初版:2022/2/3
改訂:2022/8/3

設計の考え方とやり方

2022年8月3日

有限会社システム設計　増田 亨

自己紹介

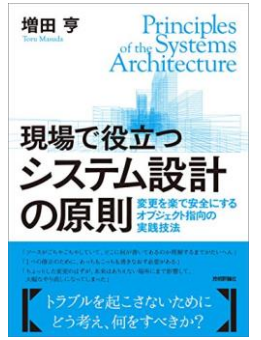
アプリケーション開発者

Java/Springを使った業務系アプリケーション開発

「ドメイン駆動設計」の開発現場への導入・実践

著書『現場で役立つシステム設計の原則』

～変更を楽で安全にするオブジェクト指向の実践技法



技術者コミュニティ「現場から学ぶモデル駆動の設計」主催

お話すること

良い設計を目指す

設計スタイルの選択

- クラス設計
- テーブル設計
- 開発のやり方

設計スキルを身につける

良い設計を目指す

良い設計は悪い設計より

変更が楽で安全である

ソフトウェアを変更する理由（複合）

- 市場の競合関係（競争優位・**競争劣位**）の変化に対応する
- 事業の環境（社会経済や消費行動）の変化に適応する
- 事業の遂行能力（組織・人材・仕事のやり方）を変える
- 事業の方針を変える

- 開発者の事業活動の理解が変わる
- 開発者の設計能力が変わる
- 利用できる技術の費用対効果が変わる（メモリ、帯域、…）

ソフトウェアの変更が楽で安全であれば

- 事業活動の変化のスピードを上げられる
- 事業活動の変化のコストを下げられる

- 開発者の学びと成長をソフトウェアに反映できる
- 開発者の成長の機会を増やせる
- 費用対効果の高い技術に移行できる

ソフトウェアの変更がやっかいで危険になると

- 事業活動の**変化のスピード**を落とす
- 事業活動の**変化のコスト**が増える

- 開発者の**成長の機会**が減る
- 開発者の**学びと成長**をソフトウェア開発に活かせない
- **費用対効果の悪い技術**を使い続ける

変更が楽で安全になる設計
それが開発者がやるべき仕事

私が実践している設計の考え方とやり方は、この価値観に基づいている

ソフトウェアの変更を楽で安全にするための

設計スタイルの選択

クラス設計のスタイル

テーブル設計のスタイル

開発のやり方

アプリケーション開発の今昔

かつての潮流

- トランザクションスクリプト方式
- 上書き更新型のデータベース
- 目標固定の分解思考（ウォーターフォール）

手続き的なプログラミング
SELECT FOR UPDATE
エクセル仕様書

今後の潮流

- ドメインモデル方式
- 追記型のデータベース
- 目標可変の組み立て思考（アジャイル）

オブジェクト指向プログラミング
(型・カプセル化・契約による設計)
イベントソーシング/マイクロサービス
IDE/git/CI-CD/コンテナ/XaaS

ソフトウェアの変更を楽で安全にするための

クラス設計のスタイル

クラス設計の分かれ道

トランザクションスクリプト方式

- データクラスと機能クラスを分ける
- 中核の関心事は入出力処理（画面・データベース・Web API）
- プリミティブな型でプログラミング
- 防御的プログラミング（

ドメインモデル方式

- ロジックをデータを一つのクラスにカプセル化する
- 中核の関心事は計算判断ロジック（ビジネスルール）
- アプリケーションで扱う値を独自の型として定義
- 契約プログラミング

なぜドメインモデル方式か？

ソフトウェアの複雑になる主な理由は、ビジネスルールに基づく計算判断ロジックの記述が複雑だから

計算判断ロジックの複雑さをどう扱うか？

トランザクションスクリプト方式の問題

- あちこちのデータ入出力処理に**計算判断ロジックが断片化し重複**する
- ビジネスルールの**暗号化**　例えば　if (区分 == 9) 処理数 = -1;
- どこにどんな計算判断ロジックが書いてあるか**探しにくい**
- ちょっとした計算判断ロジックの変更が**やっかいで危険**になる

ドメインモデル方式

計算判断ロジック中心（ビジネスルール中心）

関連する業務ロジックと業務データを一つのクラスにカプセル化

- 同じデータを使った計算判断ロジックがあちこちに断片化しない／重複しない

クラス名（型名）とメソッド名で業務の約束事を表現

- ビジネスで扱うデータの種類ごとにクラスを用意する
- データの種類ごとに必要な計算判断を洗い出して名前をつける(クラス名・メソッド名)
- 契約プログラミング：型を使って事前条件(引数の型）と事後条件（返す型）を表明

クラス設計：複雑さを分離する

ビジネスルールクラスの設計（ドメイン層）

- ✓ 事業活動の決め事（ビジネスルール）を
- ✓ 値の種類／区分定義に注目して
- ✓ 宣言的に記述

ビジネスアクションクラスの設計(アプリケーション層)

- ✓ 計算判断の実行（ドメインオブジェクトを使った計算判断の実行）
- ✓ 記録・参照の実行
- ✓ 通知・依頼の実行

ドメインモデル方式で
アプリケーション全体を
どう組み立てるか？

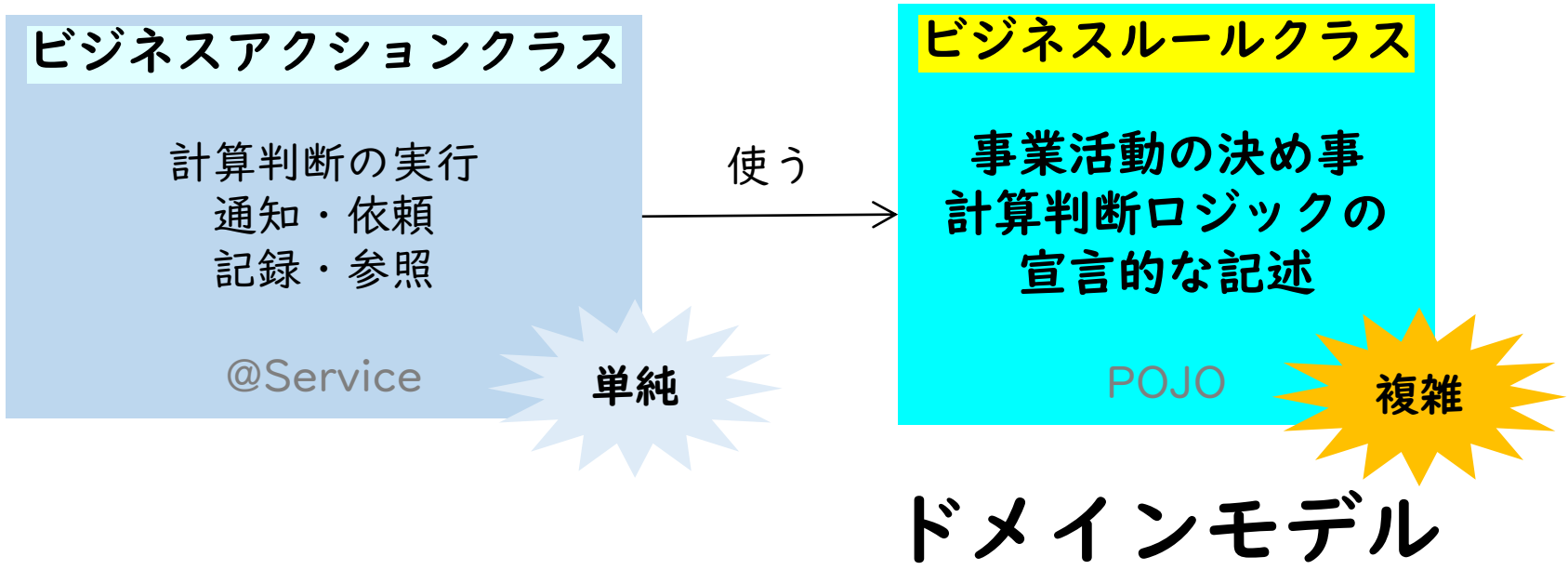
ビジネスルールクラス

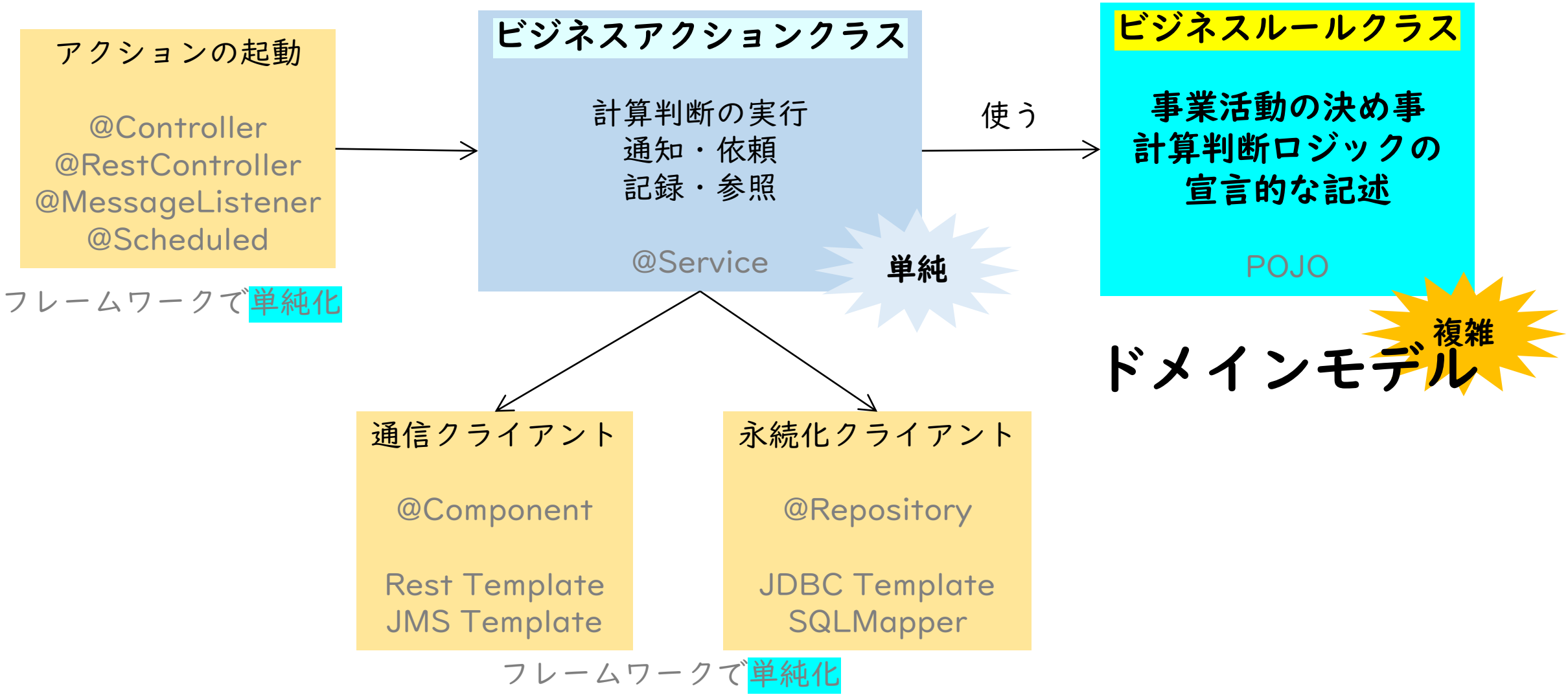
事業活動の決め事
計算判断ロジックの
宣言的な記述

POJO

複雑

ドメインモデル


ビジネスアクションクラス
計算判断の実行
通知・依頼
記録・参照
@Service
単純
使う
ビジネスルールクラス
事業活動の決め事
計算判断ロジックの
宣言的な記述
POJO
複雑
ドメインモデル


アクションの起動
@Controller
@RestController
@MessageListener
@Scheduled
フレームワークで単純化
ビジネスアクションクラス
計算判断の実行
通知・依頼
記録・参照
@Service
単純
使う
ビジネスルールクラス
事業活動の決め事
計算判断ロジックの
宣言的な記述
POJO
複雑
ドメインモデル
通信クライアント
@Component
Rest Template
JMS Template
永続化クライアント
@Repository
JDBC Template
SQLMapper
フレームワークで単純化

クラス設計を改善する（リファクタリング）

分割不足（大きなクラス＝低凝集）を改善する

- ✓メソッドの抽出（と名前づけ）→関心事の分離の地ならし
- ✓クラスの抽出（と名前づけ）
- ✓クラス数が増えるのでパッケージの追加/サブパッケージの追加（と名前づけ）

分割意図を説明する

- ✓パッケージ名を工夫する
- ✓クラス名を工夫する

分割した要素（クラス・パッケージ）の凝集度をさらに改善する

- ✓クラスの移動
- ✓メソッドの移動
- ✓インスタンス変数の移動
- ✓パッケージ/サブパッケージの移動（パッケージ構造の組み替え）

ソフトウェアの変更を楽で安全にするための

テーブル設計のスタイル

テーブル設計の分かれ道

ミュータブル（変更可能）なデータモデルでテーブル設計

- レコードの上書き更新可能（SELECT FOR UPDTE …; UPDATE … ; COMMIT ;)
- NULL 許容（後でUPDATEするカラム)
- 削除フラグ（UPDATE)
- 更新プログラムID・更新タイムスタンプ

イミュータブル（変更不可）なデータモデルでテーブル設計

- 事実の記録を徹底 INSERT only
- 予定した・変更した・完了した、これらも「事実の発生」として記録
- 事実の記録だけあれば（原理的には）状態は導出可能
- キャッシュやビューとして状態テーブルを追加することはある

イミュータブルデータモデルを選ぶ

- 事実の記録を徹底する
- 事実を消さない（上書き変更しない）
- 事実の記録があれば状態は動的に導出できる

- 上書き更新（UPDATE）は、データ操作が複雑になり、状態判定ロジックなどSQL/プログラムが複雑になる

イミュータブルデータモデルの効果

挙動が安定する

- 同じ条件で参照すれば、必ず同じデータを取得できる
- 書込みが単純になる（INSERT ONLY)
- 読み取りも単純になる（必要な事実だけ取得、WHERE句の単純化）

ドメインモデル方式と相性がよい

- ビジネスルールは、発生した事実を使った計算判断
- 同じ事実からは必ず同じ結果になるのがビジネスルール

分散システムと相性がよい

- 同じ事実（不変なデータ）をあちこちで複製しても不整合は起きない
- 上書き更新イベントを分散システムに正確に伝播するのは難しい

イミュータブルに設計したテーブル

データベース制約の徹底

- 外部キー制約
- 一意制約
- NOT NULL 制約

テーブルのカラム数が減る

- ひとつのテーブルには発生時点が同じカラムだけ
- 必須の情報だけ（任意項目は別テーブル）
- **状態を上書き記録するカラム**がなくなる
- **更新プログラムID**と**更新タイムスタンプ**がなくなる

プログラムが単純かつ明快になる

SQLが単純になる

- カラム数が少ない
- NOT NULLが保証されている
- 用途別・状態別にテーブルが分かれている

データベース操作プログラムが単純になる

- ドメインオブジェクトと事実を記録したテーブルのマッピングが単純になる

テーブル更新(状態変化）を前提にした複雑な記述がなくなる

- 状態による WHERE 句、CASE式、if 文/switch文が激減する

ソフトウェアの変更を楽で安全にするための

開発のやり方

開発のやり方の分かれ道

目標の固定から出発する分解思考の開発のやり方

- 固定のゴールを目指して**タスク分解**・**工程分割**・**分業体制**を固定する
- 詳細で具体的な問題は、実際に作る**後半の工程にしわ寄せ**
- 後工程になるほど予実差異の検出を詳細で具体的にやる（**前工程はゆるい**）

目標の仮決めから出発する組み立て思考の開発のやり方

- とっとと作りはじめて、とっとと成長させる
- 実際に動くコードで事実を積み上げ認識を合わせながら進める
- 仮決めの目標との差異を検出しながら、目標を調整 ＆ 進め方を調整

組み立て思考の開発のやり方

目標は可変

- 構想（長い目で見た理想的な姿）を思い描く（方向性の認識合わせ）
- 構想は（作ってみて）知見が増えれば変化する
- 構想は状況の見通しが変われば変化する

構想（目標）の変化を前提にソフトウェアを作る

- 変更しやすい構造を選択する
- ソフトウェアを変える環境を整備する(プロセス/開発環境/運用環境)
- ソフトウェアを変え続ける開発のやり方に習熟する

変化しやすい構造を選択する

構造の分かれ道

- ツリー構造（トップダウン）
- ピラミッド構造（ボトムアップ）
- ネットワーク構造

変更が楽で安全なネットワーク構造

- ツリー構造やピラミッド構造は部分の**差し替えがやっかいで危険**
- ネットワーク構造は部分の差し替え・追加・切り離しが楽で安全

とっとと作る

とっとと作る準備

- IDE(統合開発環境）・リポジトリ・CI/CD・実行環境の整備と習熟
- 基本構造と要素技術の選択と習熟
- 設計スタイルの選択と習熟

コードファースト

- ラフスケッチ・箇条書き程度の情報からコードを書いて動かしてみる
- 課題を発見し、事実に基づいて対応アクションを起こす

自己文書化・設計の可視化

- プログラムを仕様書・設計書として書く（動くだけなら不要な内容の丁寧に記述する）
- コードから関連図や一覧表を自動生成
- コンパイラやインスペクションツールで文書品質を保証

ソフトウェアの変更を楽で安全にするための

設計スキルを身につける

設計スキル

設計スキルの実体

- 経験則の脳内データベース
- 超高速の脳内検索
- 目の前の課題と経験則との高度なマッチング（文脈の違いの吸収）

設計のスキルアップの方策

- データベースを拡充する（経験則を増やす）
- 検索速度をあげる（脳内キャッシュ、脳内パーティショニング）
- パターンマッチ能力をあげる（文脈差異に適応）

設計スキルアップの行動計画

経験則を増やす

- 実際に作ってみる（設計スタイル、要素技術、構造、ツール、…）
- 他人の経験則を知識として学ぶ
- 他人の経験則・目の前の課題・体験知から新たな経験則を導き出す

脳内キャッシュの最適化

- 使わないと期限切れでキャッシュから消える⇒適時リフレッシュする
- リフレッシュを繰り返した内容は長期記憶に書き込まれる
- さらに繰り返すと小脳にキャッシュされる（体が覚える）

設計スキルアップの行動計画(続き）

脳内の経験則データベースのパーティショニングを調整する

- 整理の軸・枠組みのバリエーションを絵にしてみる
- 体系だった説明の書籍の構造を鑑賞する
- 脳内データベースの構造（参照経路）が変わる

経験則と目の前の課題のマッチング精度を上げる

- 微妙なパターンアンマッチから学ぶ（「なにか変」違和感センサー）
- 他のパターンに変えてみて結果を評価（マッチ度の学習）
- 文脈（目的・状況・判断基準）に依存したマッチングを工夫する

まとめ

良い設計は悪い設計より

変更が楽で安全である

ソフトウェアの変更が楽で安全であれば

- 事業活動の変化のスピードを上げられる
- 事業活動の変化のコストを下げられる

- 開発者の学びと成長をソフトウェアに反映できる
- 開発者の成長の機会を増やせる
- 費用対効果の高い技術に移行できる

変更を楽で安全にする設計
それが開発者がやるべき仕事

私が実践している設計の考え方とやり方は、この価値観に基づいている

補足資料

どうやっているか

クラス設計の基本パターン

設計のためのモデリング

複雑さとの戦い方

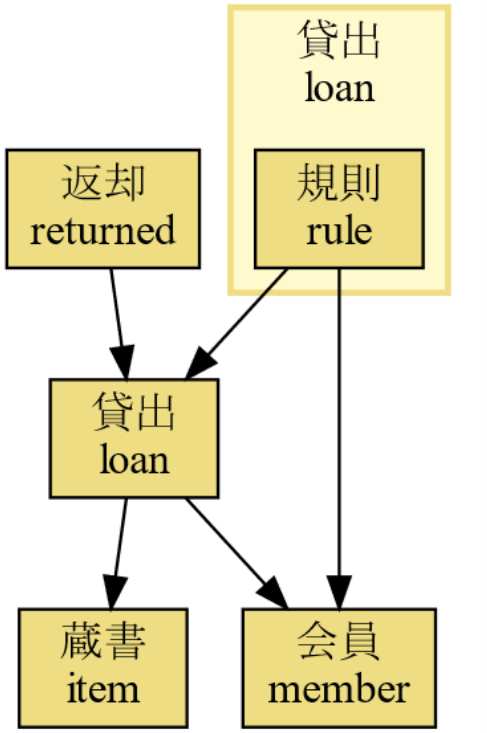
どうやっているか
・クラス設計
・テーブル設計
・自己文書化

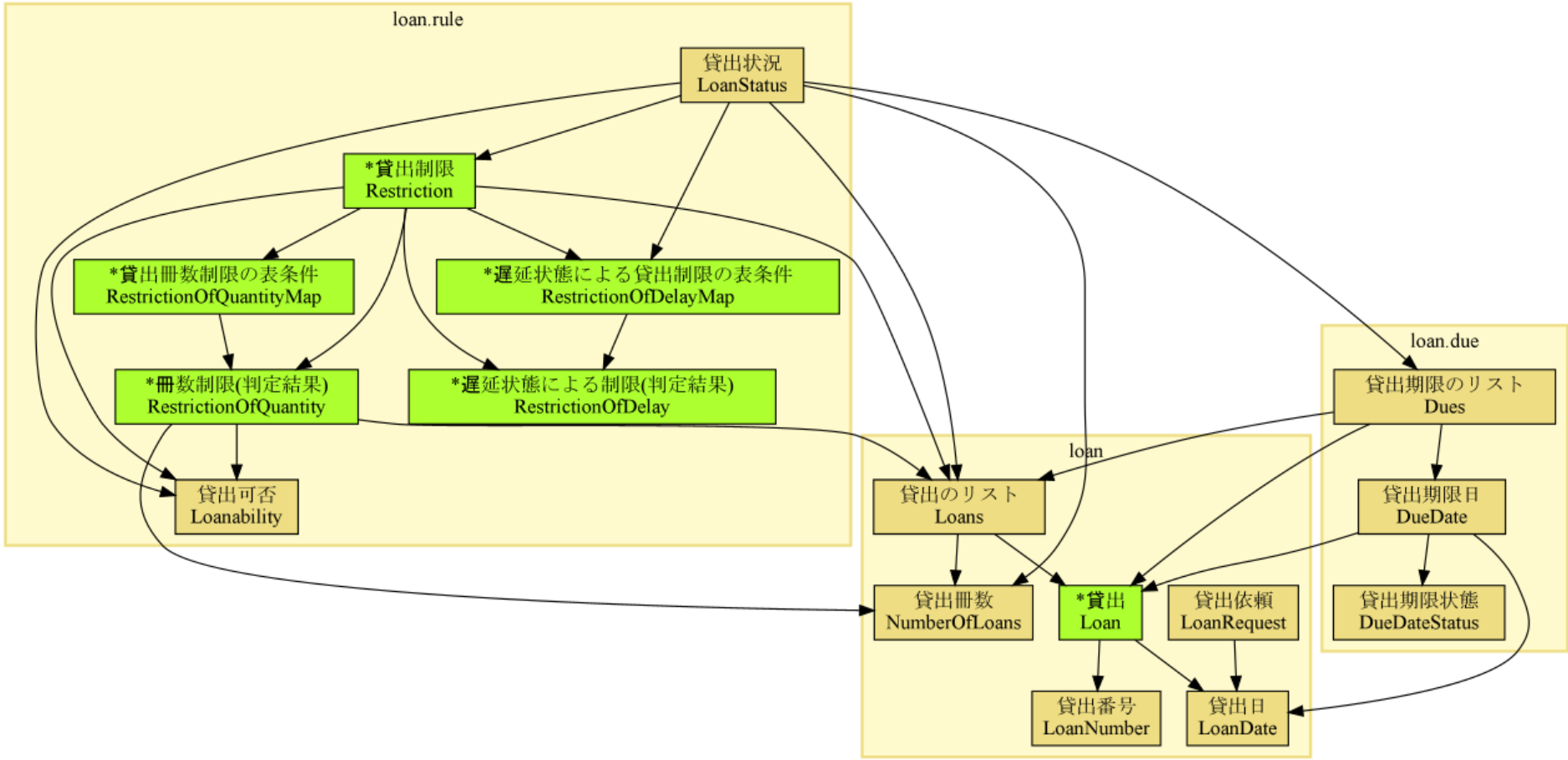
図書館の蔵書貸出アプリケーション

https://github.com/system-sekkei/library

JIGでソースコードから自動生成

ドメイン層のクラス設計

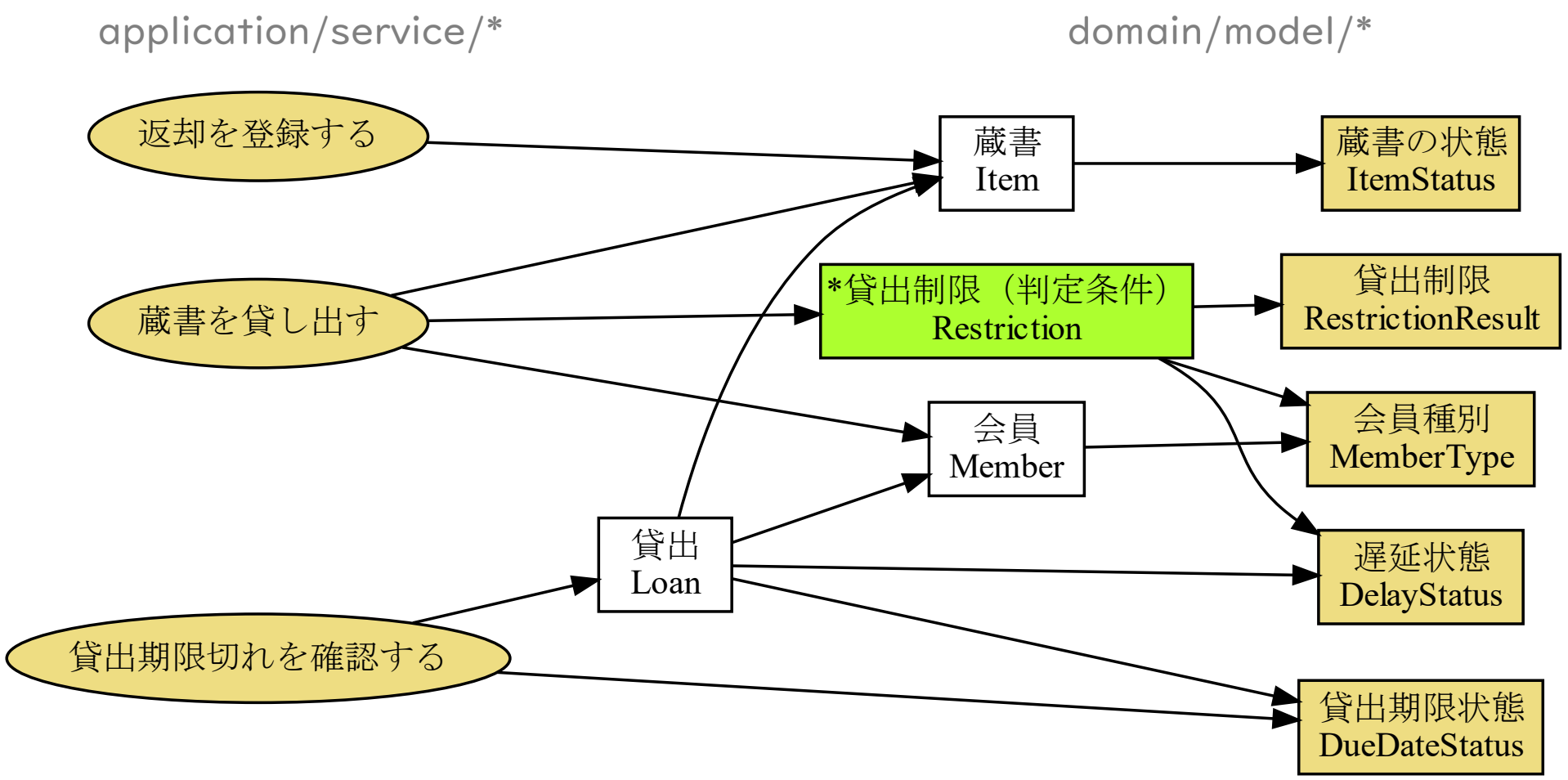




ビジネスルール関連図

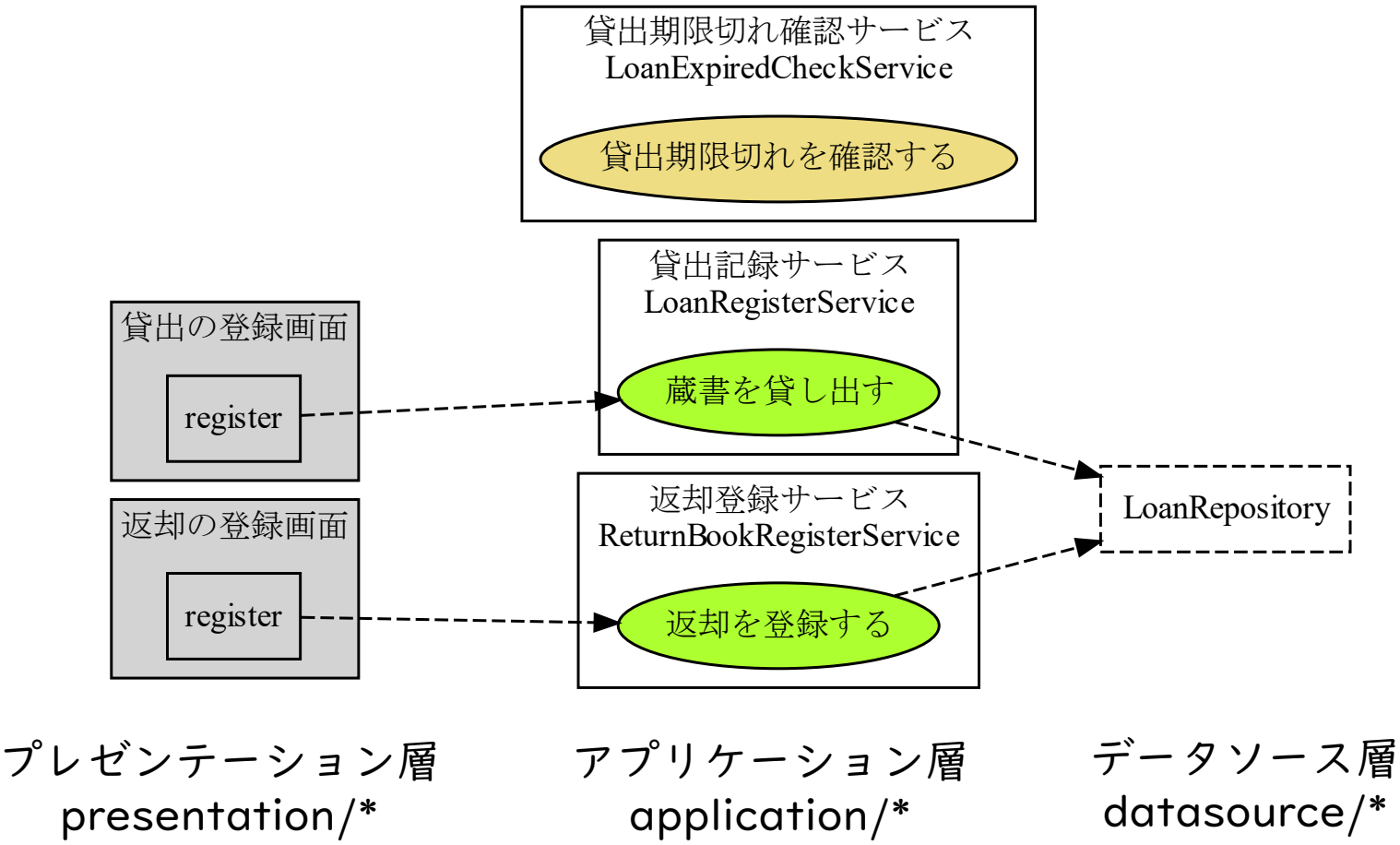
JIGでソースコードから自動生成

ソースコードから実装の中核を可視化



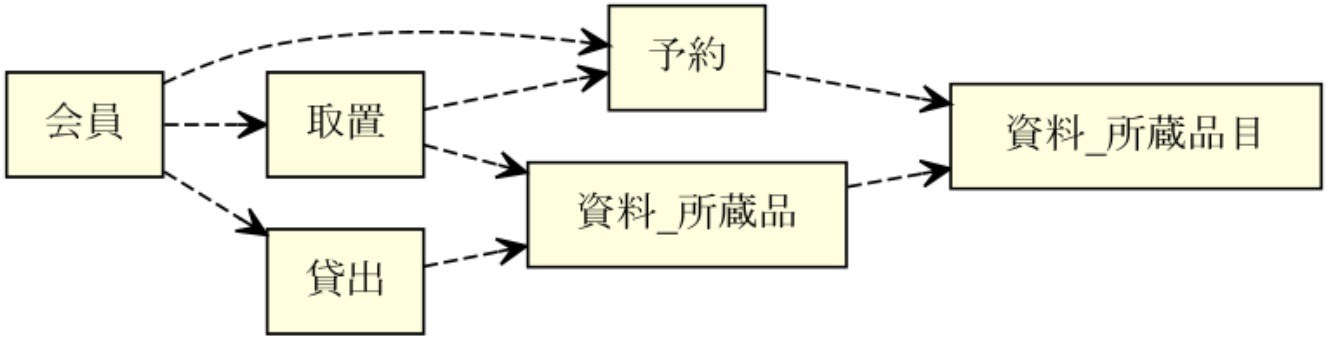
JIGでソースコードから自動生成

入出力構造の可視化

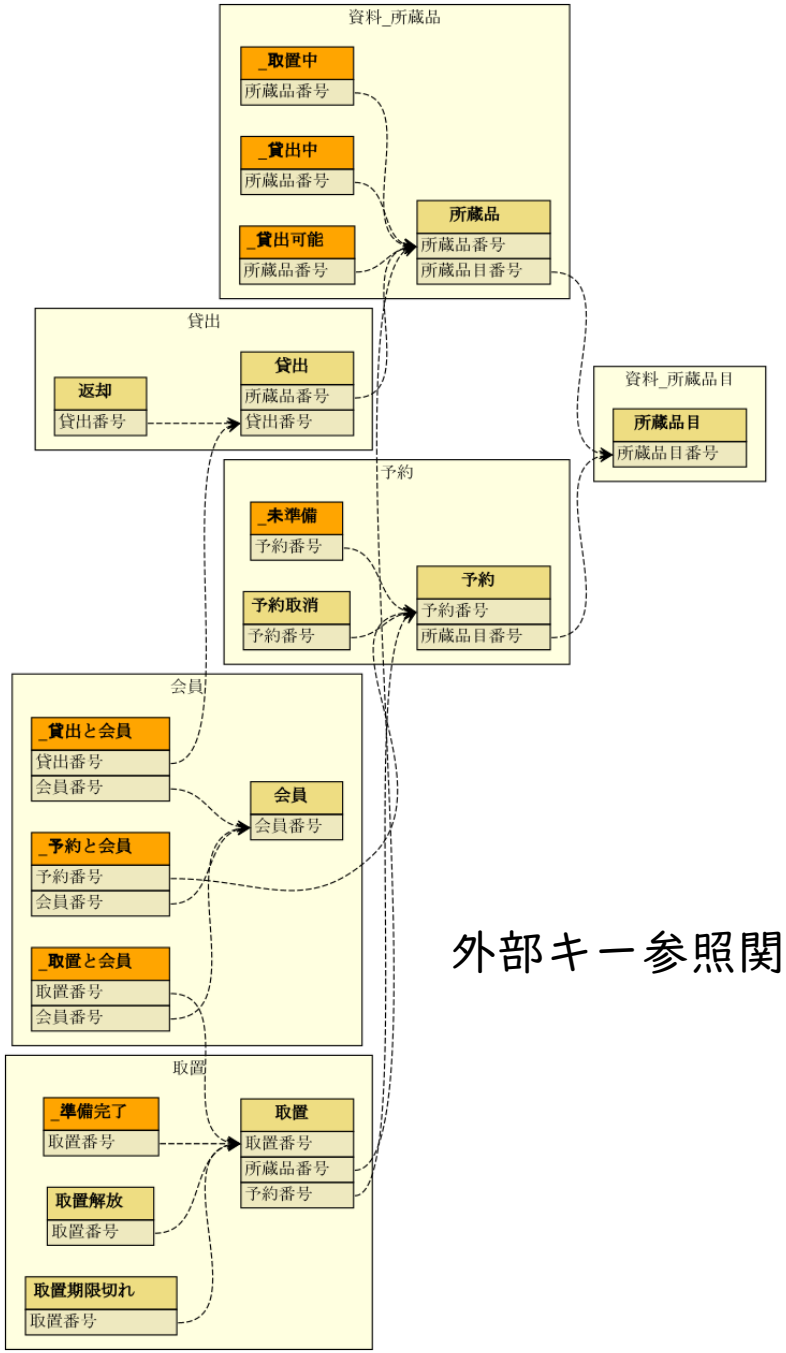


テーブル設計

JIG-ERDで自動生成



スキーマ構造



外部キー参照関係

ドメインモデル（ユビキタス言語）

JIGでソースコードから自動生成

ビジネスアクション（ユースケース）

JIGでソースコードから自動生成

クラス設計の基本パターン

値の種類で分割
対象領域の事実を扱う基本クラス

基本的な値を扱うクラス	**数量**	金額、単価、個数、人数、百分率、千分率
	日付・時刻	日付、日数、時刻、時間
区分を表す値を扱うクラス	**種類の違い**	商品種別、会員種類、料金区分、配送方法
	状態の違い	処理待・処理中・処理済、在庫有無、予約可否
範囲を扱うクラス	**数量の範囲**	価格帯（x円～y円)、数量範囲(x個～y個)
	日付・時間の範囲	期間（開始日～終了日）、時間帯（開始時刻～終了時刻）

プリミティブな計算式を隠蔽

区分を整理しロジックを集める

判断ロジックのカプセル化

対象領域で扱うこれらの値の種類ごとに、ビジネスルール（計算判断のロジック）と事実の表現（インスタンス変数）をクラスで定義する

クラスの設計：メソッドの集合として定義

事実を扱う計算判断ロジックの候補

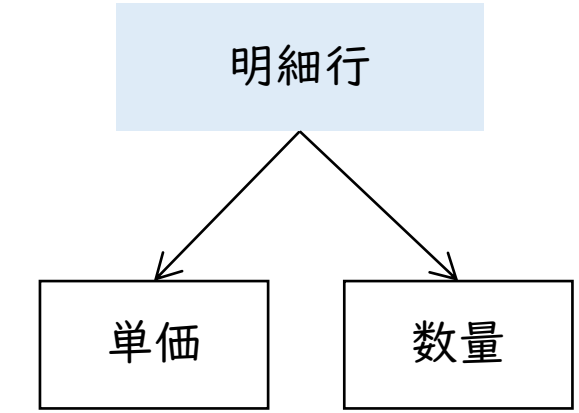
四則演算	**足し算、引き算**	add(), plus(), subtract(), minus()
	掛け算	multiply(), times()
	割り算、あまり	divide(), remainder()
比較演算	**等値**	equalsTo(), notEqualsTo()
	大小、前後	greaterThan(), lessThan() , isAfter(), isBefore()
境界	境界要素の取得	MAX, MIN
順序	前の値・次の値	previous(), next()
文字列形式	**文字列に変換**	toString(), show(), format()
	文字列から変換	from(), parse()

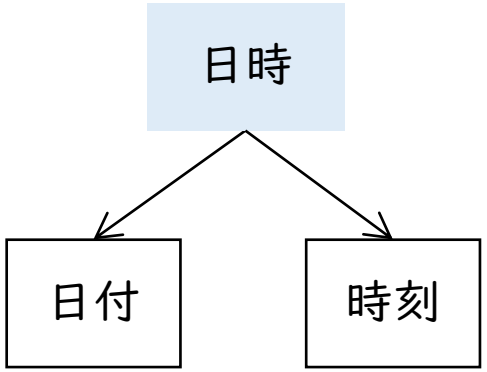
✓ 対象領域で関心のあるメソッドだけに絞り込むことで、クラスの意図（型の意味）が明確になる
✓ 引数の型やメソッドの返す型を目的特化・用途限定にするほど挙動が安定する（契約による設計）
（汎用的なライブラリクラスと設計の方向が逆）

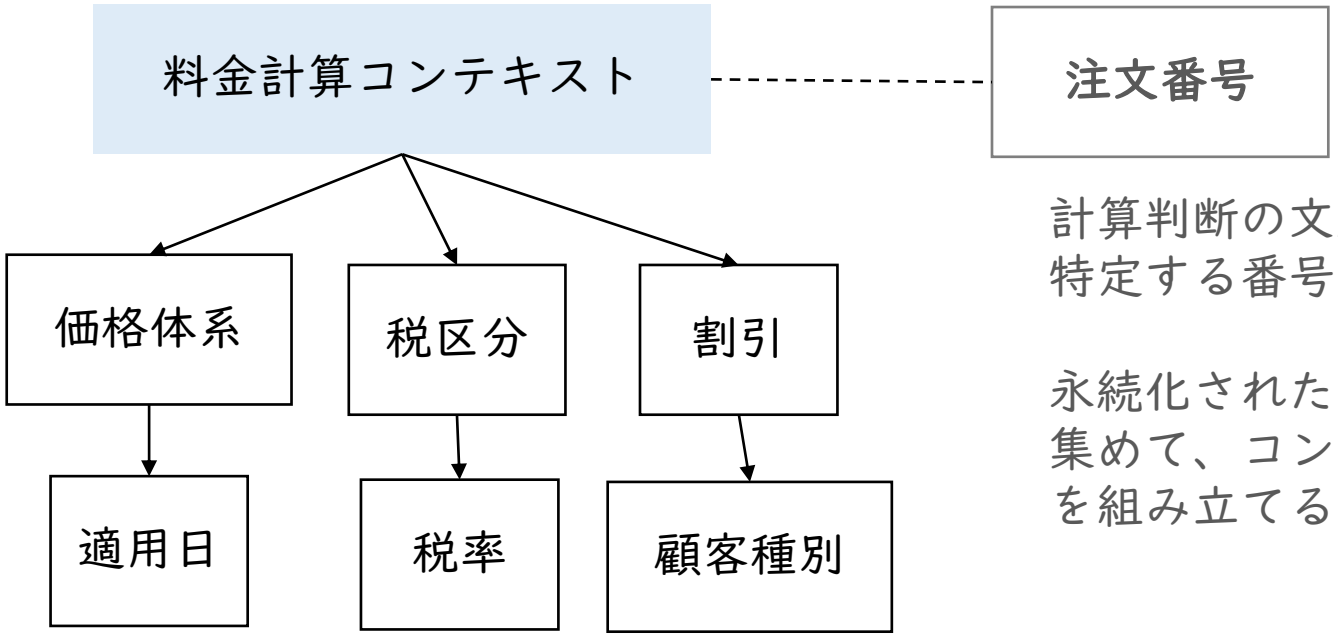
事前条件・事後条件

組み立て役のクラスの形

分割したクラスの組み合わせ方







計算判断の文脈を
特定する番号

永続化された事実を
集めて、コンテキスト
を組み立てるためのキー

組み立て役のクラスの形
コレクション操作をカプセル化

コレクションとその操作をカプセル化して独自のクラスを作る
操作の意図を公開し、操作の実装は隠蔽する

リスト操作をカプセル化

filter操作　　サブリストの抽出
map操作　　別の要素のコレクションに写像
reduce操作　合計、個数、最大、最小、…

商品一覧
メンバー一覧
注文一覧
…

セット(集合)操作をカプセル化

部分集合(subset)
和集合(union)
差集合(minus)
共通集合(intersect)

スキルセット
ルールセット

マップ(写像)操作をカプセル化

マップのマップから値の取り出し
逆写像（値からキーの特定）
写像の併合(merge)
共通写像(intersect)

表形式の計算ルールや判定ルール

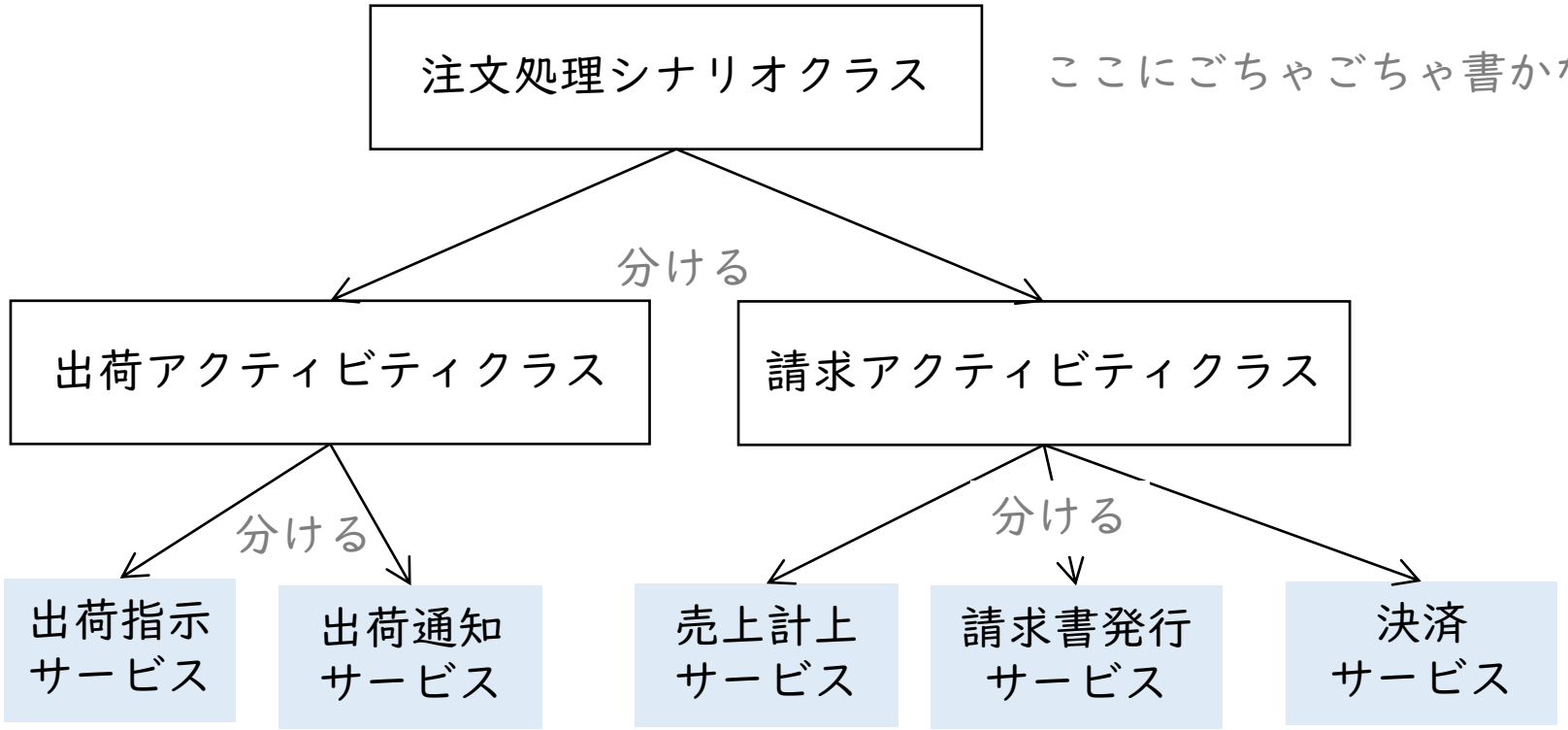
組み立て役のクラス：ビジネスアクション

アプリケーション層のクラスの分割と組み立ての形

業務全体の
流れの表現

関心事ごとの
活動体系を表現

通知・記録・参照の
基本アクションを表現

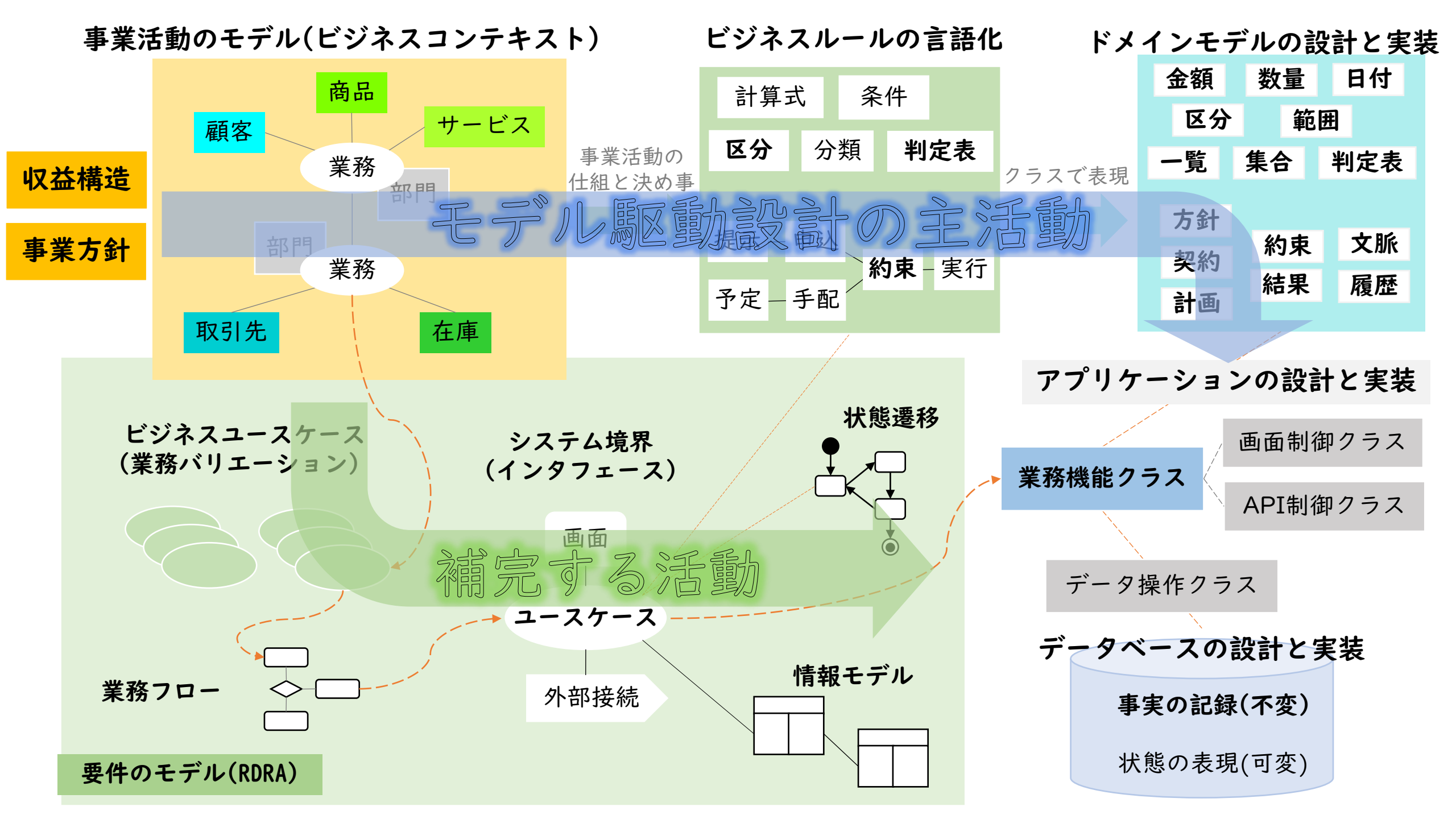


計算判断はビジネスルールクラスを利用する（ロジックをここに書かない）

事業活動を理解して設計するためのモデリングの基礎知識

設計のためのモデリング

機能一覧やユースケース一覧は
クラス設計のためのモデリングではない


事業活動のモデル(ビジネスコンテキスト)
ビジネスルールの言語化
ドメインモデルの設計と実装
収益構造
事業方針
商品
顧客
サービス
業務
部門
部門
業務
取引先
在庫
事業活動の
仕組と決め事
クラスで表現
モデル駆動設計の主活動
計算式
条件
区分
分類
判定表
予定
手配
約束
実行
金額
数量
日付
区分
範囲
一覧
集合
判定表
方針
契約
計画
約束
文脈
結果
履歴
アプリケーションの設計と実装
ビジネスユースケース
(業務バリエーション)
システム境界
(インタフェース)
状態遷移
画面
補完する活動
ユースケース
業務フロー
外部接続
情報モデル
業務機能クラス
画面制御クラス
API制御クラス
データ操作クラス
データベースの設計と実装
事実の記録(不変)
状態の表現(可変)
要件のモデル(RDRA)

事業活動を大局的に理解する（whyの理解）

- 事業活動は顧客との契約（受注）とその履行である
- 事業活動は協力企業との契約（発注）とその履行である
- 事業活動は競合との差別化行動である

- 事業活動を発展させるために金銭的利益が必要
- 事業活動を発展させるために顧客の成功が必要
- 事業活動を発展させるために仕事のやり方の改善が必要
- 事業活動を発展させるために個人と組織の学習と成長が必要

事業活動を時系列で整理する

- 事業活動を業務イベントの連鎖として捉える
  - 受注系：商流イベント・金流イベント・物流イベント
  - 発注系：商流イベント・金流イベント・物流イベント

- 事業活動には表舞台と舞台裏の二つの領域がある
  - 顧客が認知する表舞台の業務イベント
  - 顧客が関知しない舞台裏の業務イベント

ビジネスルールの発見とクラス設計

- 業務イベントには必ず業務上の決め事がある
  - 事前条件：業務イベントが発生してよい条件
  - 事後条件：業務イベントが発生した時の約束事

- 業務上の決め事はビジネスで一般的なパターンがある
- その決め事を表現するクラス設計も基本的なパターンがある
- それらの基本パターンを出発点にして、その事業の独自性・差別化の中核となる業務（コアドメイン）に焦点を合わせて、クラス設計に反映する

一般的な業務の関心事とクラスの候補

提供する価値の表現

物品(goods, product)
役務(service)
権利(right)
利用(usage)
移動(transport)
価格(pricing)
提供条件(policy, conditions)

販売機会

商品カタログ(catalogue)
引合(inquiry)
見積(estimate)
提示(offer)

約束と履行

契約(contract, order)
予約(reservation, booking)
値引(discount)
引渡(delivery)
請求支払(billing, payment)
進捗(progress, milestone)
キャンセル(cancel)

関係

顧客(customer, account)
関係(relationship)
連絡(contact)
伝達(communication)
通知(notification)

価値の提供能力

在庫(inventory)
提供能力(capacity)
提供可能性(availability)
手配(arrangement)
購入(purchase)
調達(procurement)

計画と実行

予定(schedule)
計画(plan)
行動(action)
進捗(progress, milestone)

事業活動とテーブル設計

- 業務イベント（事実の発生）を記録する
- 事実の記録は不変（イミュータブル:insert only）
- 変更・キャンセルなども事実の発生として記録

- 状態は、事実の履歴があれば動的に導出できる
  - 動的な状態の導出が性能面で実用に耐えない時は、キャッシュとして状態テーブルを利用することはある

複雑さとの戦い方

複雑さとの戦い方（１）

- 複雑さの主因は、金額・数量・日付の計算判断
- アプリケーションで扱う値を特定する
- 独自の型として定義（値オブジェクト）
- 値オブジェクトに計算ロジックをカプセル化する
  - 実装の詳細の隠蔽
  - 変更の影響の局所化
- ビジネスルールを記述する基本語彙を独自の型で表現
  - int型/String型でビジネスルールを記述しない

複雑さとの戦い方（２）

- 複雑さの主因は条件分岐
- 分岐する条件の特定と整理
  - 商品区分、顧客区分、物流区分、…
  - 提供能力（引当、手配）の状態区分（例：引当可、引当済、…）
  - 契約履行の進行状態（例：出荷待ち・出荷指示中・出荷済）

- 条件分岐の組合せを特定し整理するクラス設計
  - 区分オブジェクトの定義とリファクタリング
  - 条件分岐（if文/switch文）の記述とリファクタリング
  - (List)のフィルタリングと集約演算
  - 集合(Set)の演算
  - 写像(Map)による構造化