JJUG CCC 2024 Spring

いまどきの

分析設計パターン 10選

～複雑な業務ロジックに立ち向かう実践技法

2024年6月16日

有限会社システム設計　増田 亨

自己紹介

増田 亨（ますだ とおる）

業務系アプリケーションソフトウェアの開発者

モデル駆動設計

Java/Spring Boot/IntelliJ IDEA/JIG

有限会社システム設計 代表
コミューン株式会社 技術顧問

訳書

7月20日発売予定
予約販売中

著書

分析設計パターン 10選

業務系アプリケーションでよくでてくる課題とその実装方法

特に、**複雑な業務ロジック**の扱い方の私の経験則を言語化

複雑な業務ロジックとは？

ソフトウェア開発がたいへんになる理由

複雑な業務
ロジック

ここを扱う**分析設計パターン**を習得すると

ソフトウェア開発がもっと**楽で楽しく**なる

４つの観点

事業価値の
提供

アプリケーション
アーキテクチャ

複雑な業務
ロジック

要件定義
仕様の記述

ソフトウェア
テスト

事業価値の提供

複雑な業務ロジックに焦点を合わせることで

事業価値の高いソフトウェアを生み出せる

複雑な業務ロジック

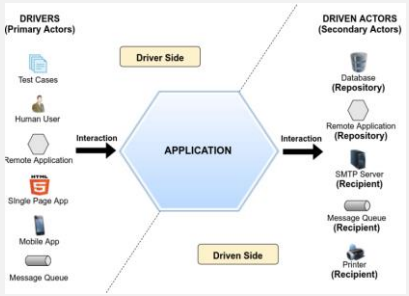
競争優位を獲得し維持するためには、

さまざまな**事業活動の最適化**が必要になる。

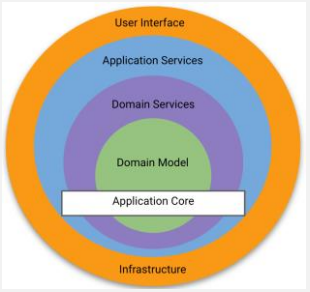
複雑な業務ロジックは、事業活動を最適化するための
ビジネスルールに基づく計算判断。

アプリケーションアーキテクチャ

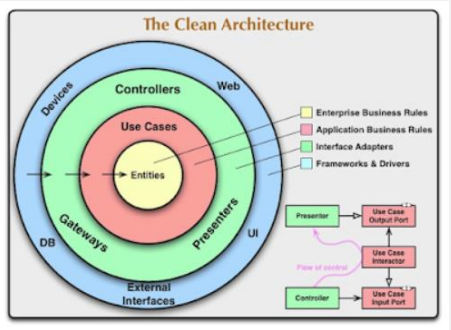
複雑な業務ロジック



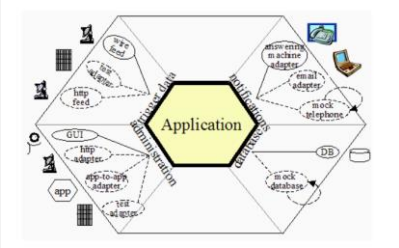
ヘキサゴナル



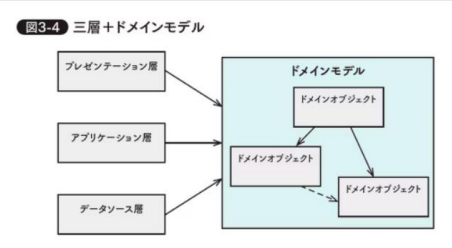
オニオン



クリーン



ポートとアダプター



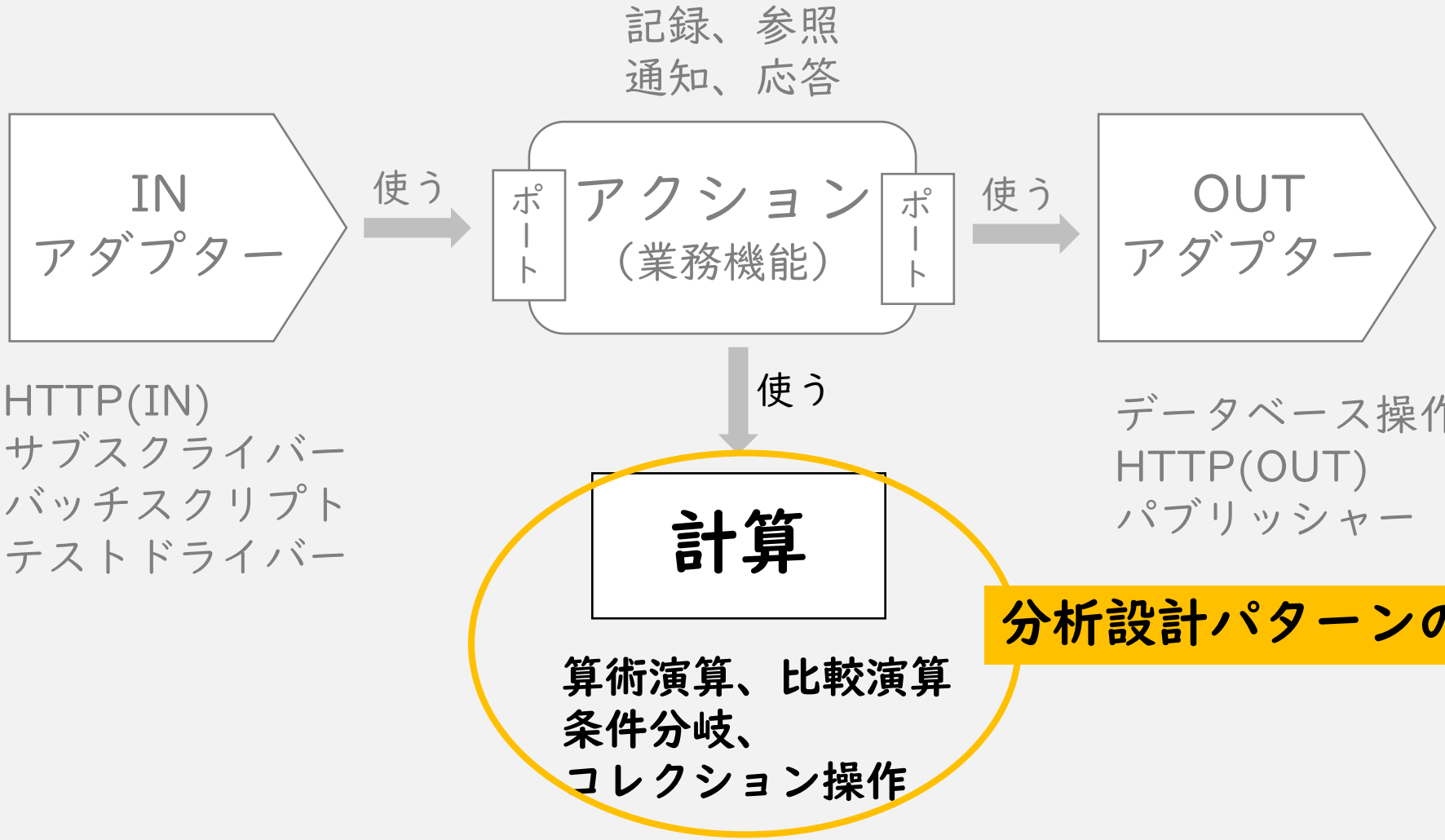
三層＋ドメインモデル

どのアーキテクチャスタイルも基本は同じこと言っている

① **複雑な業務ロジックを独立させる**

② アプリケーションのその他の部分が、複雑な業務ロジックに依存する

計算、アクション、アダプターの分離



要件定義（仕様の記述）

複雑なビジネスルールの可視化

行動の刺激と制約
計算ルール
判定ルール

複雑な業務
ロジック

ビジネスルールが不明確なまま開発を進めると、

問題の発見が遅れ、その結果、

関係者との**確認作業**や**修正作業**が大きな負担になる。

分析設計パターンは
ビジネスルールを
明確に記述する技法

ソフトウェアテスト（品質保証）

複雑な業務
ロジック

3大テスト技法

境界値テスト

状態遷移テスト

デシジョンテーブルテスト

品質保証の**重点課題**

テストの準備と実施に**多大な労力**を投入

分析設計パターンの関心事と
テスト技法の対象は重なっている

それぞれの分析設計パターンの説明の流れ

ビジネスルールや
要求仕様の具体例

分析設計の要点
解法のアプローチ

コードの中核部分
ハイライト

分析設計パターン【基礎編】

① 値の種類

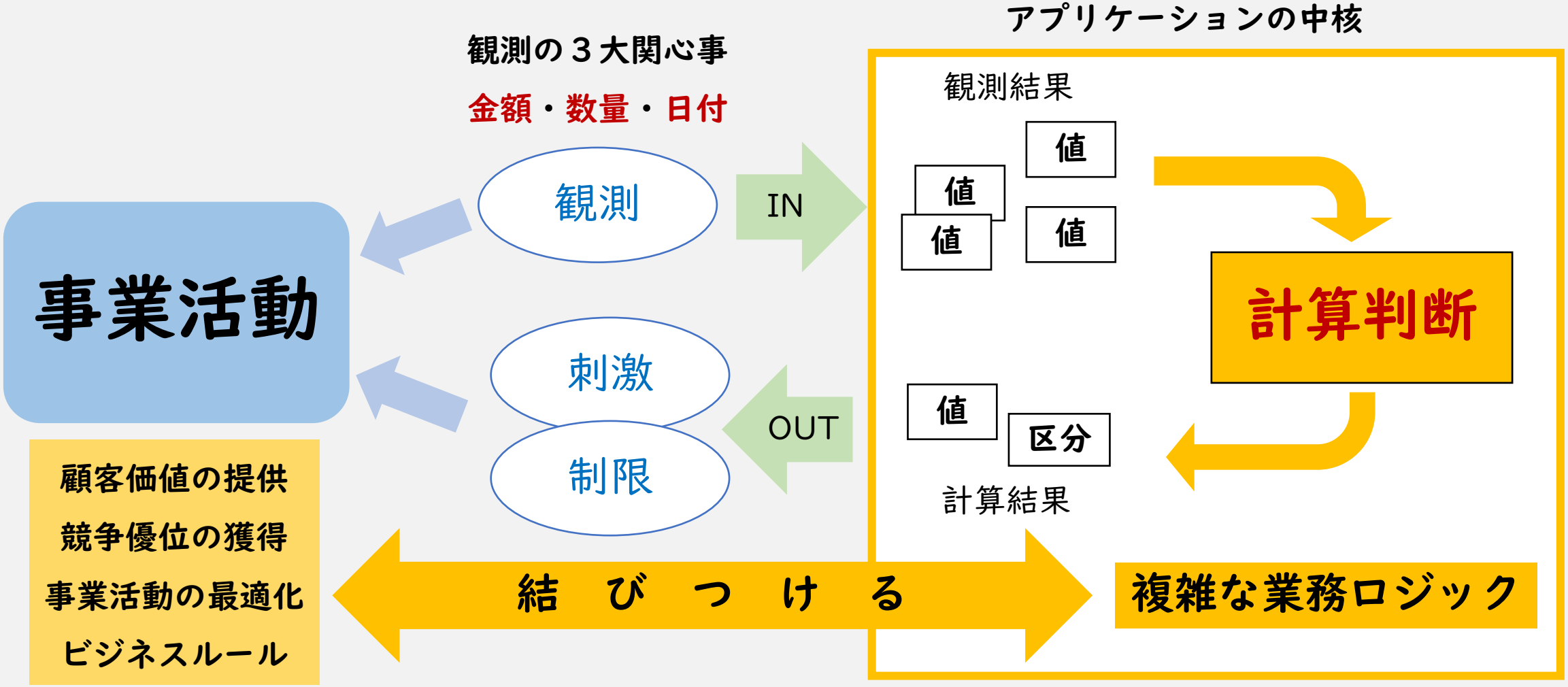
② 範囲

③ 階段型

①値の種類

複雑な業務ロジックを記述するための**基本中の基本**

事業活動を観測した値を使って計算判断



値の種類（ビジネスの関心事）を特定する

名前に注目する？

データ項目名など

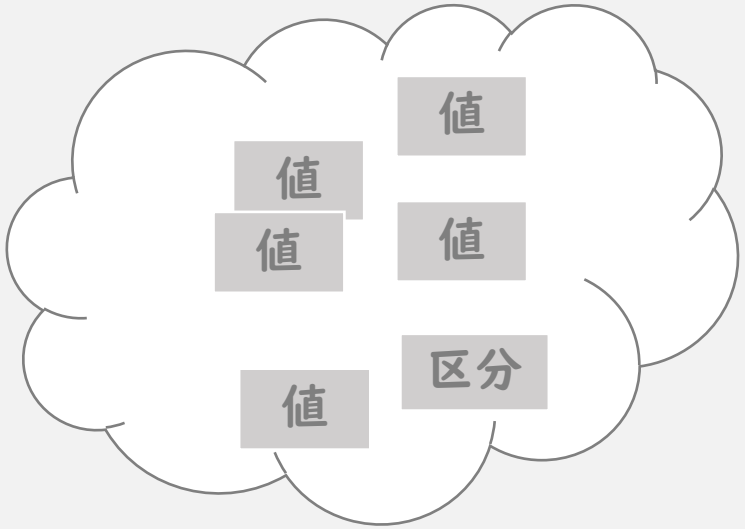
ビジネスの関心事を表現する値には

名前がついている

ただし、名前と値の種類は
必ずしも一致しない

名前が同じ　→　異なる種類の値
名前が違う　→　同じ種類の値

観測結果、計算結果



分類の着眼点

- **単位**
- **範囲**
- **関係**

値の種類を分類整理することで
ビジネスの関心事を
具体的に理解できる

値の種類①：単位に注目する

- **単位**は、**値の種類**を分類するもっとも**重要な手がかり**
  - ✓円、個、回、Kg、cm、率、年月、月日、…
- **異なる単位**の数値を、すべて同じ型（intやlong）で扱うのは、**不具合の原因**
- 値の種類ごとに**アプリケーション独自の型**を定義する
  - ✓**関心の分離**の実践的な技法
  - ✓ビジネスルールの**正確な記述**

値の種類②：範囲に注目する

事業活動で発生する値は**適切な範囲**がある

- プリミティブ型の範囲より、かなり狭い（用途限定）
- **範囲が決まる理由**の理解 → 重要な業務知識

範囲は重要なビジネスルール

事業活動最適化の制約条件と関係する

範囲が異なれば**別の種類**の値

- 「金額」を表すさまざまな値　⇒ 用途によって金額範囲が異なる

適切な値の範囲はソフトウェアテストの重点の一つ

- **境界値**テスト

観測結果だけでなく、**計算結果**も適切な**範囲**がある

値の種類③：関係（計算式）に注目する

単価	×	数量	＝	金額
円/Kg		Kg		円

計算式として表現されたビジネスルール（値の関係）

ビジネスルールとコードを結びつける

```
class UnitPrice {
    Amount 金額;
    Unit   単位;

    Amount 掛ける(Quantity 数量) {
        if (単位 != 数量.単位) throw new IllegalArgumentException("単位の不一致");
        return new Amount(金額.額() * 数量.量);
    }
}
```

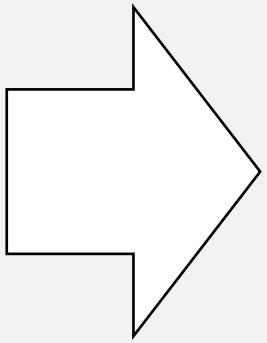
値の関係をメソッドで表現する基本パターン

演算の選択肢

比較演算
- 同じ（=、!=）
- 大きい、小さい（<、>）

算術演算
- 足す、引く（+、-）
- 掛ける（×）
- 割る（÷）

型変換
- 整数へ、整数から
- 文字列へ、文字列から



計算式

値 演算 値 ＝ 値

ビジネスルールとコードを対応させる

メソッドで表現

クラスの型#演算(引数の型)：返す型

その値の**目的**に応じて
必要な演算を選択して
メソッド（の集合）として**用途**を表明する

ビジネスルールを表現する基本部品

②範囲型

Range クラス

値の範囲の判定

業務アプリケーションのあちこちにでてくる業務ロジック

プログラミングとしては、**初歩的な比較演算**

>, >=, <, <=, ==, !=

不具合の原因になりやすい

- ✓「含む」「含まない」の取り違えや記述ミス
- ✓ **簡単な式**なので、あちこちに気軽に記述　→　同じロジックの**重複記述**
- ✓ **修正漏れ** and/or **誤修正**

範囲を適切に扱う工夫

範囲型のクラスを作って、範囲の判定ロジックの記述を**一元化**

金額範囲型

金額範囲はビジネスの重要な関心事

```
class AmountRange {
  Amount 下限; // 含む
  Amount 上限; // 含まない

  boolean が次の金額を含む(Amount この金額) {
    if (この金額.が次の金額以上である(上限)) return false;
    if (この金額.が次の金額未満である(下限)) return false;
    return true;
  }
}
```

// 金額型

プリミティブな比較演算の記述をカプセル化

```
class Amount {
  int 金額;
  boolean が次の金額以上である(Amount other) { return 金額 >= other.金額;  }
  boolean が次の金額未満である(Amount other) { return 金額 < other.金額; }
}
```

日付範囲型

日付範囲はビジネスの重要な関心事

```
class DateRange {
  LocalDate 開始日; //含む
  LocalDate 終了日; //含む

  boolean 期間内(LocalDate 日付) {
    if (日付.isBefore(開始日)) return false;
    if (日付.isAfter(終了日)) return false;
    return true;
  }
}
```

プリミティブな比較演算の記述をカプセル化

設計ノート　日付範囲どうしの演算が役に立つことがある

期間と期間の合成（期間どうしの足し算、期間どうしの引き算、重複期間）
期間と期間の関係の判定（隣接（連続）、重複、包含、離間）

③階段型

階段型のビジネスルール

購入金額	割引率
2,000円未満	0%
5,000円未満	3%
10,000円未満	5%
10,000円以上	10%

- 境界値テストが重要になる典型的なケース
- 境界を含む/含まないの取り違え
- 変更があった時に境界の不整合が起きがち

設計（コードの書き方）で品質保証する

enumを使った階段型の計算

```
enum DiscountCategory {
    少額(Amount.of(2_000), DiscountRate.of(0)),
    普通(Amount.of(5_000), DiscountRate.of(3)),
    高額(Amount.of(10_000), DiscountRate.of(5)),
    超高額(Amount.上限額, DiscountRate.of(10));

    final Amount 上限境界;
    final DiscountRate 割引率;

    DiscountCategory(Amount 上限境界, DiscountRate 割引率) {
        this.上限境界 = 上限境界;
        this.割引率 = 割引率;
    }

  static Amount 割り引く(Amount 割引対象金額) {
        DiscountCategory 価格帯 = 該当する価格帯(割引対象金額);
        return 割引対象金額.割り引く(価格帯.割引率);
    }
```

階段の**上限**と**割引率**を定義

ビジネスルールの表現

価格帯の判定：実装の詳細

```
static final DiscountCategory[] 価格帯一覧 = DiscountCategory.values();

Amount 下限境界() {
    if (this == 少額) return Amount.of(0);
    return 価格帯一覧[ordinal() - 1].上限境界;
}

static final Map<DiscountCategory, AmountRange> 価格帯別割引テーブル =   // 金額範囲型のMap
    Arrays.stream(価格帯一覧).collect(
        toMap(価格帯 -> 価格帯, 価格帯 -> AmountRange.生成(価格帯.下限境界(), 価格帯.上限境界))
    );

static DiscountCategory 該当する価格帯(Amount 元の金額) {
    return 価格帯別割引テーブル.entrySet().stream()
            .filter(価格帯 -> 価格帯.getValue().が次の金額を含む(元の金額))
            .findFirst()
            .orElseThrow().getKey();
}
```

下限の導出（一つ前の要素の上限）

金額範囲型のMapを使った判定

時間とともに変化する状態の扱い方

分析設計パターン【中級編】

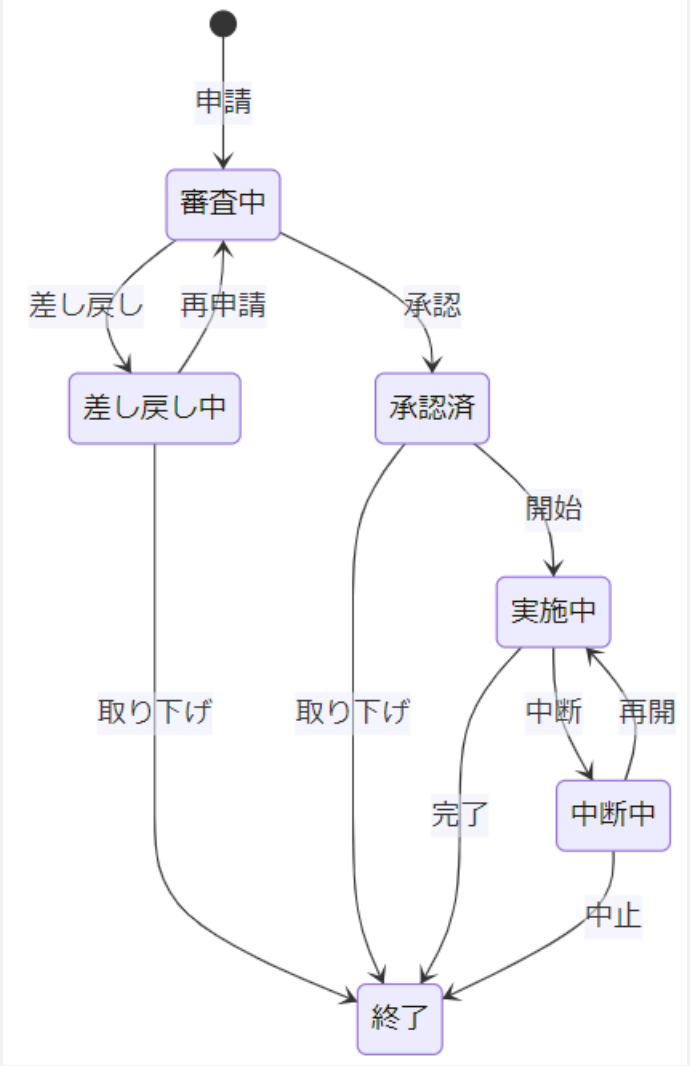
④ 状態遷移

⑤ 入出金履歴と残高

⑥ 未来在庫

④状態遷移

状態遷移図



状態遷移表

from/to	審査中	承認済	差し戻し中	実施中	中断中	終了
審査中		承認	差し戻し			
承認済				開始		取り下げ
差し戻し中	再申請					取り下げ
実施中					中断	完了
中断中				再開		中止
終了						

・ある状態で、許されるアクションに何があるか？

・ある状態で、そのアクションの実行は適切か？

こういう判定ロジックを表現する方法

設計（コードの書き方）で品質保証する

状態遷移モデルの実装

if文/switch文を使わずに宣言的に記述

```
/**
 * 状態
 */
enum State {
    審査中,
    承認済,
    差し戻し中,
    実施中,
    中断中,
    終了
}
```

```
/**
 * アクション
 */
enum Action {
        承認,
        差し戻し,
        再申請,
        取り下げ,
        開始,
        完了,
        中止,
        中断,
        再開
}
```

設計ノート
おそらく、複数の状態遷移モデルが混在している。
分割して整理するとなんらかのブレークスルーがありそう。

```
class ActionsByState {
  Map<状態, Set<アクション>> 状態遷移表 =
    Map.of(
        審査中, Set.of(承認, 差し戻し),
        承認済, Set.of(開始, 取り下げ),
        差し戻し中, Set.of(再申請, 取り下げ),
        実施中, Set.of(中断, 完了),
        中断中, Set.of(再開, 中止),
        終了, Set.of()
    );

  Set<アクション> 可能なアクションの一覧(状態) {
    return 状態遷移表.get(状態);
  }

  boolean 妥当性(状態, アクション) {
    return 可能なアクションの一覧(状態)
                .contains(アクション);
  }
}
```

⑤入出金履歴と残高

イベントソーシング（その１）

ステートソーシングとイベントソーシング

ステートソーシング

- 残高を上書き更新　可変量　挙動が不安定

日付	入金	出金	残高
~~4月5日~~	~~1,000円~~	~~1,000円~~	~~0円~~
~~4月10日~~	~~3,000円~~		~~3,000円~~
~~4月11日~~		~~2,000円~~	~~1,000円~~
~~4月19日~~		~~1,000円~~	~~0円~~
4月20日	~~1,000円~~		1,000円

最後の更新日

変動し続ける残高

イベントソーシング

- 残高を入出金の履歴から導出　不変量　挙動が安定

日付	入金	出金	残高
4月5日	1,000円	1,000円	残高()
4月10日	3,000円		残高()
4月11日		2,000円	残高()
4月19日		1,000円	残高()
4月20日	1,000円		残高()

確定した事実

入出金履歴から残高を投影（モデル）

残高フィールドではなく
履歴フィールドを持つ

確定した事実から、状態を（毎回）計算する

入出金イベントの**履歴** ⇒ 指定日付の**残高**

履歴の内容は過去の事実として確定しているので
残高は、どのタイミングで実行しても常に同じ値になる

入出金履歴から残高を投影（実装）

```
class 入出金履歴 {
    List<入出金イベント> 入出金履歴;

    金額 残高の投影(日付 対象日) {
        return 入出金履歴.stream()
                .filter(入出金イベント -> 入出金イベント.以前(対象日))
                .map(入出金イベント::金額) // 出金額はマイナスに変換
                .reduce(Amount.ゼロ, Amount::足す); //たたみこむ
    }
}
```

入出金の記録があれば、任意の時点の残高を確実に導出できる

行動分析、金銭取引の正確な記録、法的な監査記録

⑥未来在庫

イベントソーシング（その２）

入出庫の予定（未来在庫）

日付	入庫予定	出庫予定
7月1日	10	
7月2日		5
7月4日	20	
7月5日		15
7月5日		8
7月10日	10	
7月11日	20	
7月12日		15

・日付を指定して、**出荷可能数**を調べる

・希望出荷数を指定して、**出荷可能日**を調べる

考え方（計算式）

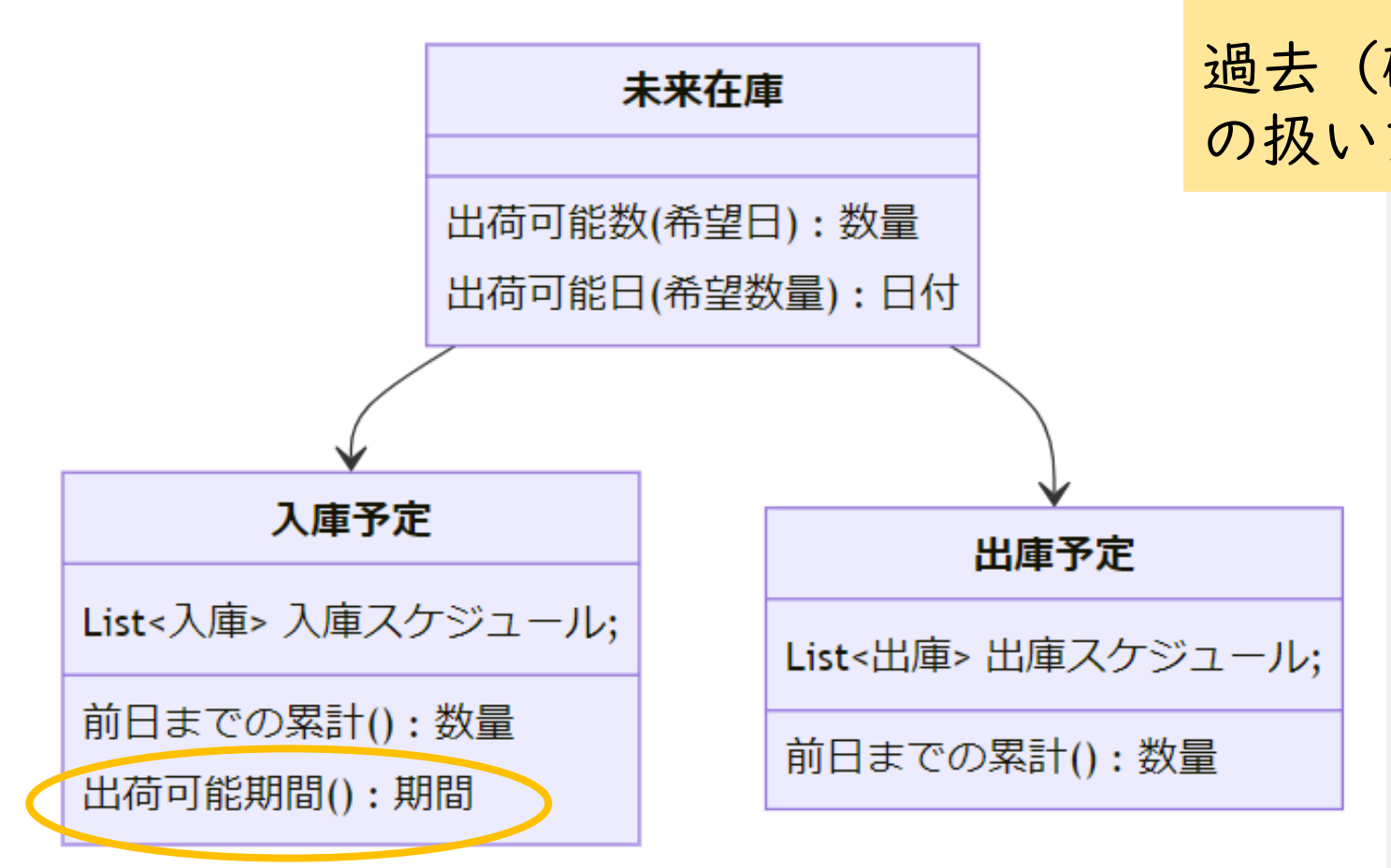
当日残高見込み ＝ 前日残高見込み ＋ 入庫予定数 − 出庫予定数

当日出荷可能数 ＝ 前日残高見込み − 当日出庫予定数

未来在庫のモデル

入庫予定と出庫予定を
別のコレクションで管理

過去（確定）と未来（不確定）
の扱い方の違い



設計ノート

入庫予定と出庫予定は本質的に異なる未来
・入庫予定：確度
・出庫予定：優先度別の割り当て

入金履歴と出金履歴は「過去」の事実なので同じに扱った

未来在庫判定の実装

```
int 出荷可能数(LocalDate 指定日) {
        int 前日残高 = 入庫予定.前日までの累計(指定日) - 出庫予定.前日までの累計(指定日);
        int 当日出荷予定数 = 出庫予定.出荷予定数(指定日);
        return 前日残高 - 当日出荷予定数;
}

LocalDate もっとも早い出荷可能日(int 出荷希望数) {
        if (!出荷可能(出荷希望数)) throw new IllegalStateException("出荷不能");

        List<LocalDate> 出荷可能日リスト = 入庫予定.出荷可能期間()
                .filter(対象日 -> 出荷可能数(対象日) >= 出荷希望数)
                .toList();

        return 出荷可能日リスト.getFirst();
}
```

ビジネスルールの表現

設計ノート　下のメソッドは、実装の詳細をもっとカプセル化したほうがよさそう

分析設計パターン【上級編】

⑦ スキルマッチング（Set演算）

⑧ 比例配分（割合と端数処理）

⑨ 複合条件（デシジョンテーブル）

⑩ 経路探索（ネットワーク構造の表現と計算）

⑦スキルマッチング

Set要素の一致度の演算

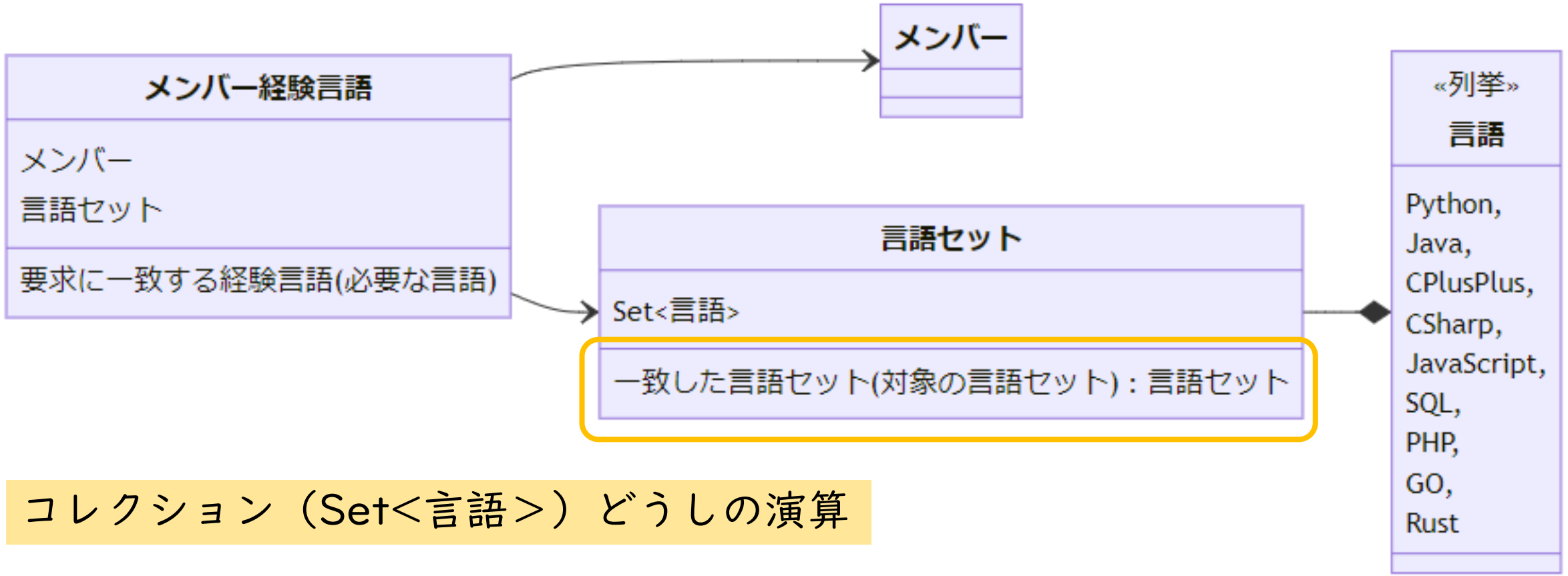
スキルマッチング

- メンバーは、いろいろなプログラミング言語を経験している
- 新たな開発プロジェクトを開始するにあたり、適切な経験を持っているメンバーを特定したい

設計ノート

技術者向けにわかりやすくプログラミング言語を例にしているが、実際に開発したのは、複数の特殊スキルの組み合わせが必要な店舗スタッフのシフトスケジュール調整機能

経験言語のマッチング（モデル）



コレクション（Set<言語>）どうしの演算

経験言語のマッチング（実装）

```
class 言語セット {
    Set<言語> 言語セット;

    言語セット 合致した言語(比較対象) {
        Set<言語> 一致した言語セット = 言語セット.stream()
                .filter(比較対象.言語セット::contains)
                .collect(toSet());
        return new 言語セット(一致した言語セット);
    }
}
```

Set<言語>どうしで演算

設計ノート

実際のロジックは、スキルのレベル分けとニーズの重要度で一致度合いを数値化して比較

⑧比例配分

割合と端数処理

分担比率と厳密な配分

例： 貸付の分担率に応じた、貸付額、返済額、手数料、利息の比例配分

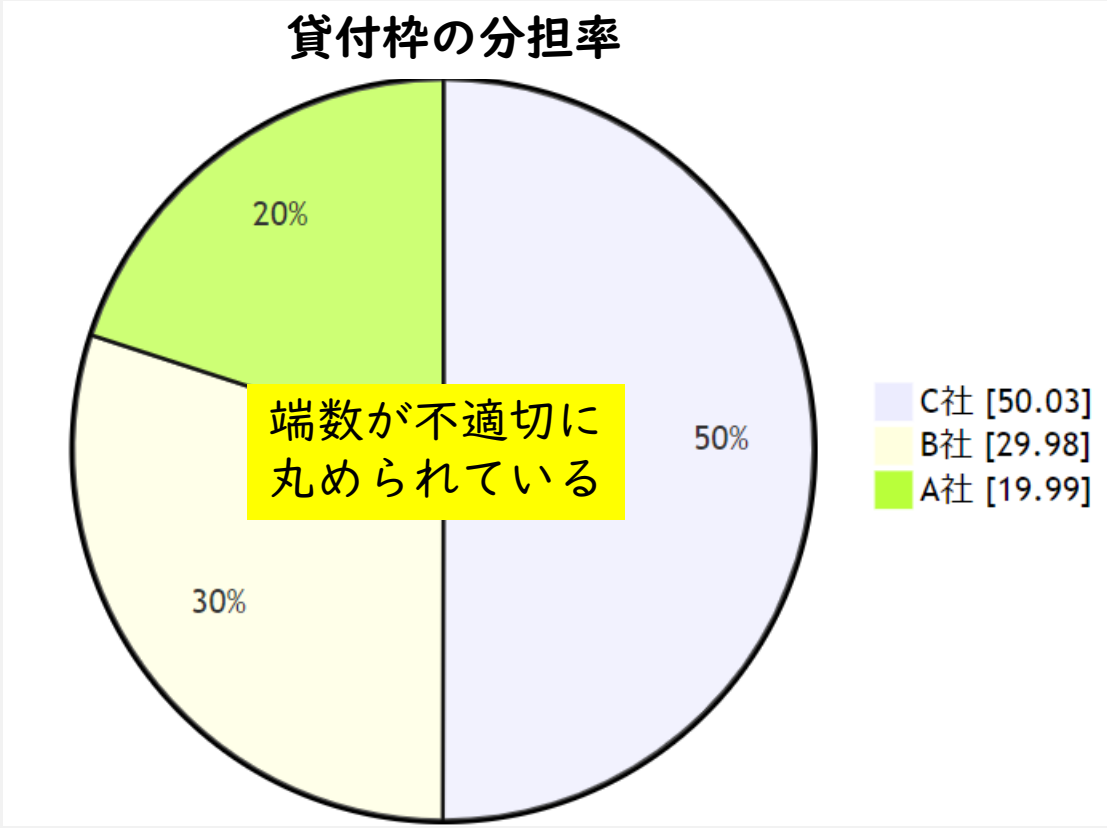
a. 貸付枠の分担率を決める（率の合計＝100%）

b. 分担率に応じて、貸付額、元金返済額、受け取り手数料、受け取り利息を比例配分する

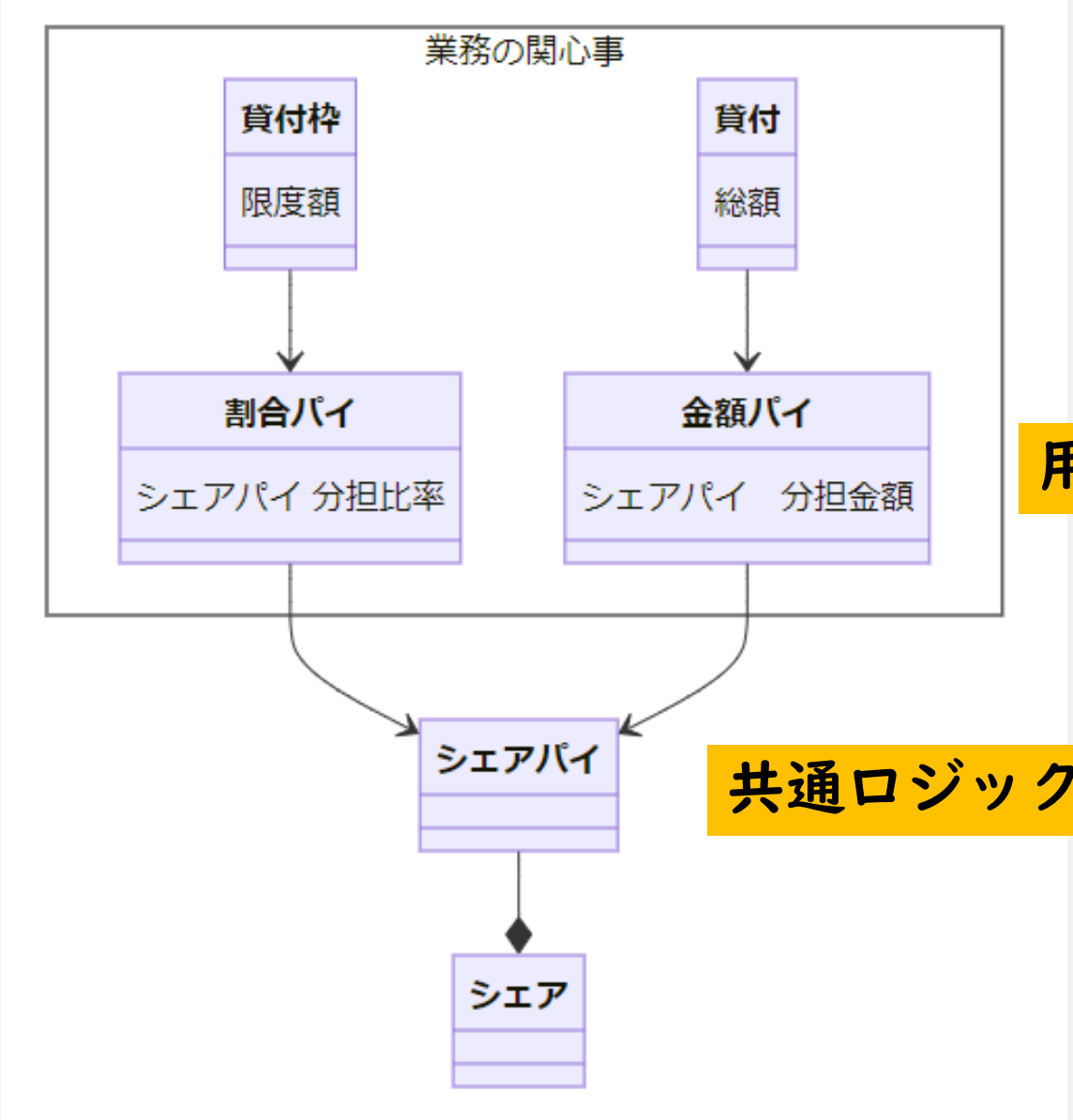
c. 端数を厳密に処理し、比例配分後の合計を配分対象額と必ず一致させる

d. 様々な配分対象に対する配分計算と端数処理の重複記述と不整合を防ぐ

参考：『ドメイン駆動設計』8章



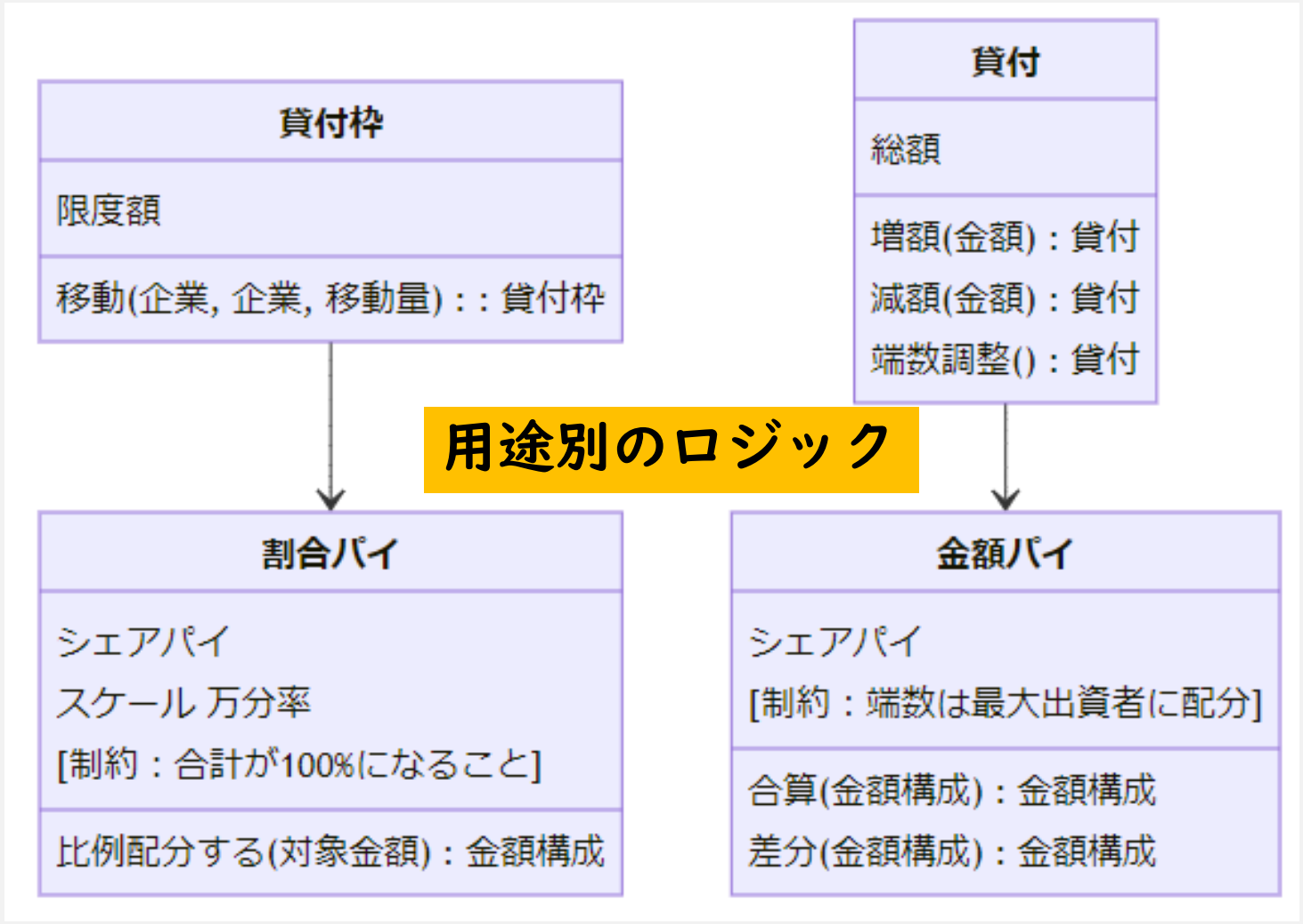
分担比率と配分（モデル）

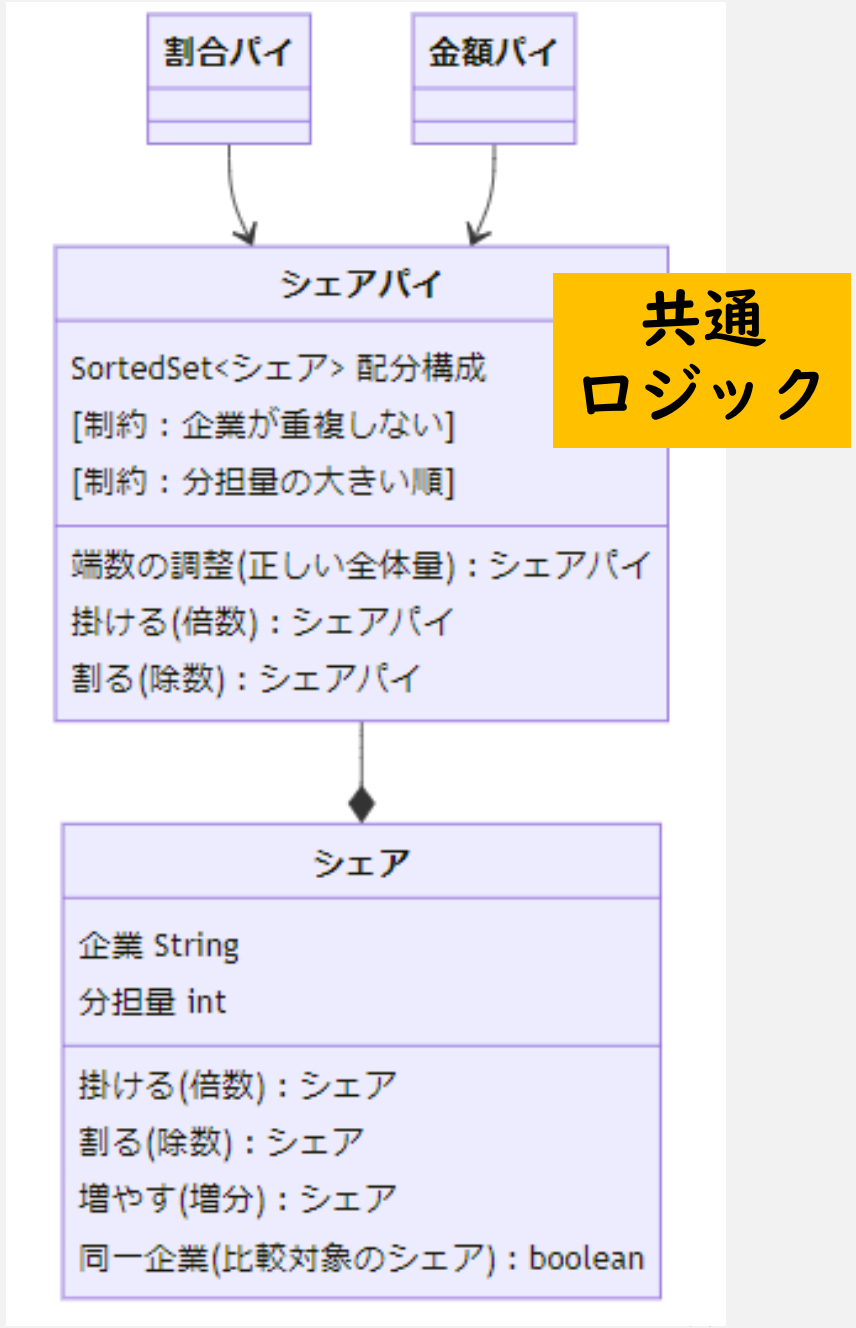


用途別のロジック

共通ロジック（計算の一貫性）

業務の関心事と計算の一貫性





比例配分の意図を表現するクラス

```
class 割合パイ {
  シェアパイ 構成比;
  static final ScaleType 尺度 = ScaleType.万分率;

  金額パイ 比例配分(対象金額) {
    シェアパイ 単純配分_金額ベース =
                構成比.掛ける(対象金額) // スケールなしの整数
                      .割る(尺度.スケール定数); // 10,000で割る（端数は切捨て）

    シェアパイ 端数調整済_金額ベース = 単純配分_金額ベース
                          .端数を最大分担者に割り当てて調整(対象金額);

    return 金額パイ.of(端数調整済_金額ベース);
  }

}
```

比例配分の詳細を実装するクラス

```
class シェアパイ {
  final SortedSet<シェア> 分担割合;

  private Collection<シェア> 端数調整(int 端数金額) {

    シェア 最大分担者の現在の分担内容 = 分担割合.first(); // 大きい順の先頭
    シェア 最大分担者の端数調整後の分担内容 = 最大分担者の現在の分担内容.増やす(端数金額);

    Set<シェア> 調整用の分担割合 = new HashSet<>(分担割合); // 作業用の可変Set

    調整用の分担構成.remove(最大分担者の現在の分担内容);
    調整用の分担構成.add(最大分担者の端数調整後の分担内容);

    return Collections.unmodifiableSet(調整用の分担割合); // 不変
  }
}
```

設計ノート　最大分担者に端数を配分する単純なルールの例

⑨複合条件

デシジョンテーブルのコード表現
if, &&, ||, ! の代わりにPredicate型を使う

複合条件

例： 貨物とコンテナの積み込みルール

a. 貨物の特性（爆発性、揮発性、一般）によって、その貨物を積載できるコンテナ種類が異なる
b. コンテナ種類には、標準型、強化型、換気装置付き、強化型かつ換気装置付きがある
c. 爆発性と揮発性の貨物は運賃を高く設定できる
d. 特殊なコンテナな必要としない一般貨物を特殊なコンテナに積んでしまうと、機会損失になる

参考：『ドメイン駆動設計』９章、１０章

20年前には、独自に述語論理を(AND, OR, NOT)を実装していた

現在は、JavaのPredicateを使ってシンプルに実装できる

デシジョンテーブル

パターン	1	2	3	4	5	6	7	8	9	10	11	12
条件												
貨物の特性	爆発性	爆発性	爆発性	爆発性	揮発性	揮発性	揮発性	揮発性	一般	一般	一般	一般
コンテナの機能	強化	強化かつ換気	換気	標準	強化	強化かつ換気	換気	標準	強化	強化かつ換気	換気	標準
判定												
積み込み可能	○	○	×	×	×	○	○	×	×	×	×	○

Enumを使って条件を定義

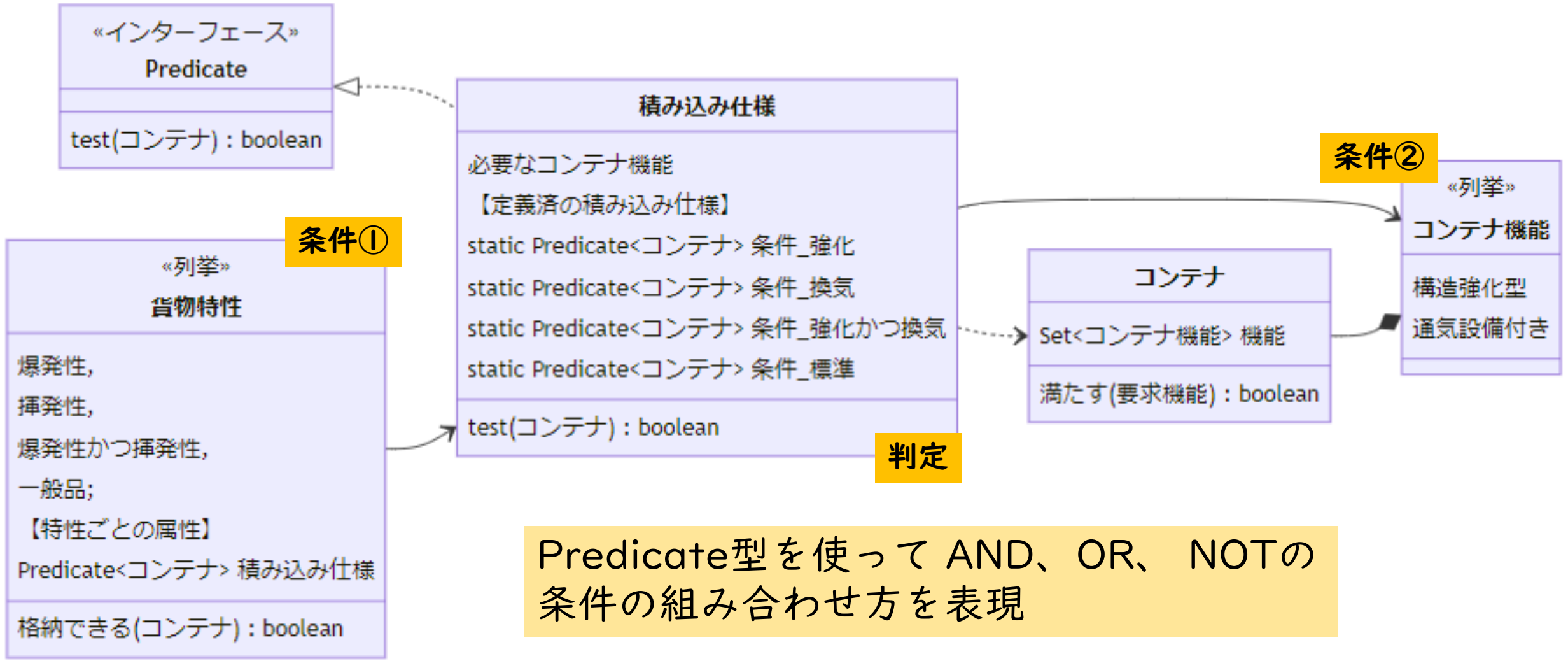
条件①

```
enum 貨物特性{
    爆発性(コンテナ条件_強化),
    揮発性(コンテナ条件_換気),
    爆発性かつ揮発性(コンテナ条件_強化かつ換気),
    一般品(コンテナ条件_標準);

    final Predicate<コンテナ> コンテナ条件;

    CargoType(Predicate<コンテナ> コンテナ条件) {
        this.コンテナ条件 =コンテナ条件;
    }

    boolean 格納できる(Container コンテナ) {
        return コンテナ条件.test(コンテナ); // 判定
    }
}
```

条件②

```
enum  コンテナ機能  {
        構造強化型,
        通気設備付き
}
```

条件を組み合わせた積み込み判定（モデル）



Predicate型を使って AND、OR、 NOTの条件の組み合わせ方を表現

Predicate型を使った条件の組み合わせ

```
class コンテナ条件 implements Predicate<コンテナ> {

  ContainerFeature 必要なコンテナ機能;

  // 定義済のコンテナ条件
  static Predicate<コンテナ> コンテナ条件_強化 = new 積み込み仕様(構造強化型);
  static Predicate<コンテナ> コンテナ条件_換気 = new 積み込み仕様(通気設備付き);
  static Predicate<コンテナ> コンテナ条件_強化かつ換気 =
                        コンテナ条件_強化.and(コンテナ条件_換気);
  static Predicate<コンテナ> コンテナ条件_標準 =
                  (コンテナ条件_強化.negate()).and(コンテナ条件_換気.negate());
  @Override
  public boolean test(Container コンテナ) {
    return コンテナ.満たす(必要なコンテナ機能);
  }
}
```

Predicate型のメソッドで条件を組み合わせる

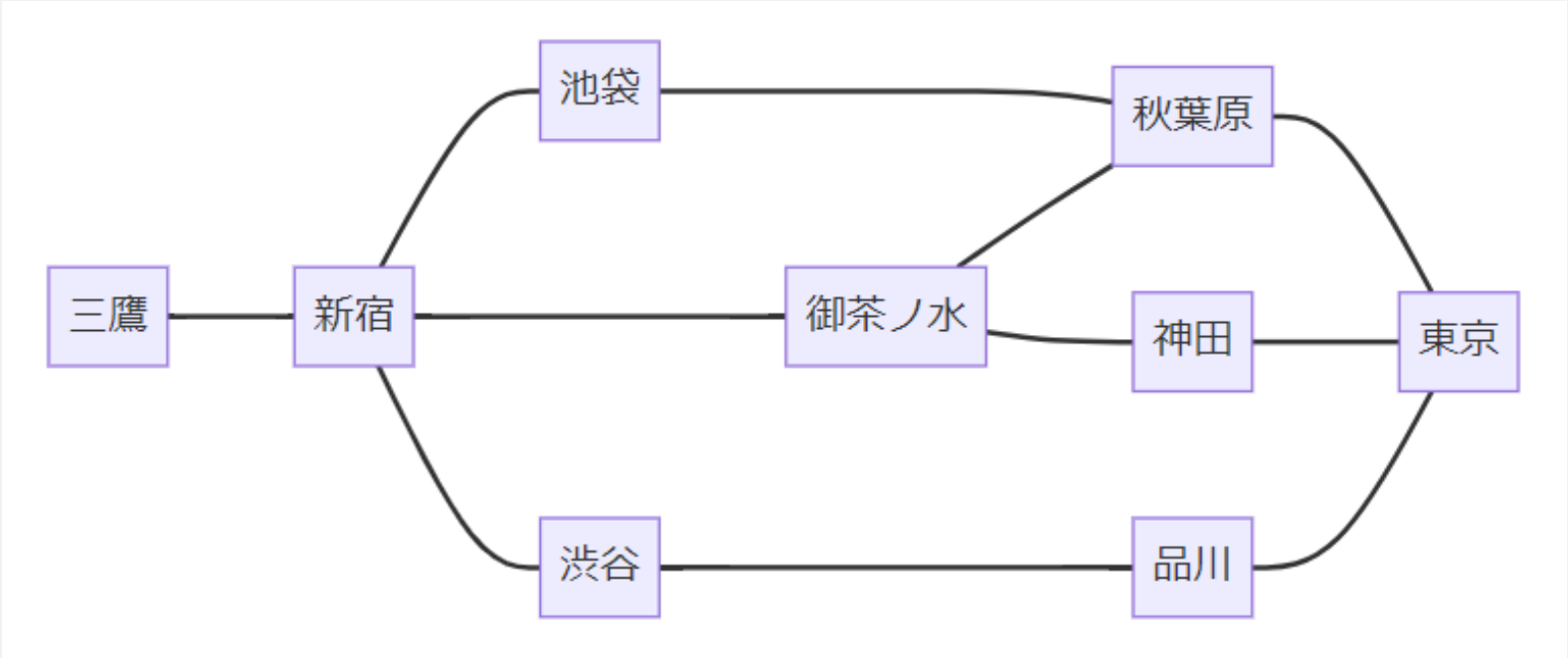
⑩経路探索

ネットワーク構造の表現と計算

経路探索

こういうネットワーク構造で扱える課題はいろいろある

経路、状態遷移、
業務フロー、工程グラフ



東京からもっとも遠い地点はどこか？
接続数がもっとも多い地点はどこか？

人間だったら、簡単に発見できるが…

①経路の表現：隣接ペアを定義し隣接リストに変換

```
Set<Path> 隣接ペア = Set.of(
  // 中央線
  new Path(東京, 神田),
  new Path(秋葉原, 御茶ノ水),
  new Path(神田, 御茶ノ水),
  new Path(御茶ノ水, 新宿),
  new Path(新宿, 三鷹),
  // 山手線 内回り
  new Path(東京, 秋葉原),
  new Path(秋葉原, 池袋),
  new Path(池袋, 新宿),
  // 山手線 外回り
  new Path(東京, 品川),
  new Path(品川, 渋谷),
  new Path(渋谷, 新宿)
);
```

変換

ネットワーク構造を扱う定石

```
Map<Place, List<Place>> 隣接リスト

Map.of(
   東京, List.of(神田, 秋葉原, 品川),
   …
);
```

設計ノート

隣接ペアはデータ構造が単純でテーブルなどで表現しやすい

しかし、プログラムで操作するには隣接リストのほうが扱いやすい

②経路の探索ロジックの例

```
private void 幅優先で探索して各地点への距離を計算する(Place 出発地) {

  Queue<Place> 探索地点のキュー = new LinkedList<>();
  探索地点のキュー.add(出発地); // 東京

  while (!探索地点のキュー.isEmpty()) {
    Place 探索地点 = 探索地点のキュー.remove(); // キューの先頭を取り出す（東京）
    探索地点に隣接する未探索地点のリスト(探索地点) // 東京から[神田, 秋葉原, 品川]
        .forEach(隣接地点 -> {
          出発地からの距離のマップ.距離を更新(探索地点, 隣接地点); // 東京から神田の距離
          探索地点のキュー.add(隣接地点); // 神田から先を探索
        });
  }
}
```

隣接リストを使った計算例①

```
@Test
void 東京からもっとも遠い地点は三鷹 () {
  Place 出発地点 = new Place("東京");
  PathLengthMap 各地点までの距離のマップ = 隣接リスト.経路マップ(出発地点);

  PathWithDistance 期待値 = new PathWithDistance(new Path(東京, 三鷹), 4);
  assertEquals(期待値, 各地点までの距離のマップ.出発地点から最も遠い地点と距離());
}
```

隣接リストを使った計算例②

```
@Test
void 接続数がもっとも多い地点は新宿() {
  Map<Integer, List<Place>> 接続数別グルーピング = 隣接リスト.接続数別グルーピング();

  List<Place> 接続数が最も多い地点のリスト = 接続数別グルーピング.entrySet().stream()
      .max(comparingInt(Map.Entry::getKey))
      .orElseThrow().getValue();

  assertTrue(接続数が最も多い地点のリスト.contains(新宿)
                  && 接続数が最も多い地点のリスト.size() == 1);
}
```

最後に

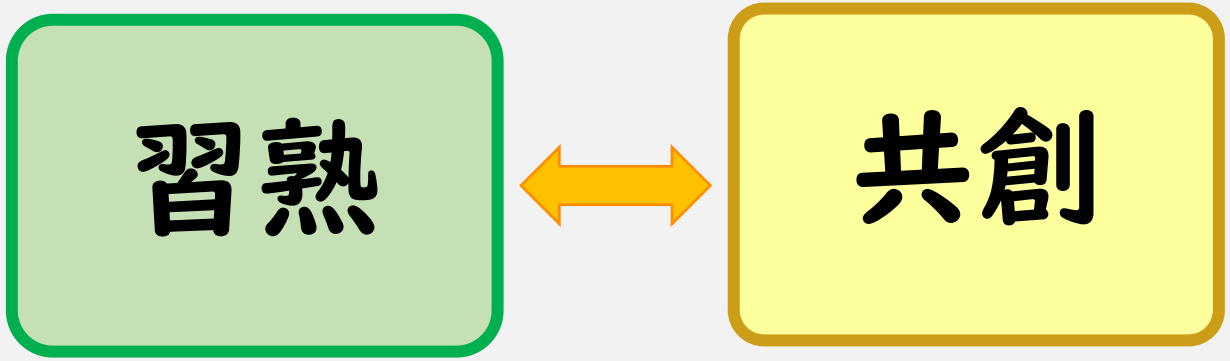
分析設計パターンの活かし方

暗黙的な経験則
技能⇒知識
成功体験
失敗体験

言語化/可視化された経験則
設計原則（文脈に依存しない一般化）
設計パターン（文脈の類型化）
体験談（個別の文脈）

今日紹介した内容

制約の多い実際の文脈で
手を動かして
体で覚える
知識⇒技能



設計の知識と技能を
持ち寄って
組み合わせる

相乗効果（三つの掛け算）　足し算ではなく掛け算

ソースコードは、以下のリポジトリで公開しています

https://github.com/**masuda220**/**business-logic-patterns**

src以下の domain/model/jjugccc2024

ワークショップや実プロジェクトで**共創**しましょう