

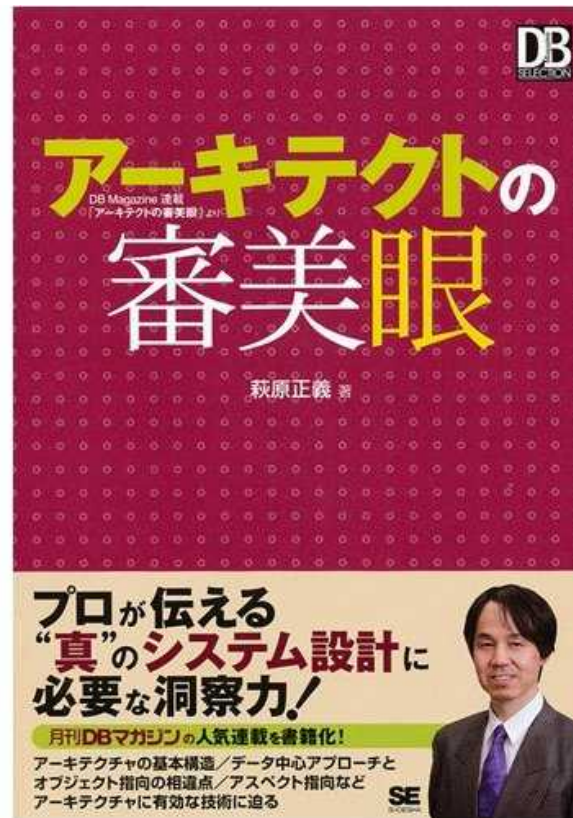
# アーキテクトの審美眼

萩原 正義

マイクロソフト株式会社

# タイトル

- クラウドになっても不変のアーキテクチャの原則
- 哲学と呼べる信念の確立



# アジェンダ

- アーキテクチャ先行定義の原則
  - データ類型化とデータアーキテクチャ
  - ドメインモデルとアプリケーションアーキテクチャ
- 抽象化
  - マルチパラダイムの進化
  - モデル要素の分割と複合化
  - パラダイムの選択: 設計モデル要素、チームモデル、分析アプローチ、計算モデルなど
- 意思決定プロセス

# アーキテクチャ実現プロセス

## Architecture-driven (解決駆動)

構造、振舞い (DOA, OOAD)

モデリング  
(論理コンポーネント)

構造、振舞い (patterns, architecture)

アーキテクチャ設計

- プラットフォーム
- フレームワーク
- データモデル
- トランザクションモデル
- セキュリティ
- スケーラビリティ
- 可用性
- 運用管理
- 保守性
- 再利用性
- ...

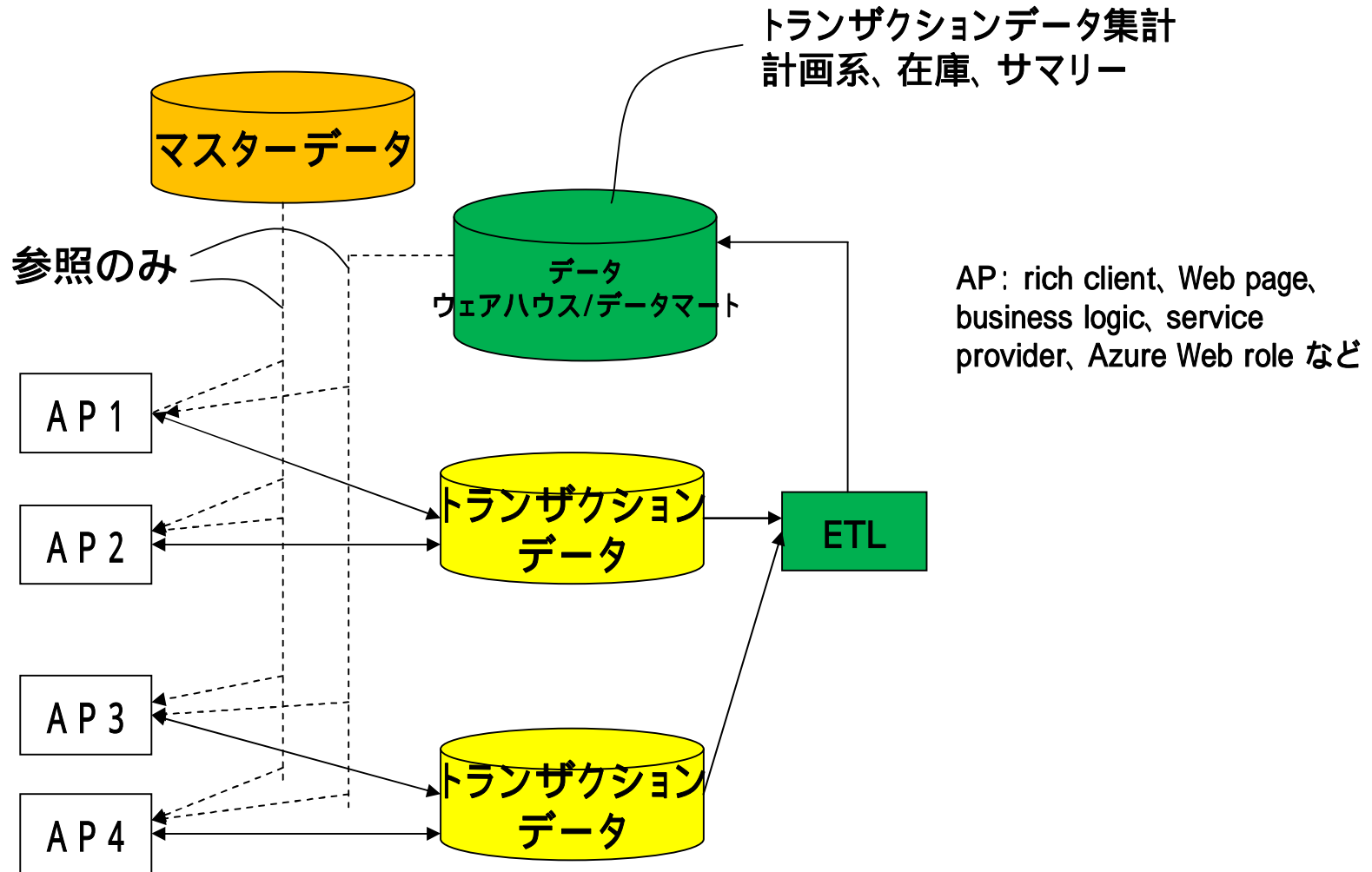
マッピング、設計

- 機能要件と非機能要件の分離、複合化による検証
- Architectural patterns を使ったマッピング
- 設計時での tradeoff の解決

実装  
(物理コンポーネント)

配置  
(物理 tier)

# データアーキテクチャの原則



# アーキテクチャの原則(1)

- Entity\* Relationship

データ項目を先行して発見し、業務の意味的グループから Entity の粒度と Relationship を同時に決定

- 意味的なグループは他との相対的關係からしか決まらない
- Relationship は関数従属性、ビジネスから決まる

Entity はアプリケーションから独立して、先行して定義(データアーキテクチャ)

アプリケーションが Entity 間の關係を決める

# アーキテクチャの原則(2)

- ドメインモデル

アプリケーションの設計モデルはデータモデルから独立して定義

ビジネスモデルをそのまま反映：ドメインモデル(POJO/POCO、naked object、Qi4j、DCI Architecture ...)

アプリケーションの設計モデルのコアはユースケースに独立して、先行して定義

# 縦と横の構造の原則

## アーキテクチャ

- **縦の構造 (資産)**

安定した保守管理可能な構造: 非冗長化、複雑さと変化のカプセル化  
OO パラダイム (継承と委譲による複合化)

“Why”に基づくアーキテクチャ

拡張点と拡張性 (variability point と variant)

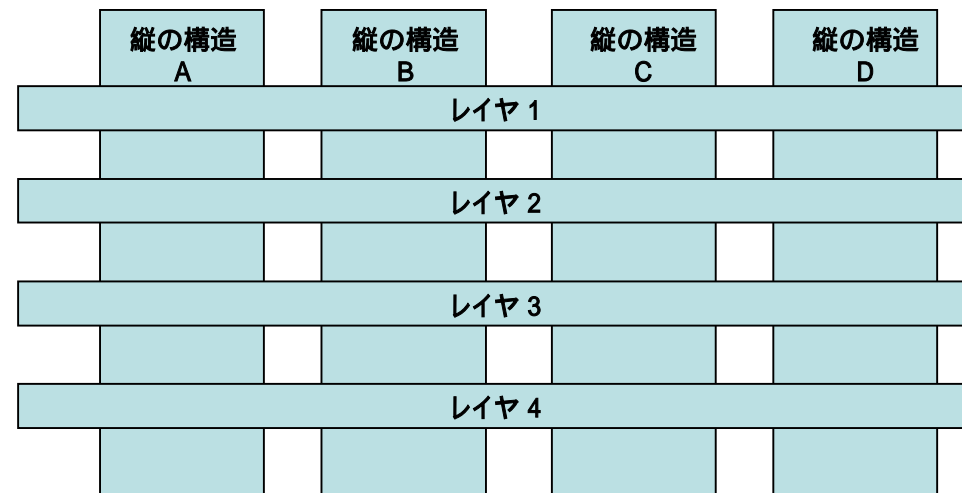
可変性には binding-time の仕様化

- **横の構造 (要求)**

横断的関心 (ユースケース、SOA サービス、非機能要求、特性)

可変性

アスペクト指向パラダイム



# ソフトウェア開発パラダイム



1970代 1980代 1990代 現在 2009+

計算機  
の発展

ネットワー  
ク  
の発展

データセン  
ター  
の発展

ビジネス  
の発展

アプリケー  
ション  
間連携

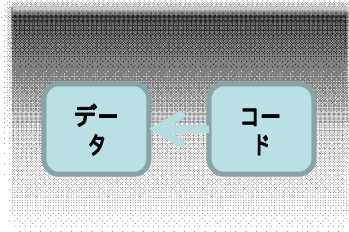
分散化

ユビキタス

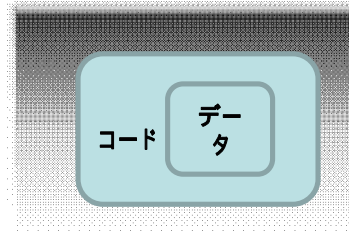
意味レベル  
抽象化

部品化

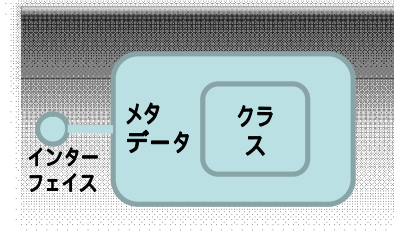
自律と協調



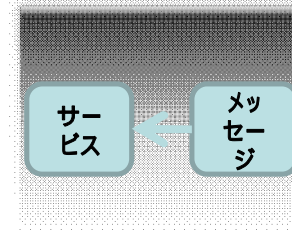
構造化



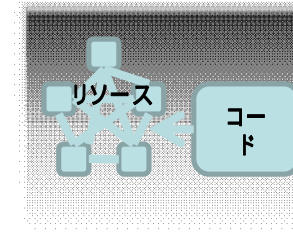
オブジェク  
ト指向



コンポーネン  
ト指向



サービス  
指向



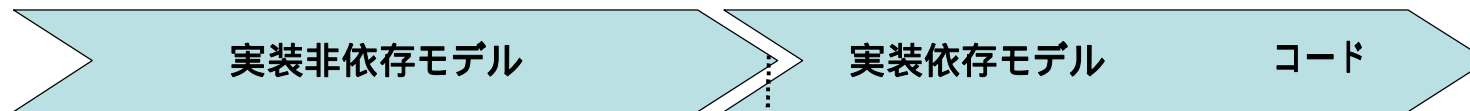
Web  
指向

# Viewpoints に依存するモデル

	ビジネス	情報	アプリケーション	インフラ
概念	<ul style="list-style-type: none"> <li>ユースケース</li> <li>シナリオ</li> <li>ビジネスゴールと目的</li> </ul>	<ul style="list-style-type: none"> <li>ビジネスエンティティと関係</li> </ul>	<ul style="list-style-type: none"> <li>ビジネスプロセス</li> <li>サービス分割</li> </ul>	<ul style="list-style-type: none"> <li>サービス配布</li> <li>QOS戦略</li> </ul>
論理	<ul style="list-style-type: none"> <li>ワークフローモデル</li> <li>役割定義</li> </ul>	<ul style="list-style-type: none"> <li>メッセージスキーマ</li> <li>文書仕様</li> </ul>	<ul style="list-style-type: none"> <li>サービス相互作用</li> <li>サービス定義</li> <li>オブジェクトモデル</li> </ul>	<ul style="list-style-type: none"> <li>論理サーバタイプ</li> <li>サービスマッピング</li> </ul>
物理		<ul style="list-style-type: none"> <li>データベーススキーマ</li> <li>データアクセス戦略</li> </ul>	<ul style="list-style-type: none"> <li>詳細設計</li> <li>インフラ技術依存設計</li> </ul>	<ul style="list-style-type: none"> <li>物理サーバ</li> <li>導入済みソフトウェア</li> <li>ネットワークレイアウト</li> </ul>

# Software Artifact (モデル)

## 分類、分割し複合化



### モデル要素

- クラス、インターフェイス
- コンポーネント
- サービス
- ビジネスルール
- ロール
- フィーチャ、アスペクト
- モジュール
- エンティティ
- プロセス
- パターン
- アーキテクチャ...

# マルチパラダイム

- Procedural
- OO
- Aspect
- Component (CbD)
- SOA
- WOA
- Declarative
- Functional
- Concurrent
- Model-driven
- ...

# オブジェクト指向パラダイム

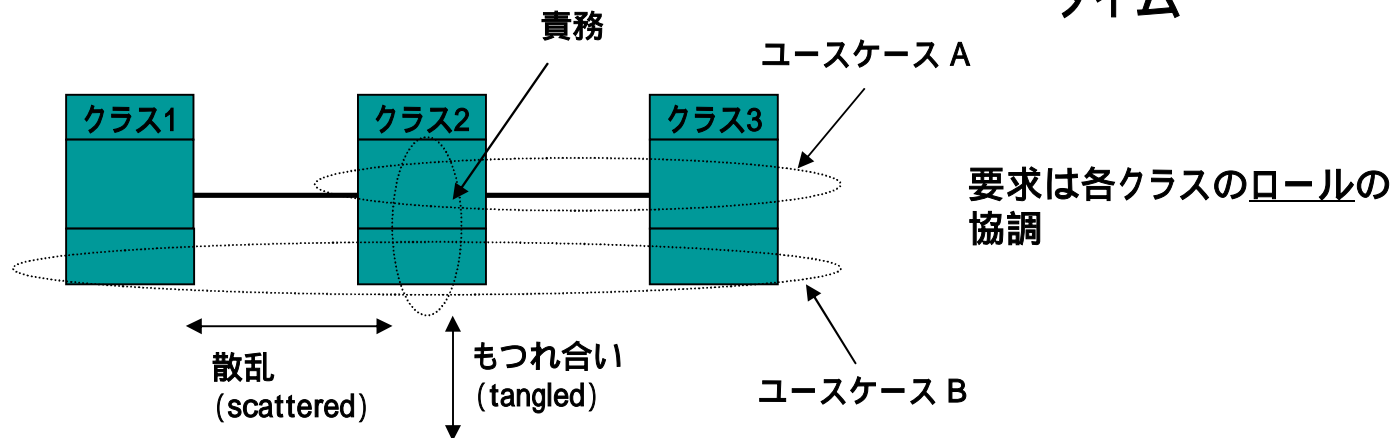
## 散乱ともつれ合い

- 要求(テスト)の変更

ユースケースはクラスとは散乱ともつれ合いの関係

ユースケース単位のモジュール化(同時更新)技術

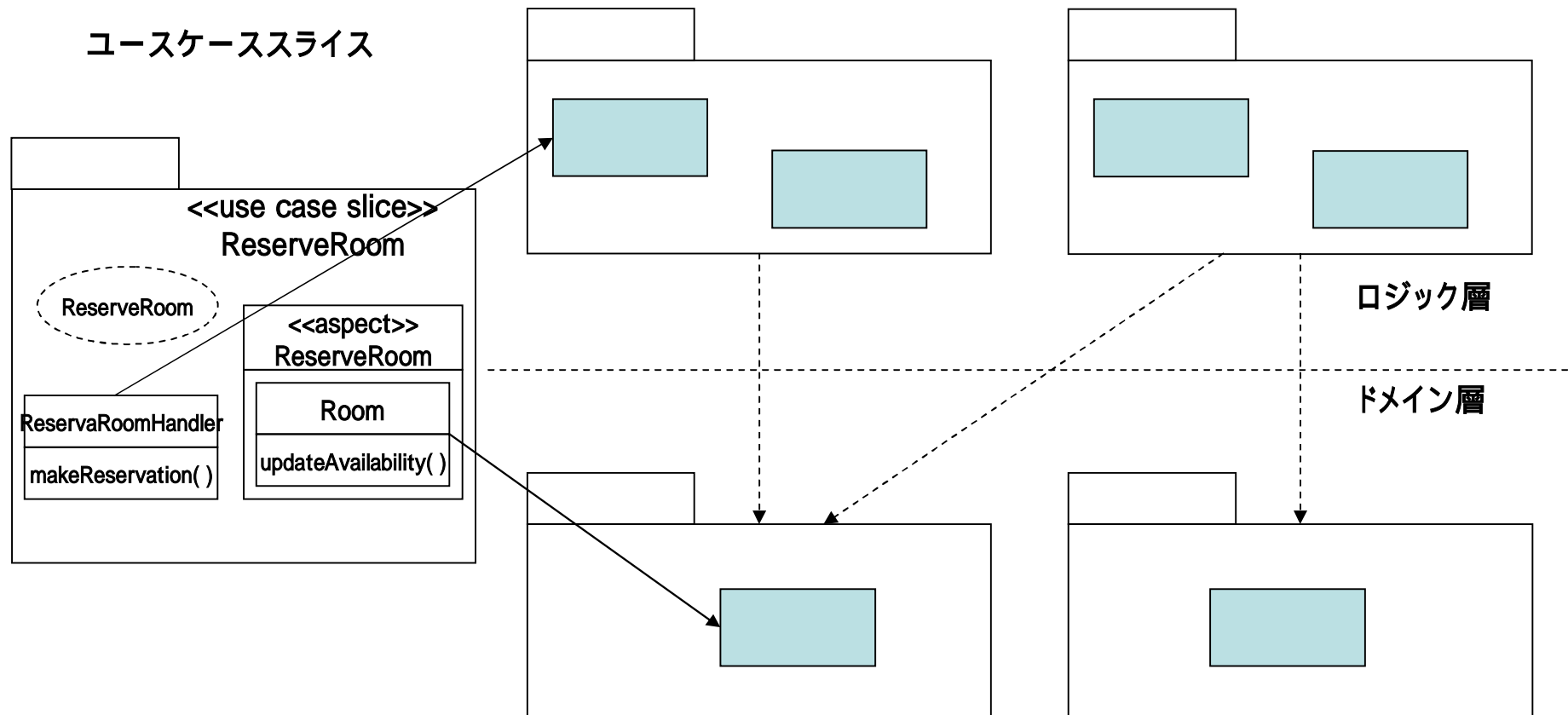
- 要求を関心ととらえると関心の分離の原則に帰着
  - 要求同士重複、矛盾、依存関係があり、関心の分離が困難
  - 既存の OOP での実装が困難: 関心はクラスより大きく、クラスより小さい
- 可変性の決定時期; バインディングタイム



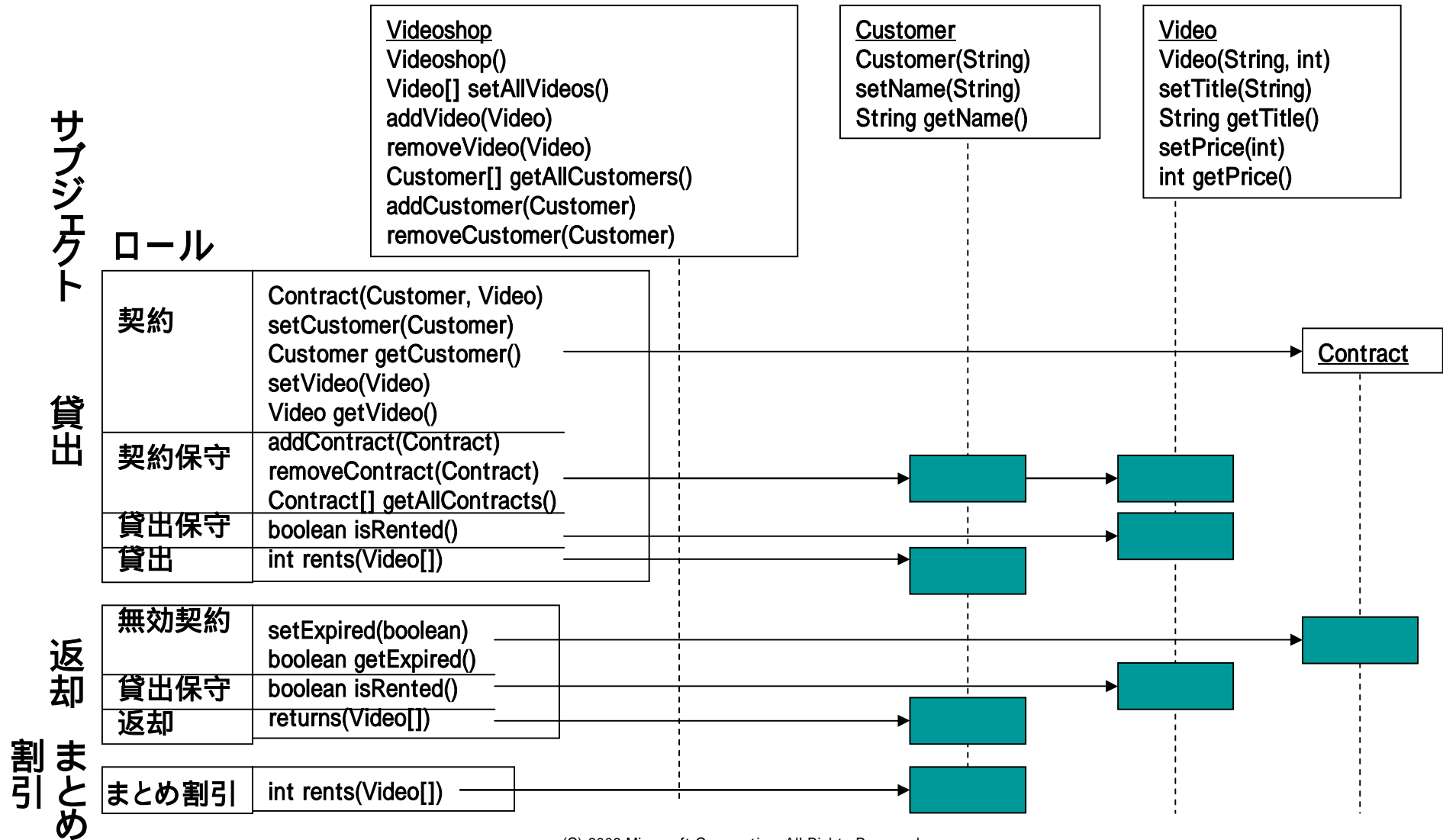
# アスペクト指向パラダイム

## 可変性単位-ユースケーススライス

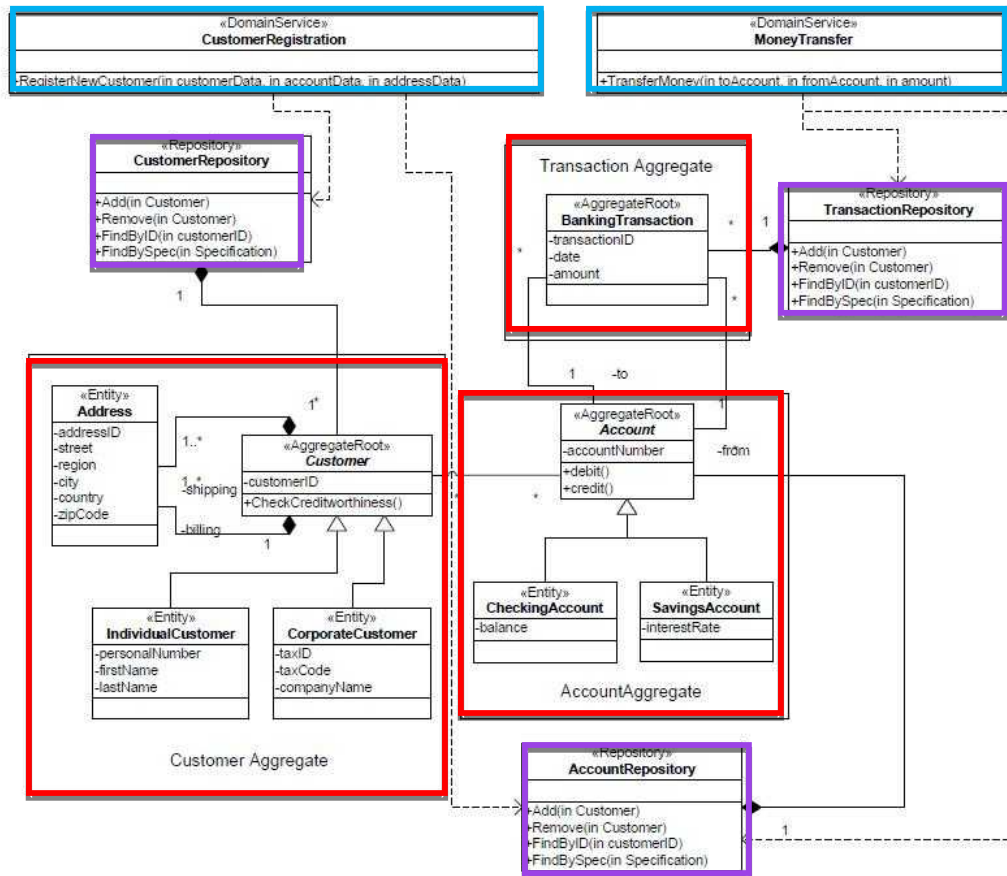
設計モデル要素構造



# サブジェクト指向パラダイム

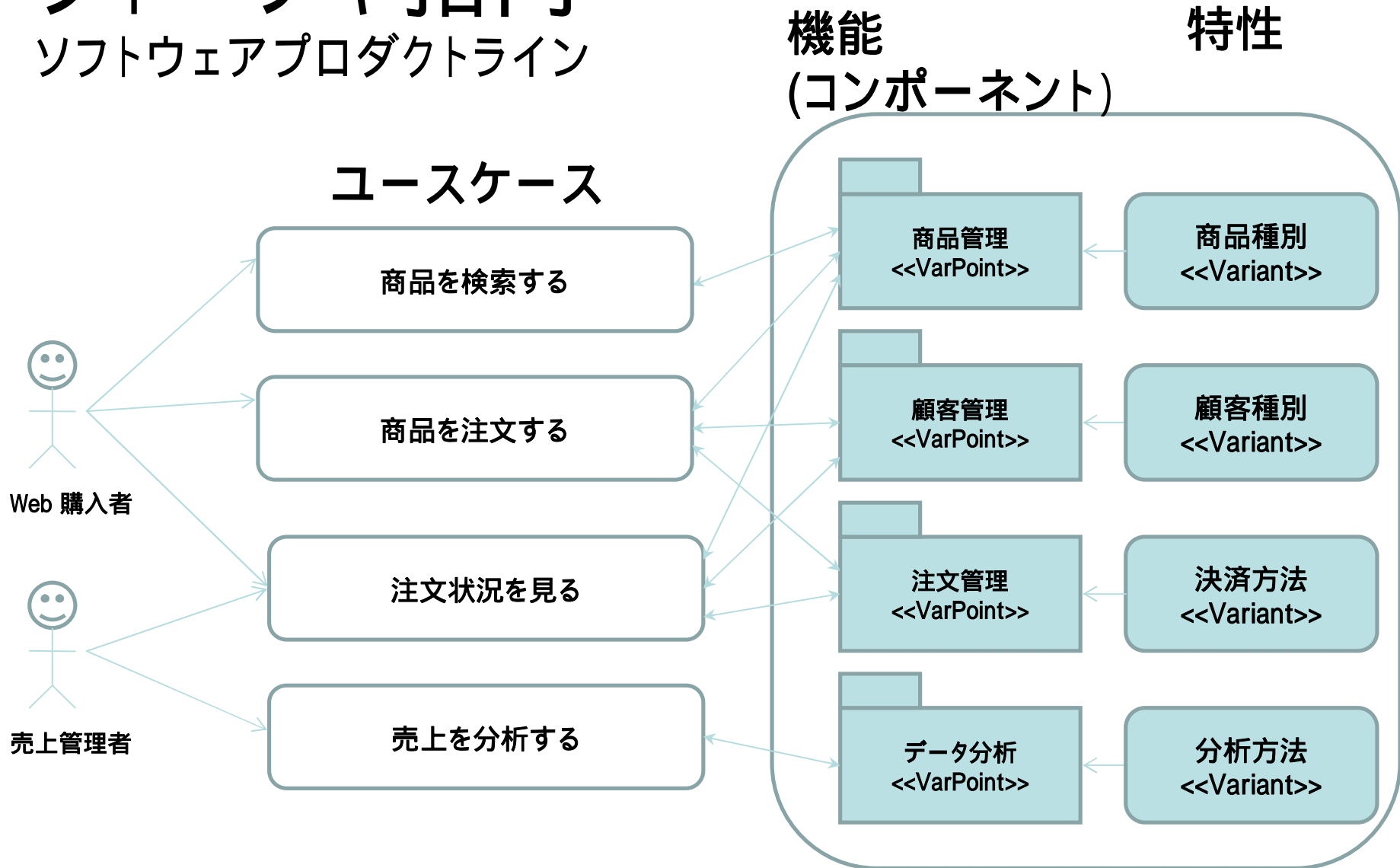


# ドメインドリブン設計

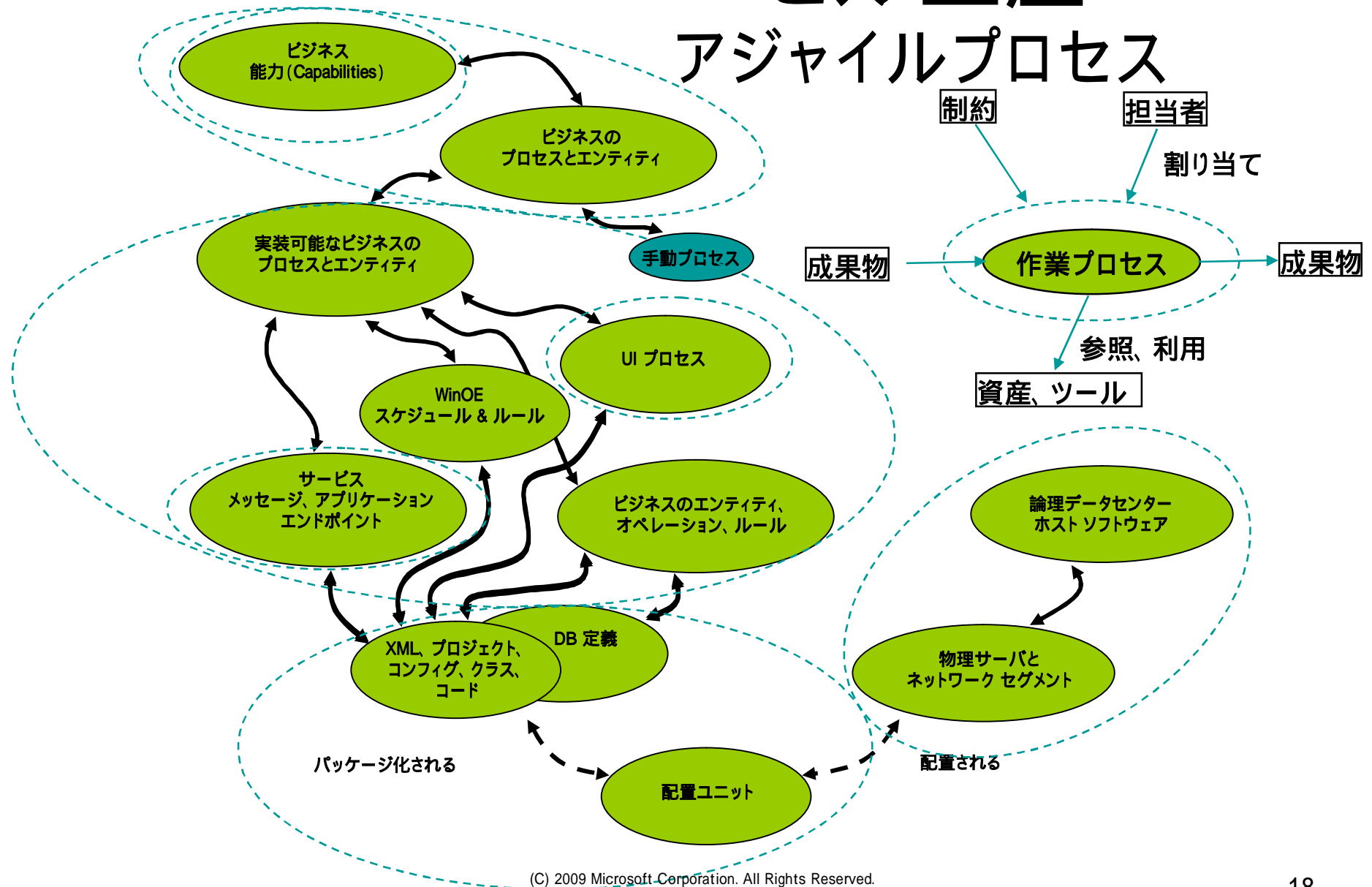


出典: Using ADO.NET Entity Framework in Domain-Driven Design: A Pattern Approach

# フィーチャ指向 ソフトウェアプロダクトライン

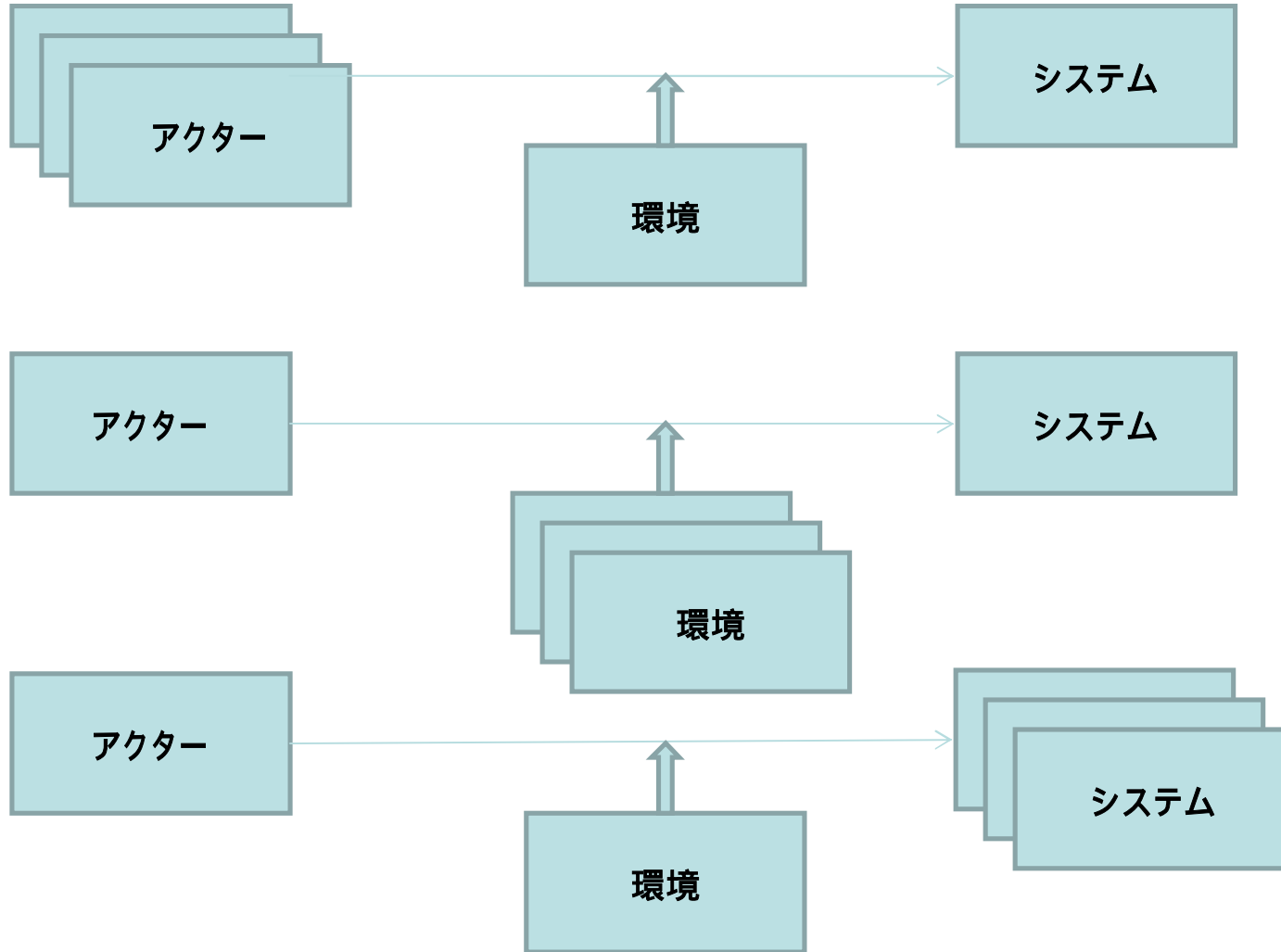


# セル生産 アジャイルプロセス



# コンテキスト指向

4次元 (method, target, source, context)



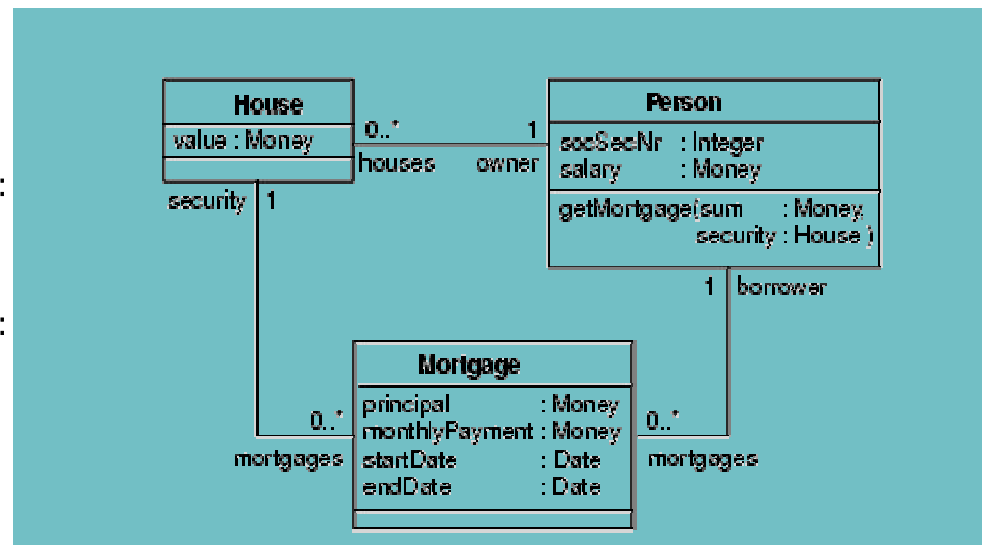
# 宣言型パラダイム (制約)

- 制約

1. 本人所有の場合に限り、住宅ローンを借りられる
2. ローンの開始日付は終了日付より前ではない
3. 所有者の SSN はユニーク
4. 新規ローンは債務者の収入が十分な場合に限る
5. 新規ローンは住宅の価値が十分な場合に限る

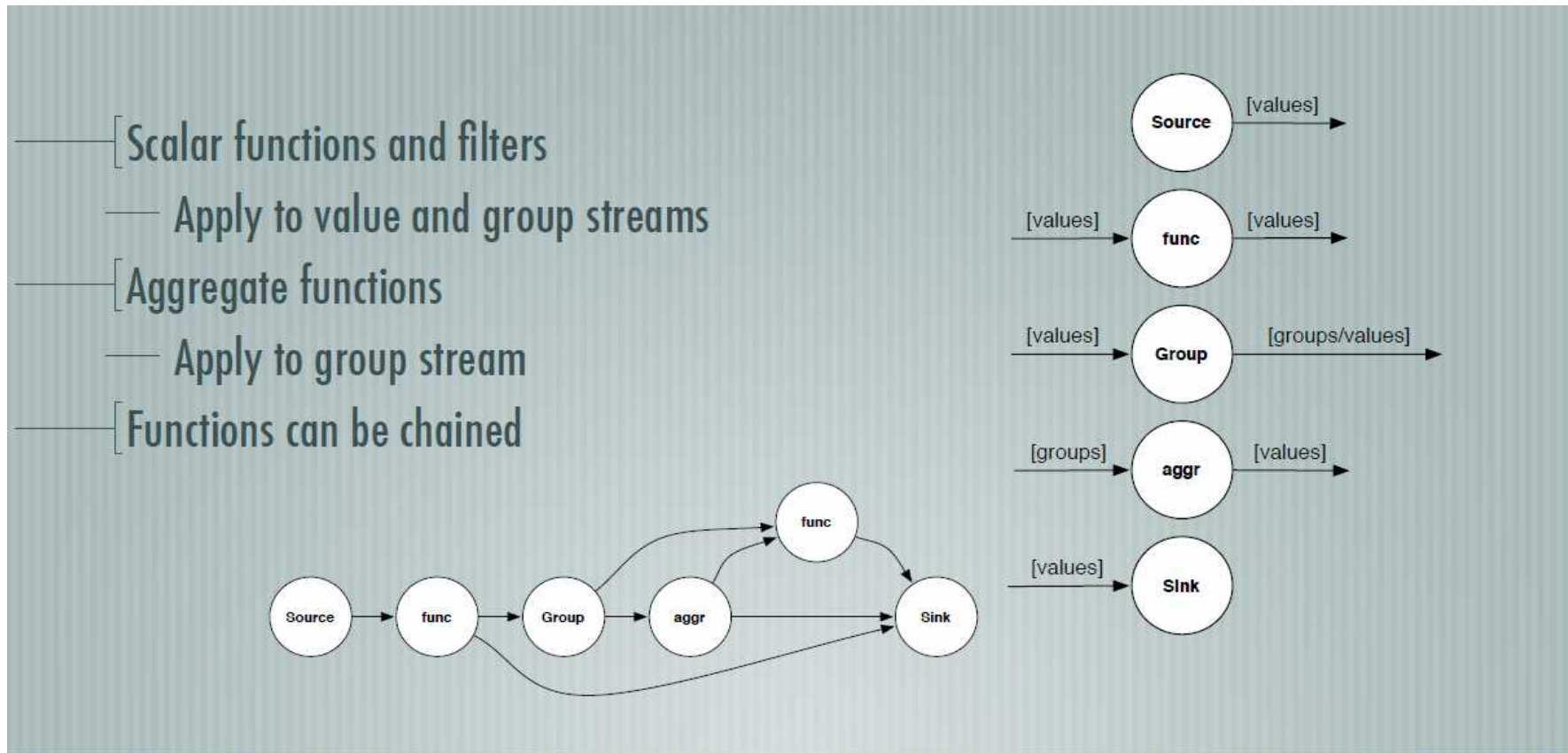
- OCL

```
context Mortgage
inv: security.owner = borrower
context Mortgage
inv: startDate < endDate
context Person
inv: Person::allInstances()->isUnique(socSecNr)
context Person::getMortgage(sum : Money, security :
House)
pre: self.mortgages.monthlyPayment->sum() <=
self.salary * 0.30
context Person::getMortgage(sum : Money, security :
House)
pre: security.value >= security.mortgages.principal-
>sum()
```



# 関数型パラダイム

## データストリームによる Unit of Work



# パラダイムの途中変更は困難

## 選択と複合化

- Scale-up vs. Scale-out
- データモデル (Relational, XML, OO)
- 同期 vs. 非同期
- Domain model vs. transaction script
- OOAD vs./with AOSD
- EDA、ワークフローなどの導入
- ソフトウェアプロダクトライン vs. context-ware ソフトウェア

# Domain Specific Paradigm

- パラダイムの選択はドメイン単位
- パラダイム適用はスナップショット  
時代とともに選択は変わる  
概念モデルの分類や体系化も同じ
- パラダイム変更は Agile では扱いにくい
- ドメインとパラダイム選択の洞察力

