# 内容

はじめに ........ 1
1. 方式設計と業務処理 ........ 1
2. 分析結果からの展開 ........ 2
2.1. 実行単位 ........ 2
2.2. 業務処理の実装部分 ........ 3
3. 方式設計 ........ 4
3.1. 方式設計の要件 ........ 4
3.2. 方式の選定条件 ........ 4
3.3. 業務処理との結合 ........ 4
4. 業務処理設計 ........ 5
4.1. クラス設計の粒度 ........ 5
4.2. 仕様変更の影響箇所 ........ 5
（1）システムの変更要因 ........ 5
（2）影響がある箇所 ........ 5
4.3. 処理を局所化するための設計 ........ 6
（1）継承による呼び出しメソッドの切り替え ........ 6
（2）Factory＋delegate(委任)パターン風 ........ 7
4.4. マスタ・データ編集の局所化 ........ 9
（1）interface の実装 ........ 9
（2）visitor パターン風 ........ 10
（3）enum を使った振り分け ........ 12

Copyright(C)2021 Future Office Coordinate Service Corporation All Rights Reserved.

# オブジェクト指向―設計

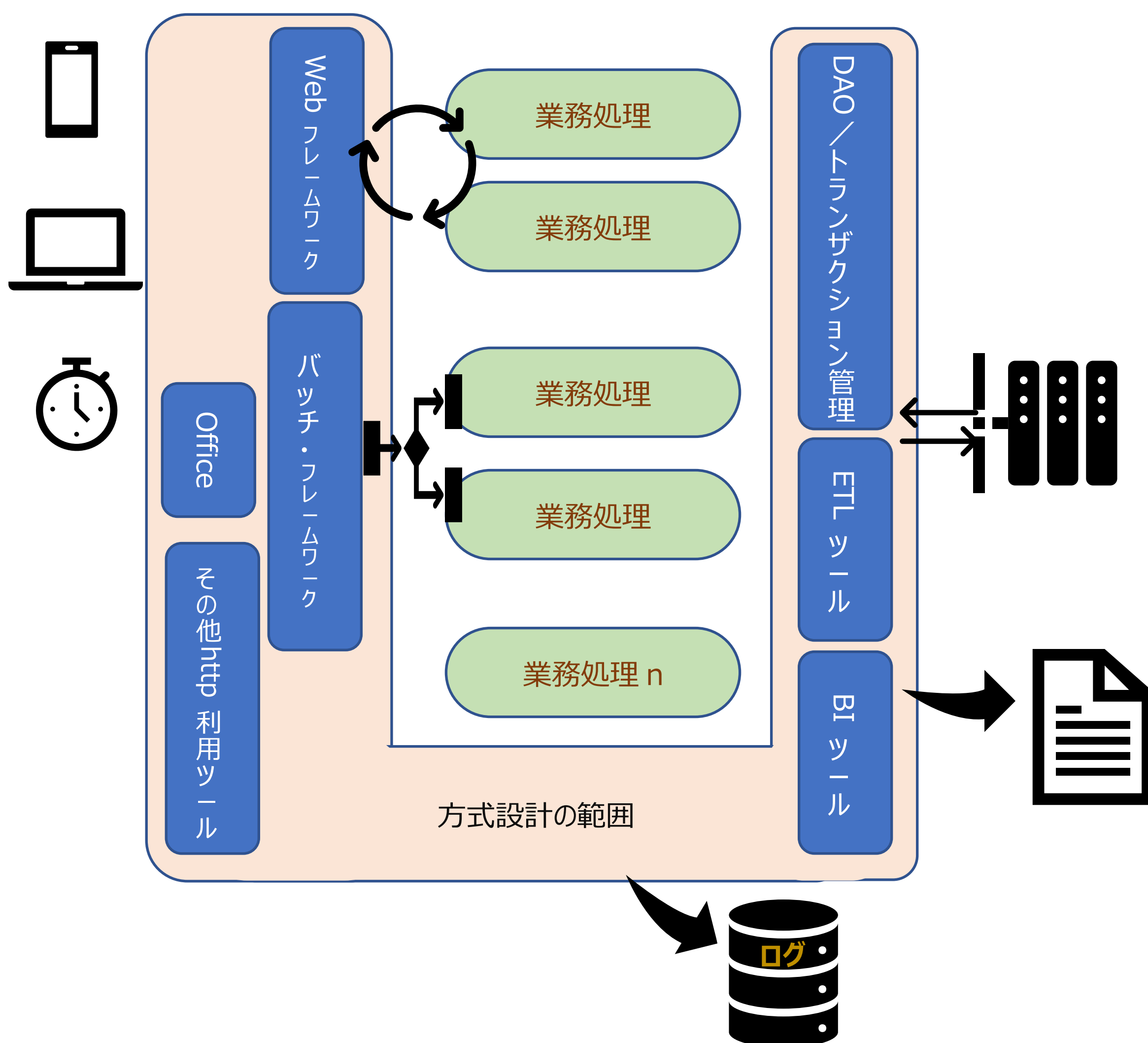
はじめに

設計工程では分析工程で決めたシステム化範囲の実現方法を決めます。

企業内システムの場合、トランザクションデータの入力／報告／保管／更新／削除のデータフローをどのような方式で実現するかという「方式設計」と業務ルールの実装を行う「業務処理設計」が必要になります。

## 1.　方式設計と業務処理

下図の“業務処理”は Java 等の言語による実装を表し、AP サーバ等のコンテナ[1] ＞ フレームワークから呼び出されて処理を行います。実装～試験で手間が掛かる“業務処理”の部分を小さくした方が品質と保守性を上げられる可能性が高くなります。




[1] JVM からみてコンテナは main メソッドを持ったアプリケーションです。フレームワークはコンテナにロードされて業務処理の呼び出し／監視／制御／後始末を行います。ETL、BI 等のツールは一般的にはコンテナ／フレームワークと関係なく単独で動作します。

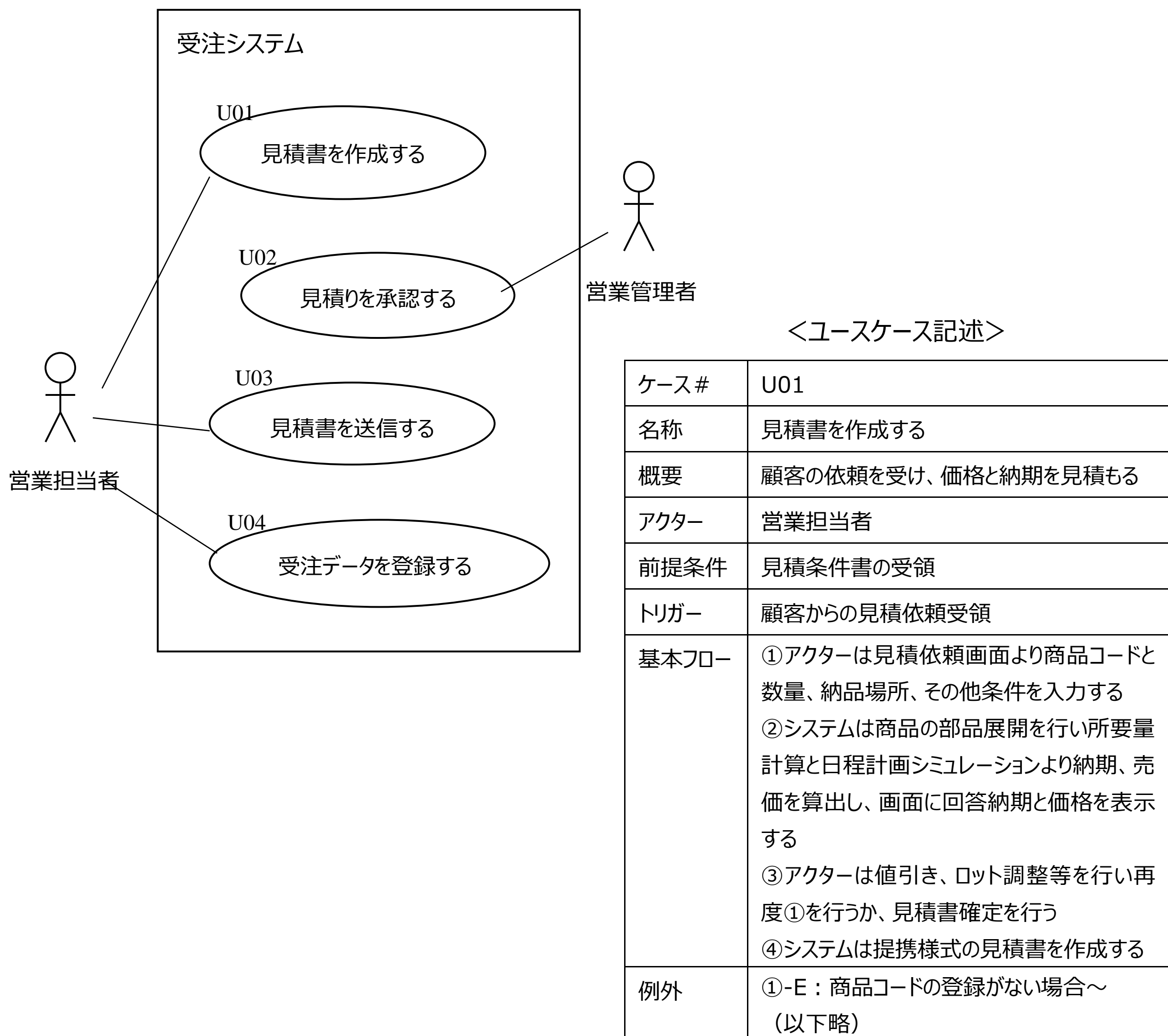
Copyright(C)2021 Future Office Coordinate Service Corporation All Rights Reserved. 

## 2.　分析結果からの展開

フレームワークや特定の設計手法を使う場合は、適用範囲・インターフェースを決め（方式設計）て業務処理を組み合わせます。例えば、以下の受注システムの設計を行う場合では…

### 2.1.　実行単位

ユースケースの粒度がユーザの認識する仕事の単位になっていると仮定し、この単位で実装内容を考えます。「見積書を作成する」では、納期／価格のシミュレーションと見積書作成に関連する一連の処理を設計します。



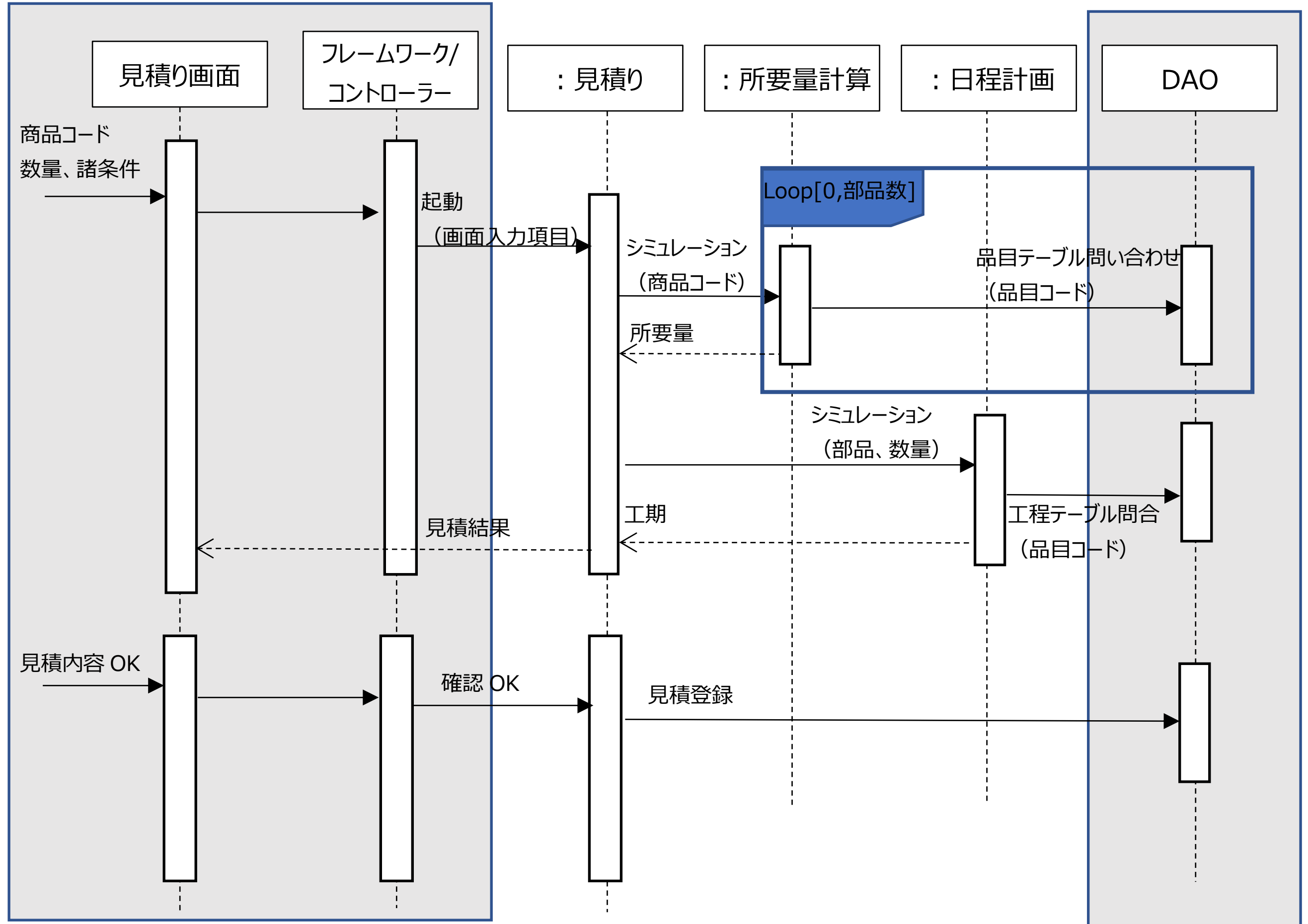

<ユースケース記述>

| ケース＃ | U01 |
|---|---|
| 名称 | 見積書を作成する |
| 概要 | 顧客の依頼を受け、価格と納期を見積もる |
| アクター | 営業担当者 |
| 前提条件 | 見積条件書の受領 |
| トリガー | 顧客からの見積依頼受領 |
| 基本フロー | ①アクターは見積依頼画面より商品コードと数量、納品場所、その他条件を入力する<br>②システムは商品の部品展開を行い所要量計算と日程計画シミュレーションより納期、売価を算出し、画面に回答納期と価格を表示する<br>③アクターは値引き、ロット調整等を行い再度①を行うか、見積書確定を行う<br>④システムは提携様式の見積書を作成する |
| 例外 | ①-E：商品コードの登録がない場合～<br>（以下略） |

Copyright(C)2021 Future Office Coordinate Service Corporation All Rights Reserved.

## 2.2. 業務処理の実装部分

フレームワークの利用が決まっている場合、大枠は方式設計で決まります[2]。
業務処理毎に設計を行うのは下図の**「方式設計に依存する部分」以外**のものになります。




方式設計に依存する部分

※このシーケンス図は業務処理設計を行う（あるいは行わない）場所を区分するためのもので、一部機能を省略しています

2 フレームワークは大枠を決めることでツール化や定型化を可能にしています

Copyright(C)2021 Future Office Coordinate Service Corporation All Rights Reserved. 

## 3. 方式設計

実装はコードを書く他に実行単位の環境定義作成も含むため、何が必要かは方式設計で決めておく必要があります。

### 3.1. 方式設計の要件

データの永続化（保存、再利用）が必要であれば RDB への接続方法（API）や接続先の記述方法、フレームワークを使うのであればその定義 - 例えば Spring であれば DI 関連の定義等 - を行うための設計が必要です。他にはログ出力 - ユーザによっては暗号化やログサーバでの集中管理を求めるかもしれません - や、エラー処理の組込み方法等が必要になります。

[決定が必要な要件の例]

① ログ出力、エラー処理の手順、ユーティリティクラスや基底クラスの API
② RDB への接続方法（API）
③ 割り込みや再実行、並行処理に関しての実装内容

### 3.2. 方式の選定条件

方式を選ぶ前提はシステムの寿命に準じる程度の期間使い続けることができるものですが、Web 開発で過去に定番と言われたフレームワークやリッチクライアント等もセキュリティの問題で使えなくなったり（例：Struts, Flash, curl 他）、バージョンアップで一部のクラスが非推奨になったり API が変わってしまう（例：Java, jQuery 他）ものがあるため、以下の点を考慮して決定します。

① 業界等で標準化されたツールを使う（SQL, HTML5, JavaScript[ECMAScript 6 以降]）
② 交換可能な単位で採用する（Web フレームワーク-MVC の Contol/View 等）
③ オープンソースの場合は情報の更新が活発なものを選ぶ（半年以上更新が無いものは避ける）

### 3.3. 業務処理との結合

フレームワークを使う場合はフレームワークから呼び出せる「型」で業務処理を実装し、環境定義ファイル等を使って関係づけます。

永続化データに関しては、対象の種類（RDB、一般ファイル、NoSql 他）から業務処理を切り離す[3]ために DAO(Data Access Object)/DTO(Data Transfer Object)を使い、更に DAO は具象クラスを意識せずに置換できるように Factory パターン[4]が多く使われます。DAO、DTO、Factory をどこまで使うか、どうやって作るかは設計工程で決めておく必要があります。

---

[3] 将来 変更になっても影響を最小限にするために、対象を“なにがしかの入れ物に入ったデータ”として抽象化しておき、入れ物自体を意識せずに済むようにしておく

[4] 1テーブルをアクセスするのにいくつものクラスを使うので冗長に見えますが、テーブルなりビューにより扱う項目／属性が決まるので簡単にクラス生成のツール化ができます

Copyright(C)2021 Future Office Coordinate Service Corporation All Rights Reserved. 

## 4. 業務処理設計

設計に関しては開発時点の最適が永続的な最適とは限りません。短サイクル（四半期毎等）でシステム更改を行う企業では、仕様変更に柔軟に応じられるか否かが死活問題になります。

### 4.1. クラス設計の粒度

クラスは仕様変更が入る前提[5]で考え、以下の単位で検討します。

① 処理／機能の内容（例えば、「見積り」、「所要量計算」、「日程計画」、「決済条件設定」）

② データの分類（対象製品が内製、外注、oem 等）

①×②がクラス分割の候補になります。次項以降の処理の局所化やデータ編集の局所化を考慮し、プロジェクト要員のスキルと習熟度で現実的なレベルに決めます。

実装の粒度は、業務的に意味のある（共通認識可能な名前が付けられる）ものが戻り値として得られる単位です。「業務的な意味」には【見積：{所要量：{部品構成、在庫、仕掛}、工期：{製造ライン、ロットサイズ、標準日程}}】のようにピラミッド構造があり、全ての業務階層が切り出し候補になります。各クラスは単機能で実装すれば使い勝手がよく再利用が進みます。

必要であればリファクタリングで実装後に見直しの機会を設けてください。

### 4.2. 仕様変更の影響箇所

システムには、取引で発生・変化するデータ（トランザクション）と永続的に維持・保存するデータ（マスタ）があります。トランザクションデータは入力／チェック／更新／集約／削除等の処理を行い、マスタは主に参照先として使用します。そして、システムの改変は以下の影響があります。

（1）システムの変更要因

システムに変更を加える要因には色々ありますが、例として以下のものがあります。

① 取引条件の変更、② 新取引先、③ 新商品・サービスやキャンペーンの新設

④ 法律／規制の変更、⑤ システム新機能、⑥ システム・アーキテクチャの変更

（2）影響がある箇所

システムの変更箇所はマスタのデータ、トランザクションのデータ及び処理が主になります。

①②…例：締・支払サイトの設定方法、連絡先、納品場所、割引率／掛目等のマスタの項目追加

③…例：クーポン、ポイント等の属性の追加及び取扱い処理の追加

④…例：規制に関わるチェックや処理の追加及び税率等のマスタの項目変更

⑤…独立した機能（属性／処理）の作成…既存機能はインターフェース追加

---

[5] 特に新規開発のシステムでは仕様変更や方式の見直しが設計中にも実装時にも発生し、全面的な作り替えがしばしば発生します。

Copyright(C)2021 Future Office Coordinate Service Corporation All Rights Reserved.

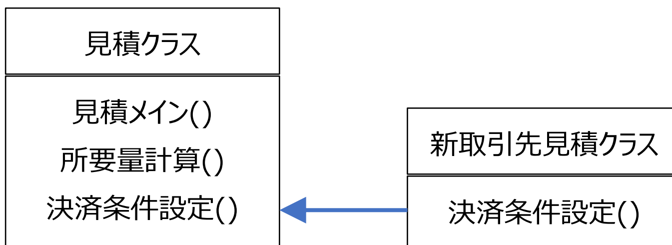
## 4.3. 処理を局所化するための設計

取引先との決済条件に新しい期日設定の処理が必要になった等、処理のバリエーションを組込むには以下の方法があります。

（1）継承による呼び出しメソッドの切り替え

新取引先用のメソッド（下図-新取引見積クラス::決済条件設定）でオーバーライドし、決済条件の設定処理を置換することができます。

★どの見積クラスをインスタンス化するか、バリエーションが増えると扱いが難しくなります




```
jshell> class 見積クラス {
   ...>     public void 見積メイン(){
   ...>         System.out.println("見積クラス::見積メイン");
   ...>         所要量計算();
   ...>         決済条件設定();
   ...>     }
   ...>     public void 所要量計算() {
   ...>         System.out.println("見積クラス::所要量計算");
   ...>     }
   ...>     public void 決済条件設定() {
   ...>         System.out.println("見積クラス::決済条件設定");
   ...>     }
   ...> }
|  次を作成しました: クラス 見積クラス

jshell> class 新取引先見積クラス extends 見積クラス{
   ...>     public void 決済条件設定() {
   ...>         System.out.println("新取引先見積クラス::決済条件設定");
   ...>     }
   ...> }
|  次を作成しました: クラス 新取引先見積クラス

jshell> (new 見積クラス()).見積メイン();
見積クラス::見積メイン
見積クラス::所要量計算
見積クラス::決済条件設定

jshell> (new 新取引先見積クラス()).見積メイン();
見積クラス::見積メイン
見積クラス::所要量計算
新取引先見積クラス::決済条件設定
```

サンプルコードは分かり易いように日本語を使いましたが、プロジェクトでの実装にはコメント以外で日本語を使わないでください…各プロジェクト内規約を確認

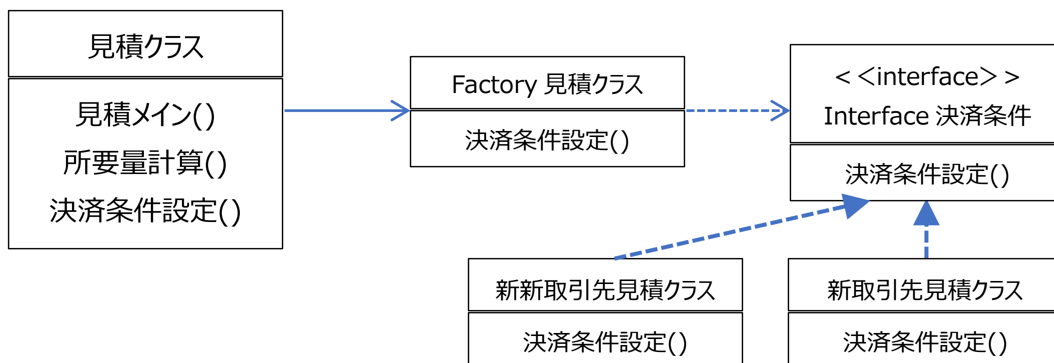
Copyright(C)2021 Future Office Coordinate Service Corporation All Rights Reserved.

（2）Factory＋delegate(委任)パターン風[6]

同一の interface を実装したクラスを作り、この interface *型* として扱うと継承に比べクラス間の結合度が低く、バリエーションの増加に対応しやすくなります。

これは以下のように動作します。

① 見積クラス::決済条件設定 の中で Factory 見積クラスのインスタンスを生成
② Factory 見積が、Interface 決済条件を実装したクラスの中から条件にあうものをインスタンス化
③ インスタンス化した Interface 決済条件が null でなかったら、その決済条件設定を実行
⇒Factory 見積からの戻りが null だった場合は、デフォルトの処理を行う




※上図に“決済条件設定()”が 5 カ所に出てきますが、シグネチャ（名前、引数）は Interface 決済条件とそれを実装している 2 つのクラスの間だけが一致していれば問題ありません

[6] この例はデザインパターンの Factory パターン、delegate パターンで行う抽象化(interface 実装)を簡略化してあります。本物はその手の教材等で確認してください

Copyright(C)2021 Future Office Coordinate Service Corporation All Rights Reserved.

```
jshell> interface Interface 決済条件 {
   ...>       public void 決済条件設定();
   ...> }
|  次を作成しました: インタフェース Interface 決済条件
jshell> class 新取引先見積クラス implements Interface 決済条件{
   ...>       public void 決済条件設定() {
   ...>             System.out.println("新取引先見積クラス::決済条件設定");
   ...>       }
   ...> }
|  次を作成しました: クラス 新取引先見積クラス
jshell> class Factory 見積{
   ...>       public Interface 決済条件 決済条件設定(String 取引先) {
   ...>             switch (取引先) {
   ...>                   case "新":
   ...>                         return new 新取引先見積クラス();
   ...>                   default:
   ...>                         return null;
   ...>             }
   ...>       }
   ...> }
|  次を作成しました: クラス Factory 見積
jshell> class 見積クラス {
   ...>       public void 見積メイン(String 取引先){
   ...>             System.out.println("見積クラス::見積メイン");
   ...>             所要量計算();
   ...>             決済条件設定(取引先);
   ...>       }
   ...>       public void 所要量計算() {
   ...>             System.out.println("見積クラス::所要量計算");
   ...>       }
   ...>       public void 決済条件設定(String 取引先) {
   ...>             Interface 決済条件 var = (new Factory 見積()).決済条件設定(取引先);
   ...>             if (var == null) {
   ...>                   System.out.println("見積クラス::決済条件設定");
   ...>             } else {
   ...>                   var.決済条件設定();
   ...>             }
   ...>       }
   ...> }
|  次を作成しました: クラス 見積クラス

jshell>

jshell> (new 見積クラス()).見積メイン("旧");
見積クラス::見積メイン
見積クラス::所要量計算
見積クラス::決済条件設定

jshell> (new 見積クラス()).見積メイン("新");
見積クラス::見積メイン
見積クラス::所要量計算
新取引先見積クラス::決済条件設定
```

Copyright(C)2021 Future Office Coordinate Service Corporation All Rights Reserved. 

## 4.4. マスタ・データ編集の局所化

納品先を編集する場合に相手先が社内工場か取引先か、更に取引先が別取引先の子会社でといったように編集方法を変える（マスタ項目の使い方を変える）ようなケースがあります。この違いを処理に組込むには以下の方法があります。

### （1）interface の実装

個別機能の interface *型* を作り実装します。編集結果を受け取る側は同一の *型* として扱うことで異なるバリエーションを意識する必要がありません。

```
jshell> interface Interface 納品先 {
   ...>        String 納品場所(String[] str);
   ...> }
|  次を作成しました: インタフェース Interface 納品先

jshell> class 得意先 implements Interface 納品先{
   ...>        public String 納品場所(String[] str) {
   ...>            return str[0]+" 社 "+str[1]+" センター "+str[2]+" 棟";
   ...>        }
   ...> }
|  次を作成しました: クラス 得意先

jshell> class 社内倉庫 implements Interface 納品先{
   ...>        public String 納品場所(String[] str) {
   ...>            return str[0]+" 工場 "+str[1]+" 倉庫";
   ...>        }
   ...> }
|  次を作成しました: クラス 社内倉庫

jshell> String[] 住所 = {"A", "B", "C"};
住所 ==> String[3] { "A", "B", "C" }

jshell> Interface 納品先 納品先 = new 得意先();
納品先 ==> 得意先@68c4039c

jshell> System.out.println(納品先.納品場所(住所));
A 社 B センター C 棟

jshell> 納品先 = new 社内倉庫();
¥u7d0d¥u54c1¥u5148 ==> 社内倉庫@3c0ecd4b

jshell> System.out.println(納品先.納品場所(住所));
A 工場 B 倉庫
```

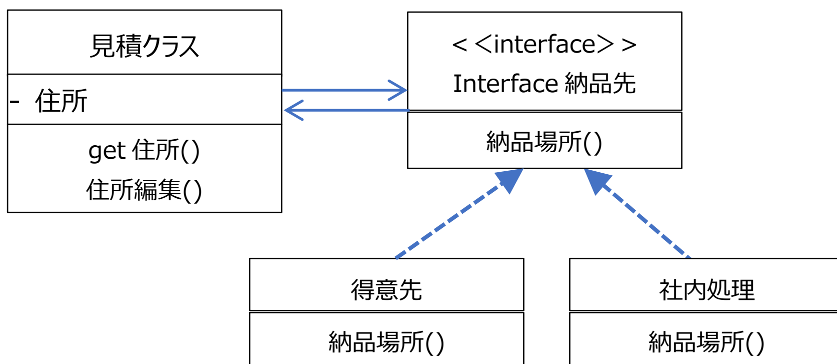
※得意先クラスと社内倉庫クラスに Interface 納品先を実装し、どちらのインスタンスも Interface 納品先 *型* として扱いながら編集の異なる出力を得ている。

Copyright(C)2021 Future Office Coordinate Service Corporation All Rights Reserved. 

（２）visitor パターン風[7]

編集対象のデータと、これを参照して編集する処理を分ける組み合わせです。

データを持っている側（見積クラス）と処理を行う側（Interface 納品先）で参照を交換し、相互に呼び出します。




① 見積クラス::住所編集の引数に Interface 納品先の具象クラス（得意先｜社内処理）参照を渡す
② 見積クラス::住所編集は Interface 納品先::納品場所の引数に自分の参照(this)を渡す
③ Interface 納品先の具象クラス::納品場所は、引数の見積クラス参照(this)を使い get 住所を呼ぶ
④ Interface 納品先の具象クラス::納品場所は、get 住所で得た情報を編集し、②に制御を戻す

[7] この例はデザインパターンの Visitor パターンを簡略化してあります。本物はその手の教材等で確認してください

Copyright(C)2021 Future Office Coordinate Service Corporation All Rights Reserved. 

```
jshell> class 見積クラス {
   ...>       private String[] 住所 = {"A", "B", "C"};
   ...>       String[] get 住所() {
   ...>           return 住所;
   ...>       }
   ...>       void 住所編集(Interface 納品先 i 納品先) {
   ...>           System.out.println( i 納品先.納品場所(this) );
   ...>       }
   ...> }
|  次を作成しました: クラス 見積クラス。しかし、 class Interface 納品先が宣言されるまで、参照できません
jshell> interface Interface 納品先 {
   ...>       String 納品場所(見積クラス cls);
   ...> }
|  次を作成しました: インタフェース Interface 納品先
jshell> class 得意先 implements Interface 納品先{
   ...>       public String 納品場所(見積クラス cls) {
   ...>           String[] str = cls.get 住所();
   ...>           return str[0]+" 社 "+str[1]+" センター "+str[2]+" 棟";
   ...>       }
   ...> }
|  次を作成しました: クラス 得意先
jshell> class 社内倉庫 implements Interface 納品先{
   ...>       public String 納品場所(見積クラス cls) {
   ...>           String[] str = cls.get 住所();
   ...>           return str[0]+" 工場 "+str[1]+" 倉庫";
   ...>       }
   ...> }
|  次を作成しました: クラス 社内倉庫

jshell> Interface 納品先 納品先 = new 得意先();
jshell> (new 見積クラス()).住所編集(納品先);
A 社 B センター C 棟

jshell> 納品先 = new 社内倉庫();
jshell> (new 見積クラス()).住所編集(納品先);
工場 B 倉庫 A
```

Copyright(C)2021 Future Office Coordinate Service Corporation All Rights Reserved. 

（３）enum を使った振り分け

enum は識別用定数（個々を“列挙子”と呼ぶ）を列挙して定義する型（列挙型）です。メソッドを書くことができ、区分等で識別と同時にそれぞれのメソッドで処理ができます。

```
jshell> enum Enum {
   ...>      //列挙子
   ...>      得意先("A", 1) {   //得意先の場合の編集パターン
   ...>          String 納品場所(String[] str) {
   ...>              return str[0]+" 社 "+str[1]+" センター "+str[2]+" 棟";
   ...>          }
   ...>      }
   ...>    , 社内倉庫("B", 2)   //これはデフォルトのメソッドで編集
   ...>    , UNDEF("", 0)
   ...>    ; //";"で列挙子終わり
   ...>
   ...>      private String kbn;
   ...>      private int no;
   ...>      private Enum(String kbn, int no){
   ...>          this.kbn=kbn;
   ...>          this.no=no;
   ...>      }
   ...>      //デフォルト編集パターン
   ...>      String 納品場所(String[] str) {
   ...>          return str[0]+" 工場 "+str[1]+" 倉庫";
   ...>      }
   ...>      //以下はコード("A","B")で列挙子を逆探するためのコード
   ...>      String getKbn(){ return this.kbn; }
   ...>      static Enum byId(String kbn) {
   ...>          return Arrays.stream(Enum.values())
   ...>              .filter(e -> e.getKbn().equals(kbn))
   ...>              .findFirst()
   ...>              .orElse(Enum.UNDEF);
   ...>      }
   ...> }
|  次を作成しました: 列挙型 Enum
jshell> String[] 住所 = {"A", "B", "C"};
住所 ==> String[3] { "A", "B", "C" }

jshell> Enum.得意先.納品場所(住所);
$3 ==> "A 社 B センター C 棟"

jshell> Enum.社内倉庫.納品場所(住所);
$4 ==> "A 工場 B 倉庫"

jshell> Enum.byId("B").納品場所(住所);
$5 ==> "A 工場 B 倉庫"
```

以上

Copyright(C)2021 Future Office Coordinate Service Corporation All Rights Reserved.