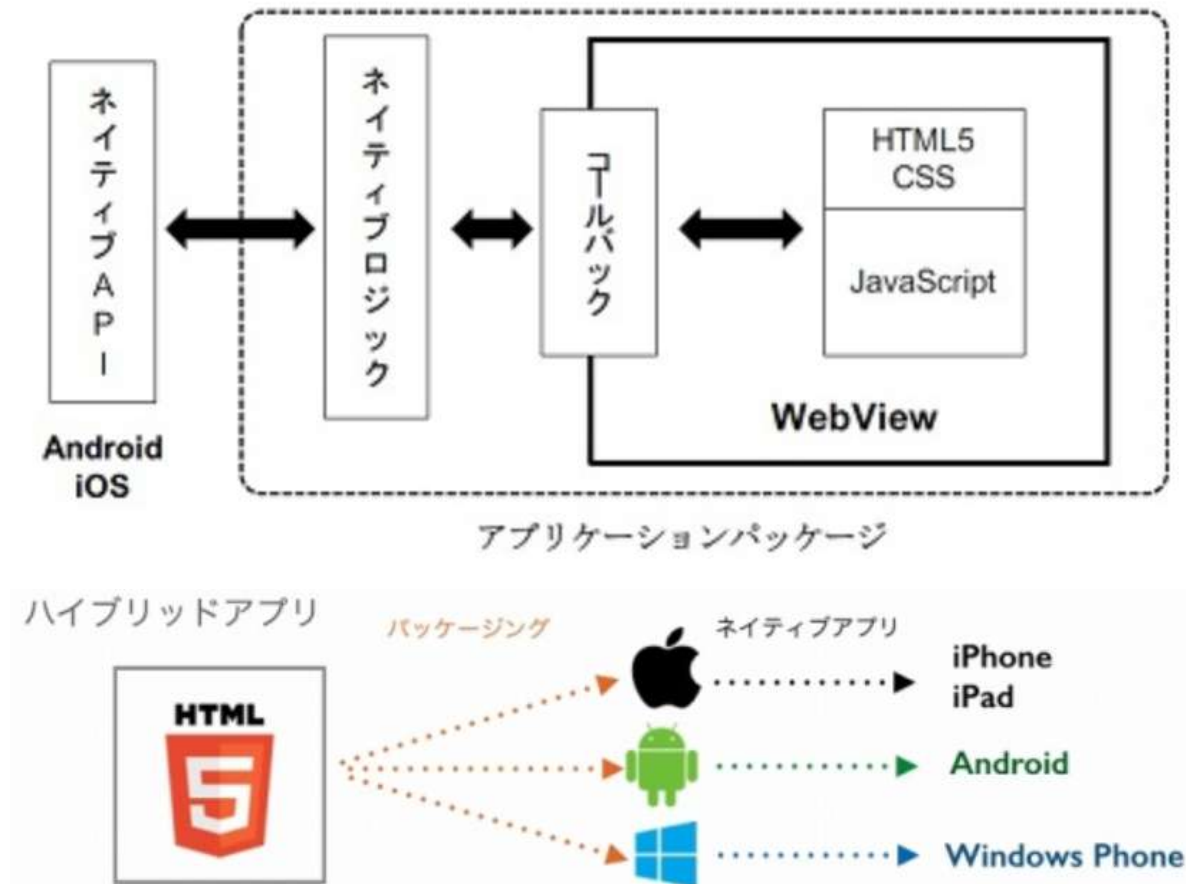
- クラウドアプリケーション

Webサーバ上のプログラムによって稼働するアプリのうち、クラウドサービスを提供するものと。

【クラウドアプリ例】

Google Drive、Dropbox

## ◇ ハイブリッドアプリケーション



Webview上でWebアプリが実行されることによって稼働するアプリのこと。

【アプリ例】

クックパッド