Modeling Forum 2018 技術公演 トラック

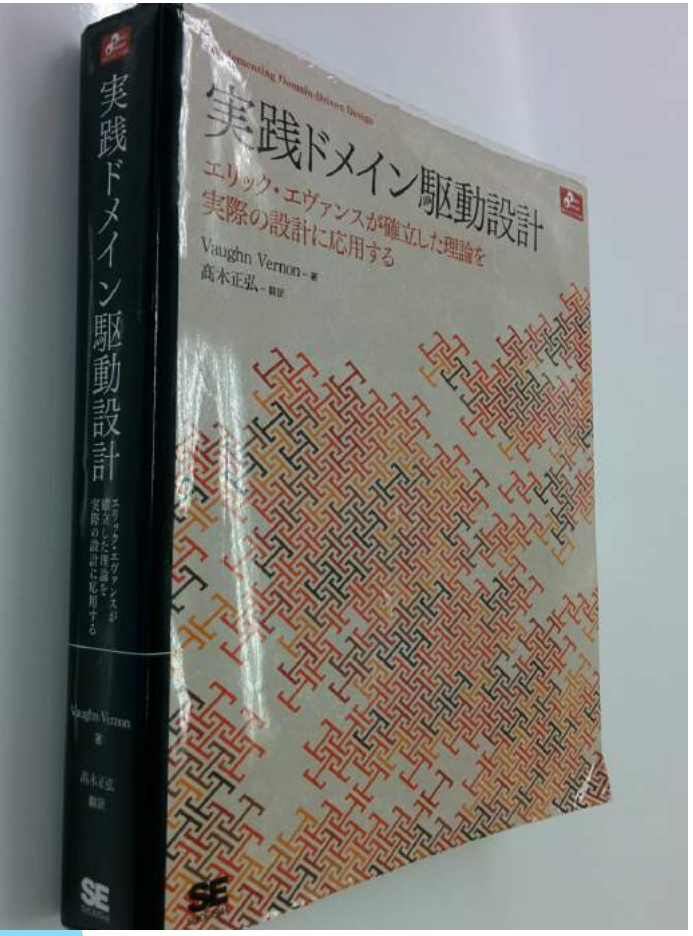
「実践ドメイン駆動設計」から理解するDDD

NEXTSCAPE
株式会社ネクストスケープ
青木 淳夫

DDD(ドメイン駆動設計)とは

- ドメイン駆動設計（DDD: Domain-Driven Design）とは2003年にEric Evans氏が提唱した「顧客と開発者が業務を戦略的に理解し、共通の言葉を使いながらシステムを発展させる手法」です。
- DDDの登場から15年以上が経ち、DDDは着実に普及しつつあります。
- 本セッションでは、2013年にVaughn Vernon氏が発表した書籍「実践ドメイン駆動設計（通称：IDDD）」の流れに沿って、DDDの基本からモデリング手法までを幅広く紹介します。

IDDD(実践ドメイン駆動設計)とは



原書名：略称（日本語翻訳名）	著者（翻訳・監修）	原書発売年（日本語版発売年）	内容	日本語版の総ページ数
Domain-Driven Design - Tackling Complexity in the Heart of Software：DDD本／エヴァンス本 **『エリック・エヴァンスのドメイン駆動設計』**	Eric Evans （和智右桂、牧野裕子訳 今関剛 監修）	2003年 （2011年）	パターン・ランゲージ形式によるDDDの理論	538P
Implementing Domain-Driven Design：IDDD本 **『実践ドメイン駆動設計』**	**Vaughn Vernon （高木正弘）**	**2013年 （2015年）**	**DDDの理論とJavaでの実践例**	**583P**

当セッションではCodeZine連載「IDDD本から理解するドメイン駆動設計」 (14回分)を50分で紹介します

- https://codezine.jp/article/corner/655 参照

CodeZine
開発者のための実装系Webマガジン

「IDDD本から理解するドメイン駆動設計」連載一覧

1〜13件(全13件)

2018/11/01
実践DDD本 第13章「境界づけられたコンテキストの統合」〜分散システム設計〜
ドメイン駆動設計（DDD）は、顧客と開発者がビジネスを戦略的に理解し、共通の言葉を用いてシステムを発展させていく設計手法です。前回は集約を格納／取得する「リポジトリ」について紹介しました。第13回となる今回は「境界づけられたコンテキストの統合」について解説します。
開発プロセス

2018/09/10
実践DDD本 第12章「リポジトリ」〜集約の永続化管理を担当〜
ドメイン駆動設計（DDD）は、顧客と開発者がビジネスを戦略的に理解し、共通の言葉を用いてシステムを発展させていく設計手法です。前回は集約の生成を担う「ファクトリ」について紹介しました。第12回となる今回は、集約を格納／取得する「リポジトリ」について紹介します。
開発プロセス

2018/06/26
実践DDD本 第11章「ファクトリ」〜複雑な生成をユビキタス言語でシンプルに〜
ドメイン駆動設計（DDD）は、顧客と開発者がビジネスを戦略的に理解し、共通の言葉を用いてシステムを発展させていく設計手法です。前回は「集約」について紹介しました。11回目となる今回は「ファクトリ」について紹介します。ファクトリを使うことで、ユビキタス言語に沿ってモデルを構築できます。
開発プロセス

2018/04/16
実践DDD本 第10章「集約」〜トランザクション整合性を保つ境界〜
ドメイン駆動設計（DDD）は、顧客と開発者がビジネスを戦略的に理解し、共通の言葉を用いてシステムを発展させていく設計手法です。前回は「モジュール」について紹介しました。10回目となる今回は「集約」について解説します。

2018/02/22
実践DDD本 第9章「モジュール」〜高凝集で疎結合にまとめる〜
ドメイン駆動設計（DDD）は、顧客と開発者が業務を戦略的に理解し、共通の言葉を用いてシステムを発展させていく設計手法です。前回は「ドメインイベント」について紹介しました。9回目となる今回は「モジュール」について紹介します。
開発プロセス

2017/09/13
実践DDD本 第8章「ドメインイベント」〜出来事を記録して活用〜
ドメイン駆動設計（DDD）は、顧客と開発者が業務を戦略的に理解し、共通の言葉を用いてシステムを発展させていく設計手法です。前回は「ドメインサービス」について紹介しました。8回目となる今回は「ドメインイベント」について紹介します。
開発プロセス

2017/07/31
実践DDD本 第7章「ドメインサービス」〜複数の物を扱うビジネスルール〜
ドメイン駆動設計（DDD）は、顧客と開発者が業務を戦略的に理解し、共通の言葉を用いてシステムを発展させていく設計手法です。前回は「値オブジェクト」について紹介しました。7回目となる今回は「ドメインサービス」について紹介します。
開発プロセス

2017/06/02
実践DDD本 第6章「値オブジェクト」〜振る舞いを持つ不変オブジェクト〜
ドメイン駆動設計（DDD）は、顧客と開発者が業務を戦略的に理解し、共通の言葉を用いてシステムを発展させていく設計手法です。前回はエンティティについて紹介しました。第6回となる今回は値オブジェクトについて紹介します。
開発プロセス

2017/04/07
実践DDD本 第5章「エンティティ」 〜一意な識別子で同一性を識別〜
ドメイン駆動設計（DDD）は、顧客と開発者が業務を戦略的に理解し、共通の言葉を用いてシステムを発展させていく設計手法です。前回はDDDのアーキテクチャについて紹介しました。5回目となる今回は、「エンティティ」に対するモデリングの流れを通して、一意な識別子やコーディング例について紹介します。

2017/01/30
実践DDD本 第4章「アーキテクチャ」 〜レイヤからヘキサゴナルへ〜
ドメイン駆動設計（DDD）は、顧客と開発者が業務を戦略的に理解し、共通の言葉を用いてシステムを発展させていく設計手法です。前回は「境界づけられたコンテキスト間のチーム関係を示す「コンテキストマップ」について紹介しました。4回目となる今回は、DDDのアーキテクチャについて紹介します。
開発プロセス

2016/12/16
実践DDD本 第3章「コンテキストマップ」〜「境界づけられたコンテキスト」の関係を俯瞰する地図〜
ドメイン駆動設計（DDD）は、顧客と開発者が業務を戦略的に理解し、共通の言葉を用いてシステムを発展させていく開発手法です。第2回の前回は「ドメイン、サブドメイン、境界づけられたコンテキスト」の概要／分割方法／評価方法について紹介しました。第3回となる本稿では、境界づけられたコンテキスト間のチーム関係を示す「コンテキストマップ」について紹介します。
開発プロセス

2016/11/08
実践DDD本 第2章「ドメイン」「サブドメイン」「境界づけられたコンテキスト」を読み解く
ドメイン駆動設計（DDD）は、顧客と開発者が業務を戦略的に理解し、共通の言葉を用いてシステムを発展させていく開発手法です。第1回となる前回は「DDDの概要」「ユビキタス言語」「DDDを始める方法」について紹介しました。第2回となる本稿では「戦略的設計」において重要な概念「ドメイン」「サブドメイン」「境界づけられたコンテキスト」について紹介します。
開発プロセス

2016/08/01
ドメイン駆動設計のメリットと始め方 〜 1章「DDDへの誘い」
本連載では、書籍『実践ドメイン駆動設計』（以下、IDDD本）の流れに沿って、重要な部分にフォーカスすることで、ドメイン駆動（DDD）について理解することをお手伝いします。初回となる今回は、1章「DDDの誘い」を解説します。

NEXTSCAPE

スピーカー所属企業

- 株式会社ネクストスケープ
  - 企業理念「お客様のビジョンに共感し、感動を実現します」
  - https://www.nextscape.net/
  - 新しいことに挑戦したいエンジニアを募集中です

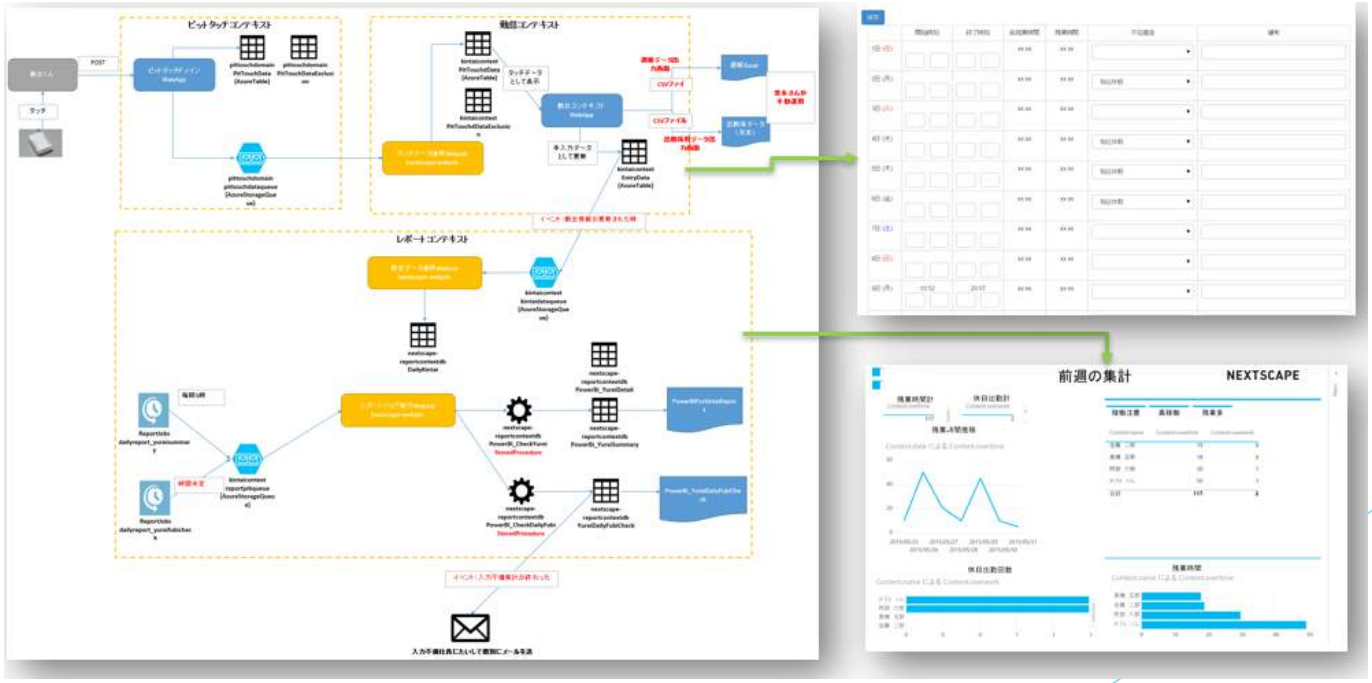
自己紹介

- 株式会社ネクストスケープ
  クラウドソリューションアーキテクト
  青木 淳夫

  - 主にMicrosoftテクノロジーを活用した開発に従事。
    Azureなどのクラウド技術、SitecoreやMarketoといったデジタルマーケティングソリューションを用いてシステム価値を提供。
    最近はHoloLensを用いたMR(Mixed Reality)ソリューションのビジネス活用を支援。
  - 15年前にエクストリーム・プログラミングに出会って以来、アジャイルによるチーム作りを推進中。Certified Scrum Master/Product Owner/Web解析士
  - 著書3冊に加え、100本以上の技術記事を執筆。
    - http://profile.hatena.ne.jp/aoki1210/

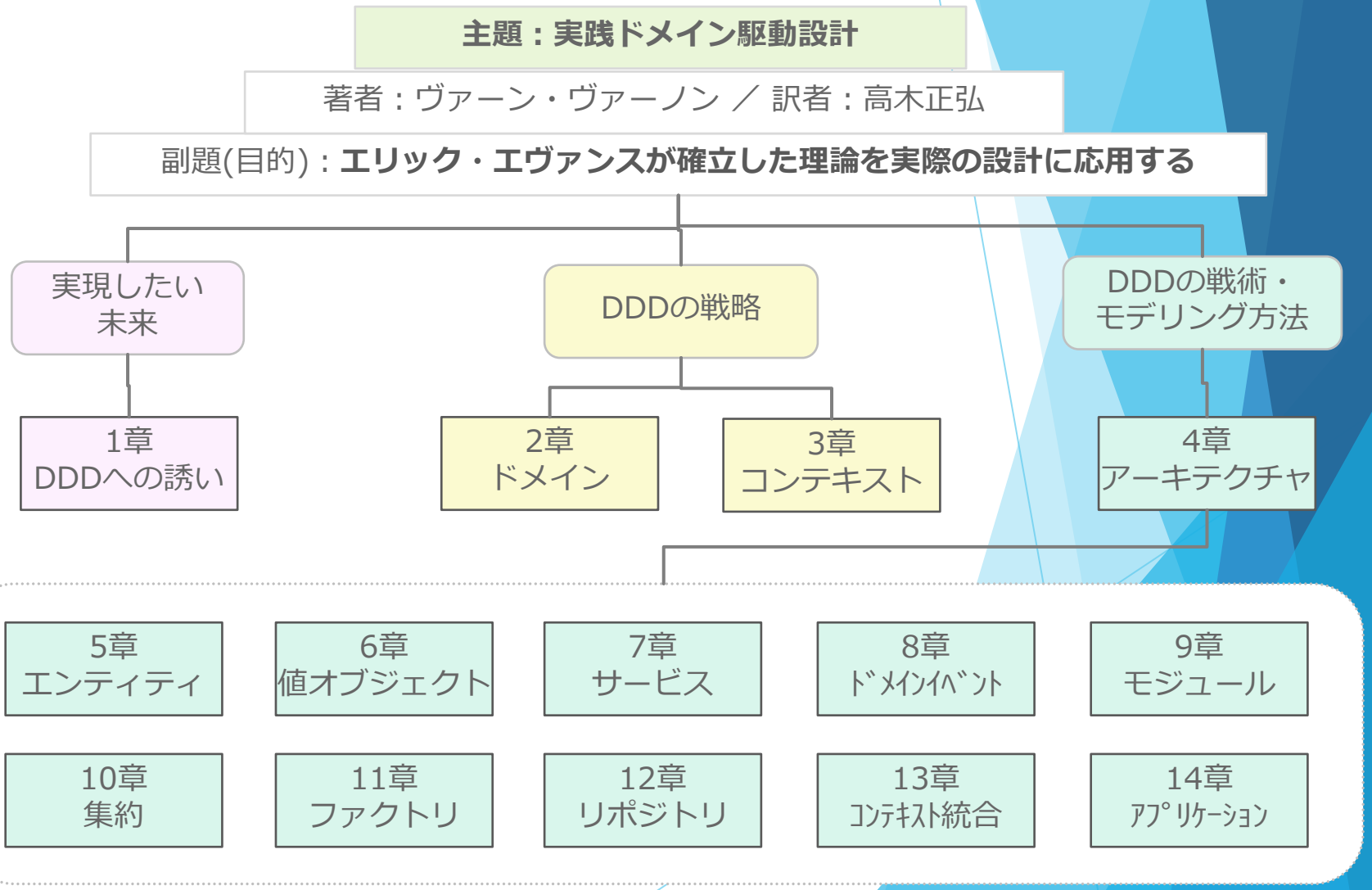
社内でのDDD実践例

- ピットタッチで登録した勤務情報を編集したり登録管理するアプリケーションをDDDで構築しました。
  - （使用技術→CentOS / C#, ASP.NET MVC, Azure, SQL Database, Azure Table / Power BI 等）



IDDD本の構成

- IDDD本の目的は「DDDの理論を実際に適用する」ことで、それらを順序だてて説明しています。
- またSaasOvation(サーズオベイション)という疑似プロジェクトを例としたサンプルコードも提供されています。



IDDDサンプルソース

- IDDD本SaasOvation - Javaのサンプル
  - https://github.com/VaughnVernon/IDDD_Samples
- IDDD本SaasOvation - C#のサンプル
  - https://github.com/VaughnVernon/IDDD_Samples_NET

1章：DDDへの誘い

ドメイン駆動設計のメリットと始め方

https://codezine.jp/article/detail/9546

DDDの原則

- DDDの3原則（エヴァンス本より）
  - コアドメインに集中すること
  - ドメインの実践者とソフトウェアの実践者による創造的な共同作業を通じて、モデルを探究すること
  - 明示的な境界づけられたコンテキストの内部で、ユビキタス言語を語ること

DDD導入で得られるメリット

- 開発費用を**「コスト」から「事業投資」へ**
  - 開発者視点を顧客視点へ揃えることで、ビジネス的な価値、事業価値を高めることができる。
- ドメインモデルによる「複雑さ」への対処
  - ソフトウェア開発が難しい理由のひとつとして対象領域の「複雑さ」にあります。DDDはこの**複雑さを「ドメインモデル」を用いて対処**する。

DDDを実践する人のメリット

- 若手開発者
  - クリエイティブで面白い開発をしたい。
- 中堅開発者
  - 適切で正しいソフトウェア開発をしたい。
- ベテラン開発者
  - 開発者とビジネス側との距離を縮めたい。事業価値を出したい。
- **ドメインエキスパート**
  - 開発者とスムーズな会話をして良いソフトウェアを作りたい。
- マネージャ
  - 開発者が業務知識を得て、ドメインエキスパートと協力して結果を出してほしい。

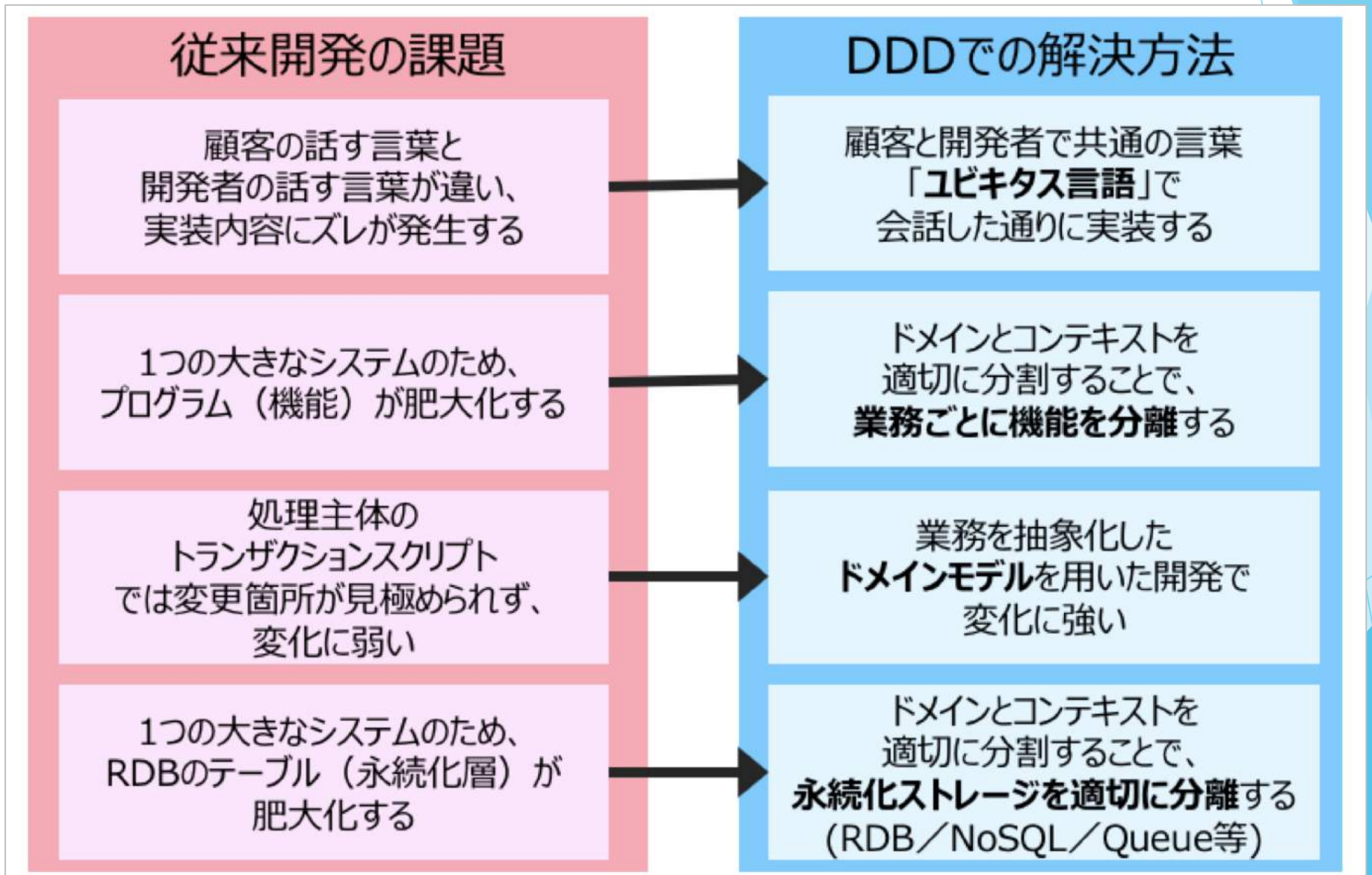
DDDを導入する理由

- プログラマーとドメインエキスパートが「共通言語」を用いて視点をあわせることにより、**ひとつのチーム**として、あたかも顧客が開発するようにソフトウェアを構築できる。
- ドメインエキスパートですら理解が浅い業務領域を深く検討することによって、**業務知識をチームで洗練／共有し、理想像を検討**できる。
- 「共通言語」をそのままプログラムと実装していく。**「設計がコードであり、コードが設計である」**と表現されるように、開発時の翻訳コストが不要になる。合わせて、実装時の課題に設計段階で気づくことができる。

DDDの共通言語「ユビキタス言語」

- ユビキタス言語とは
  - 同じ用語を使って話すという表面的なものではなく、**日本語という自然言語を元にした「チーム全体で作り上げる特別な共有言語」**
  - ユビキタス言語は、そのままコードとして実装される
- ユビキタス言語の見つけ方
  - ドメインに登場する**用語の名称**と**アクション**を記載
  - 「用語集」を作成し、用語の候補と採用／却下理由を記載
  - すでに存在しているドキュメントを集め、重要な用語やフレーズを取得

ソフトウェア課題とDDDが解決すること



DDDの導入方法

- 関係者にDDDがもたらす**事業価値を明示**する
  - ビジネス的価値
    1. コアドメインへの注力
    2. 業務の正しい理解
    3. 顧客（ドメインエキスパート）による設計
    4. より良いユーザー体験
  - 技術的価値
    1. モデル間の境界を明確に
    2. エンタープライズアーキテークチャの整理
    3. アジャイルな継続的なモデリングの実施
    4. ユビキタス言語や技術的戦術の活用

DDD推進のポイント

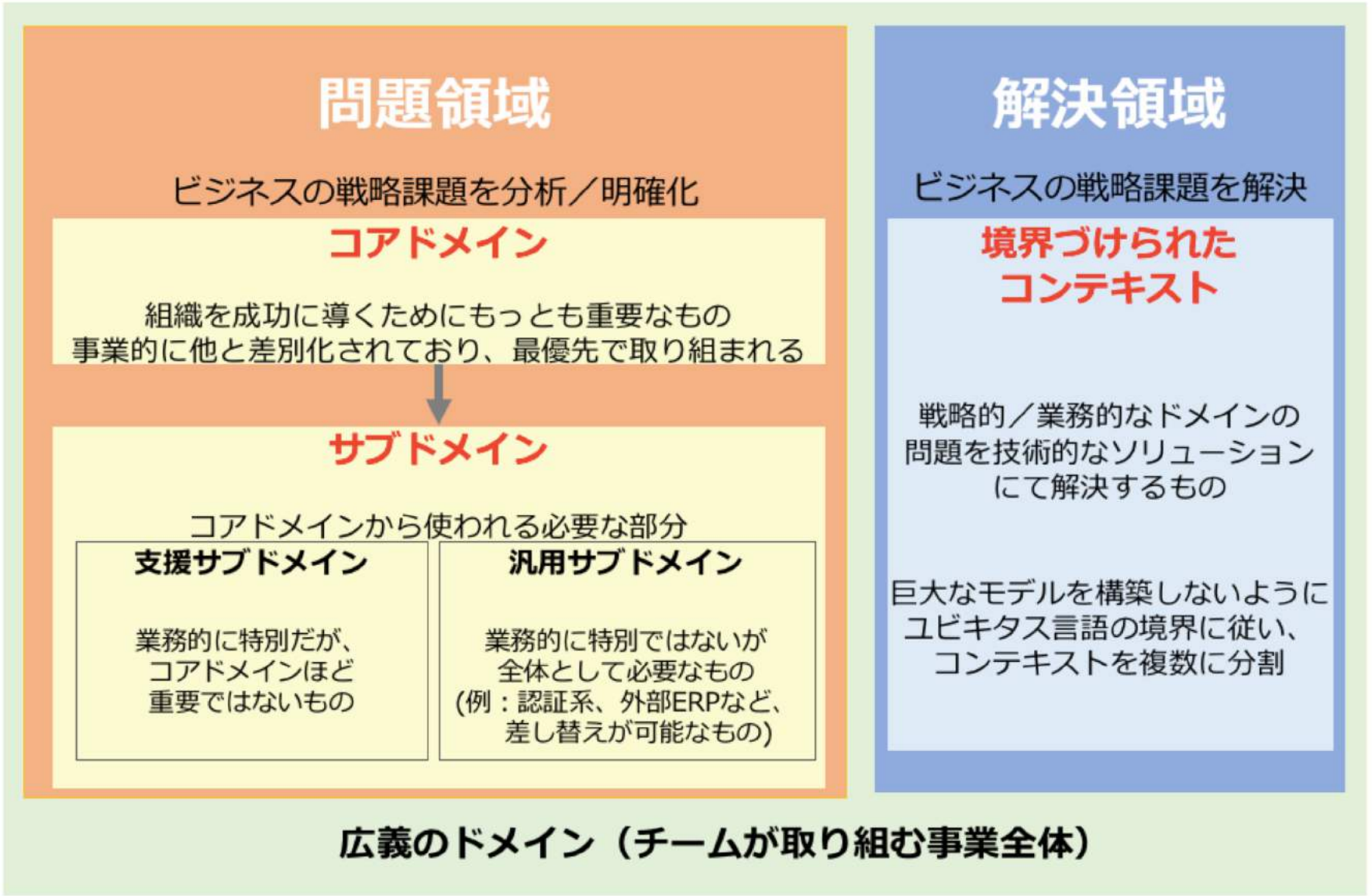
- DDD向けリソース計画
  - コアドメインにエースエンジニアを投入
- 阻害リスク回避
  - 新たな検討時間・作業時間の増加
  - 多忙な顧客（ドメインエキスパート）
  - 保守的な開発者の抵抗
- アジャイル技術前提
  - DDDはアジャイル開発を前提のため、経験あるメンバーを準備

2章：ドメイン、サブドメイン、境界づけられたコンテキスト

https://codezine.jp/article/detail/9744

「問題領域」と「解決領域」の全体像

- 広義のドメインとは
  - 広義のドメインには「分析対象となる問題領域」と「事業課題の改善に取り組む解決領域」が含まれています。
  - **分析対象となる問題領域が「ドメイン」であり、その問題や課題に取り組む解決領域が「境界づけられたコンテキスト」となります。**



コアドメインとサブドメイン

- 「コアドメイン」とは
  - 「コアドメイン」とは事業的に最も重要で戦略的に不可欠な部分です。
- 「サブドメイン」とは
  - 「コアドメイン」ではない補助的な部分を「サブドメイン」と呼びます。
    サブドメインはコアドメインにとって必要な部分となります。
  - 「支援サブドメイン」と「汎用サブドメイン」とは
    - 「支援サブドメイン」はコアドメインほど重要ではないものの、業務的に特別なもの
    - 「汎用サブドメイン」は、業務的に特別ではないが、今回のシステムにおいて必要な箇所

ドメイン（問題領域）の評価ポイント

- 構築システムの対象ドメインの整理後、以下の項目において評価を行います。

1. 戦略的コアドメインの名前、ビジョン、検討するべき概念が正しいか
2. 必要な支援サブドメインと汎用サブドメインの抜け漏れがないか
3. 各ドメインの担当者を招集可能か

「境界づけられたコンテキスト」とは

- **ドメインの課題を解決する部分を「境界づけられたコンテキスト」と呼びます。**
  - IDDDでは、**1つの「コアドメイン（もしくはサブドメイン）」に、1つの「境界づけられたコンテキスト」が対応している状態が最適**だとされています。
- ユビキタス言語を明確にする「境界づけられたコンテキスト」
  - DDDでは共通言語として「ユビキタス言語」を作り上げ、そのモデルに沿って実装します。
  - **ユビキタス言語の意味が変わる境界で「境界づけられたコンテキスト」を分割**して管理します。
  - DDDの**「境界づけられたコンテキスト」は「企業や組織の文化」に近い意味**を持ちます。つまり「境界づけられたコンテキスト」はユビキタス言語が複数の意味を持たないようにするための明示的な境界と言えます。

ユビキタス言語が業種によって変わる例

- 業種によって「アカウント」という用語の意味が変わる例。

業種	「アカウント」という用語の意味
会計システム	勘定科目
営業管理システム	顧客
コンピューターシステム	ログイン権限

ユビキタス言語がタイミングによって変わる例

- 業務フローのタイミングによって、 ECサイトにおける「商品」という用語の意味が変わる例。

順番	「商品」の取り扱い状態	「商品」の呼び名
1	予約中	入荷待ち
2	入荷時	着荷品
3	販売中	在庫品
4	販売後	出庫品
5	トラブル発生時	不良品

境界づけられたコンテキストを扱う体制

- 1つの「境界づけられたコンテキスト」は複数チームではなく、**1つのチームだけで担当**します。
  ドメインエキスパートは自分の裁量でユビキタス言語を定義し、シンプルなモデルを構築できます

- ヴァーン氏はIDDD本でモーツァルトの映画のセリフを引用し『**素晴らしいドメインモデルは、余計な音もなければ、足りない音もない。調和した交響曲のようなもの**』と紹介しています。

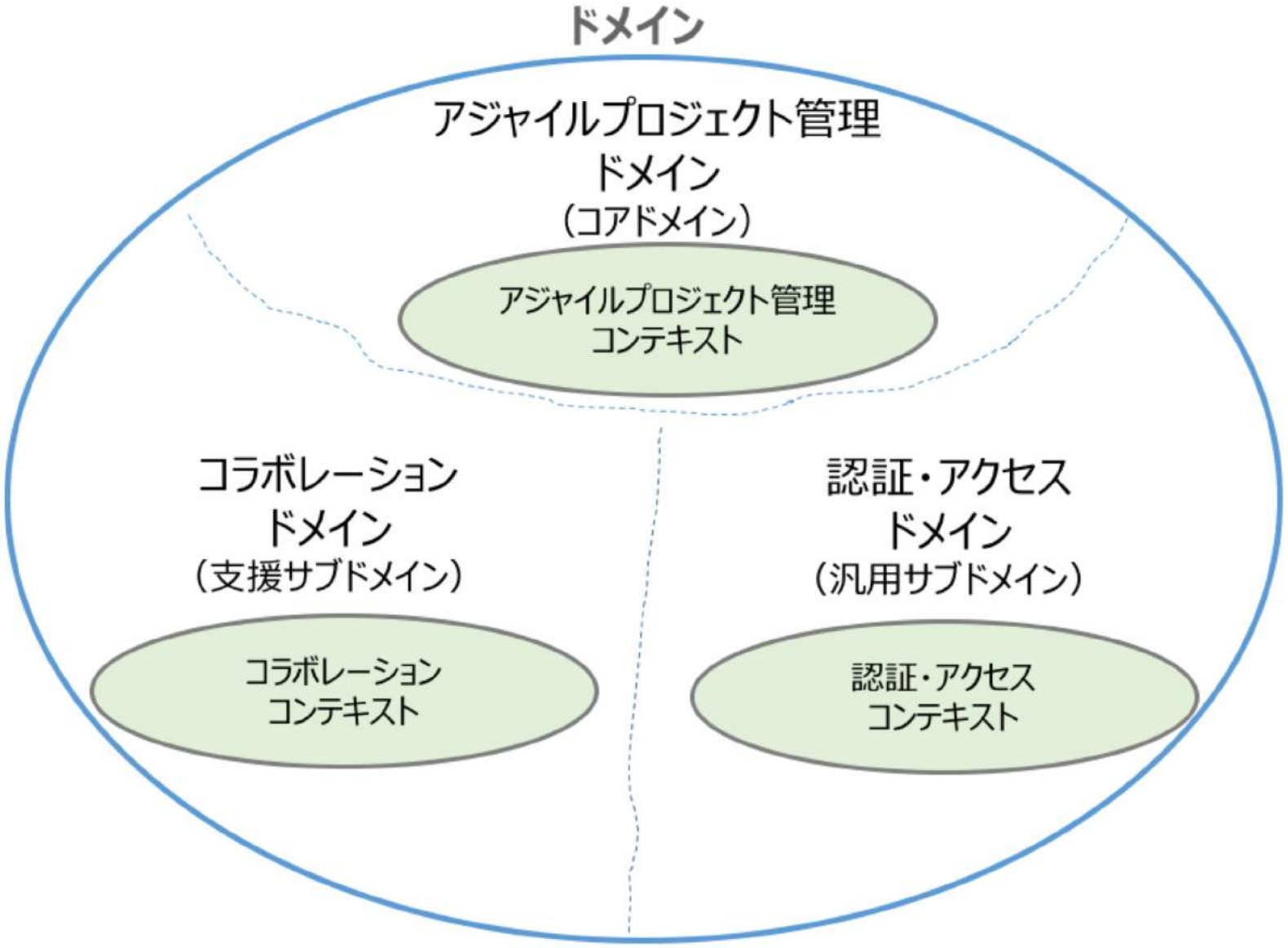
  DDDのチームメンバーはドメインエキスパートによる言語的な境界に注目して、高度な技術と豊かな感性にて「境界づけられたコンテキスト」と「ドメインモデル」を洗練させていきます。

境界づけられたコンテキスト（解決領域）の評価ポイント

- 「境界づけられたコンテキスト」が正しいかを検証する指針

1. 既存ソフトウェア資産の把握（再利用可否と相互接続状況の調査を含む）
2. 新規ソフトウェア資産の検討（開発可能かの調査を含む）
3. 既存ソフトウェアと新規ソフトウェアの統合方法検討
4. 依存する関連プロジェクトのリスク検討
5. ユビキタス言語の抜け漏れの確認
6. 境界づけられたコンテキスト間における「重複または共有しているユビキタス言語」の調査とマッピング方法／変換方法の検討
7. コアドメインの概念が境界づけられたコンテキストに適切に含まれているか確認

IDDDサンプルプロジェクト「SaaSOvation」



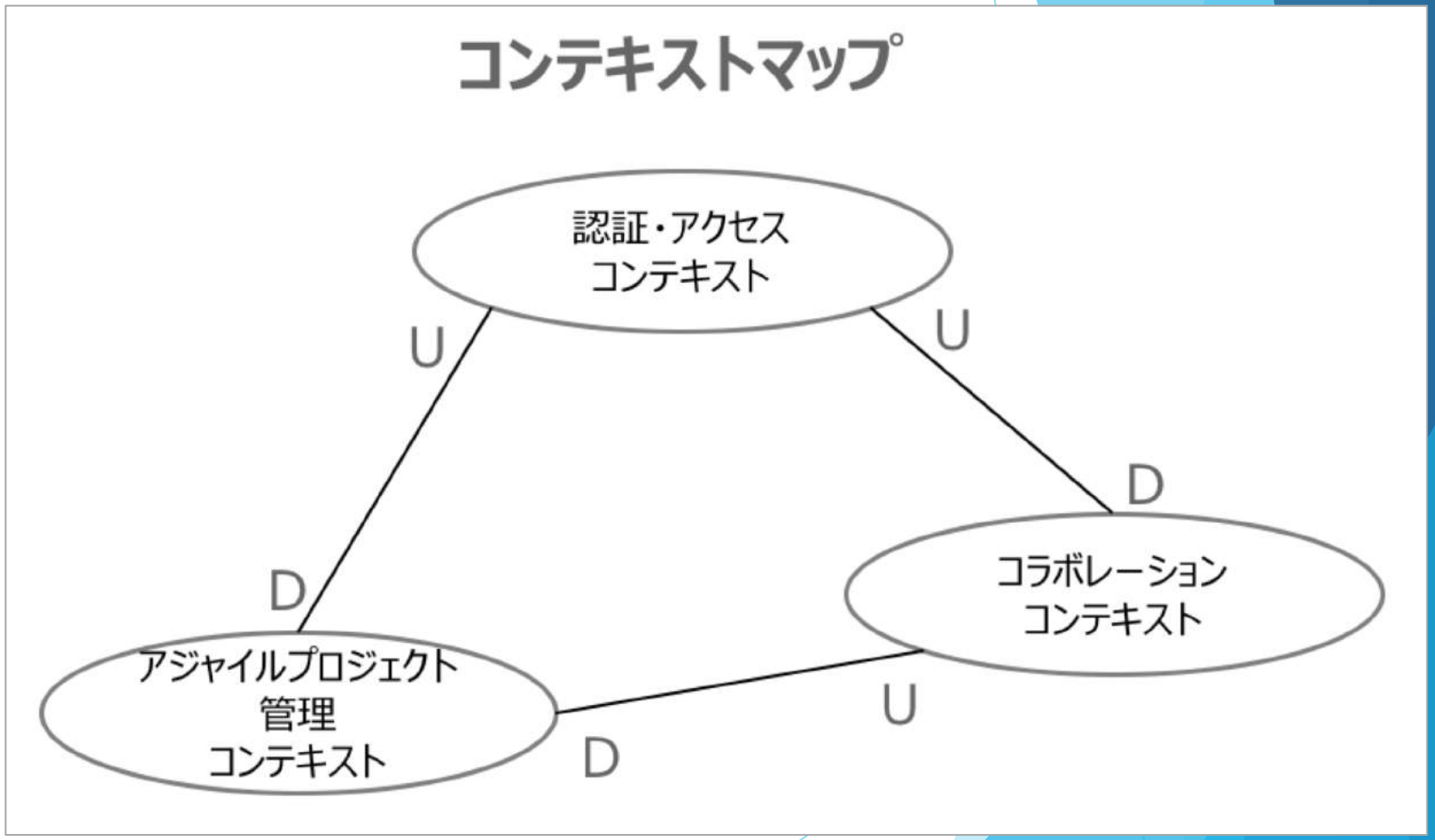
3章：コンテキストマップ

「境界づけられたコンテキスト」の関係を俯瞰する地図

https://codezine.jp/article/detail/9837

コンテキストマップとは

- 「コンテキストマップ」は複数の**「境界づけられたコンテキスト」間の関係を俯瞰する地図**です。
- システムの全体像や相互関係を把握することができます。
- U(上流)、D(下流)



コンテキストマップを描く理由

- コンテキストマップを描くことによって、システム間の関係を適切に把握できるメリットがあります。
- DDDチームは**既存システムとの連携方法を把握でき、他チームとのコミュケーションの必要性を判断**できるようになります。
- コンテキストマップはアーキテクチャ図というよりも、**チーム間のコミュケーション関係を示す図**の意味合いが強くなります。

コンテキストマップを書くタイミング

- 連携するチームとの関係性を示すドキュメントであるため、**なるべく早いうちに作成**することが良いとされています。
- 遠い将来の図ではなく**「現時点」の図**として記述します。
- 機能を追加する時に、随時更新します。

コンテキストマップの「組織パターン」
～チーム間の関係を示す～

（1）パートナーシップ

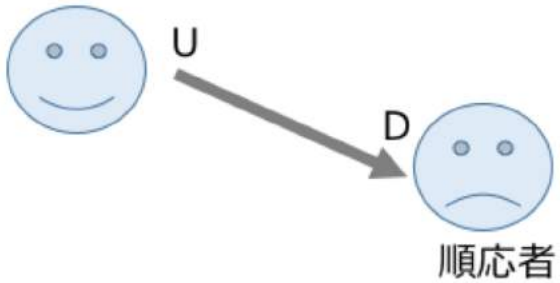
（両者の利害が一致しており、協力関係にある）

（2）別々の道

（統合しない。お互いに関与しない）

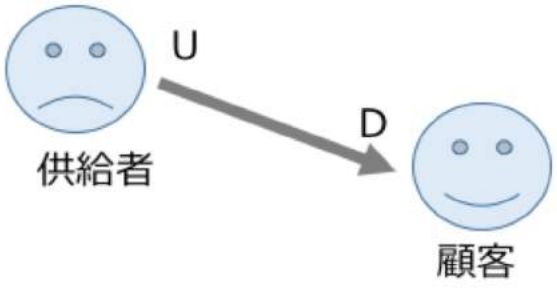
（3）順応者

（上流側が下流側の要望に応えない状態
順応者側が腐敗防止層を用意して受け入れる）



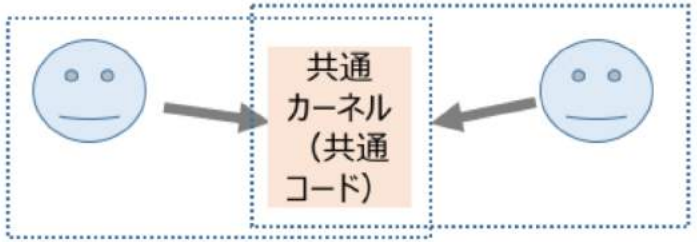
（4）顧客/供給者

（上流側が供給側、下流側が顧客
上流側が下流側のサポートを確約）



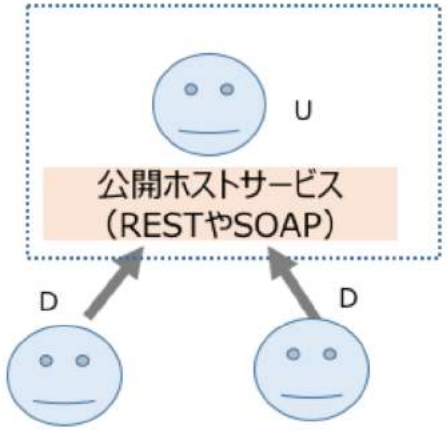
コンテキストマップの「統合パターン」
～データとプログラムの連携方法を示す～

(1) 共有カーネル
(共有専用のドメインモデルを、ソースコードレベルで共通利用)

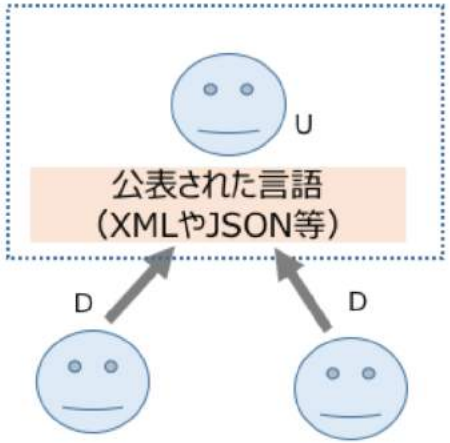


(2) 巨大な泥団子
(適切なサイズの「境界づけられたコンテキスト」分割を締め、
1つの大きな固まりとて使用)

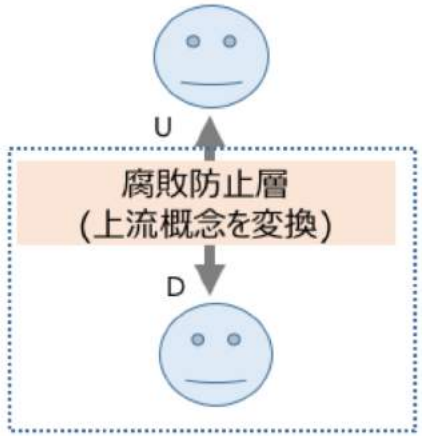
(3) 公開ホストサービス
(コンテキストにアクセスするためのサービス)



(4) 公表された言語
(コンテキスト内のモデルを示すための共通言語)



(5) 腐敗防止層
(上流のドメインモデルの影響を受けない
ように、下流側で変換用のレイヤを用意)



4章：アーキテクチャ

レイヤからヘキサゴナルへ

https://codezine.jp/article/detail/9922

特定のアーキテクチャに依存しないDDD

- DDDは特定の技術に依存していないため、**自由にアーキテクチャ**を選択することができます。
- アーキテクチャの選定においては、構築するシステムに求められる「**機能要求**（ユースケース、ユーザーストーリー、ドメインモデルのシナリオ等）」と「**品質要求**（性能、エラー制御、SLA、リアルタイム性等）」が大きな判断材料となります。
- 最適なアーキテクチャを選択することは、プロジェクト成功に向けて重要なポイントとなります。

IDDDのサンプル（SaaSOvation）でのアーキテクチャの変遷例

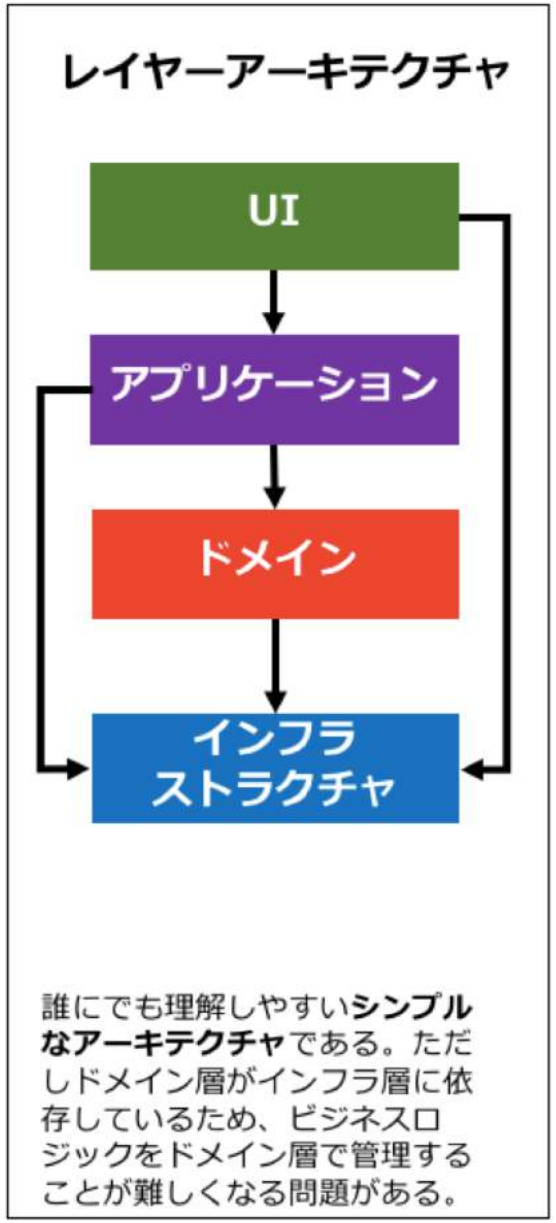
時間の流れ

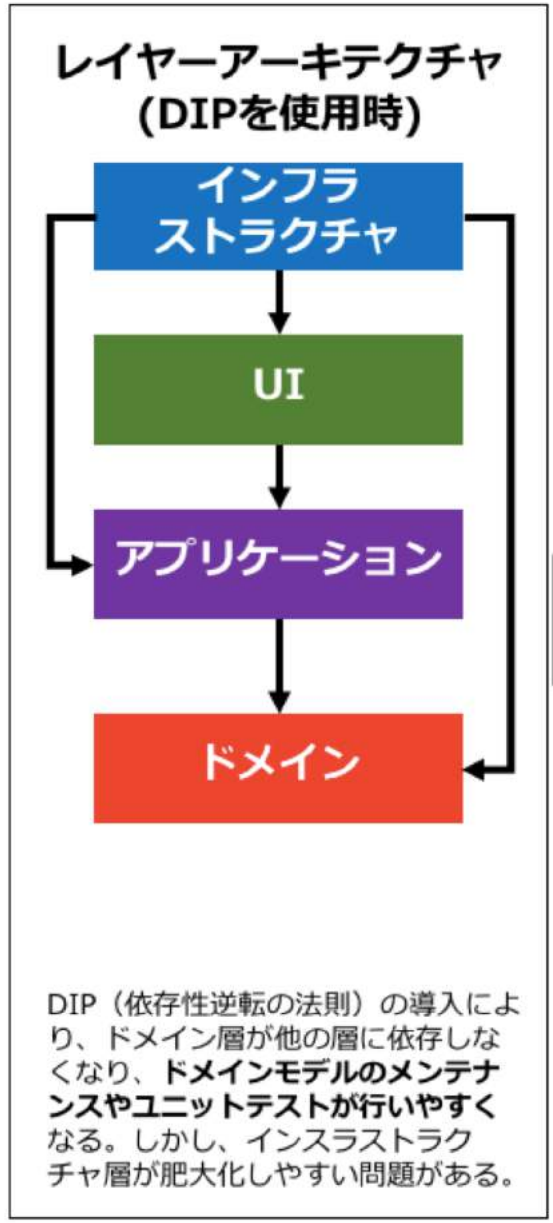
項番	背景	選定アーキテクチャ	詳細
1	**デスクトップアプリケーション**として初期開発	**レイヤアーキテクチャ**	C／Sシステムのためクライアント用のアプリケーション層とDB層で構築
2	SaaSモデルへの移行に加え、**アドオン機能**の複雑化	レイヤアーキテクチャに「**依存性逆転の原則（DIP）**」を導入	DIとユニットテストを導入し品質向上。ドメインに集中し、後にUI層や永続化層を入れ替え
3	**スマホへの対応、認証やセキュリティの一元管理**、管理ツール、BIツールなど多数の要望	**ヘキサゴナルアーキテクチャ**への移行／**REST**の導入	ポート＆アダプターの手法により、NoSQL、メッセージングなどに対応できたことで新機能への要望に対応
4	レガシーシステムのユーザーデータを**クラウドへ移行するサービス**の提供	**SOA**の導入	アプリケーション統合フレームワーク（ESB Mule）を用いてデータをサービスで集約
5	ユーザー別の情報が表示される**ダッシュボード**の機能拡張	**CQRS**の導入	コマンド（入力）とクエリ（描画イベント）を分解して複雑なダッシュボードの表示が可能
6	**時間がかかる処理**のユーザーストレスやタイムアウトへの対応	**イベント駆動アーキテクチャ**の導入	「パイプ＆フィルター」パターンを導入
7	30億ドルで買収され、新ユーザー増加とシステム統合により**データが爆発的に増加**	**分散変更処理**への対応	「パイプ＆フィルター」を分散処理対応するため、「長期プロセス（サーガ）」を導入しインフラを改善
8	政府より**法的な監査**の要求	**イベントソーシング**の導入	ドメインモデルでの全ての変更内容を追跡することでコンプライアンス問題に対応

レイヤからDIP(依存性逆転)

- **レイヤアーキテクチャ**の構成
  - UI層
  - アプリケーション層
  - ドメイン層
  - インフラストラクチャ層

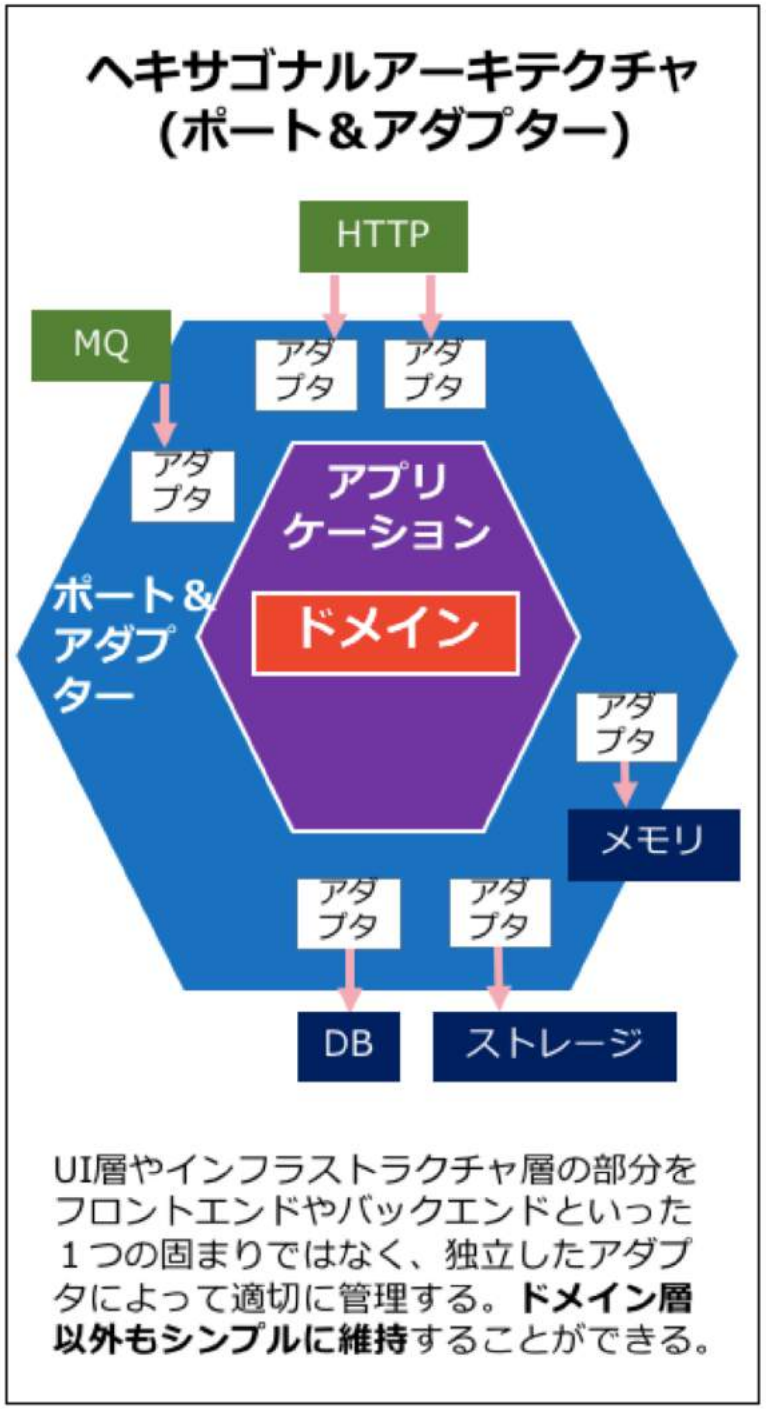
- **依存性逆転の原則（DIP）**を用いたレイヤアーキテクチャ
  - Robert C. Martin氏が提唱したDIPとは「上位が下位に依存する従来の形をやめ、**抽象が詳細に依存するのではなく、実装が抽象に依存するべき**」という指針。
  - ドメインが他のレイヤに依存しなくなるため、ドメイン層を独立させ、シンプルに管理しやすく





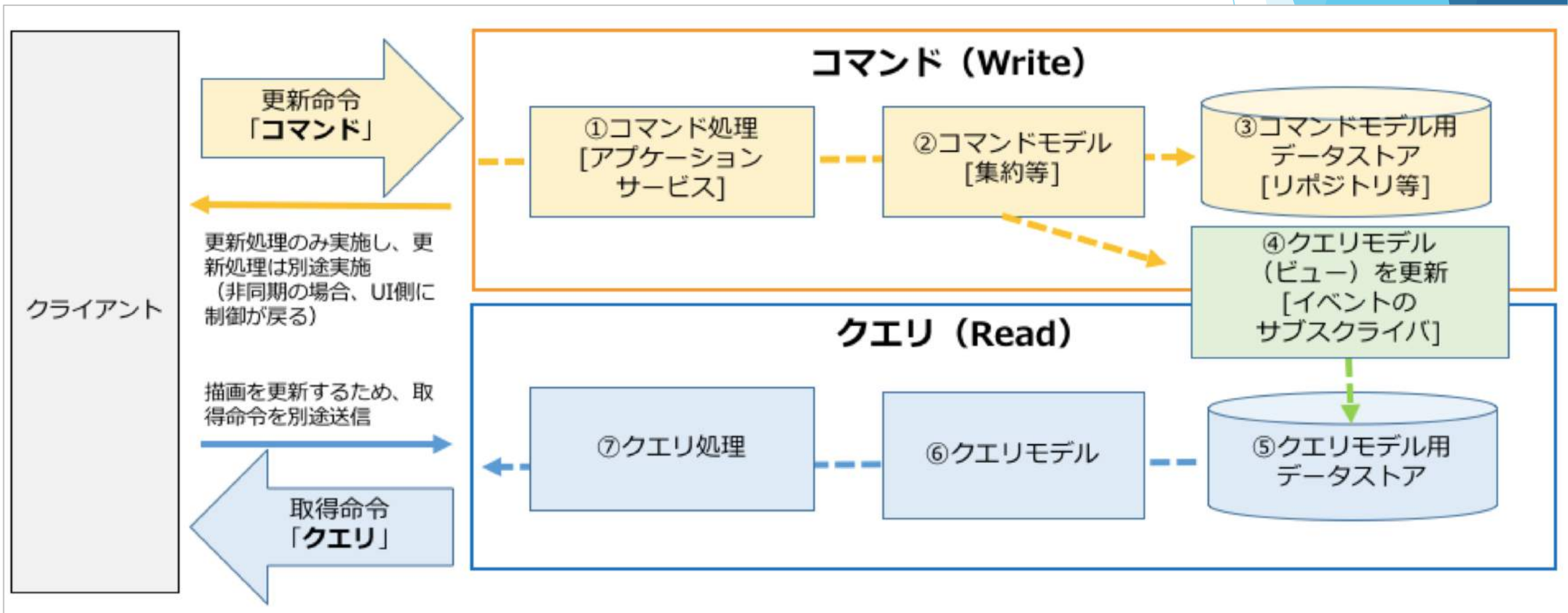
ヘキサゴナルアーキテクチャとは

- Alistair Cockburn氏が提唱したパターンで、要約すると**「アプリケーション（ドメイン）層を中心に捉え、ユーザー操作／自動テストといった入力側もデータベース／モックといった出力側も、全てまとめて差し替え可能な外部インターフェイスとして扱う」**という考え方です。
- レイヤアーキテクチャでは、上位（もしくはフロントエンド）と下位（もしくはバックエンド）という「非対称性」な構成でしたが、ヘキサゴナルアーキテクチャでは、**ドメインを中心において「処理を駆動するプライマリ」側と「処理が駆動されるセカンダリ」側と連携します。例えば、何かの処理がある場合、プライマリ側（左上部分）のアクションによってドメインが呼び出されます。そして、セカンダリ側（右下部分）によって保存したりイベントを生成したりします。**
- ヘキサゴナルアーキテクチャは、構成上の仕組みから**「ポート&アダプタ」**とも呼ばれます。システムの目的に応じて「ポート」が設計され、技術的に差しかえが可能な部分が「アダプタ」として用意されます。「ドメイン」部分は機能やユースケースを基に設計し、「ポート&アダプタ」部分は技術的なインターフェース別で設計するイメージとなります。



CQRS(Command Query Responsibility Segregation)f

- 従来のプログラミングでは、更新と取得を同じメソッドを記述していたかもしれませんが、CQRSでは「更新メソッド」と「結果取得メソッド」として明確に2つに分離します。



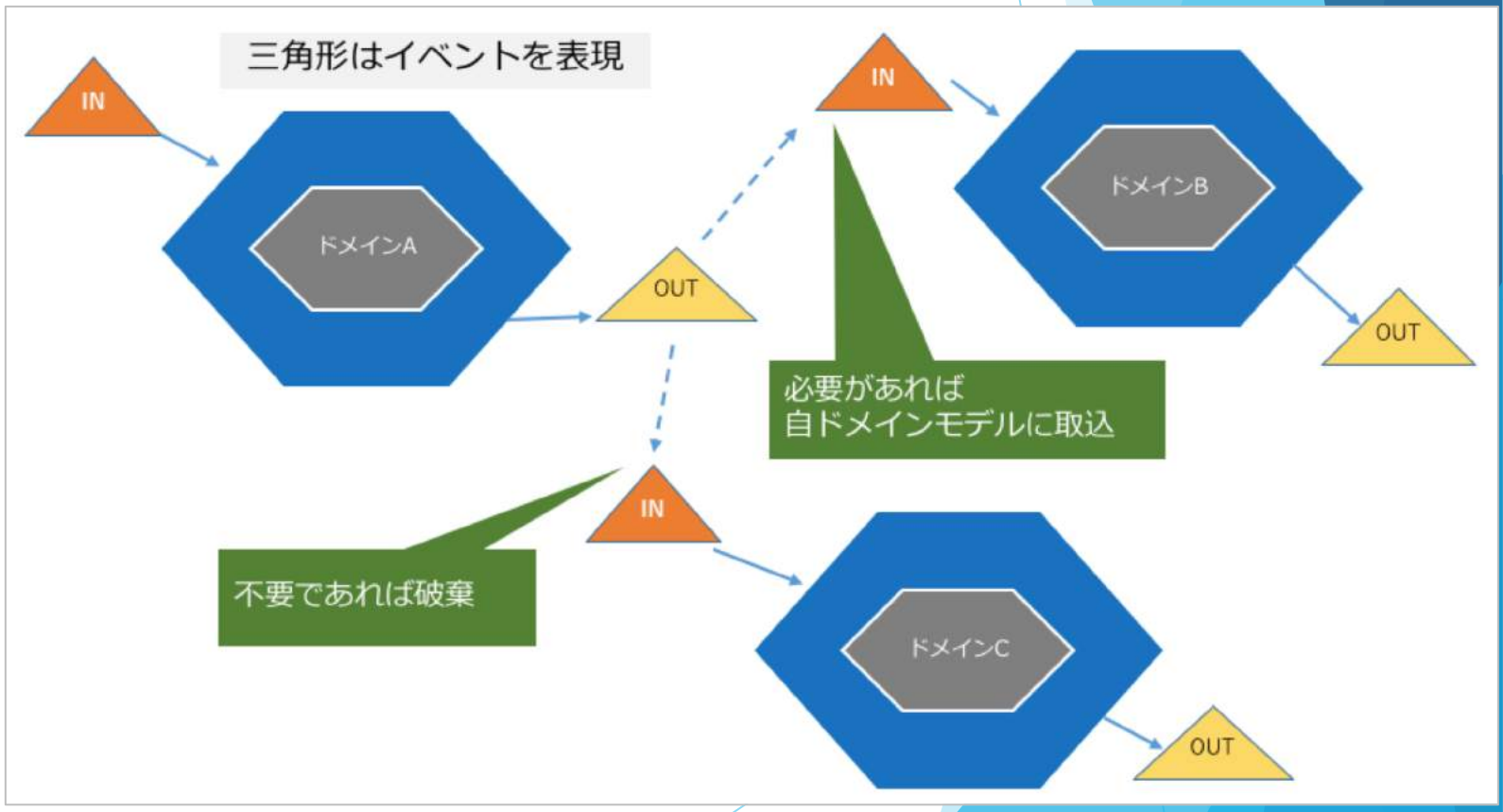
結果整合性

- 「結果整合性」とは「**結果として一貫性が保たれることが保証されていれば問題ない**」という考え方です。

- 標準的なRDBMSアプリケーションでは、ひとつの処理でビジネスルールを正しく保つ「トランザクション整合性」が一般的です。
  （例：銀行口座の入出金）
- これに対しDDDでは、ドメインエキスパートの観点から、最終的に一貫性が保たれれば**多少のタイムラグが合っても問題がないと判断される場合、「結果整合性」を用いることで複雑性を排除**することができます。

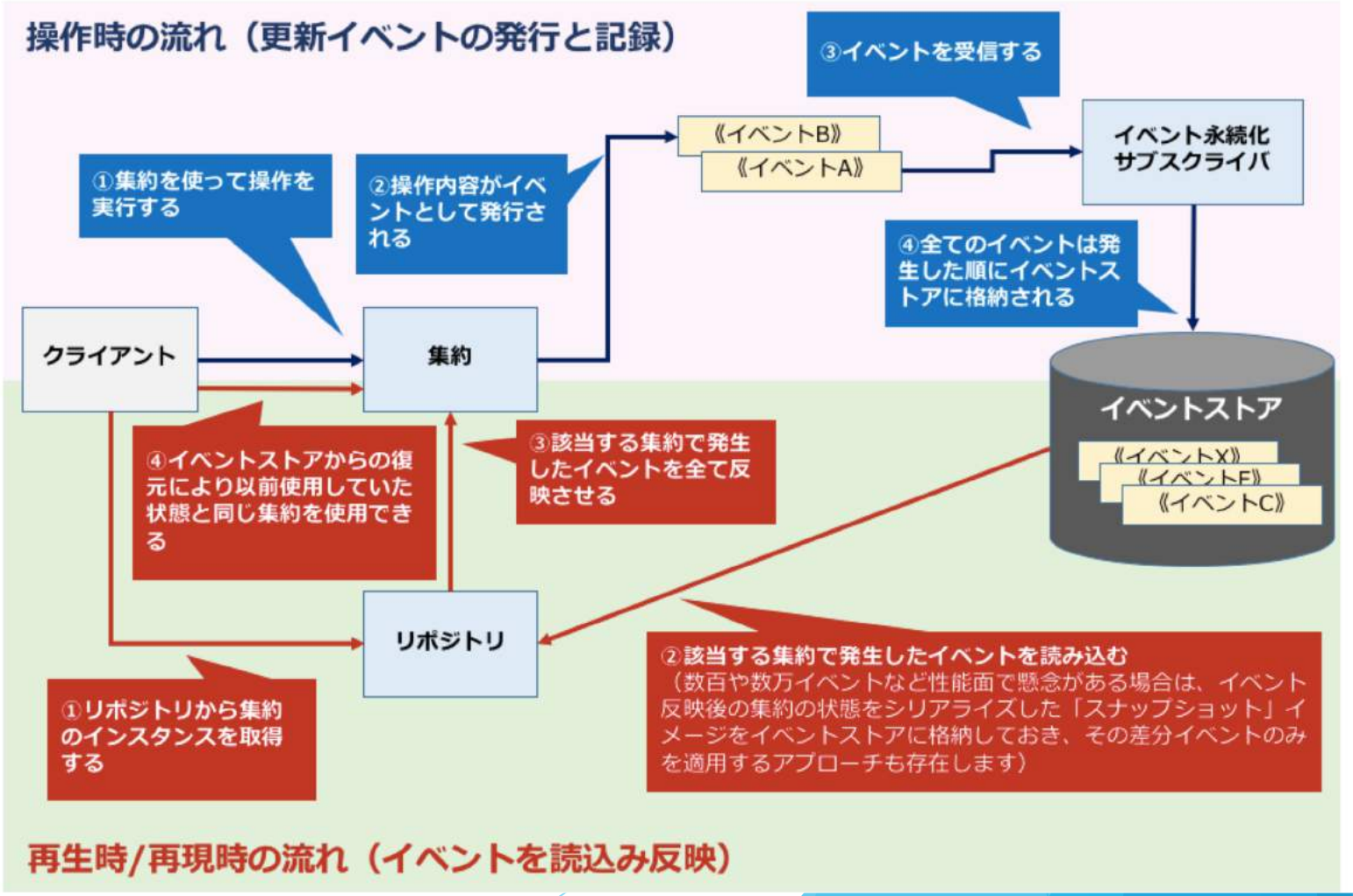
イベント駆動アーキテクチャ

- イベント駆動アーキテクチャ（EDA:Event Driven Architecture）とは、イベントを待機し、起こったイベントに応じて処理を行うプログラムスタイルです。
- ヘキサゴナルアーキテクチャの場合、複数ドメイン間で連携を行うような分散処理に適しています。



イベントを記録/再現できるイベントソーシング

- イベントソーシングの流れは、**発生したすべてのイベントを「イベントストア」に格納**する点が特徴です。
- 記録されたイベントを最初から順番に再生することでオブジェクトの状態を復元できるため、監査やバグ調査において力を発揮します。
- またイベントストアを使って、データパッチを追加することも可能です。
- GitやSubversionといったピンポイントで過去の状態に復元できるリビジョン管理ツールの概念に似ています。



5章：エンティティ

一意な識別子で同一性を識別

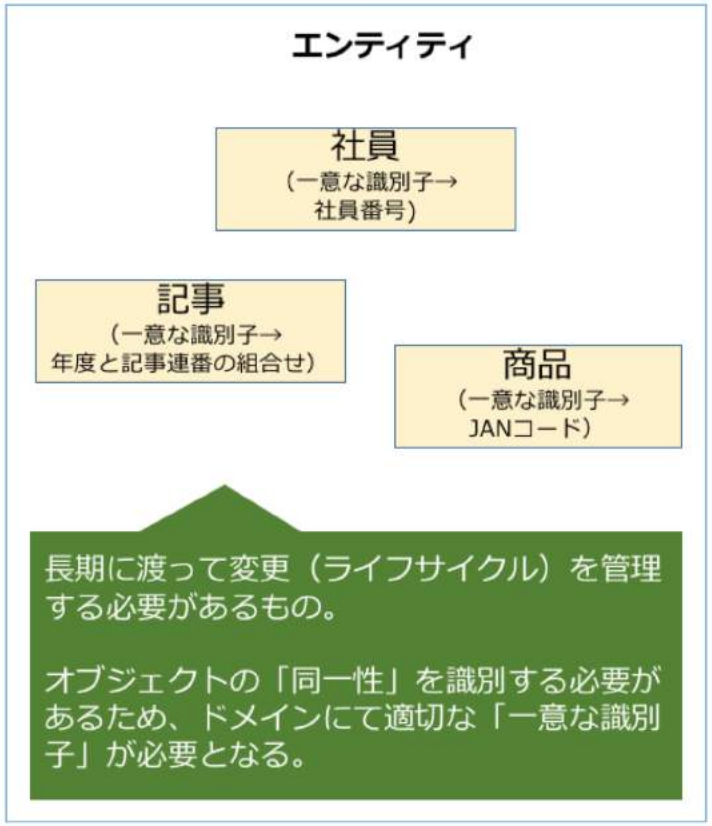
https://codezine.jp/article/detail/10038

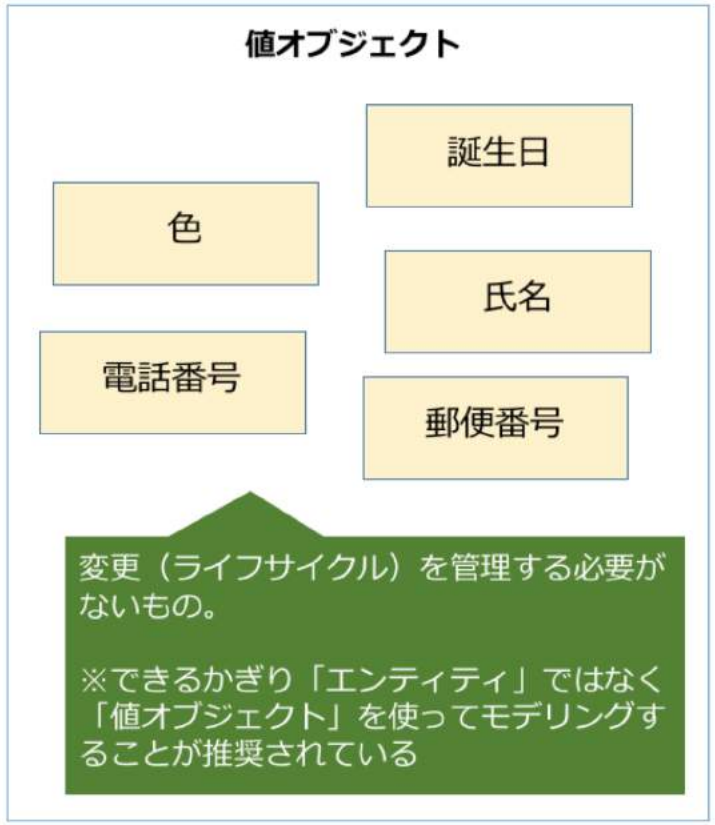
DDDのモデリングでよくある失敗

- DDDで設計を始めてみると、**データの格納方法ばかり気になってしまう**ことがよくあります。
- このようなERモデリング（テーブル構成とリレーション関係）を先に検討されたエンティティは、DB項目のプロパティしか存在しない機能不足なモデル「**ドメインモデル貧血症**」になりがちです。
- ドメインモデル貧血症とは、オブジェクト指向設計の基本である「データと処理を一緒に取り扱う」ことを行わない、単なる手続き型設計のことです。**一見、ドメインモデルと同じような構造を持ちつつも、オブジェクトの「振る舞い」が不足している状態**を指します。
- そのため、DDDで設計する場合には、DOA（データ中心アプローチ）のようにデータ中心で検討するのではなく、OOA（オブジェクト指向アプローチ）のような振る舞いも含めたモデルから検討する方法が適しています。

エンティティと値オブジェクト

- DDDにおける**「エンティティ」とは一意なものを表現する概念**です。一意であるため、**長期にわたって変化できるオブジェクト**となります。
- 例えば「社員」というエンティティの場合、**社員番号で社員を一意に識別することで、同じ人物であることを把握**し、住所や所属といった属性を適切に変更できます。
- なお、一意に識別して変更を管理する必要がないものは「値オブジェクト」として取り扱います。





エンティティ発見の流れ

- DDDでの設計は（DB設計ではなく）オブジェクトモデリングを中心に進めていきます。
  エンティティの発見と検討の流れは次の通りです。

1. ソフトウェア要件の理解
2. モデリングを検討しエンティティを抽出
3. エンティティを識別する属性と振る舞いを検討
4. 「一意な識別子」の設計
   - 一意な識別子の生成方法
   - 一意な識別子の生成タイミング
5. エンティティの振る舞い（メソッド）を検討
6. エンティティの作成方法（コンストラクタ／ファクトリ）を検討
7. エンティティのバリデーションを検討

「一意な識別子」の設計

- エンティティのモデリングでは、エンティティを識別する「一意な識別子」を検討します。
- 4つの一意な識別子の生成パターン

No	概要	値の例	メリット	デメリット
1	ユーザーが「入力」	USR123	識別子を生成する仕組みが不要	重複せず適切な識別子を利用者に登録してもらうワークフローやチェックの仕組みが必要
2	アプリケーションが「生成」	"912649a0-0ed2-4071-8af6-1a350d4adbd0"　(UUIDのみ)や"2017-12-31-USER-1a350d4adbd0" (意味ある値とUUIDの組み合わせ)	UUID(GUID)の生成アルゴリズムを用いて一意な値を軽量に生成	人間には理解しがたい文字列（そのため、通常は利用者には非表示）
3	永続化メカニズム（DB）が「生成」	10001	Oracleのシーケンス等の仕組みを用いて連番を簡単に生成	DB依存が高く性能面の課題がある（早期生成時におけるキャッシュの仕組みの検討が必要）
4	他の境界づけられたコンテキストから「割り当て」	USR123やUUID等	識別子生成の仕組みを他のコンテキストに託せる	他のコンテキストに対して識別子を取得するための検索／マッチング／割り当ての仕組みが必要

「一意な識別子」の生成タイミング

- 生成タイミングについては、「早期生成」と「遅延生成」の2パターンが存在します。

No	概要	メリット	デメリット
A	早期生成（オブジェクト作成時に生成）	オブジェクト作成と同時に識別子を使用可能	DBから識別子を生成する場合は、オブジェクト生成時にリポジトリ経由でDBから識別子を取得する必要がある
B	遅延生成（オブジェクト保存時に生成）	昔ながらの生成方法で、馴染みやすい	オブジェクト生成時にドメインイベントを発行させる場合に、一意な識別子が存在しない。またオブジェクト生成時に識別子が未設定（nullや0）のためオブジェクト間の比較に失敗する

「一意な識別子」で、エンティティの同一性を判定

- エンティティが同じかという**「同一性」を判定する場合、一意な識別子を使ってオブジェクト同士を比較**します（IDDDでは、equalsメソッドとhashCodeメソッドをオーバーライドして、一意な識別子の値の組み合わせが同じであれば、同じエンティティであるとみなしています）。
- 従来のシステム開発ではDB保存時にIDを生成する「遅延生成」が多かったと思います。しかし遅延生成の場合、一意な識別子にデフォルト値（nullや0）が設定されているため、異なるオブジェクトが同一と判定されてしまう問題があります。
- この問題に対応するには、遅延生成をやめるか、早期生成でも問題ないようにequalsメソッドとhashCodeメソッドをオーバライドして、識別子以外で同一性を判断できるようにする必要があります。

エンティティの「一意な識別子」を変更しない「不変性」

- エンティティの一意な識別子を設定した後は、**その識別子の値を変更しないようにします（不変性）**。
- もし、識別子が設定済みにもかかわらず、変更処理が呼び出された場合はエラーとなるようにしなければいけません。

6章：値オブジェクト

振る舞いを持つ不変オブジェクト

https://codezine.jp/article/detail/10184

値オブジェクトの特徴

- 値オブジェクトの特徴として、次の6つを必ず満たすべきと述べられています

No	値オブジェクトの特徴	説明
1	計測 / 定量化 / 説明	ドメイン内の何かを計測したり定量化したり説明したりする
2	不変性	状態を不変に保つことができる
3	概念的な統一体	関連する属性を不可欠な単位として組み合わせることで、概念的な統一体を形成する
4	交換可能性	計測値や説明が変わったときには、全体を完全に置き換えられる
5	等価性	値が等しいかどうかを、他と比較できる
6	副作用のない振る舞い	協力関係にあるその他の概念に「副作用のない振る舞い」を提供する

（1）計測／定量化／説明

- **値オブジェクトはドメインの何かを計測、定量化、説明した結果**となります。
- 例えば年齢や名前の場合、以下の定量化／説明を行っています。

値オブジェクトの例	計測／定量化／説明する値
年齢	何年生きてきたのかを計測し、定量化した値
名前	何と呼ばれているかを説明した値

（2）不変性

- **値オブジェクトはオブジェクトの生成時に値を設定することはできますが、その後、値を変更することはできません。**
- 従来のプログラムでは値の変更は当然だったと思いますが、DDDの値オブジェクトでは値が変更できない「不変性」を持っています。このことにより以下の**メリット**を受けることができます。

メリット	説明
シンプルで安全	利用者の想定外の更新操作が原因で、矛盾した状態に陥ることがない。そのため利用者がオブジェクトの内部状態を気にせずに安心して使える。
スレッドセーフ	マルチスレッドプログラミングにて、単一オブジェクトに対する複数スレッドにおける更新競合を制御する考慮が不要となる。
キャッシュ	値が変化しないことが保障されているため、安心してオブジェクトをキャッシュできる。

（3）統一体

- 一つの属性値だけでは意味を持たず、それぞれが**組み合わさることで適切な説明**をできることを**「概念的な統一体」**と呼びます。
- 通貨の場合、「100」や「円」だけでは意味を持ちませんが、「100円」であれば、完結した意味を持つ値となります。

（4）交換可能性

- 不変性の特徴を持つため、**途中で値の変更を行うことができません。しかし、実際のプログラムでは値の変更を行いたい**ことがあると思います。
- そのようなときは、**変更後の値を設定した新しいオブジェクトを生成して交換**します。このように交換できることを「**交換可能性**」と呼びます。

（5）等価性

- 値オブジェクト同士のインスタンスを比較する場合、**等しいかどうかを判断する「等価性」の判定方法を提供**する必要があります。

- エンティティでは「一意な識別子が同じか」で判定しましたが、**値オブジェクトでは「各属性が持つすべての値が同じか」で判定**します。

（6）副作用のない振る舞い

- 「副作用のない関数」とはエヴァンス氏がDDD本の「しなやかな設計」の部分で紹介しているパターンで、値オブジェクトの条件の一つです。
- **「副作用のない操作」とは、「どのような状態で何回呼び出してもオブジェクトの状態が変わらない」操作**を意味します。

7章：ドメインサービス

複数の物を扱うビジネスルール

https://codezine.jp/article/detail/10318

DDDにおけるサービスとは

- DDDでは、エンティティ、値オブジェクト、集約といった「ドメインオブジェクト」だけではなく、それらの外に記述したほうがよいロジックも存在します。
- そのようなときに、**状態を持たないステートレスな「サービス」**を使用できます。

ドメインサービスの特徴

- ドメインサービスの役割は、ドメインモデルが扱う「粒度の細かい処理」を担うものです。
- その処理がエンティティ／値オブジェクト／集約でもない場合に、ドメインサービスとして実装します。
- そのため、**ドメインサービスはユビキタス言語として表現**されます。
- ドメインサービスで実装する主な内容としては**「さまざまなエンティティ／値オブジェクト／集約を利用して計算するビジネスルール」**が挙げられます。
- また、クライアント側が複雑にならないように、ドメインオブジェクトの構成を変換したり、ビジネスロジックをまとめたりする際にもドメインサービスは使用されます。

ドメインサービスの失敗例

- ドメインサービスの多用／誤用
  - **ユビキタス言語ではないビジネスロジックを大量**に記述してしまう誤りがあります。
  - 特にDDDに慣れていない場合、**トランザクションスクリプト**で処理を書いてしまう可能性があります。
- ドメインサービスのミニレイヤ
  - 検討することなく最初からドメインサービスのレイヤを作ってしまうと、このレイヤが肥大化する傾向にあります。

8章：ドメインイベント

出来事を記録して活用

https://codezine.jp/article/detail/10392

「ドメインイベント」がもたらすメリット

- ドメインイベントには、**結果整合性（結果として一貫性が保たれること）を用いたシステムの構築が容易**となるメリットがあります。
- ドメインイベントを利用することで、**密結合になりがちな他システムとの連携を疎結合**（他の集約へのデータ反映含む）にできます。

ドメインイベントのモデリング方法

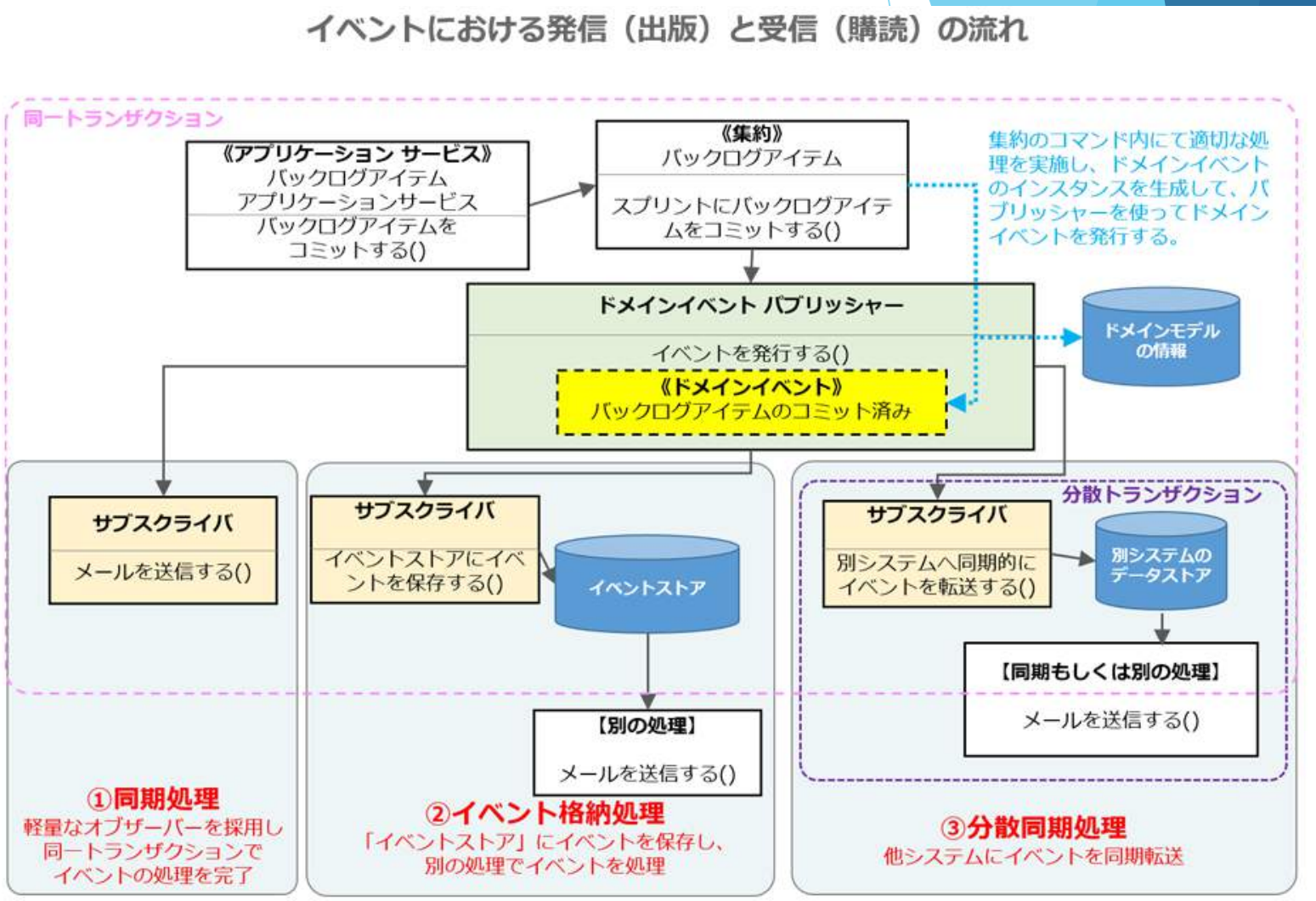
- **ドメインイベントを見つけるには「ドメインエキスパートが注目する出来事」にフォーカス**します。
  - 具体的にはドメインエキスパートが話す「～する時に」や「～した場合」といったフレーズに注目します。
- 「システム的な観点」からイベントを導入するケースもあります。
  - 例えば、**特定のシステムで発生した出来事を外部システムに通知**する場合や、境界づけられたコンテキスト内のトランザクションを分離する場合などです。

ドメインイベントの導入

- イベントを導入していくには、ドメインエキスパートの**「その結果どうなるのか」「いつ無視するべきか」といった業務的な観点からイベントの要否を判断**します。
- 次に「他システム連携は必要か」「イベントソーシングは必要か」などの技術的な観点からイベントの連携方式を検討します。

イベントを用いた処理の例

1. 同期処理パターン
   - イベント発行と同じトランザクションで実施
2. イベント格納処理パターン
   - イベントの記録だけ行います。後述する「イベントストア」等を活用して、イベントに応じた処理は別途実行
3. 分散処理パターン
   - イベントを他システムにリアルタイムで転送し、2フェーズコミットなどの「分散トランザクション」で制御



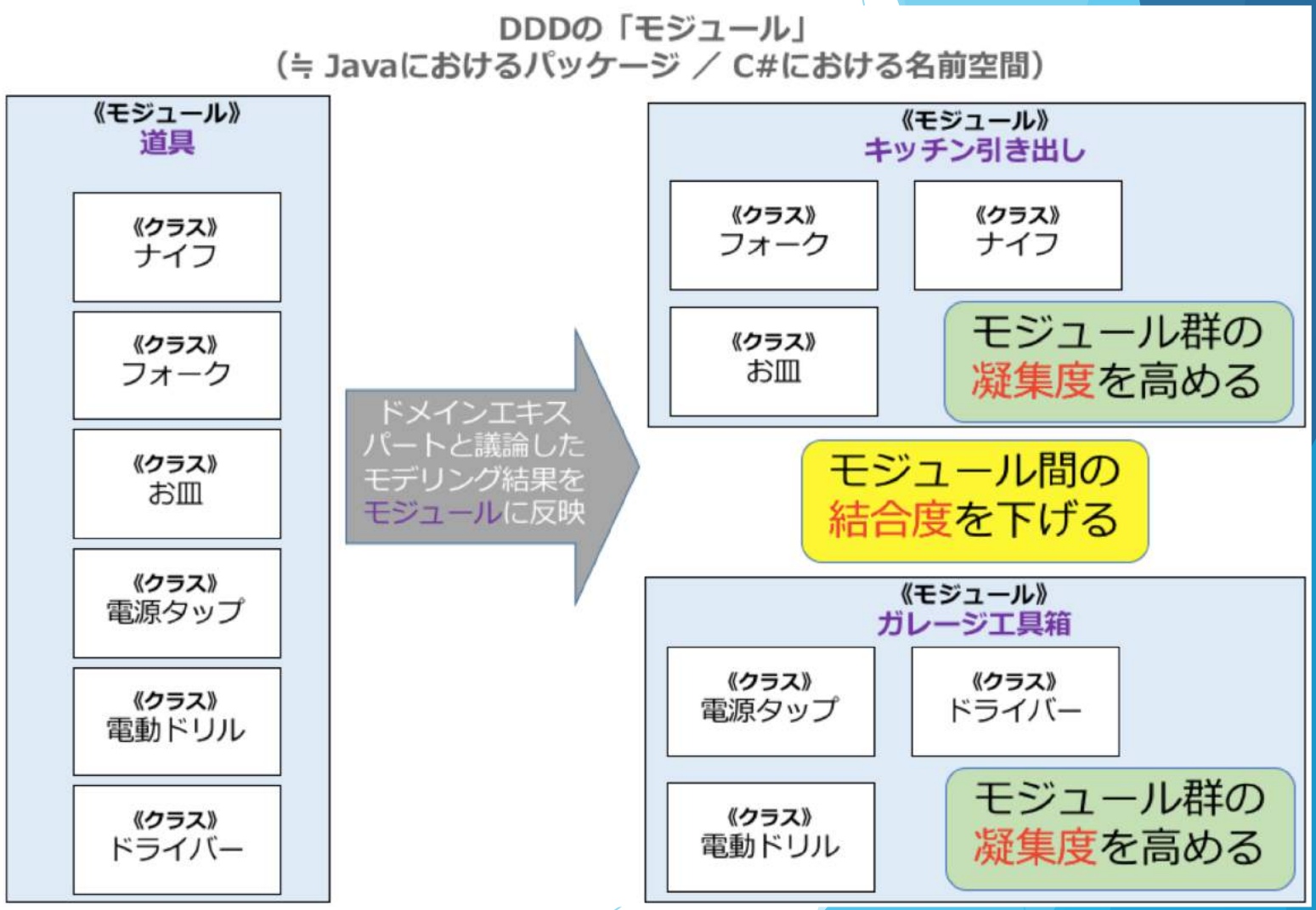
9章：モジュール

高凝集で疎結合にまとめる

https://codezine.jp/article/detail/10651

DDDでのモジュールとは

- **「モジュール」はプログラム言語における「パッケージ」や「名前空間」にあたり、クラス群をまとめて管理する入れ物**となります。
- この機能により、クラス群の責任と役割を明確にして、保守性や可読性を上げることができます。



モジュールの設計方法

- DDDでは**ソースファイルを物理的に分けるだけではなく、高凝縮で低結合なモジュール化されたモデルを、ユビキタス言語に従って構築**することを推奨しています。
- **重要なポイントとして、まず「モジュールの存在を軽視しない」こと**を挙げています。エンティティ／値オブジェクト／サービス／イベントと同じ扱いで、モジュールについても十分に議論するようにします。
- モデリングする際には、正確な意図が伝わるようにモジュール名を検討します。必要があれば恐れずにモジュール名の変更を行います。ドメインエキスパートとディスカッションしている最新情報がモジュールに反映されるように努めます。

モジュールの設計ルール

- モジュールを設計する指針として、IDDDでは以下のように整理しています。

番号	モジュールの設計指針
1	モジュールをDDDにおいて非常に重要な概念と認識し、モデリングの概念にフィットさせ設計する。例えば、集約に対して1つのモジュールを用意する。
2	モデリングの概念に従い、モジュール名をユビキタス言語に従う。
3	モジュール名を機械的に決めない。例えば、クラスの型や役割だけでまとめるようなモジュールを作らない。
4	疎結合に設計する。極力、他のモジュールに依存しないようにする。
5	モジュール同士の結合が必要な場合に、循環依存が起きないようにする。意味的に双方向な依存関係があっても実装上は単一方向の依存関係が望ましい。
6	モジュール間の循環依存は避けるべきだが、親子関係（上位と下位関係）のモジュールに限り、やむをえない場合がある。
7	取りまとめているオブジェクト群に合わせた名前をつけ、そぐわない場合はリファクタリングする。

10章：集約

トランザクション整合性を保つ境界

https://codezine.jp/article/detail/10776

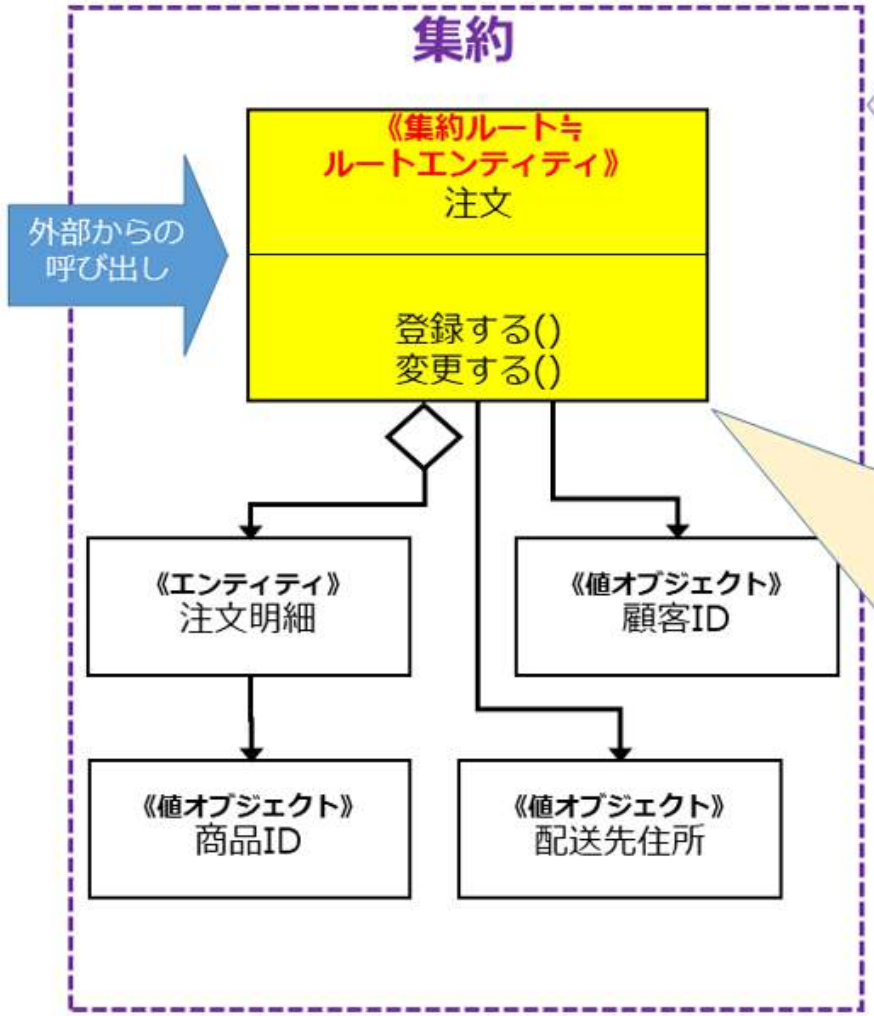
DDDにおける集約（Aggregates）とは

- 集約はオブジェクトのまとまりを表し、**整合性を保ちながらデータを更新する単位**となります
- 通常はオブジェクトの集まりの「境界線」の意味で使われ、**オブジェクト群の生成／読み込み／変更／保存といったライフサイクル管理**を行います

外部から集約を操作できる「集約ルート」

- 外部から集約を操作する場合、代表オブジェクトである「集約ルート（≒ルートエンティティ）」のみ参照することができます。
- 集約ルートを操作することで集約全体の整合性を保ちながらデータを変更できます。

DDDの「集約」のイメージ



・集約は、エンティティや値オブジェクトを内包する**境界**を定義できる。

・集約は内部のオブジェクトが変更される場合、**トランザクション整合性**を正しく保つ。

・集約の利用者は、**集約ルート「注文」のみを介して、集約の操作を行える。**（外部から「注文明細」を直接操作しない。ただし、外部に一時的に参照を渡すことはできる。）

・集約ルートは、集約全体のビジネスルールである**不変条件**を守る責務を持つ。

・ルートエンティティ（≒集約ルート）はグローバルで一意となり、境界内部のエンティティは集約内で一意となる。

集約が担う境界内の「整合性」

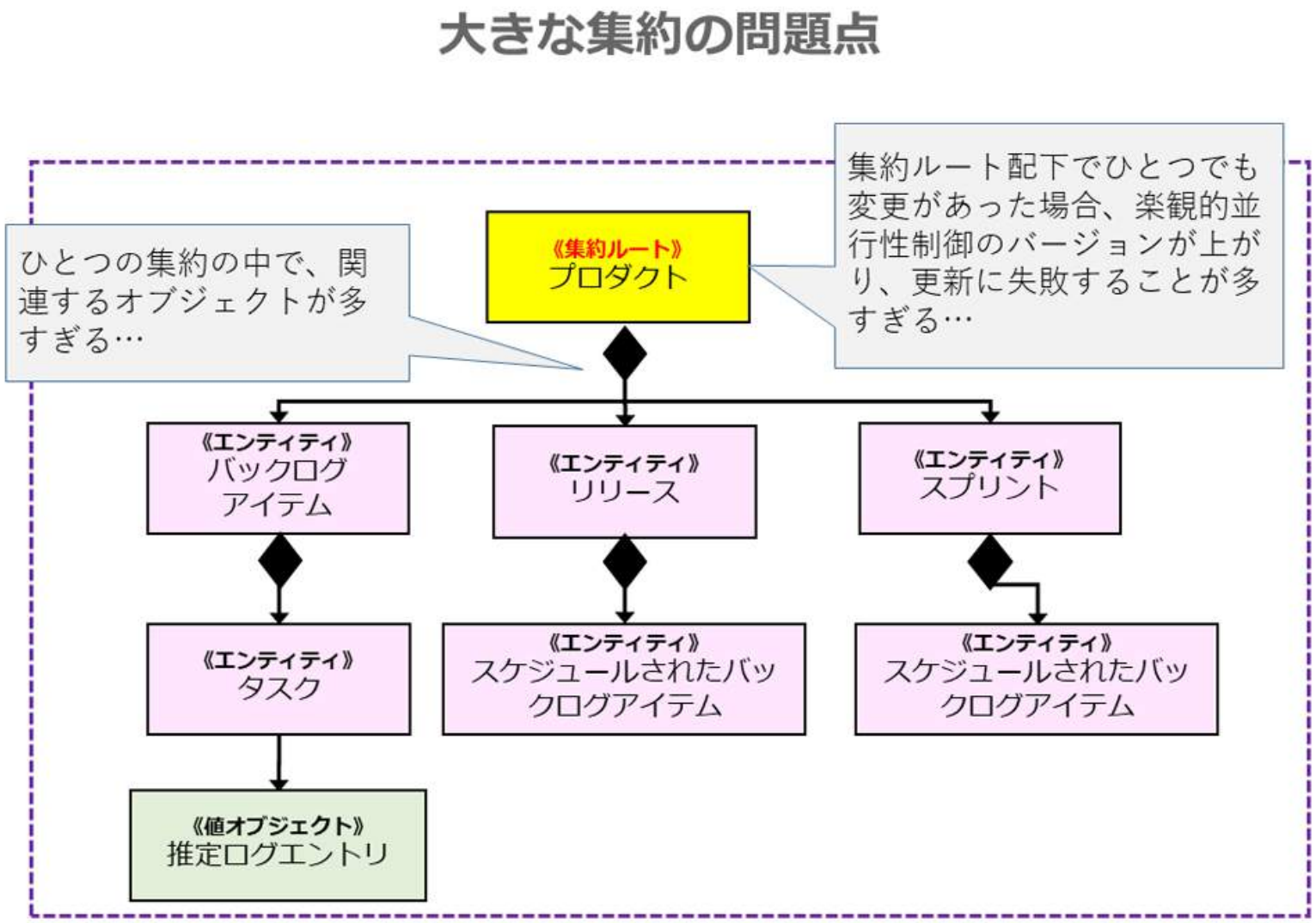
- 集約はオブジェクト群の「整合性」の境界を保ちながらデータを更新します。
- **「整合性」には以下の2つ**が存在しています。

1. **トランザクション整合性**
2. **結果整合性**

- DDDの集約は、即時的な整合性である「トランザクション整合性」と同義です。
  - 特定のDBのトランザクション命令そのものではありません。あくまで（結果整合性とは対照的に）同期的に整合性を保つ必要性を表しています。

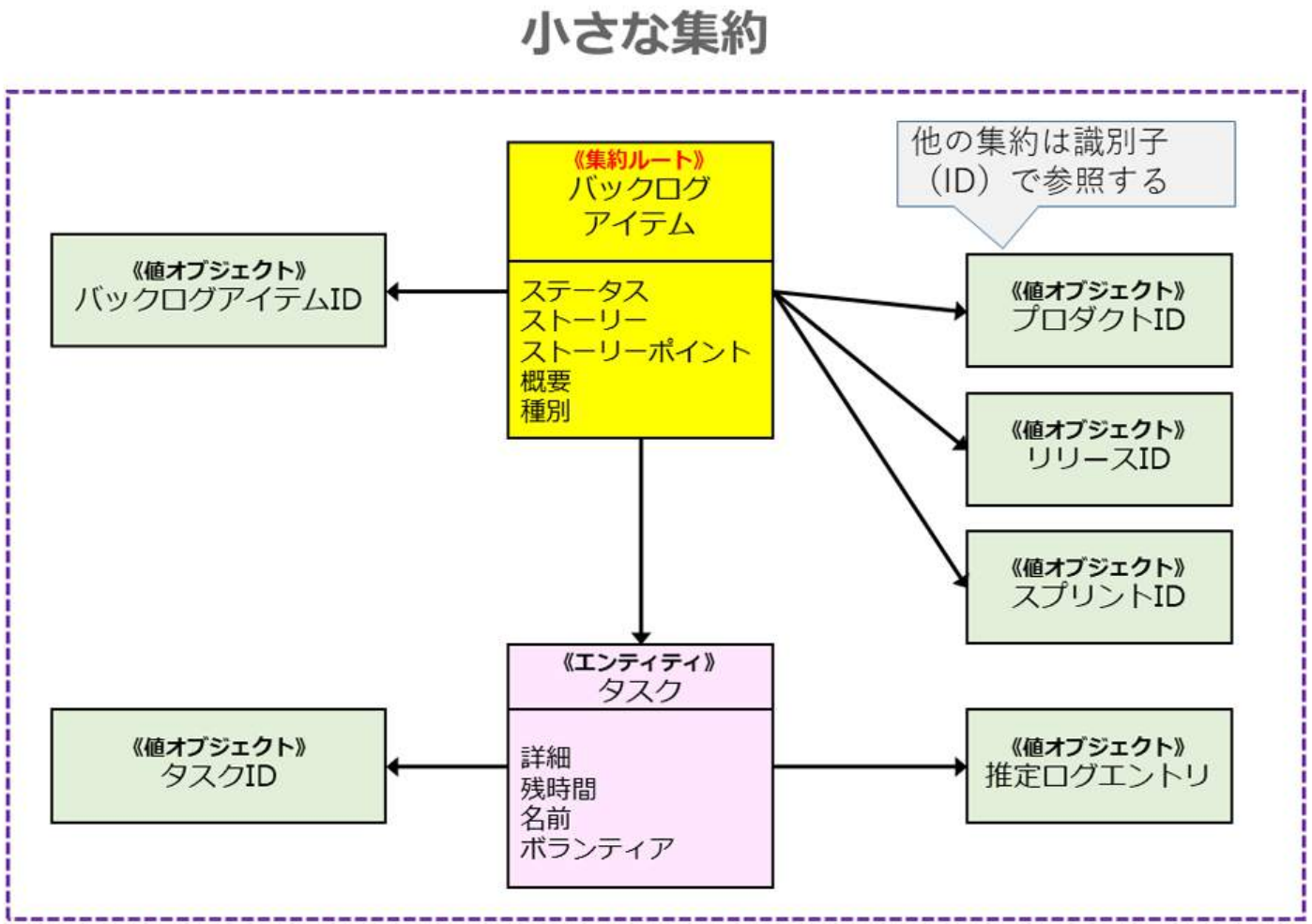
巨大な集約の問題点

- **何も考慮せずにモデリングした場合、大きな集約**ができ上がります。
- こうした大きい集約を作成した場合、数千のオブジェクトを毎回メモリに読み込み、更新する必要があり、**性能面で期待できません。**



小さな集約の特徴

- 大きな集約の問題を回避する場合、小さい集約を複数作成することになります。
- これまでひとつだった集約を複数に分解し、各集約それぞれに集約ルートが存在することになります。
- 大きな集約を分割し、エンティティは、別の集約のルートエンティティとしてモデリングしています。
- そして、他の集約ルートを直接参照するのではなく、識別子（ID）のみを参照しています。



集約の設計ルール

1. 小さな集約を設計する
   - パフォーマンスやスケーラビリティといった非機能要件の観点から、小さな集約になるように設計
2. 極力、値オブジェクトから構成する
   - 集約の内部は、ルートエンティティ以外は「値オブジェクト」だけであることが望ましい
3. 真の不変条件（ビジネスルール）を、整合性の中にモデリングする
   - 不変条件とは、どのような操作を行おうとも維持される「ビジネスルール」
4. 他の集約への参照へは、その一意な識別子を使用する
   - 集約では、他の集約ルートへの参照を保持するようにします。
   - 密結合ではなく識別子（ID）を介した参照を使うことで、疎結合に集約を組み合わせることができます。一般的に複雑になるため双方向の参照も行いません。
5. 境界の外部では結果整合性を用いる
   - 複数の集約にまたがるビジネスルールがある場合には、結果整合性を使うことになります。
   - この場合、メインとなる集約でイベントを発生させ、受信した側の集約にて処理を継続します。

大きい集約と小さい集約の比較

- 大きい集約と小さい集約の比較

項目	大きい集約	小さい集約
トランザクションの衝突	多い	**少ない**
設計難易度	**低**	中
スケーラビリティ・性能	低	**高**
エンティティ間の依存	直接参照	ID参照

  - 集約を小さくすることでトランザクションが衝突しなくなります。
  - なお、要件を満たすだけの性能が出せるかは、**実際のシナリオに基づいた検証を行う**必要があります。

11章：ファクトリ

複雑な生成をユビキタス言語でシンプルに

https://codezine.jp/article/detail/10903

DDDのファクトリとは

- 開発における一般的な「ファクトリ」とは、複雑になりがちなオブジェクト群を簡単に組み立て、呼び出し側に対して内部構造を隠して生成する仕組みのことを指します。
- これに対して、**DDDのファクトリでは「ユビキタス言語を用いて集約をシンプルに生成する」という意味合い**が強くなります。
- DDDのファクトリとしては、下記のようなシナリオが中心となります。
  1. **特定の集約にて、別のオブジェクト（主に集約）を生成**
  2. サービスにて、別の「境界づけられたコンテキスト」のオブジェクトを、ローカルの「境界づけられたコンテキスト」の型のオブジェクトに変換して生成

ファクトリを使う理由

- エヴァンス氏は、「車のエンジン」や「銀行口座」を例にファクトリを利用する理由を紹介しています。
  - エンジンの場合「複数部品を使ってエンジンを組み立てたてる処理」と「回転する処理」を分けています。
  - 銀行口座の場合「口座の開設処理」と「口座の取引処理」を分けています。
- このように**複雑な生成処理を分離することで、全体を理解しやすくできることがファクトリのメリット**と紹介しています。

ファクトリの設計

- オブジェクトの生成パターンとしては、次の3パターンが想定できます。

IDDDでのファクトリの種類	メソッドの場所	メソッド名	特徴
(1)コンストラクタ	クラス本体	クラス名	コンストラクタのためシンプル。ユビキタス言語ではない。複雑な生成（他の集約の生成など）に適さない。引数が多い。
(2)**集約上のファクトリ**	**集約**	**ユビキタス言語**	複雑な生成が可能。不変条件を満たしたり、引数を減らしたりすることができる。
(3)サービス上のファクトリ	ドメインサービス	ユビキタス言語	境界づけられたコンテキスト間の変換が可能。

不変条件を満たすファクトリ

- DDDのファクトリにおいては**「一貫した状態のオブジェクトだけを返す」**という制約があります。
- 利用者は、複雑な生成ロジックを意識することなく、簡単に正しいオブジェクトを取得できます。
  - その際に、生成される集約の状態が一貫して正しいこと、つまり、**不変条件が正しく**守られていることが重要な役割となります。
  - このように、ファクトリは生成するオブジェクトに対して、**アトミック性（原子のように、それ以上細かい単位や要素に分割されないこと。すべて完了するか1つも実行されないこと）を保ち**、中途半端に生成されないことを約束します。

12章：リポジトリ

集約の永続化管理を担当

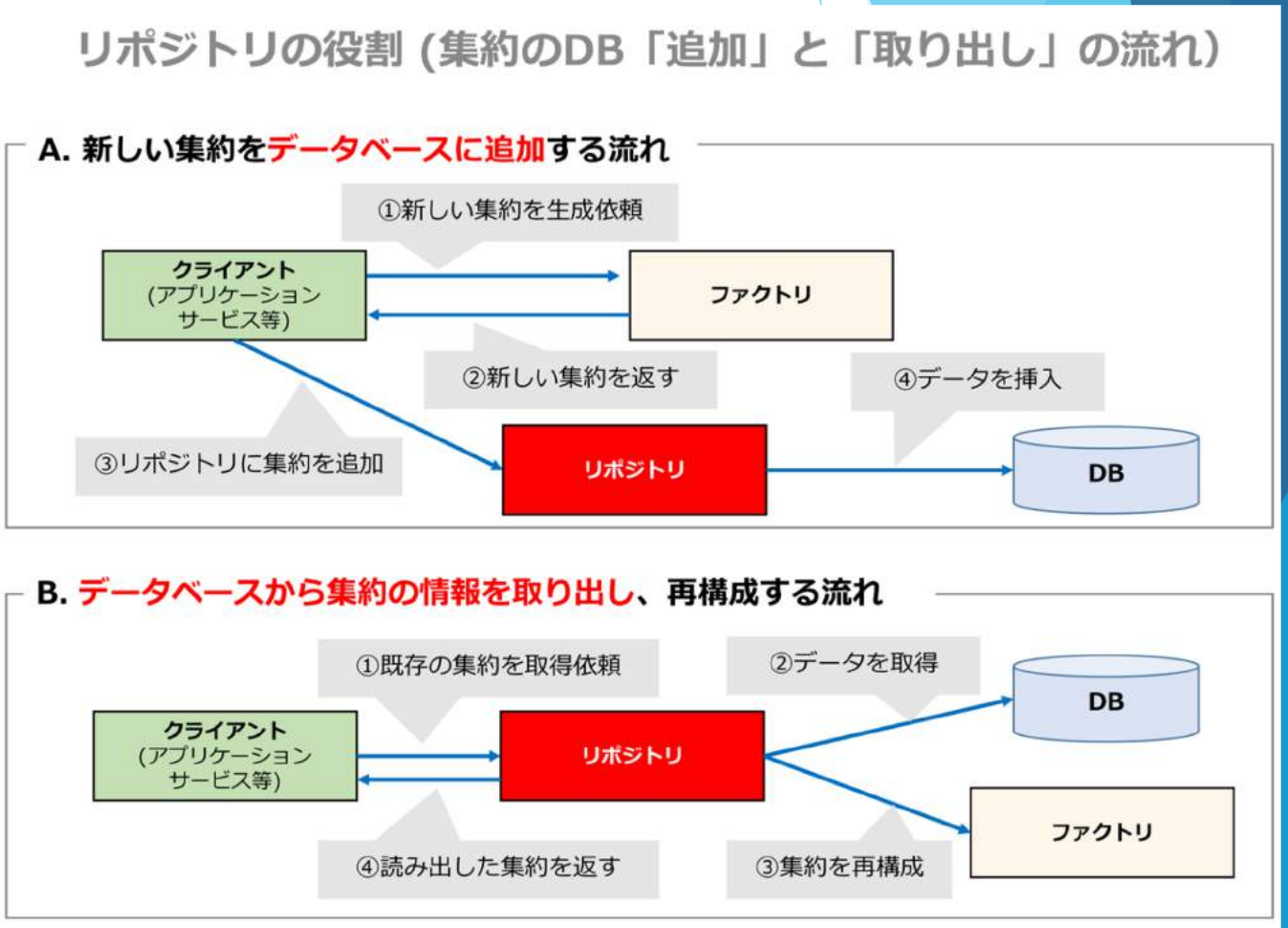
https://codezine.jp/article/detail/11048

リポジトリとは

- DDDにおけるリポジトリは、エンティティや値オブジェクトから構成される**集約の格納と取得**を担当します。
  - リポジトリは、**クライアントへ集約を提供し、背後のデータベースとのやり取りを隠ぺい**します。
  - **通常、集約とリポジトリの関係は一対一**になります。
  - 例えば「注文」の集約を利用したい場合「注文リポジトリ」を使用します。
  - クライアント側はリポジトリのおかげで、物理的な構成（RDBなのか、NoSQLなのか等）を意識せずに、簡単に集約を操作できます。

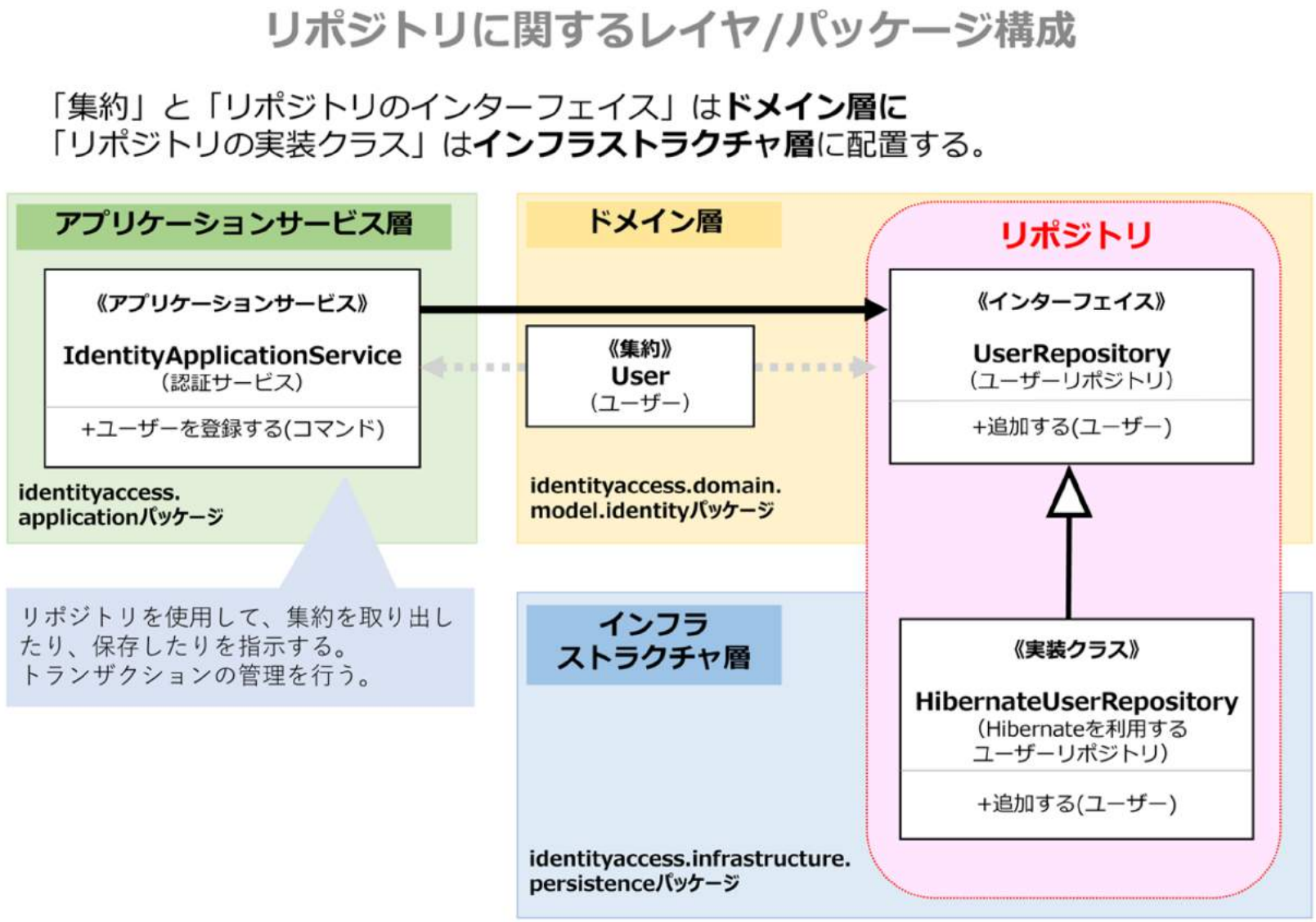
リポジトリで集約を操作する流れ

- 「A.新しい集約をデータベースに追加する流れ」の場合
  - クライアント側からファクトリを呼び出して集約を生成します。
  - そして、**リポジトリの追加メソッドを呼び出すことで永続化の対象として登録します**。
  - これによってデータベースに集約の情報が挿入されます。
- 「B.データベースから集約の情報を取り出し、再構成する流れ」の場合
  - **リポジトリはデータベースにクエリを投げてデータを取り出します**。
  - そしてファクトリを使って集約のモデルを再構成し、クライアントに集約を戻します。
  - 取り出した集約の変更内容はデータベースへ反映されます。



リポジトリのモジュール構成

- **リポジトリのインターフェイスは、集約と同じドメイン層**のパッケージに配置します。
- そして**リポジトリの実装クラスはインフラ層**に配置します。4章「アーキテクチャ」で紹介したDIP原則に従うことで、ドメイン層への影響なく、リポジトリの交換が可能になります。



「コレクション指向」と「永続指向」リポジトリ

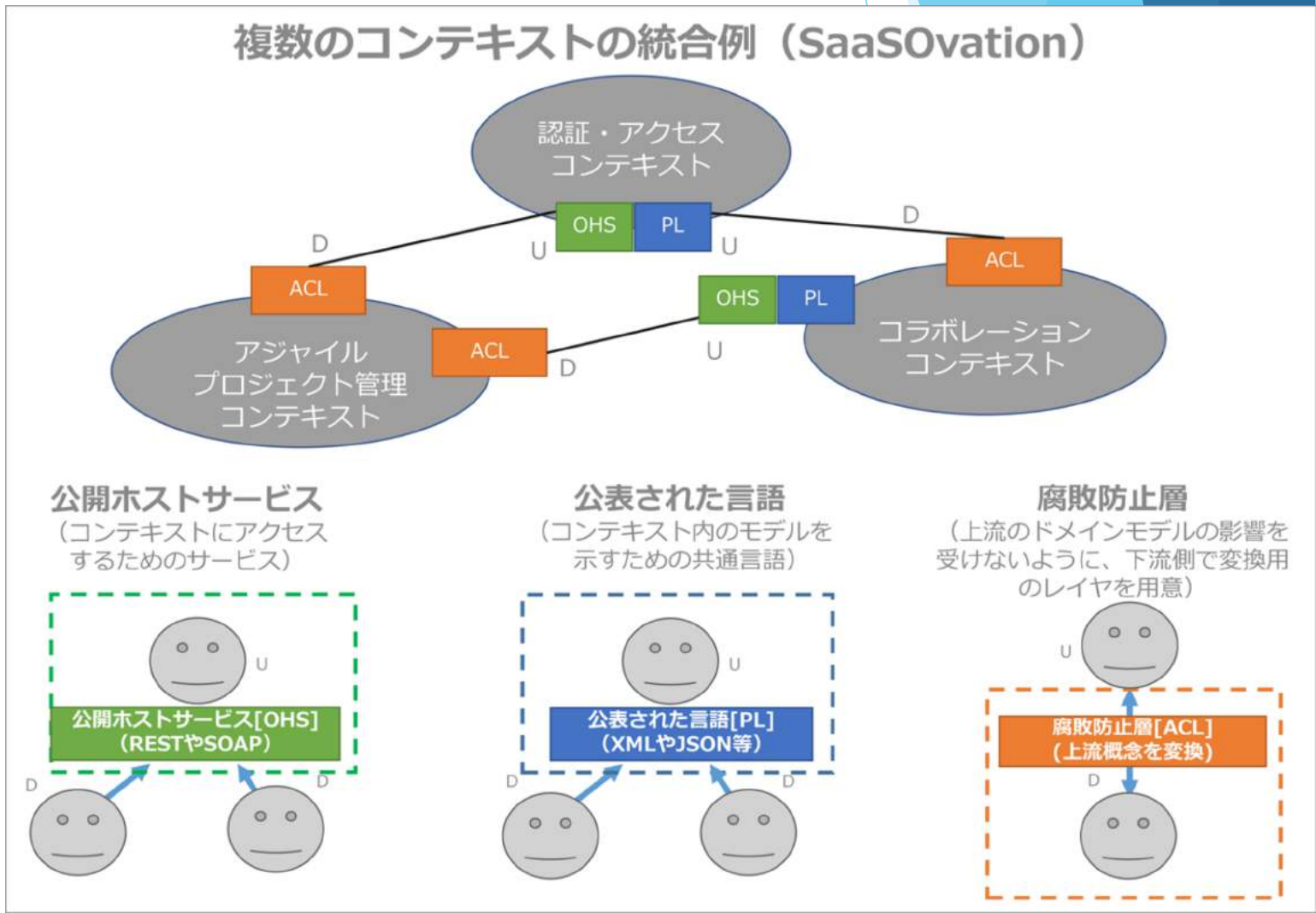
- 「コレクション指向のリポジトリ」
  - **集約の登録時に「追加メソッド（Add）」を用います。**
  - リポジトリは、登録された集約の情報をRDBに書き込みます。
  - IDDDではO/Rマッパー（Hibernate等）の仕組みでオブジェクトの変更を検知し、複数のテーブルに反映します。
- 「永続指向のリポジトリ」
  - **集約の作成／変更時に「保存メソッド（Save）」を呼び出します。**
  - リポジトリでは、保存メソッドが呼ばれたタイミングで、集約のキーを元に、NoSQLに保存します。
  - 集約の階層構造を1つのデータ（ドキュメント）として保存します。そのため、差分的な変更ではなく、集約全体を一括で更新します。

13章：境界づけられたコンテキストの統合

分散システム設計

分散システムの基本

- DDDではシステムをひとつの大きな固まりではなく、ビジネス的な境界線（ユビキタス言語の境界線）にあたる「境界づけられたコンテキスト」の単位で分割して管理します。
- 複数の境界づけられたコンテキスト間での連携する場合、**「分散システムの原則」**「データ交換フォーマット」**「Notification（通知）の仕組み」**「RESTアーキテクチャ」「メッセージングアーキテクチャ」について理解する必要があります。



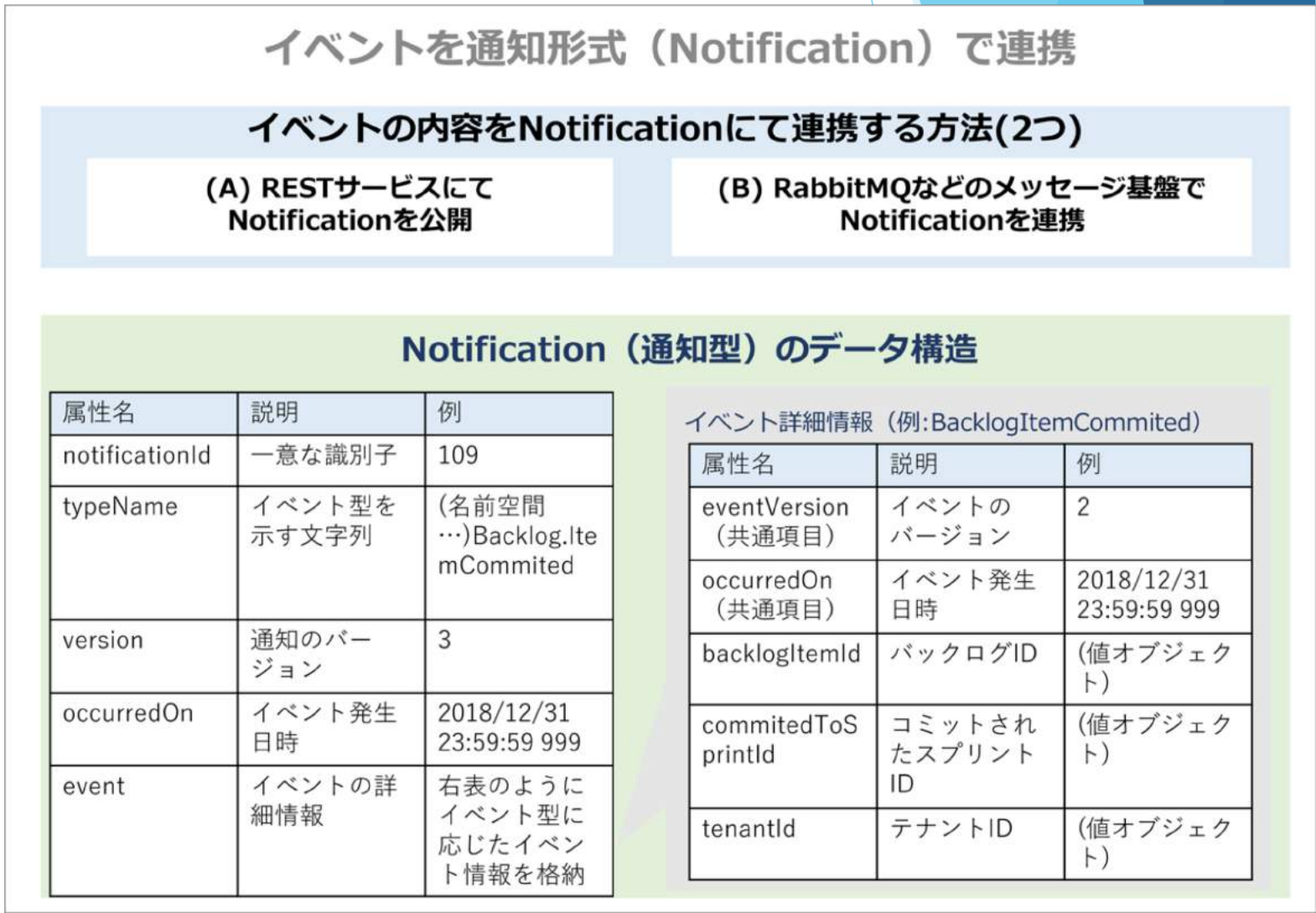
分散システム構築時の原則

- 分散システムの構築は一般的な集中システムに比べると簡単ではありません。
- そのため開発者は次の「**分散コンピューティングに関する原則**」を理解する必要があります。

1. ネットワークは信頼できない
2. ある程度の（時にはかなりの）遅延が常に発生する
3. 帯域幅には限りがある
4. ネットワークはセキュアではない
5. ネットワーク構成は変化する
6. 管理者は複数人存在する
7. 転送コストはゼロではない
8. ネットワークは一様ではない

イベントを連携する「Notification（通知）」の仕組み

- DDDで外部の境界づけられたコンテキストに情報を連携するには「ドメインイベント」を使用できます。
- IDDDでは、**このイベントを「Notification（通知）」という仕組みを用いて連携**します。
- RESTfulサービスで公開される場合もあれば、RabbitMQのようなメッセージング基盤を用いて連携される場合もあります。



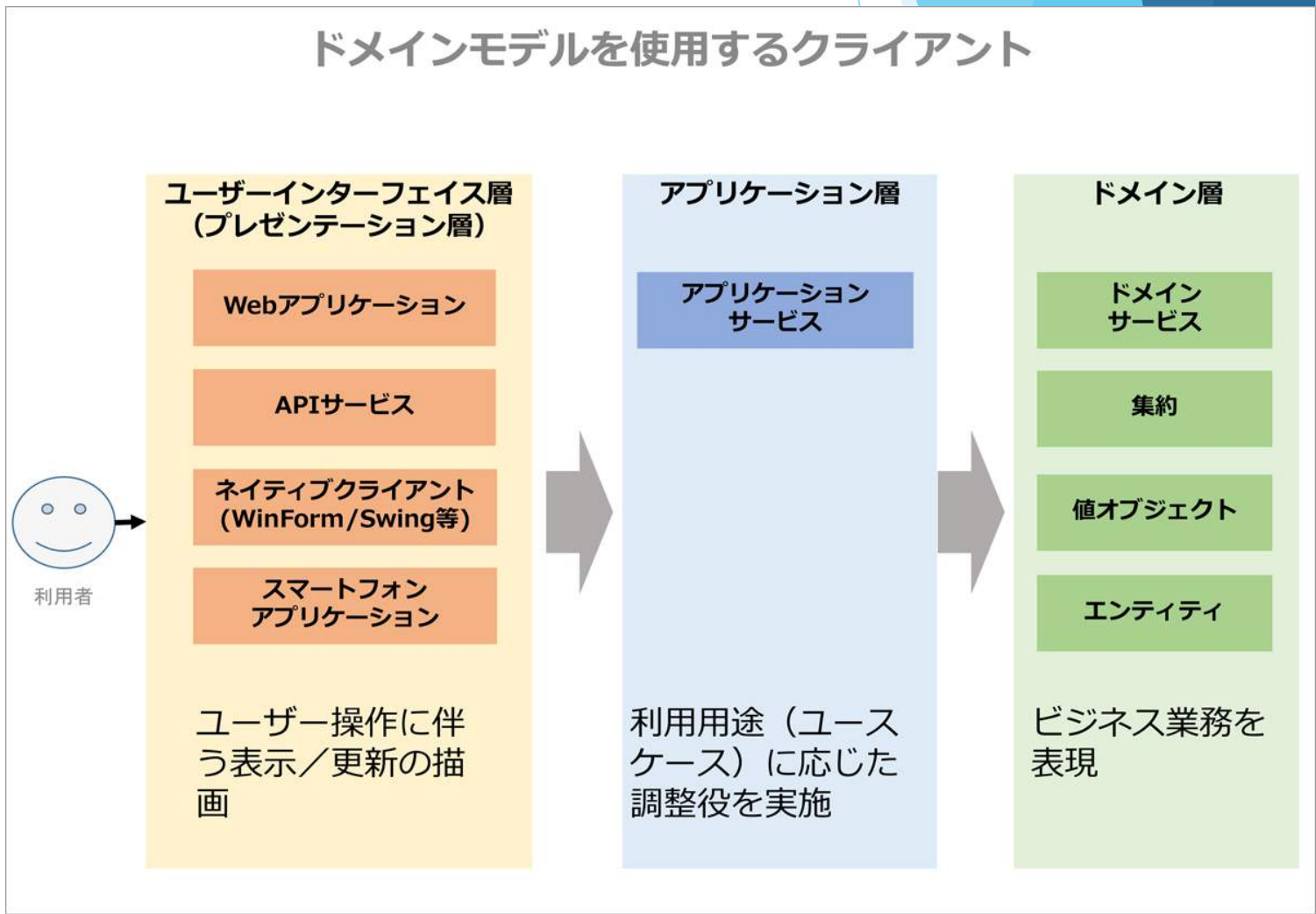
14章：アプリケーション

ドメインモデルを利用するクライアント

https://codezine.jp/article/detail/11221

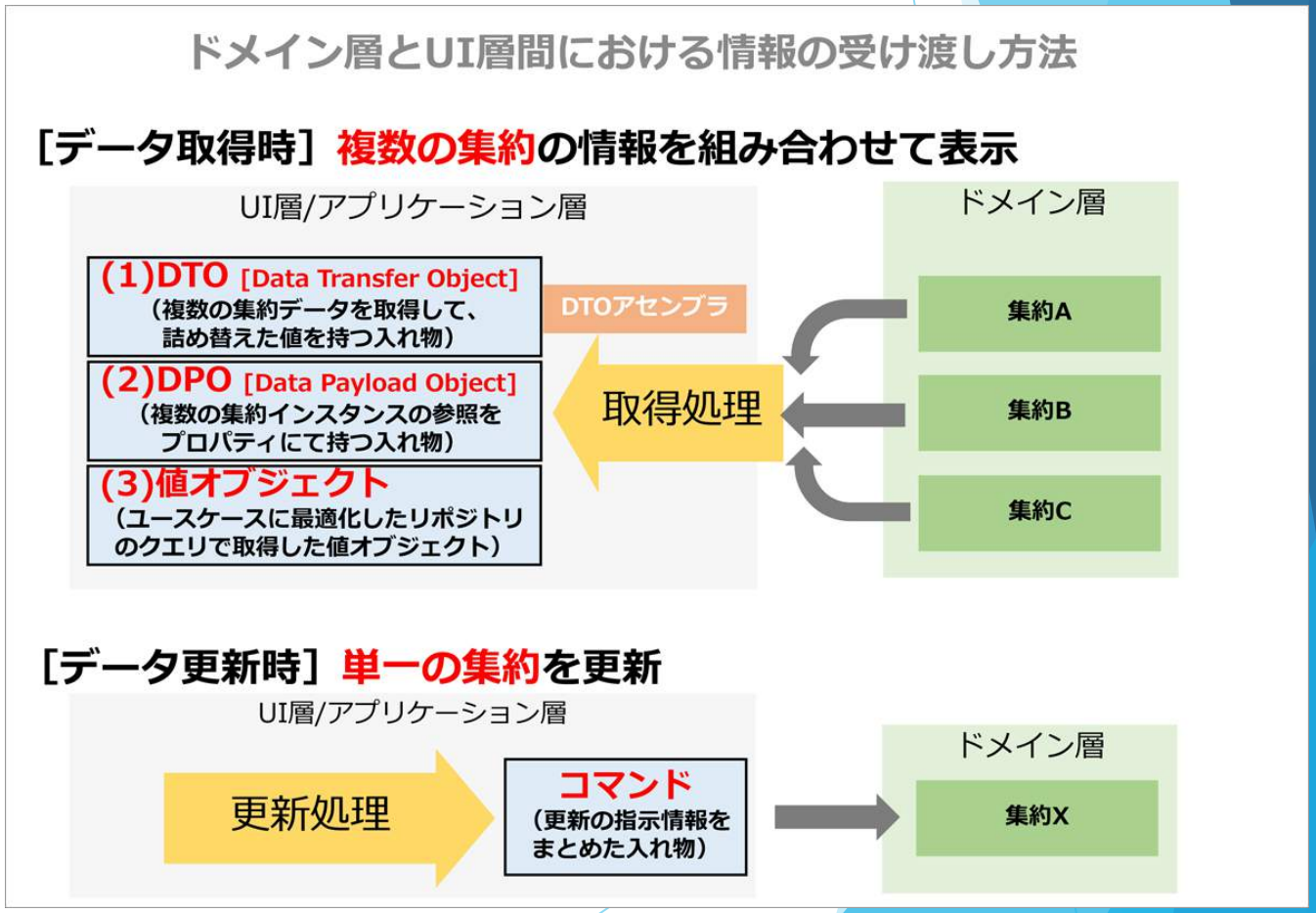
ドメインモデルを利用するアプリケーションとは

- 「アプリケーション」とは、広義の意味では「システム」全体と同じ意味となります
- ドメインモデルを使用するクライアントである**「ユーザーインターフェイス層」「アプリケーション層」**について紹介します。



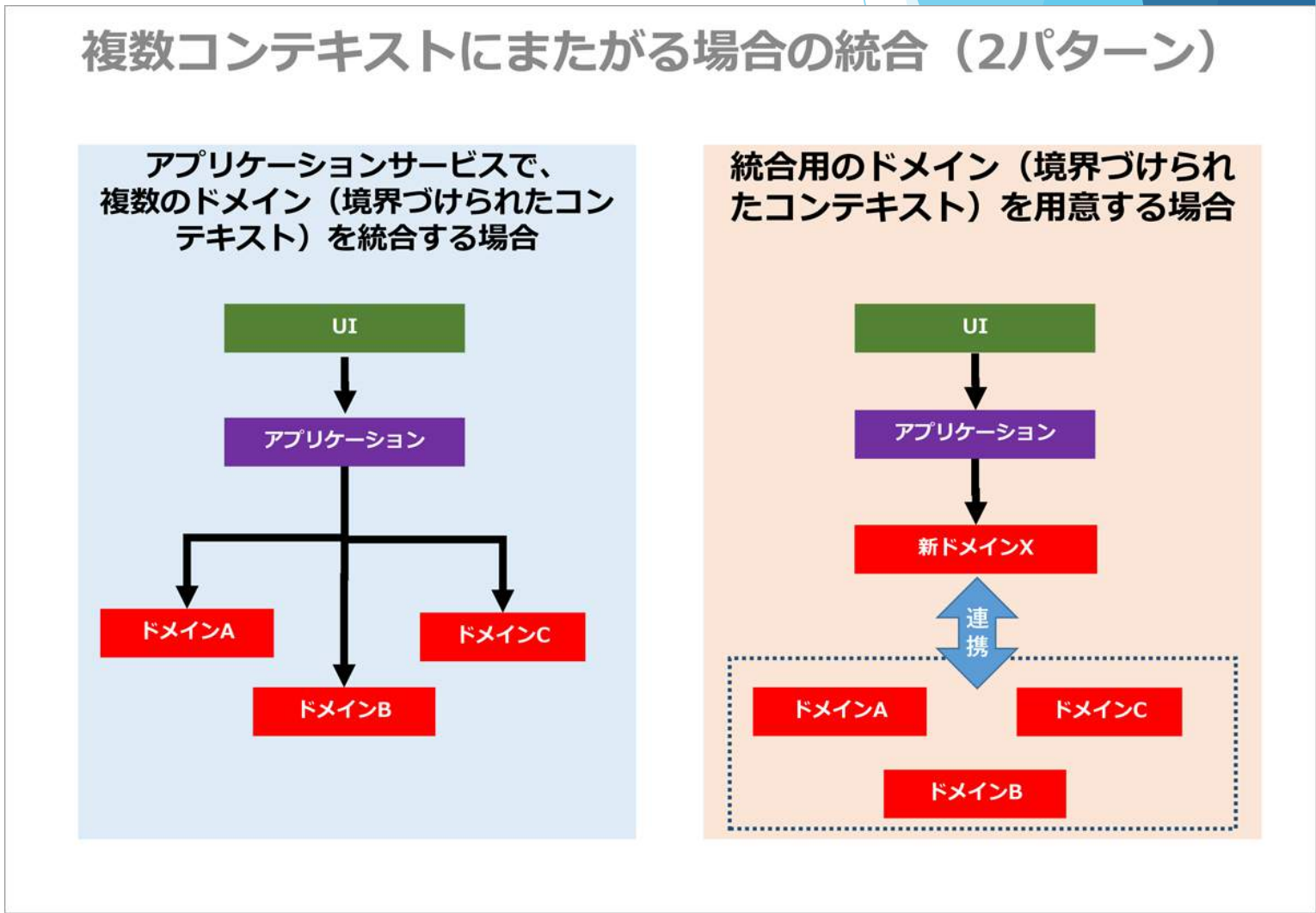
ドメイン層とUI層間における情報の受け渡し方法

1. 複数の集約情報を**DTO**へ詰め替え
2. 複数の集約へ参照を保持する**DPO**を返送
3. ユースケースに最適化した**クエリ**による取得



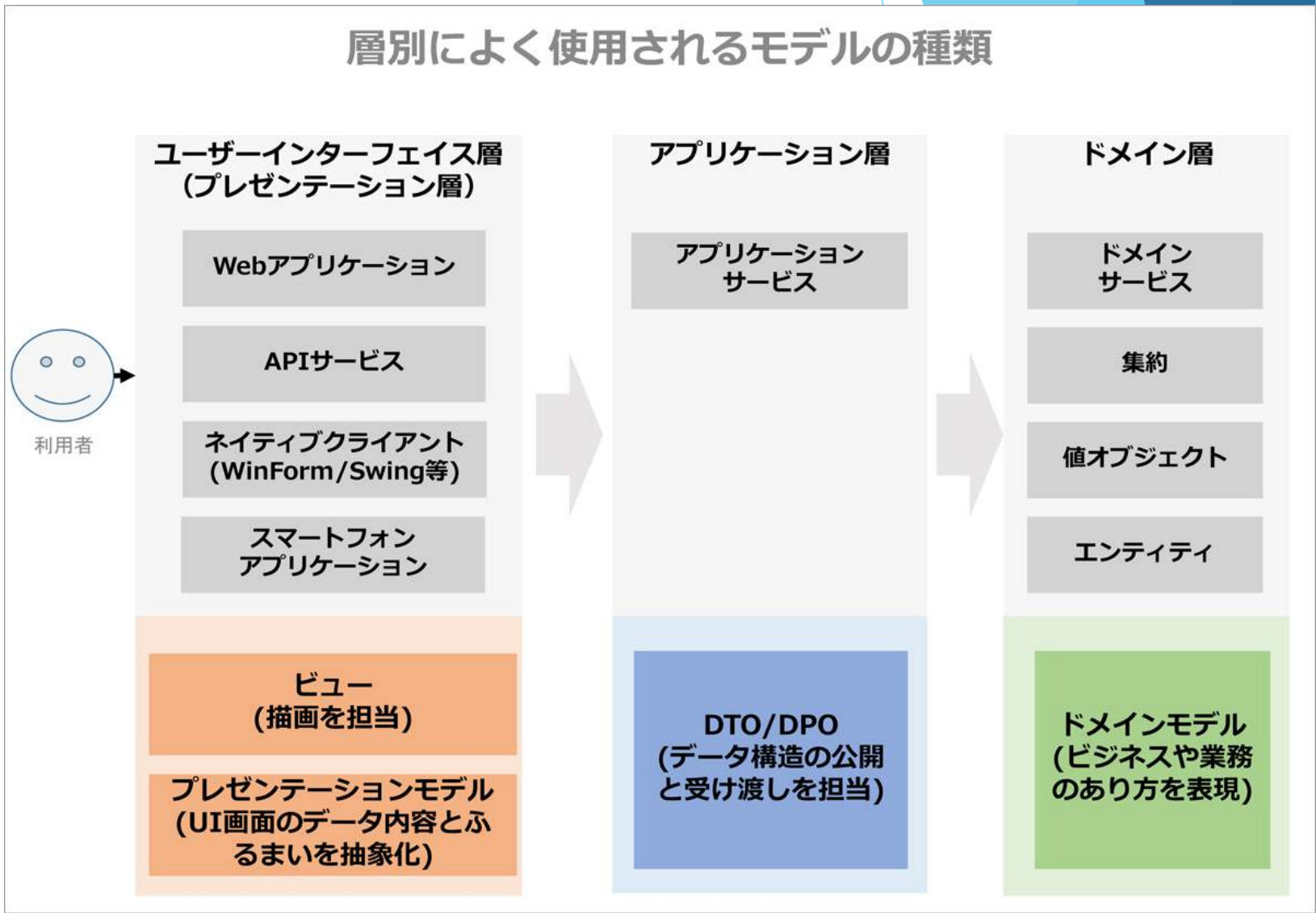
複数の境界づけられたコンテキストの統合・合成

- 1つのUI画面にて、**複数の境界づけられたコンテキストの情報が必要になる場合**があります。
- 例えば、3つのコンテキストをアプリケーションサービスでまとめる必要がある場合、大きく次の2つの方法があります。



UI層での描画に関わるモデル

- 「ビュー」と「プレゼンテーションモデル」
  1. UIの表示（描画）を行う「**ビュー**」
  2. UIの状態とUI固有ロジックを持つ「**プレゼンテーションモデル**」



UI層にドメインモデルを公開するか の判断

- **ドメインモデルを公開**
  - ユビキタス言語で使いやすく、バリデーションチェックが有効なモデルを利用できるといったメリットがあります。
  - しかし、UIとドメインモデルが**密結合**となるため相互に変更の影響を受けやすくなります。
- **ドメインモデルを非公開**
  - UI層は、DTOとプリミティブ型（Int,String等）のみを参照することになります。
  - これによりUIとドメインモデル間を**疎結合**にできますが、中間的なメッセージモデルへの詰め替えコードが冗長になってしまいます。
- IDDD本では好みや最終目標のトレードオフにて選ぶことを推奨しています。

項目	ドメインモデル公開時	ドメインモデル非公開時
性能	○（変わらない）	△（DTO詰め替えによるメモリ使用とガベージコレクションコストが発生）
依存関係	△（ドメインモデルへの依存性が高く結合度が高い）	○（ドメインモデルへの依存がなく結合度が低い）
型による妥当性チェック	○（使える）	△（使えない）
コード量	○（増えない）	△（DTO部分や詰め替え部分が増える）
複数クライアント時の対応	△（ドメインモデルにおいて影響がないように複数クライアントを考慮する）	○（複数クライアント別にDTOを用意するためドメインモデルに影響がない）

ご清聴ありがとうございました。

このセッションが皆様の開発の一助になれば幸いです。