「ソフトウェア工学」

(2) システム分析・設計

東大理学部情報科学科講義

石川　冬樹（本務先：国立情報学研究所）
f-ishikawa@nii.ac.jp / @fyufyu
http://research.nii.ac.jp/~f-ishikawa/

大学共同利用機関法人 情報・システム研究機構
国立情報学研究所
National Institute of Informatics

目次

- システム分析
- 設計原則
- アーキテクチャ設計
- 詳細設計
- デザインパターン

システム分析

- System Analysis （システム分析）
  - 要求およびドメイン（概念）のモデルに基づき，「機能をどうシステムとして実現するか」に関するシステムモデルを構築する
  - ただし抽象的な本質にとどめ，実装詳細はまだ定めない
- Robustness Analysis（ロバストネス分析）
  - システム分析の代表的な手法
  - 要求，特にユースケースの実現を考えることで，それらの妥当性確認を行い，ロバスト（頑健）にする
  - これからの設計の第一歩となる指針を示す

システム分析：全体像

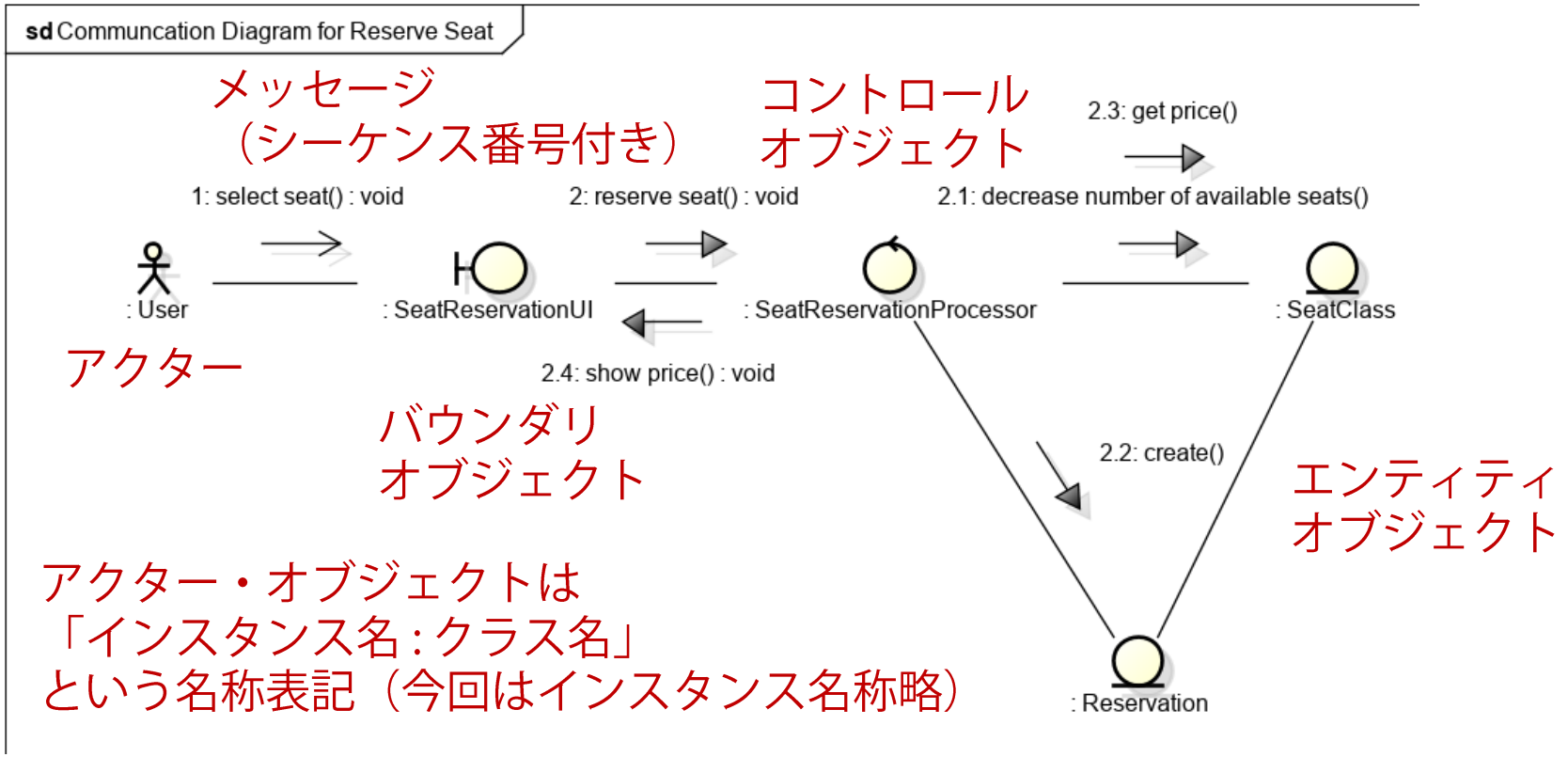
1. 各ユースケースにおいて，システムによる実現の流れを分析，定義
   - ロバストネス分析の場合，システムの構成要素として，バウンダリ，コントロール，エンティティ3つの役割からクラスを定め，各ユースケースにおける相互作用を分析する
2. ユースケースごとの定義をシステム全体へと統合し，各構成要素が提供する操作や構成要素間の関係を整理，確認
   - 別のビューとして，例えばクラス図でも整理する

ロバストネス分析：3種類のクラス

- システムの構成要素として下記の3つを考え，各ユースケースの実現方法を抽象的に定める
  - バウンダリ： アクターとシステムのインターフェースとして働くクラス
  - コントロール： バウンダリからの刺激（入力）に応じ，エンティティや他のコントロールに働きかけるクラス
  - エンティティ： 情報を維持的あるいは恒久的に保存するクラス

ロバストネス分析：ユースケースに対する分析の例 (1)

- 航空券予約に関するユースケースでの例
  - UMLコミュニケーション図において，ロバストネス分析の3種シンボルを使っている



ロバストネス分析：留意事項

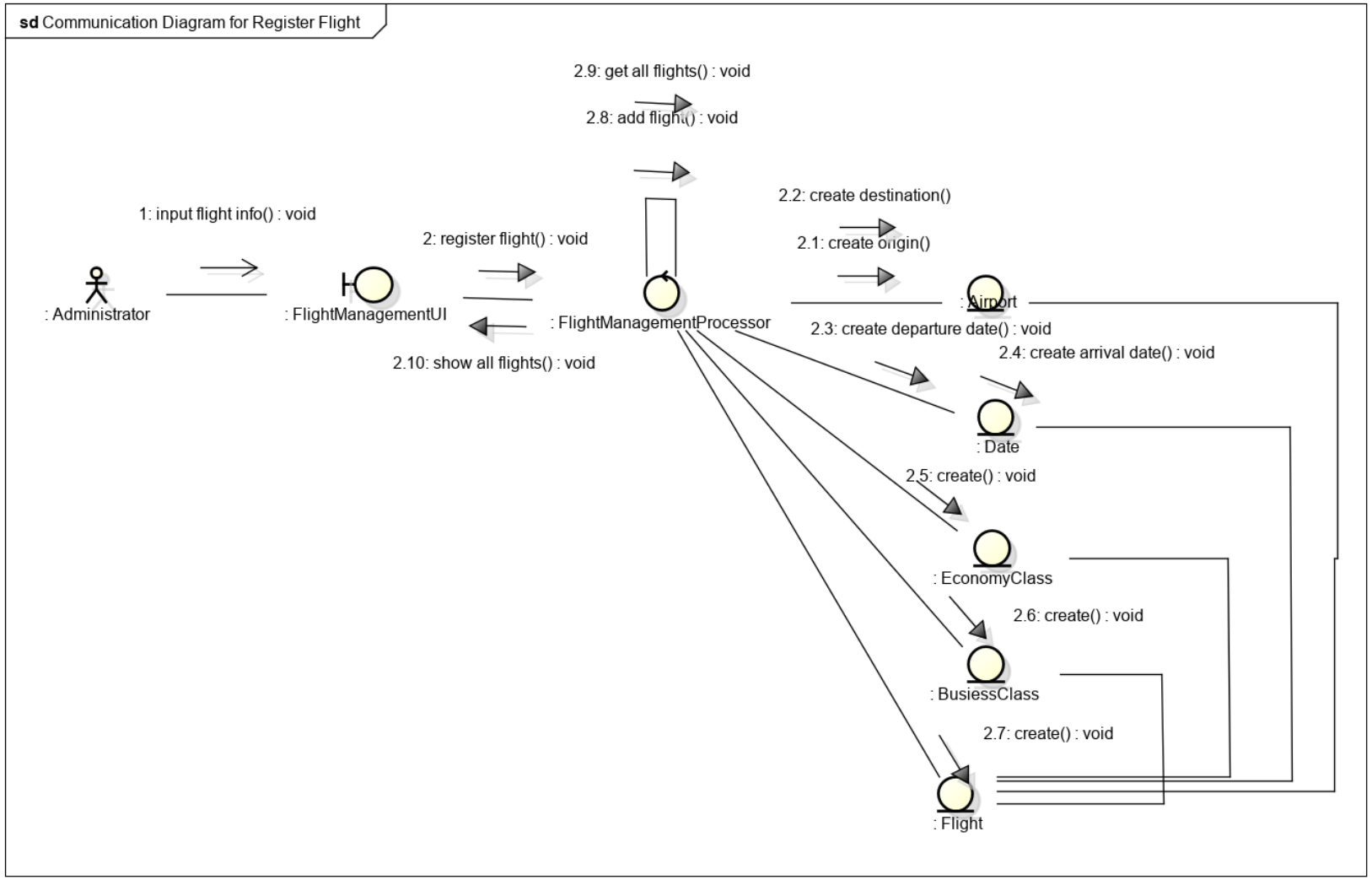
- 高い抽象度にとどめている
  - プログラミング言語固有の慣習などは気にしていない
    （命名規則，新規オブジェクト生成の方法等）
  - データ型は規定していない
  - 実装を考えるとナイーブな部分もある
    - 今回での例： オブジェクト間が双方向に呼び出し合っている
      → 実装コードではこの相互依存性は望ましくなく，
      単に戻り値とその処理として実現すべき

システム分析：全体への統合

- ユースケースごとのロバストネス分析で得られたクラス定義を統合
  - ロバストネス分析で得られた各メッセージから，それを受け取るクラスが提供する機能（操作）の定義が得られる
  - ユースケース間で共通のクラスと固有のクラスがある
  - ドメイン分析で得られた概念は，エンティティとなっているべき
  - クラス図をビューとして用いるとわかりやすい
  - ユースケースに関係のない初期化処理などは考えない
  - クラス間の関連における依存方向は未決定

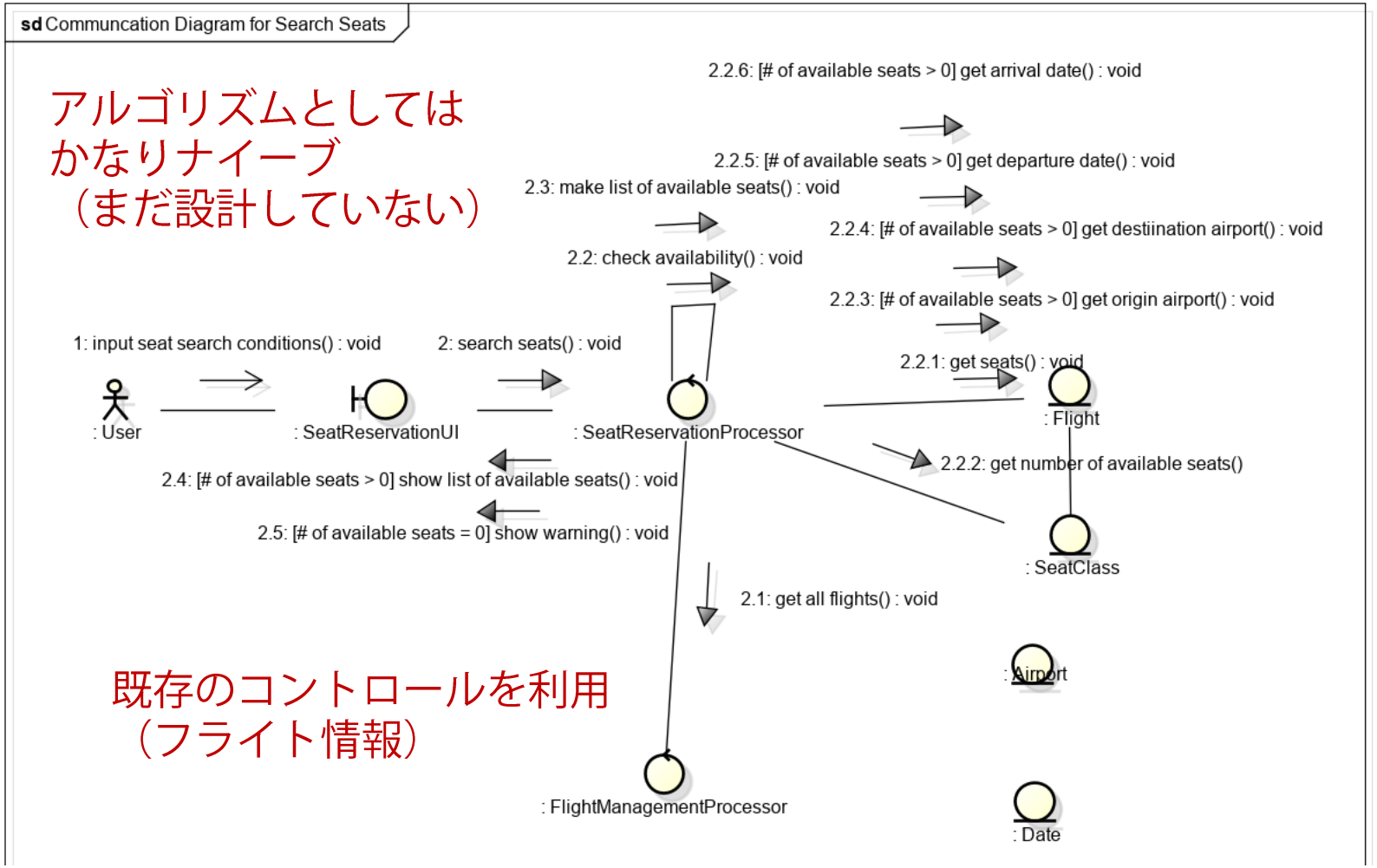
ロバストネス分析：ユースケースに対する分析の例 (2)

■フライト登録

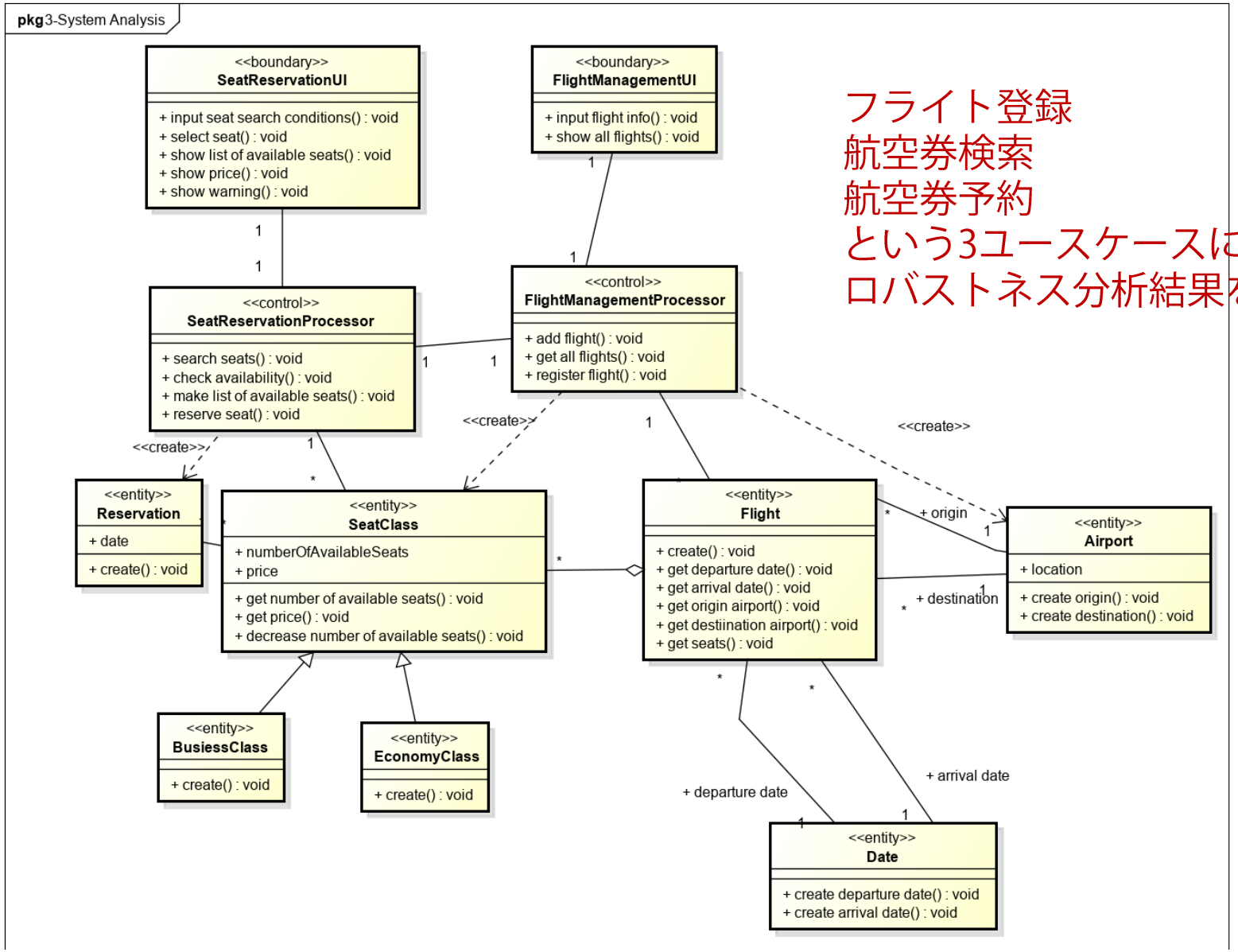


ロバストネス分析：ユースケースに対する分析の例 (3)

■航空券検索



ロバストネス分析： 全体の統合結果



フライト登録
航空券検索
航空券予約
という3ユースケースに対する
ロバストネス分析結果を統合

目次

- システム分析
- 設計原則
- アーキテクチャ設計
- 詳細設計
- デザインパターン

設計

- 設計（Design）
  - どのように要求を実現するか（How）の本質を定めていく
  - 特に非機能要求（Non-Functional Requirements）を反映する
  - アーキテクチャ（Architecture）レベルの設計と
    部品（Component）レベルの設計

設計原則

（様々な粒度・抽象度において）

- カプセル化（Encapsulation）
- 情報隠蔽（Information Hiding）
- 抽象化（Abstraction）
- モジュール化（Modularization）
- 分割統治（Divide-and-Conquer）
- 凝集・結合の考慮（Cohesion and Coupling）
- 関心事の分離（Separation of Concerns）

[Buschmann et al., Pattern-Oriented Software Architecture, Wiley, 1996] を参考に改

カプセル化・情報隠蔽

- カプセル化（Encapsulation）
  - 構造・振る舞いをグループ化して抽象的な要素として定義し，アクセスのためのインターフェースを設計する
  - 情報隠蔽，抽象化，変更容易性，再利用性に寄与
- 情報隠蔽（Information Hiding）
  - 部品の実装詳細を，そのクライアントから隠蔽する
  - クライアントは内部を知らなくてよい
  - プロバイダは，クライアントに影響を与えることなく，内部を変更することができる

抽象化

■抽象化（Abstraction）

ある定義

“An abstraction denotes the essential characteristics of an object that distinguish it from all other kinds of object and thus provide crisply defined conceptual boundaries, relative to the perspective of the viewer.”

[Booch, 1991]

結合度

結合度（Coupling）

- 表では，上の方がモジュール間の依存関係が高く，それは悪いことだと考えられる

Content	A module uses and alters data in another
Control	A modules communicate with another by passing a control flag to affect the behavior
Common	Two modules communicate through global data
Stamp	Two modules communicate by passing a shared structure (including redundancy)
Data	Two modules communicate by passing the minimum required data

凝集度

凝集度（Cohesion）

- 表では，上の方ほど要素間の関連性が低く，それらが同じモジュールにあるのは悪いことだと考えられる

Coincidental	Elements are unrelated
Logical	Elements have similar activities (e.g., all I/O actions)
Temporal	Elements perform in similar timing (e.g., initialization)
Procedural	Elements work sequentially (on different data) (e.g., a part split from a flow chart)
Communicational	Elements work on the same input
Informational	Elements cover all the functions on one data structure
Functional	Elements are related with each other to perform one function

分離

- 関心事の分離（Separation of Concerns）
  - 異なる，互いに無関係な責務は，異なる部品に割り当てられるなど分離されるべきである
- インターフェースと実装の分離
  - 既出

- 結局のところ，各担当者が知るべきこと，
  そして変更時の影響を最小にとどめることが重要

目次

- システム分析
- 設計原則
- アーキテクチャ設計
- 詳細設計
- デザインパターン

アーキテクチャ

- アーキテクチャ（Architecture）：

  構成される部品，部品が持つべき特性，部品間の関係を定めたもの
  - ステークホルダー間のコミュニケーションを促進
  - トレードオフがあるデザインの意思決定（Design Decision）について明示化，比較
  - リスクを明確化
  - 「アーキテクト」というエンジニアの職務区分もある

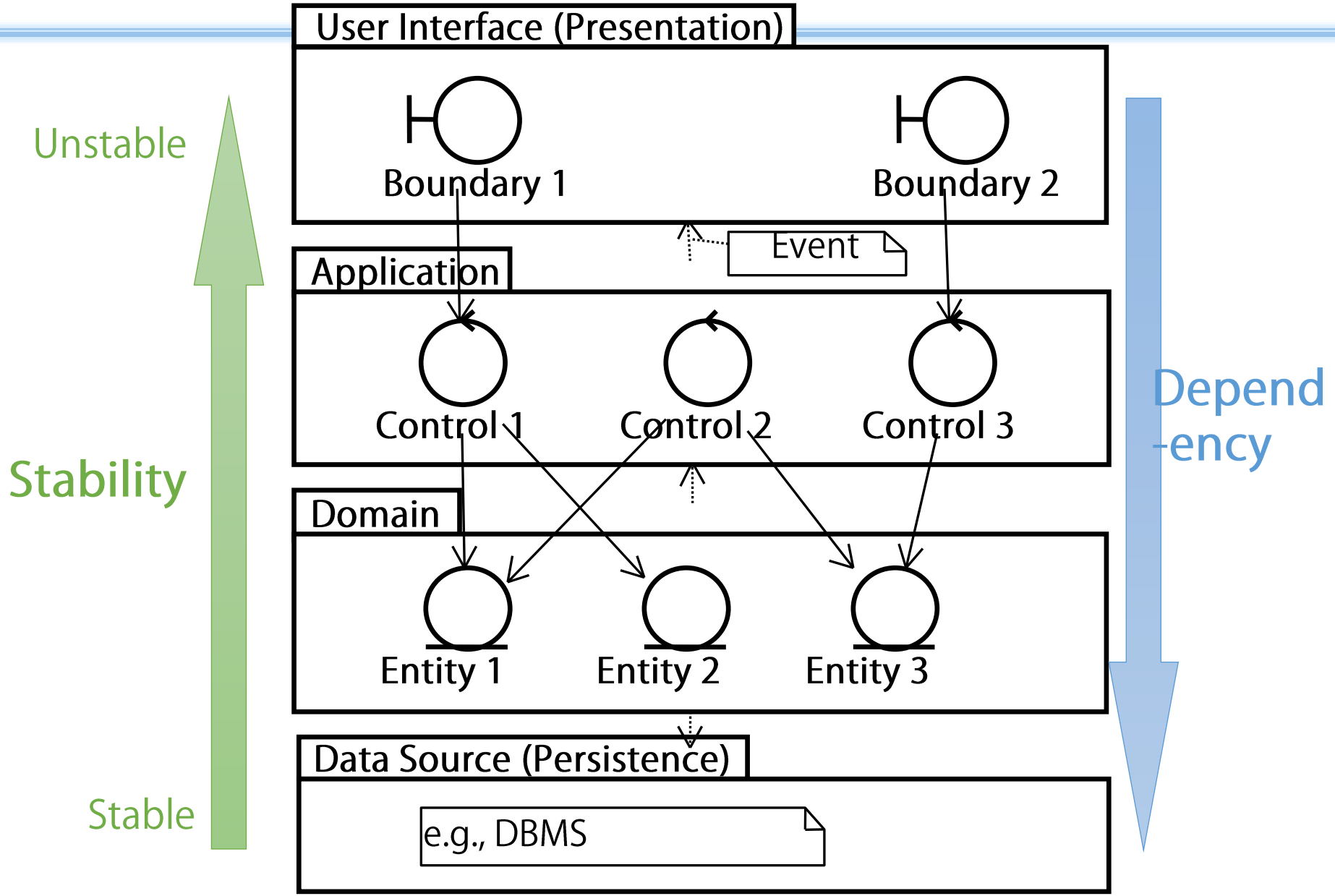
アーキテクチャパターン

- アーキテクチャパターン（Architecture Patterns）
  - 頻出する問題への典型的な解法としての
    汎用化されたアーキテクチャ
  - 例
    - 多層アーキテクチャ（共通原則）
    - MVCアーキテクチャ（人と相互作用するシステムのため）
    - ブローカーアーキテクチャ（分散システムのため）

多層アーキテクチャパターン

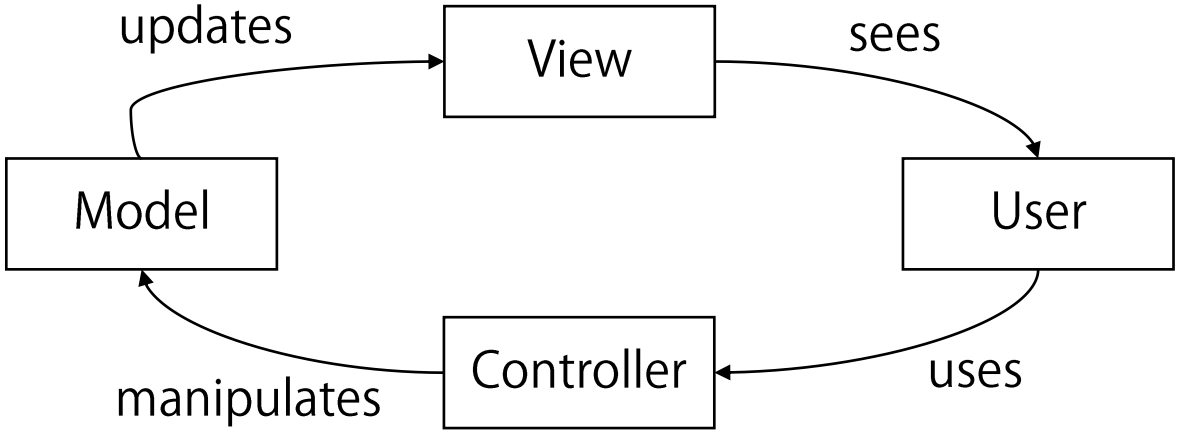
- 多層アーキテクチャパターン（Layer Architecture Pattern）
  （Layers, Layeredのときもあり）
  - システムを複数のレイヤーに分解
  - 各レイヤーは，同様な抽象度・安定性を持つオブジェクトで構成
  - 下位レイヤーは，直につながる上のレイヤーに対してサービスを提供
  - 言い換えると，各レイヤーは直下のレイヤーの知識だけ持てばよい（変更容易性・再利用性）
  - 下位にあるレイヤーの方が安定しているべきである

多層アーキテクチャパターンの適用例



MVCアーキテクチャパターン

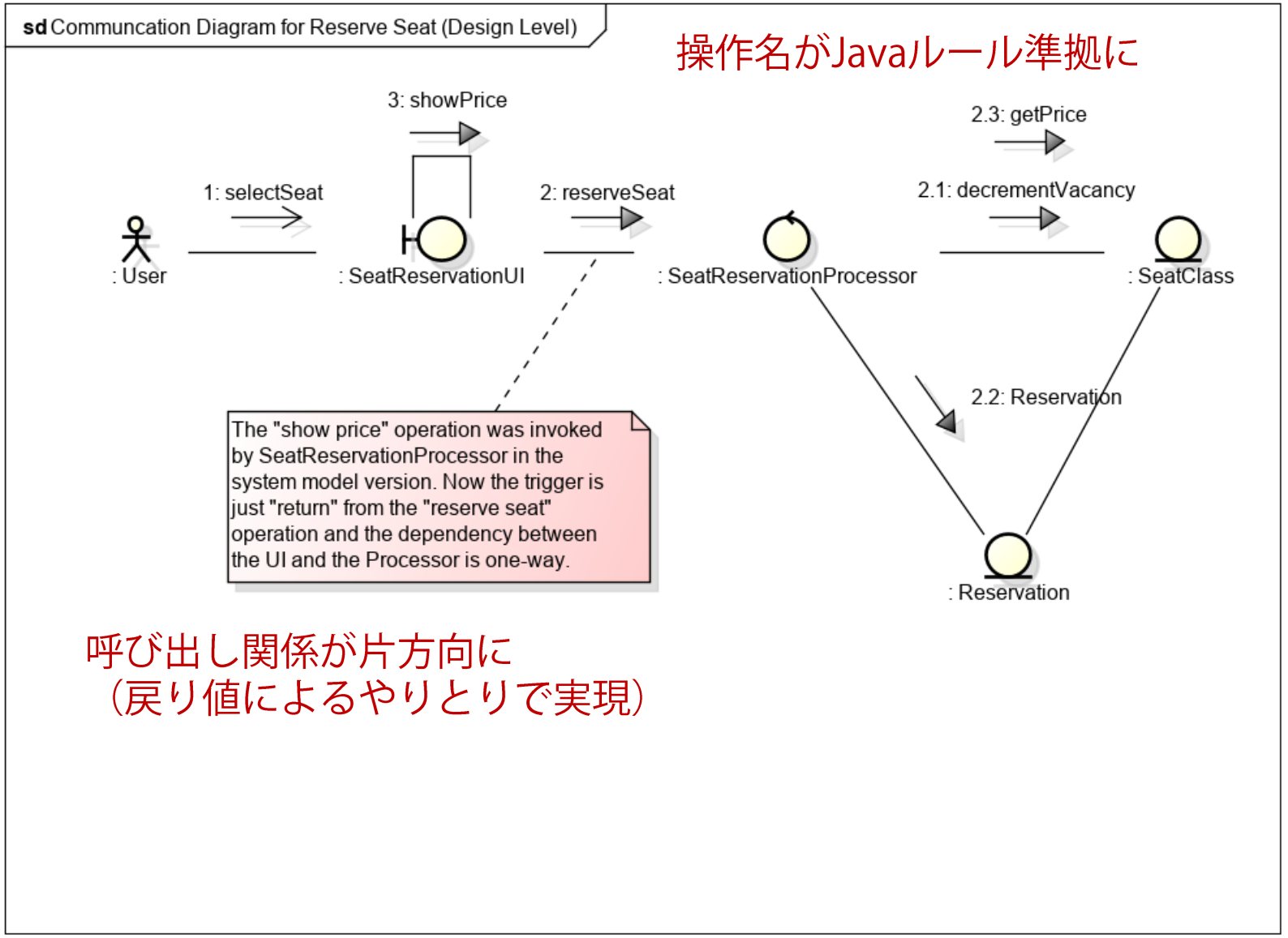
- MVCアーキテクチャパターン
  - 人と相互作用するシステムのためのアーキテクチャパターン
  - 明確な役割分担による変更容易性
  - Model はドメインのデータおよびその処理を
  - View はユーザインターフェースへの出力を
  - Controller はユーザインターフェースからの入力を



アーキテクチャ設計

- システム分析の結果から，非機能要求も踏まえ，アーキテクチャを設計を行う
  - アーキテクチャパターンを活用
- 実現方針を検討する
  - プログラミング言語，プラットフォーム・フレームワーク，GUI形式，…
- 実現手段を踏まえ，システム分析の結果を洗練する
  - プログラミング言語等の制約・慣習に合わせる
  - ナイーブであった部分を洗練する

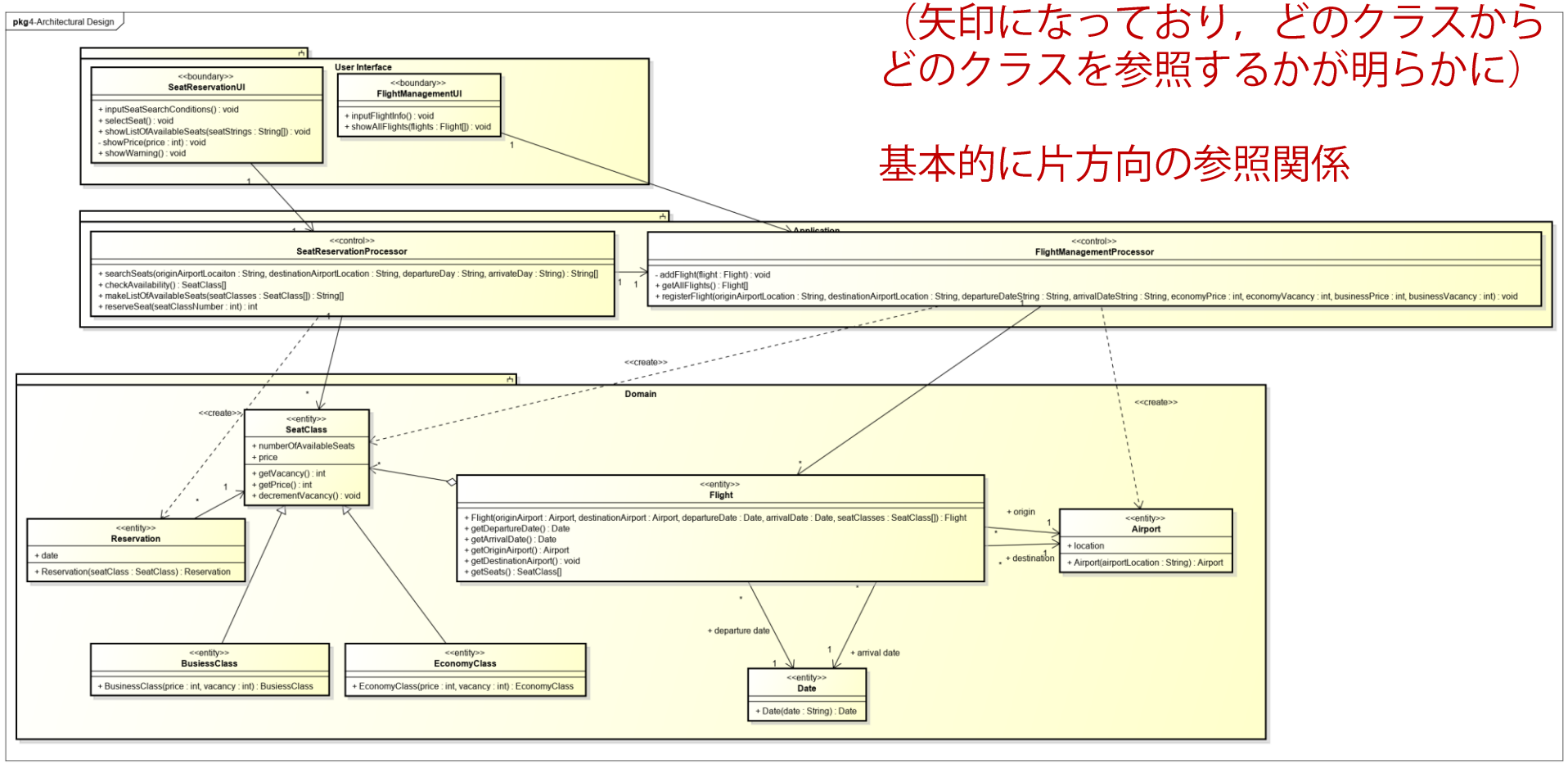
相互作用モデル洗練の例



アーキテクチャ設計の例

すべての関係に方向性が付いている
（矢印になっており，どのクラスから
どのクラスを参照するかが明らかに）

基本的に片方向の参照関係

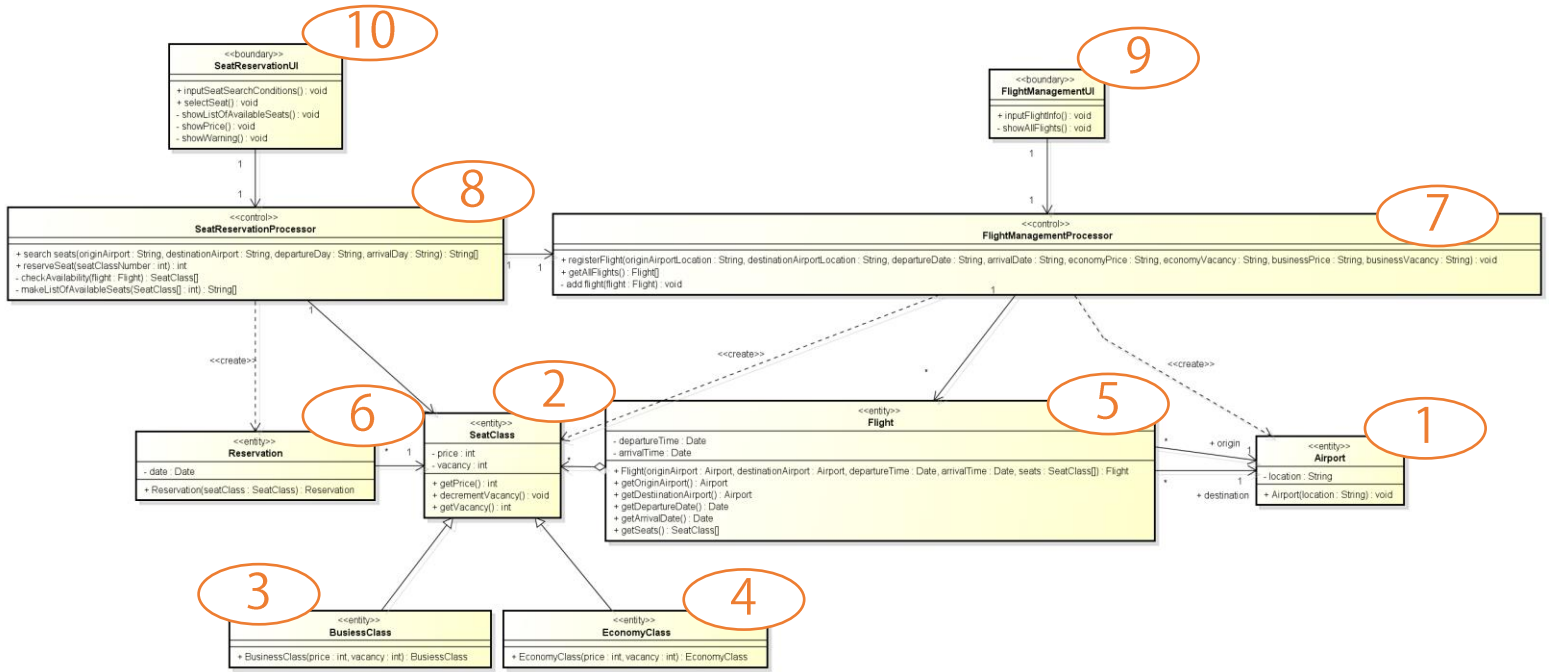


目次

- システム分析
- 設計原則
- アーキテクチャ設計
- 詳細設計
- デザインパターン

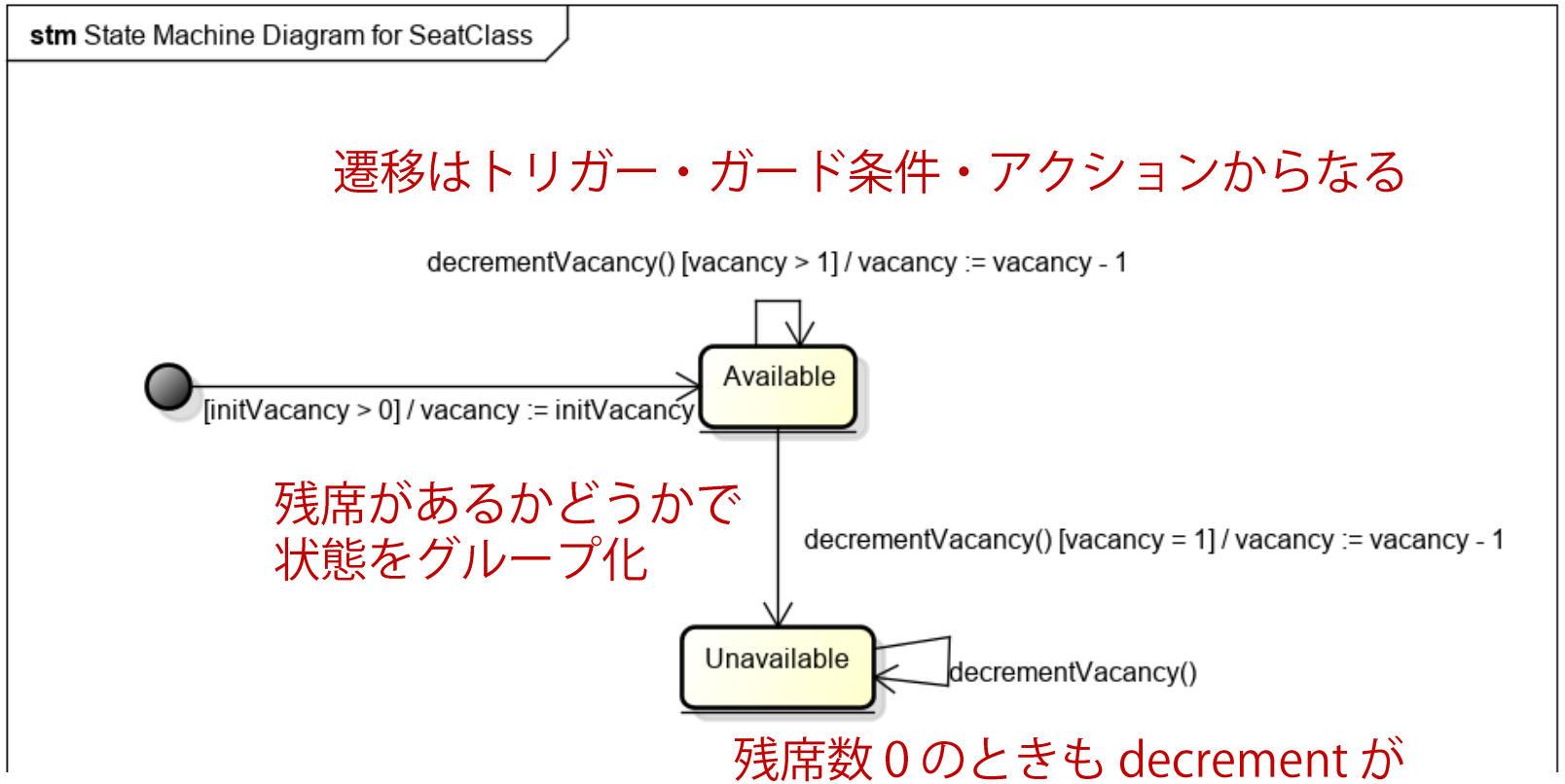
詳細設計

- アーキテクチャ設計後，各部品の設計へ
  - 特に状態遷移が複雑なクラス
  - Inside-outside Principle： 安定性が高く変更しづらいところ（被依存側）から先に詳細化していく



状態遷移設計の例

- UMLのステートマシン図
  - ここから各メソッドの振る舞いを写し取ればよい



目次

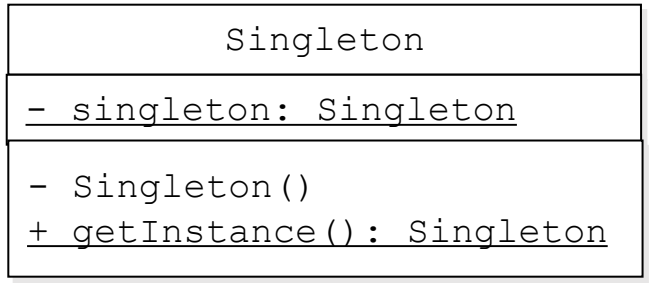
- システム分析
- 設計原則
- アーキテクチャ設計
- 詳細設計
- デザインパターン

デザインパターン

- デザインパターン（Design Patterns）
  - 頻出する設計の解法を汎用化し再利用できるようにしたもの
    （具体的な解法そのものではない！）
  - カタログの形で整理されることが多い
    - コンテキスト： 頻出する状況
    - 問題： 目的と制約，パターン適用を促すフォース
    - 解法： 問題を解決するための設計の指針・ルール
- GoFパターン：オブジェクト指向に対する23のパターン
  - Gang of Four: E. Gamma, R. Helm, R. Johnson, J. Vlissides

例： Singleton Pattern

- システム内において，あるクラスのインスタンスが1つしかないことを保証したい
  - 設定情報，固有名詞など，
- インスタンス生成機能のカプセル化



通常のコンストラクタは private にする

唯一のインスタンスを保持し，専用メソッドで取得させる
このメソッドは，インスタンスがすでにあればそれを返す

下線は Static （クラス変数・クラスメソッド）・ +/- は public/private

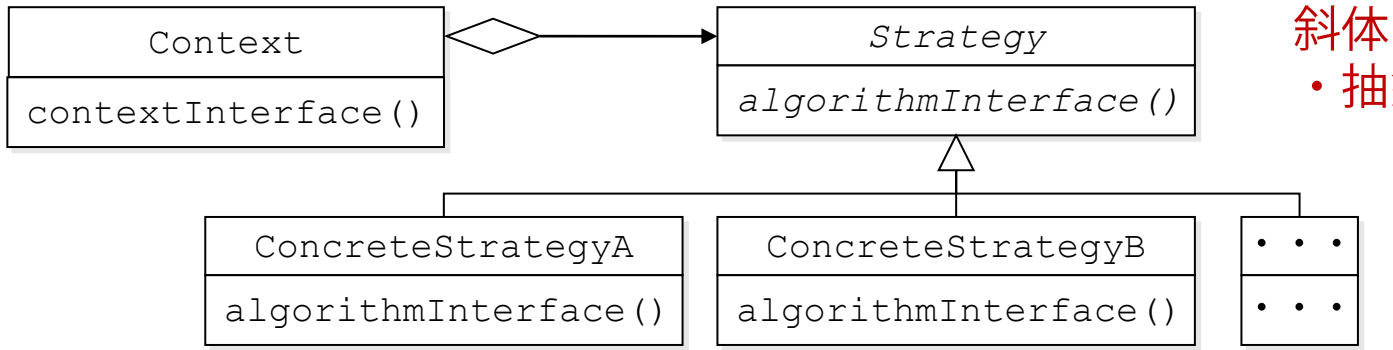
例： Singleton Pattern

```
...
import java.util.HashMap;

public class Airport extends Object {
  private String location = null;
  private static HashMap<String, Airport>
      airportMap = new HashMap();

  private Airport(String locate) {
    location = locate;
  }

  public static Airport getInstance(String location) {
    Airport result = airportMap.get(location);
    if(result == null) {
      result = new Airport(location);
      airportMap.put(locate, result);
    }
    return result;
  }
  ...
}
```

例： Strategy Pattern

- 複数アルゴリズムの選択肢があるが，それらを，利用側のクラスに埋め込みたくない
- 振る舞いのカプセル化とポリモフィズム
  - 多数のクラスができ実行オーバーヘッドあり

利用側クラス Context が Strategy を構成要素として所有



斜体は抽象クラス
・抽象メソッド

異なるアルゴリズムの実装をサブクラスとして提供
利用側はそれから選択・設定して利用

例： Strategy Pattern

- 非パターン（の例）とパターン適用後
  - アルゴリズム追加時の変更先は？
  - 実行オーバーヘッドは？

```
class Mathematic {
  public Data sort(Data data){
    switch(settings) {
    case QUICK:
      return quickSort(data);
    case BUBBLE:
      return bubbleSort(data);
    default: ...
  }

  public void doSettings(...){
     settings = ...;
  }
}
```

```
class Mathematic {
  Sorter sorter;
  public Data sort(Data data){
    return sorter.sort(data);
  }
  public void setSorter(Sorter s){
    sorter = s;
  }
}
```

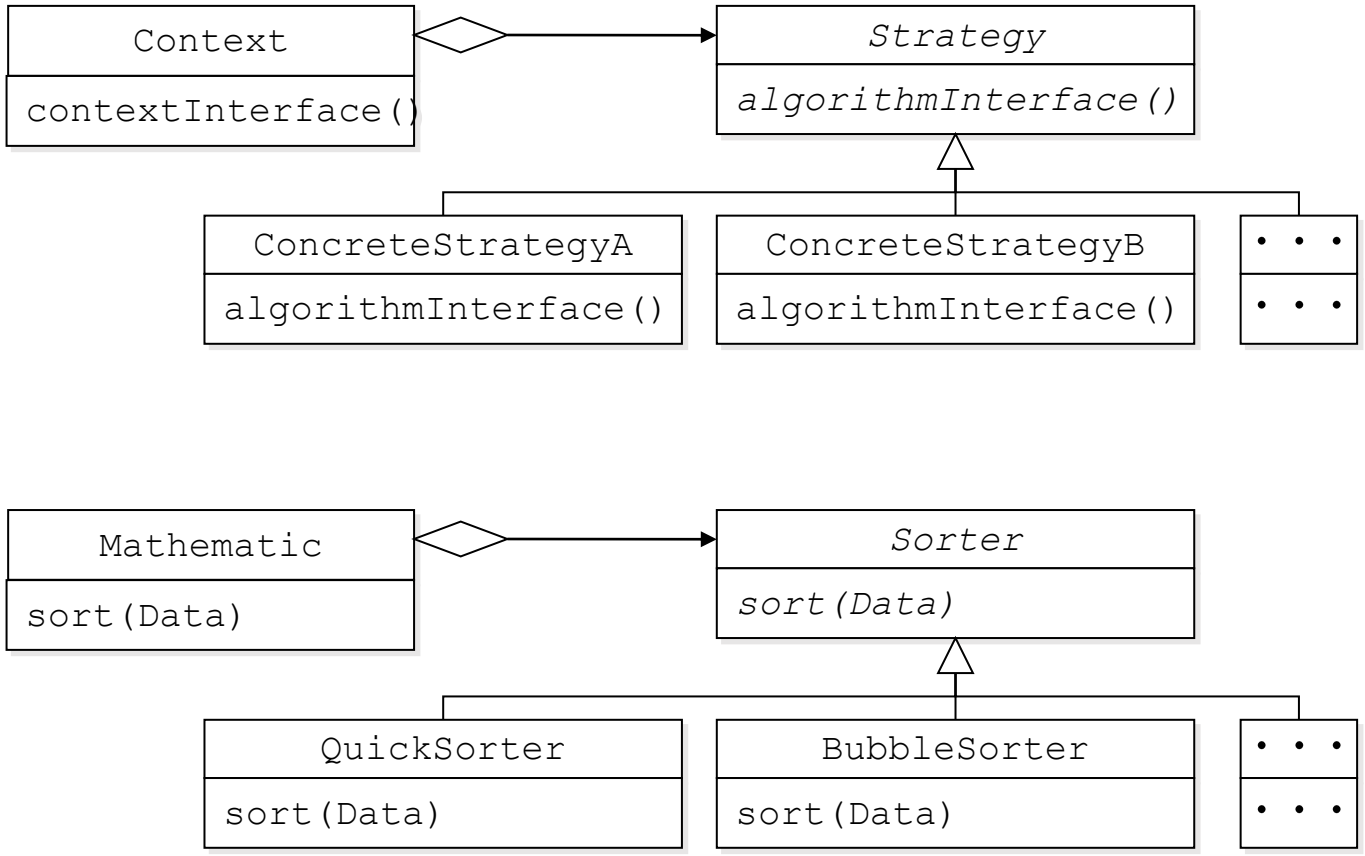
```
abstract class Sorter {
  public abstract Data sort(Data);
```

```
class QuickSorter extends Sorter {
  public Data sort(Data) { ... }
```

```
class BubbleSorter extends Sorter {
  public Data sort(Data) { ... }
```

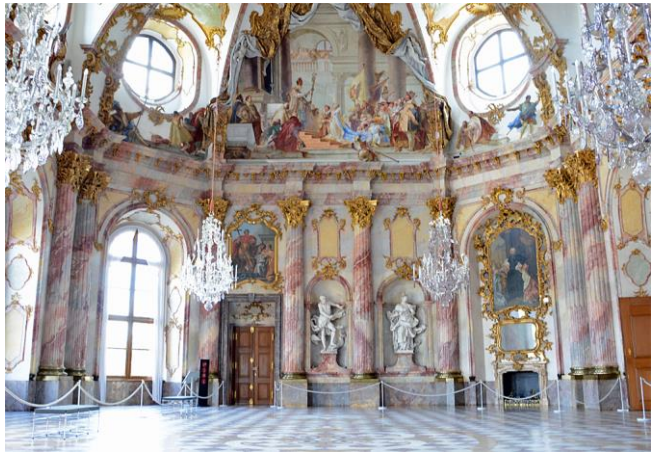
例： Strategy Pattern

- パターンとその適用

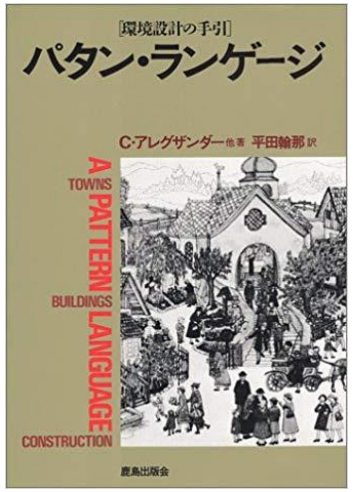


おまけ

- アーキテクチャもデザインパターンも建築用語
  - Rococo Architecture



[https://en.wikipedia.org/wiki/Rococo#/media/File:Kaisersaal_W%C3%BCrzburg.jpg]



[クリストファー・アレグザンダー (著), 平田訳, パタン・ランゲージ―環境設計の手引, 鹿島出版会, 1984]

今回の参考文献（１）

- ユースケース駆動開発実践ガイド
  - Doug Rosenbergら，翔泳社，2007
  - ロバストネス分析を含む手法ICONIXの解説
- Clean Architecture 達人に学ぶソフトウェアの構造と設計
  - R. C.Martin, 角ら訳, (著), KADOKAWA, 2018
  - アーキテクチャレベルの設計に関する良書の一つ

今回の参考文献（２）

- オブジェクト指向における再利用のためのデザインパターン
  - E. Gamma et al., 本位田ら訳,ソフトバンククリエイティブ, 1999
  - デザインパターンの大元の本翻訳
- Java言語で学ぶデザインパターン入門
  - 結城 浩,ソフトバンククリエイティブ, 2004
  - デザインパターンをプログラミングを交えて解説

まとめ

- システム分析
  - 実現の検討を高い抽象度で行うことで，実装非依存な設計の基礎を得つつ，要求・ユースケースがロバストであるように妥当性確認を行う
- 設計
  - 様々な非機能要求も踏まえつつ，要求・ユースケースから実現のための指針を定めていく
  - アーキテクチャレベルから部品レベルへと移っていく