Modeling Forum 2016

オブジェクト指向プログラミングのための モデリング入門

ギルドワークス株式会社　取締役
増田　亨

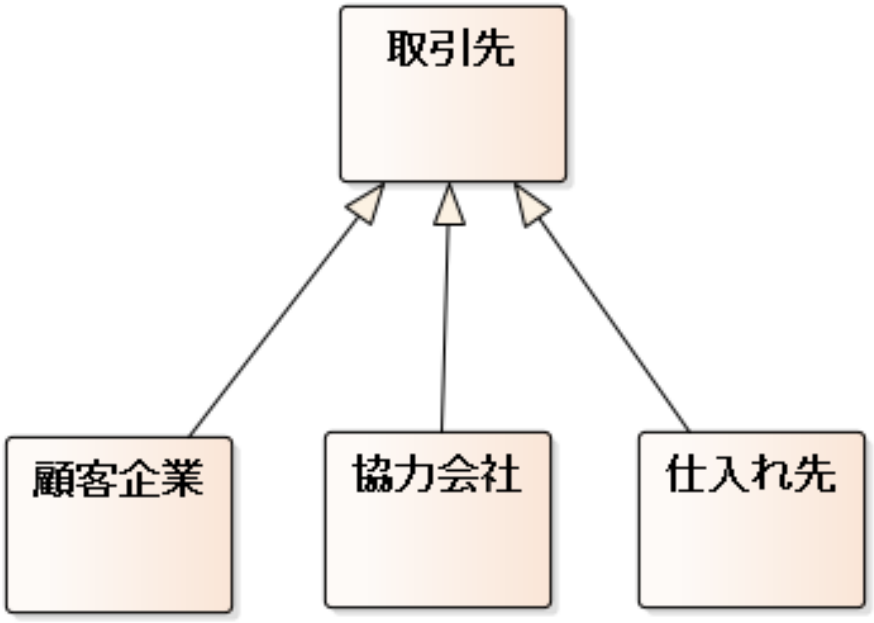
UML クラス図　三つの使い方

説明用のモデル

データモデル

オブジェクトの設計

説明用のモデル

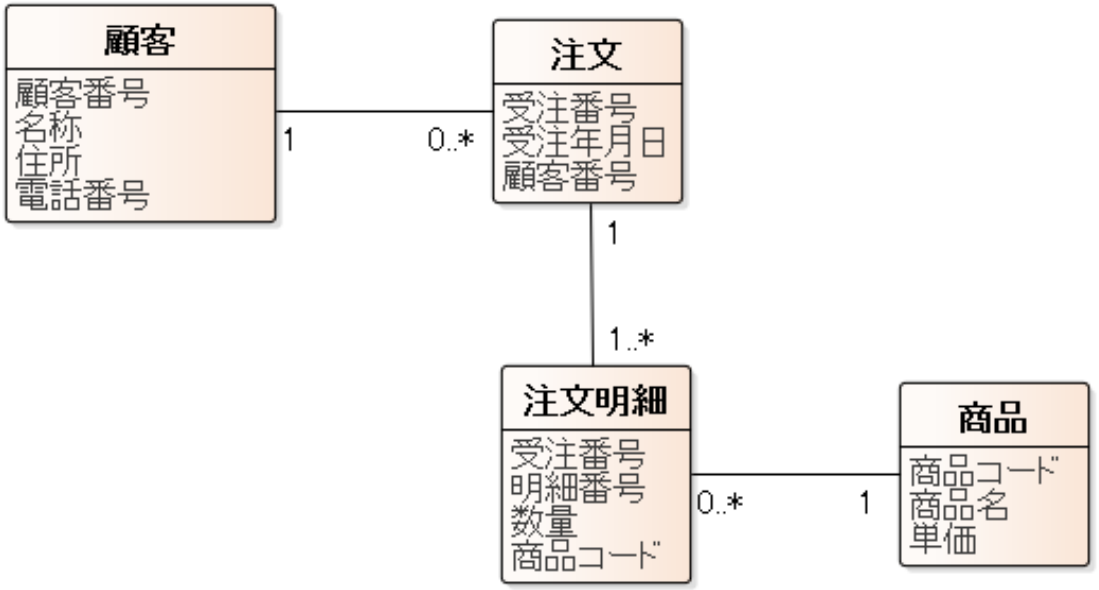


こういう説明は可能
どう実装するかは別の問題
図はきれいだが実装には不適切

実装に不適切なクラス図をもとに設計してはいけない

実装するとしたら、たぶん継承じゃなくコンポジション

データモデル



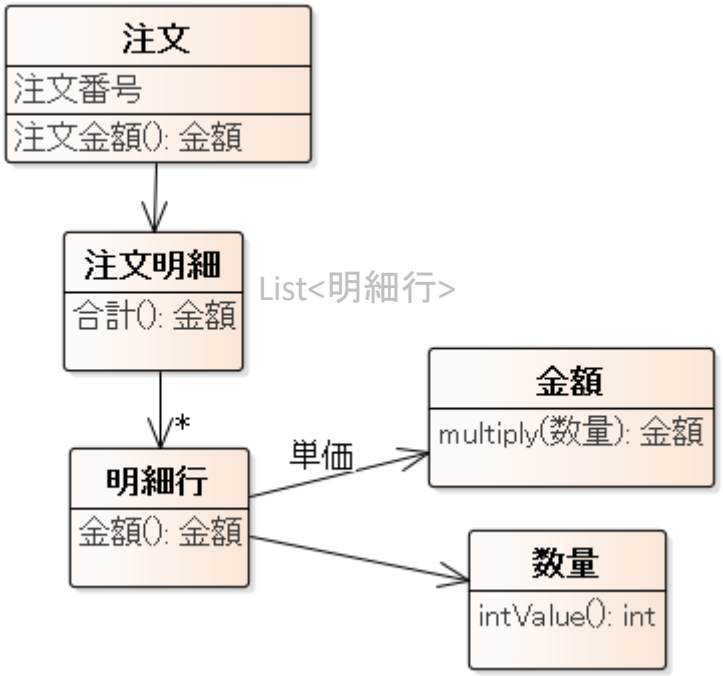
ERモデルをクラス図風に描いた図
関心事は「データの関係」

プログラムの設計図にはならない
(ER図はプログラムの設計図ではない)

注意：こういう「クラス設計に使えないクラス図」が多い

オブジェクトの設計

注文番号を指定して
注文の合計金額を得る



ロジックの置き場所（メソッド）の設計

クラスが持つメソッド（ロジック）は何か
メソッドが返す型は何か
メソッドに渡す型は何か

オブジェクト指向設計のおさらい

何が問題か

どうすれば良いか

実際にどうやるか

何が問題か：拙い設計

- コードが重複しまくっている
- 条件分岐の密林があちこちにある
- どこに何が書いてあるかわからない
- 変更した時にどこで何が起きるか推測できない
- やっていることは解読できるが、なぜ、そこでその処理が必要か意味がわからない
- パッケージ名/クラス名/メソッド名/変数名/コメントが嘘だらけ

- でも動いてる

動いていなければ、設計が悪いことは誰でもわかるのに…

オブジェクト指向で設計すると

- コードの重複がなくなる
- 条件分岐の密林がなくなる
- どこに何が書いてあるかわかりやすくなる
- 変更した時に影響範囲を限定しやすくなる
- 処理の意図がすぐわかる
- パッケージ名/クラス名/メソッド名/変数名がわかりやすく正確
  - コメントはむしろノイズ

オブジェクト指向で設計する勘所

- データ（演算対象）とロジック（演算）を同じクラスに置く
- ロジック（演算）に注目する
  - メソッドはロジックの置き場所
  - クラスはメソッドをグルーピングする手段
  - 同じインスタンス変数を使うメソッドを一つのクラスに集める
- メソッドはインスタンス変数を対象にして、判断／加工／計算をすること
  - return インスタンス変数; のgetterはNG
  - インスタンス変数 = 引数; のsetterはNG

オブジェクト指向設計：どうやるか

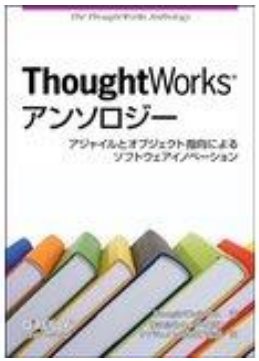
- class 宣言すれば自動的にオブジェクト指向のメリットが生まれるわけではない
- 拙い設計で書かれたコードを対象に、リファクタリングを繰り返すのは効率が悪すぎる
- 最初からオブジェクト指向らしい設計でコードを書きたい
- しかし、頭でわかっていても、なかなかオブジェクト指向らしいコードにならない
- オブジェクト指向らしい書き方を、理屈ではなく、体で覚えるのが一番の早道

体で覚える
オブジェクト指向設計

頭でわかっているつもりではだめ
オブジェクト指向らしい設計が
直観的にわかり
オブジェクトらしいコードに
自然に手が動くように

ロジックに注目することを体で覚える

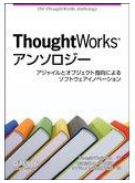
ロジックの良い置き場所を体で覚える



オブジェクト指向エクササイズ

オブジェクト指向の良さを生み出す9つのルール

オブジェクト指向エクササイズ



９つのルールで、１０００行くらい書いてみると
オブジェクト指向らしい設計を体感できる

既存のコードから9つのルールの適用例を探し、
値オブジェクトやファーストクラスコレクションを
実際に作ってみると
before/afterでオブジェクト指向設計を体感できる

９つのルール

1. 一つのメソッドでインデントは一段階
2. else 句は使わない
3. すべてのプリミティブ型と文字列をラップする
4. 一行につきドットは1つまで
5. 名前を省略しない
6. すべてのエンティティを小さくする
7. 一つのクラスのインスタンス変数は2つまで
8. ファーストクラスコレクションを使う
9. getter, setter, プロパティを使わない

どこまで体で覚えていますか？

名前をつける

1. 一つのメソッドでインデントは一段階
2. else 句は使わない
3. すべてのプリミティブ型と文字列をラップする
4. 一行につきドットは1つまで
5. 名前を省略しない
6. すべてのエンティティを小さくする
7. 一つのクラスのインスタンス変数は2
8. ファーストクラスコレクションを使う
9. getter, setter, プロパティを使わない

パッケージ
クラス
メソッド

小さい単位で
名前をつける

データとロジックを凝集させる

1. 一つのメソッドでインデントは一段階
2. else 句は使わない
3. すべてのプリミティブ型と文字列をラップする
4. 一行につきドットは1つまで
5. 名前を省略しない
6. すべてのエンティティを小さくする
7. 一つのクラスのインスタンス変数は2つまで
8. ファーストクラスコレクションを使う
9. getter, setter, プロパティを使わない

値オブジェクト

ファーストクラス
コレクション

データとロジックを別の場所におかない

複数の関心事を持たない

ルールに違反している時は、複数の関心事が混在している

1. 一つのメソッドでインデントは一段階
2. else 句は使わない
3. すべてのプリミティブ型と文字列をラップする
4. 一行につきドットは1つまで
5. 名前を省略しない
6. すべてのエンティティを小さくする
7. 一つのクラスのインスタンス変数は2つまで
8. ファーストクラスコレクションを使う
9. getter, setter, プロパティを使わない

データとロジックを凝集させる

ルール 8

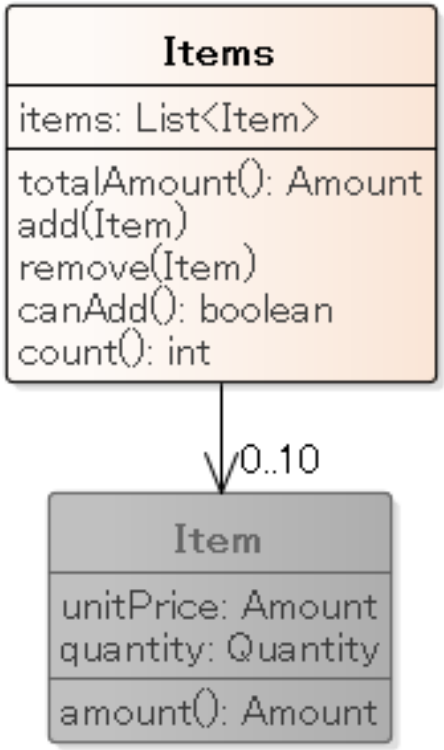
ファーストクラス
コレクション

ルール 3

すべてのプリミティブ型と
文字列をラップする

ルール8

ファーストクラスコレクション



コレクション List<Item>だけを持つクラス
コレクション操作のロジックを集める場所

外部にコレクションを公開しない
正しい状態を保証する

コレクション操作のロジックがあちこちのクラスに散らばらない/重複しない

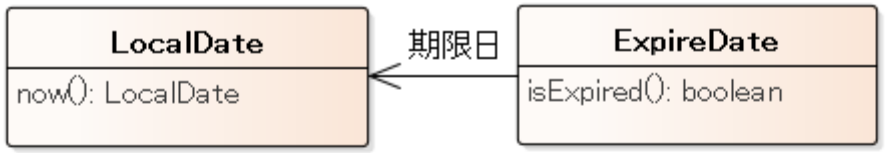
Item の件数制限や、合算方法に変更があっても他のクラスに波及しない

Itemの一覧画面に関する変更は、まずここを見れば良い

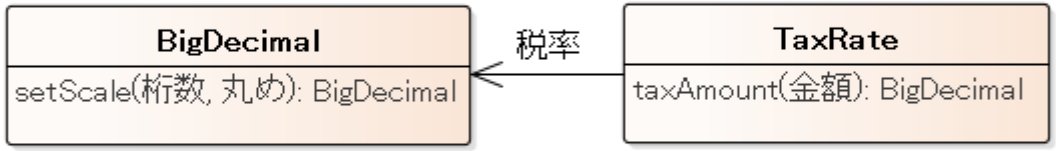
ロジックを集める感覚を体で覚える
関心事とクラスを一致させる感覚を体で覚える

ルール3

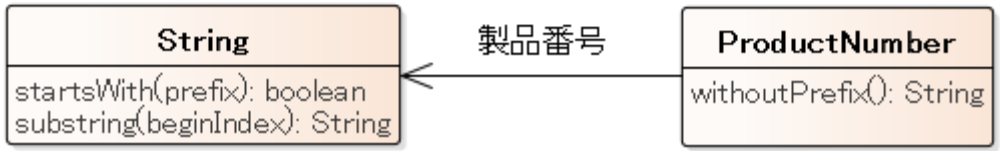
すべてのプリミティブと文字列をラップする



目的を限定した日付
日付の計算ロジックの置き場所



目的を限定したBigDecimal
丸め計算のロジックの置き場所



目的を限定した文字列
文字列加工ロジックの置き場所

ロジックをデータを持つクラスに置く感覚を体で覚える
関心事とクラスを一致させる感覚を体で覚える

データとロジックを凝集させる

ルール 8

ファーストクラス
コレクション

ルール 3

すべてのプリミティブ型と
文字列をラップする

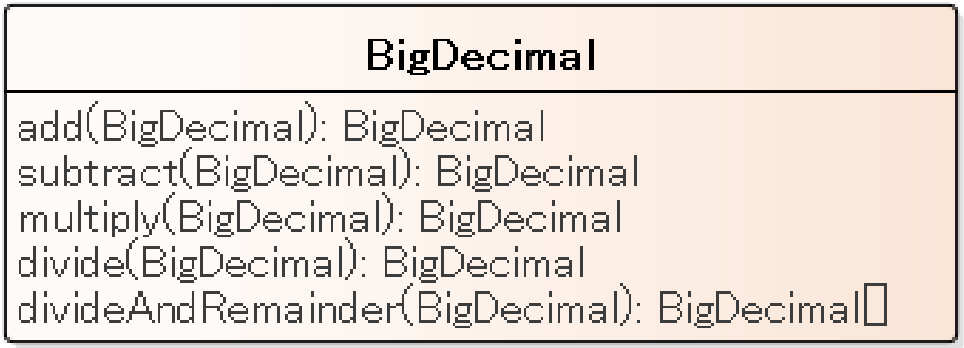
この二つを体で覚えると以下3つは自然にルール通りになる

ルール9 getter/setter は使わない
ルール6 すべてのエンティティを小さくする
ルール7 ひとつのクラスにインスタンス変数は2つまで

ルール9

getter/setterは使わない

- データのある場所に、ロジックを置けば、getterは不要
- 値の変更が必要な時は、setter ではなく、別の値を持った同じクラスのインスタンスを返す＜閉じた操作＞



クラスの用途をメソッドで表現

別のインスタンスを返す
BigDecimal型に閉じている

返す型も渡す型も BigDecimal

数値や日付をラップしたクラスの基本パターン

ルール 7

ひとつのクラスに インスタンス変数は2つまで

- ロジックの置き場所としてクラスを考える
- クラスのメソッドは、どのメソッドもインスタンス変数を「すべて」使うのが良い設計
  - インスタンス変数を使わないメソッドは、そのクラスに存在する理由がない
  - 特定のインスタンス変数だけを使うメソッドは、そのインスタンス変数とメソッドを抜き出して、別のクラスにする

ルール 6

すべてのエンティティは小さく

- 大きなクラスは、インスタンス変数を持ちすぎ
- 長いメソッドは、さまざまなインスタンス変数／ローカル変数への操作をごちゃまぜにしている
- 一つの変数に注目し、その変数だけに関心のあるロジックを集めてメソッドにする
- あるインスタンス変数を使うメソッドを集めて、別のクラスに抽出する

ロジックとインスタンス変数が密接に関係する凝集したクラスの感覚を体で覚える

複数の関心事を持たない

ルール 1 — インデントは一段階

ルール 2 — else 句は使わない

ルール 4 — 一行につきドットは1つ

ルール 7 — 一つのクラスに
インスタンス変数2つまで

ルール１

インデントは一段階

- インデントは関心事の階層構造の表現
  - 深いインデントは、複数の関心事の階層構造
- 改善のやり方
  - 最も深いインデントをメソッドに抽出
  - メソッドをクラスにすることを検討する
    - メソッドの引数をインスタンス変数にしたクラス
    - ロジックをそのクラスに移動
    - そのクラスのインスタンスに仕事をさせる
  - これをインデントの数だけ繰り返す
- 階層ごとのロジックを別クラスに分離する

インデントに埋め込まれた構造が、
クラス構造として現れる

ルール 2

else 句は使わない

- else 句は異なる関心事の表現
- 改善のやり方
  - 判断部も処理部もすべてメソッドに抽出する
  - 事前条件のチェックはガード節で早期リターンする
    - if(invalid()) return ;
  - 残った条件を、相互排他かつ全体網羅になるようにグルーピングする
    - 複雑なif-else構造は異なるグループが混在している
  - グルーピングした条件区分を列挙型(enum)で宣言する
  - ロジックをenumに移動する

if文の条件分岐の異なる関心事の整理を体で覚える
相互排他かつ全体網羅の構造にすることを体で覚える

ルール 2

else 句は使わない（続き）

- 区分ごとのロジックが複雑な場合
  - 多態を使う
  - 区分ごとのロジックを別のクラスにする
  - インタフェース宣言で、区分ごとのクラスを同一グループとして宣言する
  - 区分ごとのロジックを使う側は

条件分岐の構造を、if文/switch文ではなく、
区分ごとのクラスで表現することを体で覚える

クラス図が大きく変化する

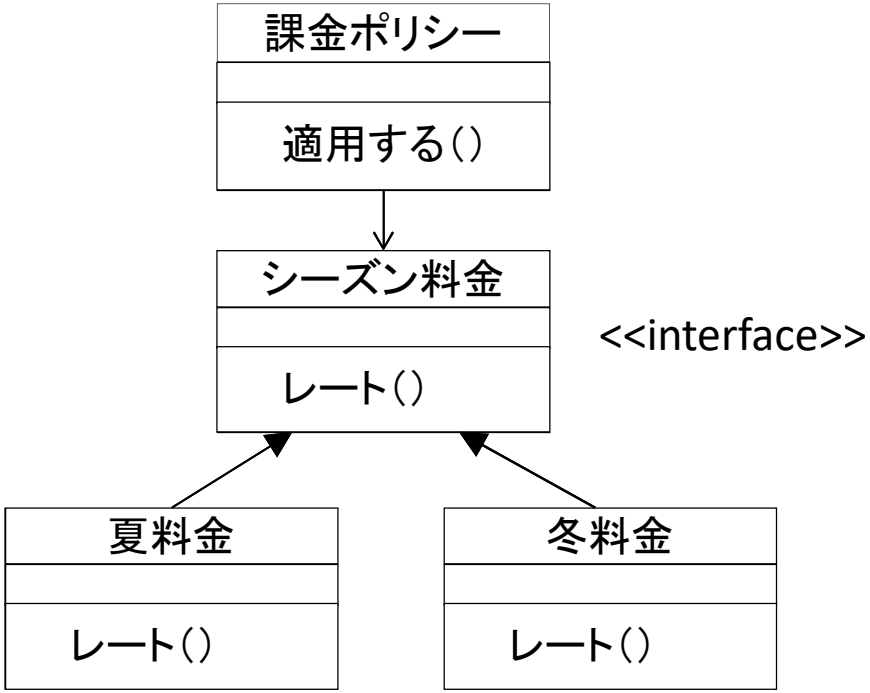
区分をクラスで表現する

計算方法(How)を表現

計算 () メソッドに埋もれ、
暗黙化する業務知識

シンプルな設計に見えるが、
ルールの変更・追加のたびに
計算 () メソッドが肥大化し、
if 文が増殖する

料金区分(What)をクラスで表現



業務知識をそのまま、クラスとして表現
複雑に見えるが、ルール変更・追加が、
楽で安全になる

ルール 4 一行につきドットは1つ

- person.preference.contactMethod.doSomething()
  - クラスの構造が、コードに埋め込まれている
  - 変更に弱い
  - 意図があいまいになる
- 改善のやり方
  - 説明用変数を使ってドット構造を分解してみる

```
Preference preference = person.preference();
ContactMethod method = preference.contactMethod();
method.doSomething();
```

ルール 4 一行につきドットは1つ

- person.preference.contactMethod.doSomething()
- doSomething() を実行する場所を再考する
  このクラスがpersonをインスタンス変数に持っているとして
  - person にdoSomething()をやらせる？
  - method をインスタンス変数として持つ？
  - methodを持った別のクラスを作って、そっちで doSomething() する？

なんでも階層構造で整理するのではなく、もっと柔軟に
ロジックの置き場所を考えることを体で覚える練習

ルール 7

一つのクラスに
インスタンス変数は二つまで

- メソッドはインスタンス変数を使って判断／加工／計算を行う
- 同じクラスのメソッドは、そのクラスのインスタンス変数をすべて使うことが原則
- もし三つ以上の変数を同時に使うロジックがあるなら、ロジックの分解と、別クラスへの抽出ができる

メソッドとインスタンス変数を強く関係づけること、
クラスはそのための道具であることを体で覚える練習

名前をつける

ルール 6

すべてのエンティティを
小さく

ルール 5

名前を省略しない

ルール 6

すべてのエンティティを小さく

- クラスは50行以内
- パッケージは10ファイル以内
- メソッドは3行以内（追加ルール）
- 他のルールを実践すると、小さなクラスがたくさんできる
- パッケージで整理する
- 名前を考えるのがたいへん

意図やロジックを、メソッドの中の記述ではなく、
メソッド名、クラス名、パッケージ名で表現することを体で覚える練習

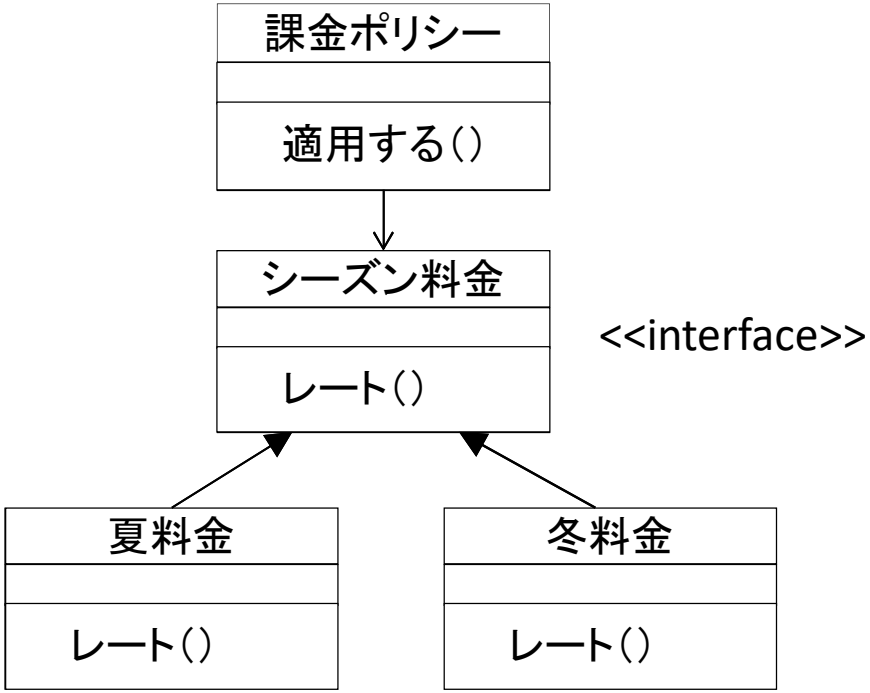
ロジックをクラスで表現する

計算方法(How)を表現

計算 () メソッドに埋もれ、
暗黙化する業務知識

シンプルな設計に見えるが、
ルールの変更・追加のたびに
計算 () メソッドが肥大化し、
if 文が増殖する

料金区分(What)をクラスで表現



業務知識をそのまま、クラスとして表現
複雑に見えるが、ルール変更・追加が、
楽で安全になる

ルール 5

名前を省略しない

- メソッド名、クラス名、パッケージ名
  - 考えるのがたいへん
  - ついつい安易な名前をつけがち
- どこに何が書いてあるかわかりやすくする
  - 「良い名前」を見つける
  - 「良い名前」を明記する

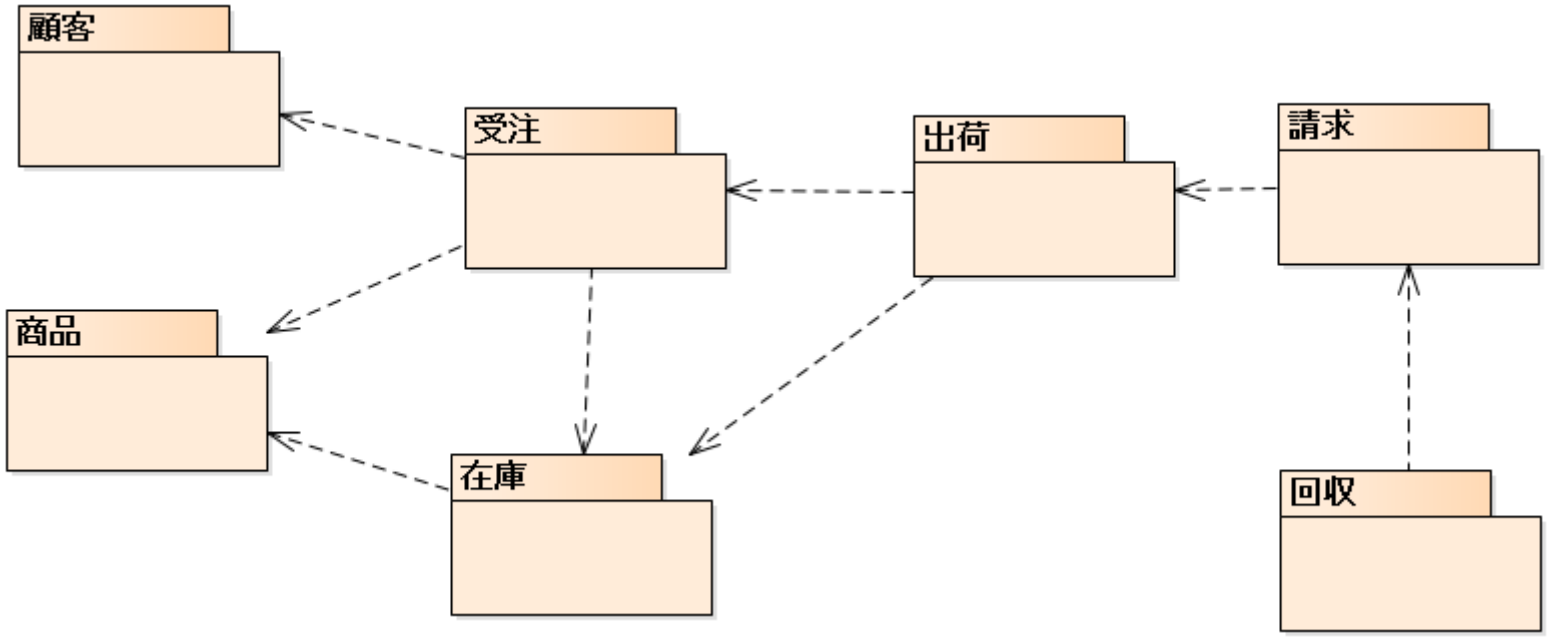
プログラミング言語の匿名化の仕組みは設計を劣化させる
ダックタイピング／タプル／ラムダ

ルール 5

名前を省略しない

- 名前たいせつ
- クラス図やパッケージ図は、名前の関係の俯瞰と整理に効果的
- ただし
  - 実装から遊離した概念モデルにしないこと
  - データに注目した構造にしないこと

パッケージ図を使った
ロジックの置き場所の俯瞰



与信限度額を考慮するとどうなるか？
発注と仕入れまで視野を広げるとどうなるか？
矢印で示した一方向の依存関係で正しく実装しているか／実装できるか？

オブジェクト指向で
プログラミングするためのモデリング

UML クラス図　三つの使い方

説明用のモデル

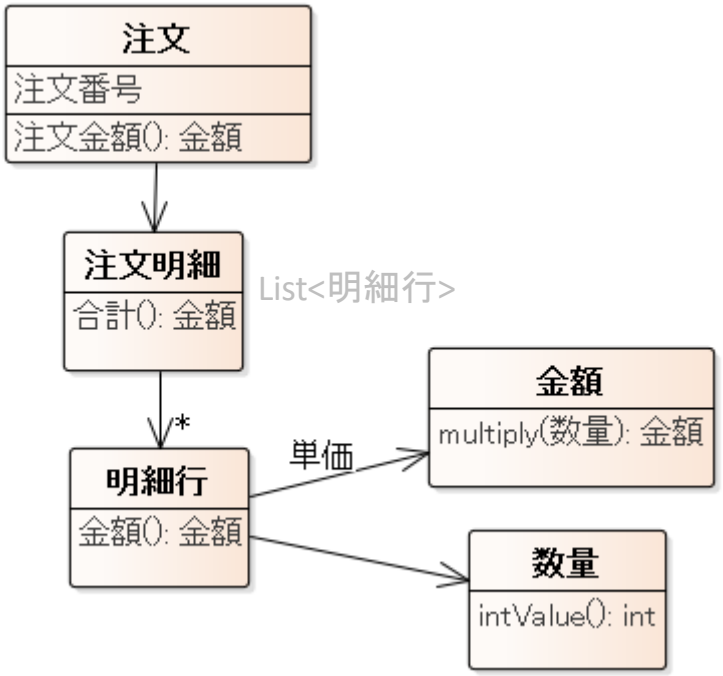
実装には不適切

データモデル

プログラムの
設計図ではない

オブジェクトの設計

オブジェクトの設計



プログラムを書くために、こういうクラス図を必ず描くか？

コードで書けるなら
クラス図を描く必要はない

コードで書くのが難しい時
クラス図でスケッチしてみる

コードで書いていておかしいと感じた時
別の記法（ex. クラス図）で設計を見直す

主要なクラスの関係を俯瞰したい時
図にしてみるとわかりやすいことがある

インクリメンタルな設計

毎日設計する

パッケージ構成の変更／名前の改善
クラス構成の変更／名前の改善
メソッド構成の変更／名前の改善

インクリメンタルな設計

- 良い設計は最初からは見つからない
  - パッケージ構成／名前
  - クラス構成／名前
  - メソッド構成／名前
- クラス図だけで格闘していても良い設計にはたどり着けない

インクリメンタルな設計

ざくっとラフスケッチをしてみる
コードで書いて、動かしてみる
リファクタリングをする
行き詰まったら
絵に描いてみる
会話をしてみる
結果をコードに反映する
そうやって設計を改善し続ける
それがもっとも確実で効率的