Software Engineering

(4) Design

Sokendai / National Institute of Informatics

Fuyuki Ishikawa / 石川　冬樹

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

TOC

(Review) Design

- Design
  - Defines “How” to realize the requirements
  - Needs to reflect the non-functional requirements
  - Deals with the whole system (architecture) or individual parts (components)

(Review) Design Principles

- Encapsulation
- Information Hiding
- Abstraction
- Modularization
- Divide-and-Conquer
- Consideration of Cohesion and Coupling
- Separation of Concerns

Extended from [Buschmann et al., Pattern-Oriented Software Architecture, Wiley, 1996]

Architecture

- Architecture :
  Definition of components in the system as well as their characteristics and relations
  - Promote communication among stakeholders
  - Make design decisions explicit to allow for comparison
  - Clarify potential risks
  - Significant, sometimes done by experts called “architects”

Architecture Pattern

- Architecture Patterns :
  Generalized architectures as typical solutions for reoccurring problems
  - Well-known examples
    - Layer architecture (common principle)
    - MVC architecture (for interaction with human users)
    - Broker architecture (for distributed systems)

Layer Architecture Pattern

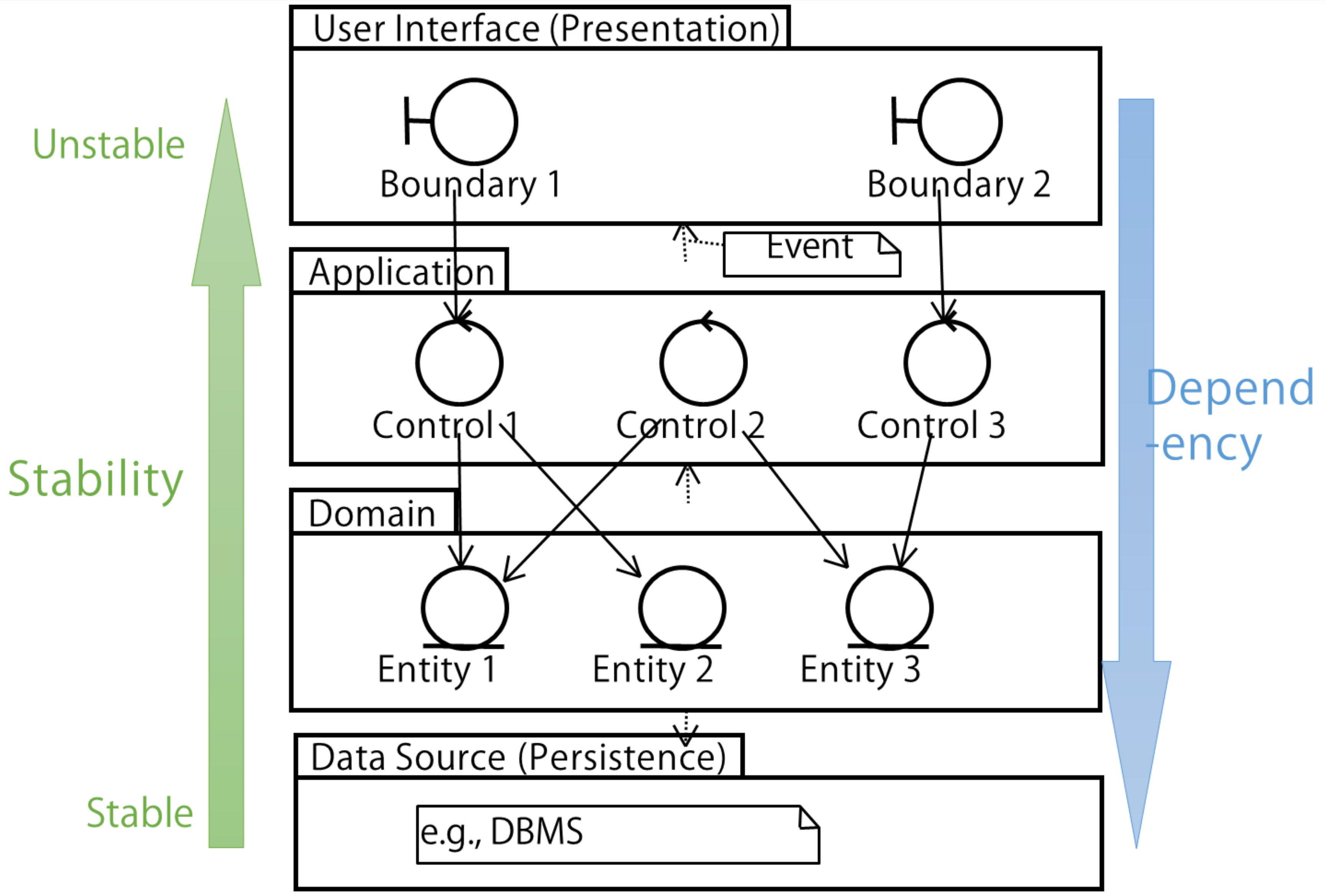
- Layer Architecture Pattern

  (sometimes “layers” or “layered”)

  - We decompose the system into multiple layers
  - Each layer consists of objects with similar abstraction/stability levels
  - Lower layers should be more stable
  - Each layer provides its service to the direct upper layer
  - Each layer is required knowledge only on the direct lower layer

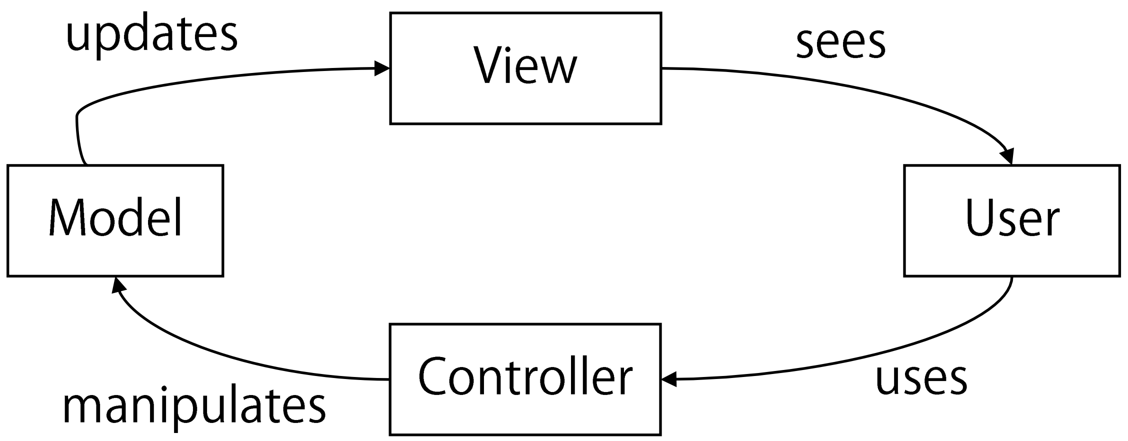
➡ Modifiability and reusability

Layer Architecture Pattern: Example



MVC Architecture Pattern

- MVC Architecture Pattern
    - Architecture pattern for interaction with human users
    - Modifiability with clear separation of responsibility
    - Often embedded in web frameworks

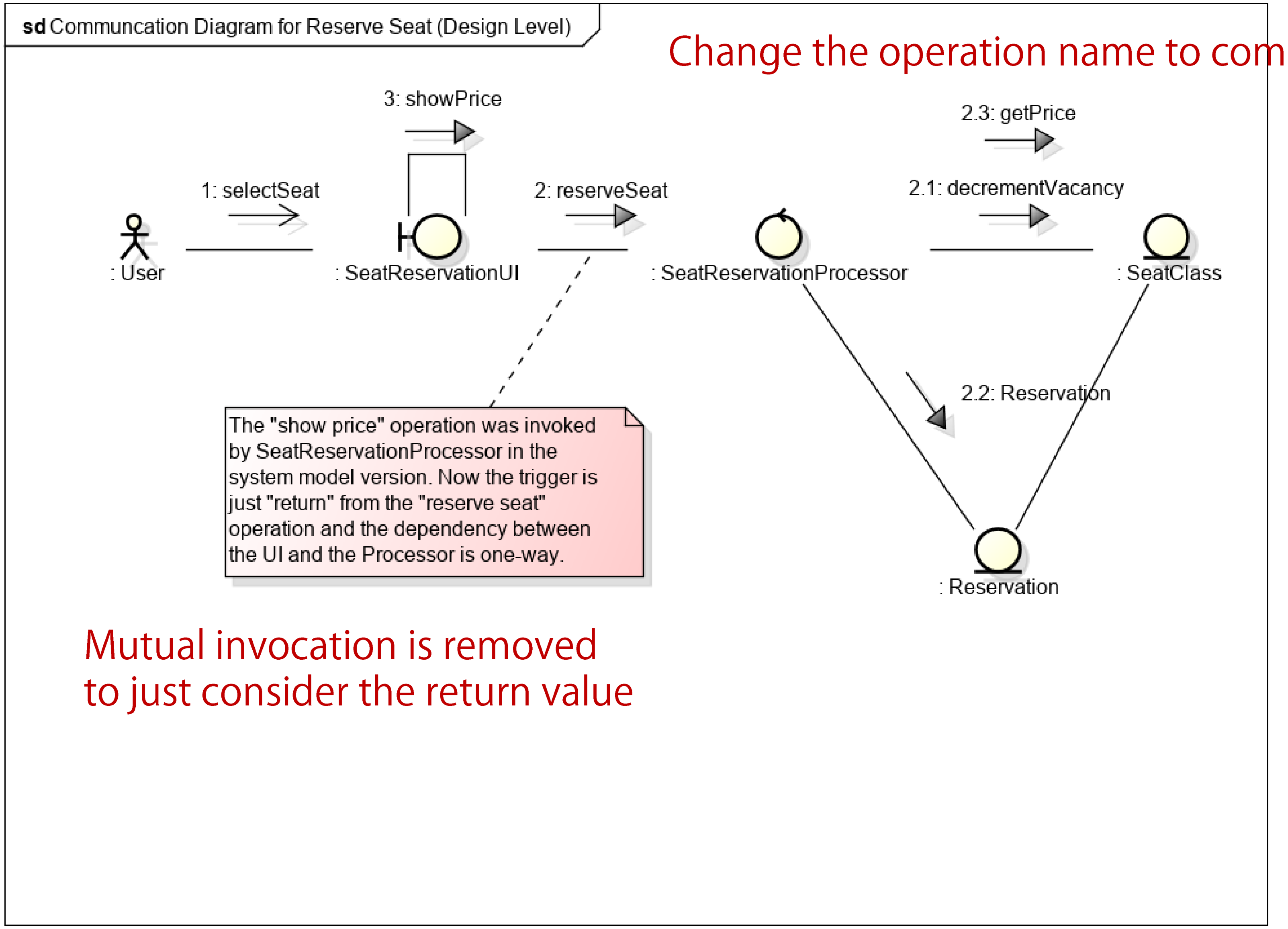


- Model: deals with domain data and their processing
- View: deals with output to the user interface
- Controller: deals with input from the user interface

Architecture Design

- Architecture design
  - Make decisions on the architecture
  - Make use of available architecture patterns
  - Consider non-functional requirements
  - Decide implementation strategies, such as platforms, libraries/frameworks, GUI styles, etc.
  - Refine the result of system analysis to match with the implementation strategies as well as removing naïve points

Example: Refinement of System Analysis Result

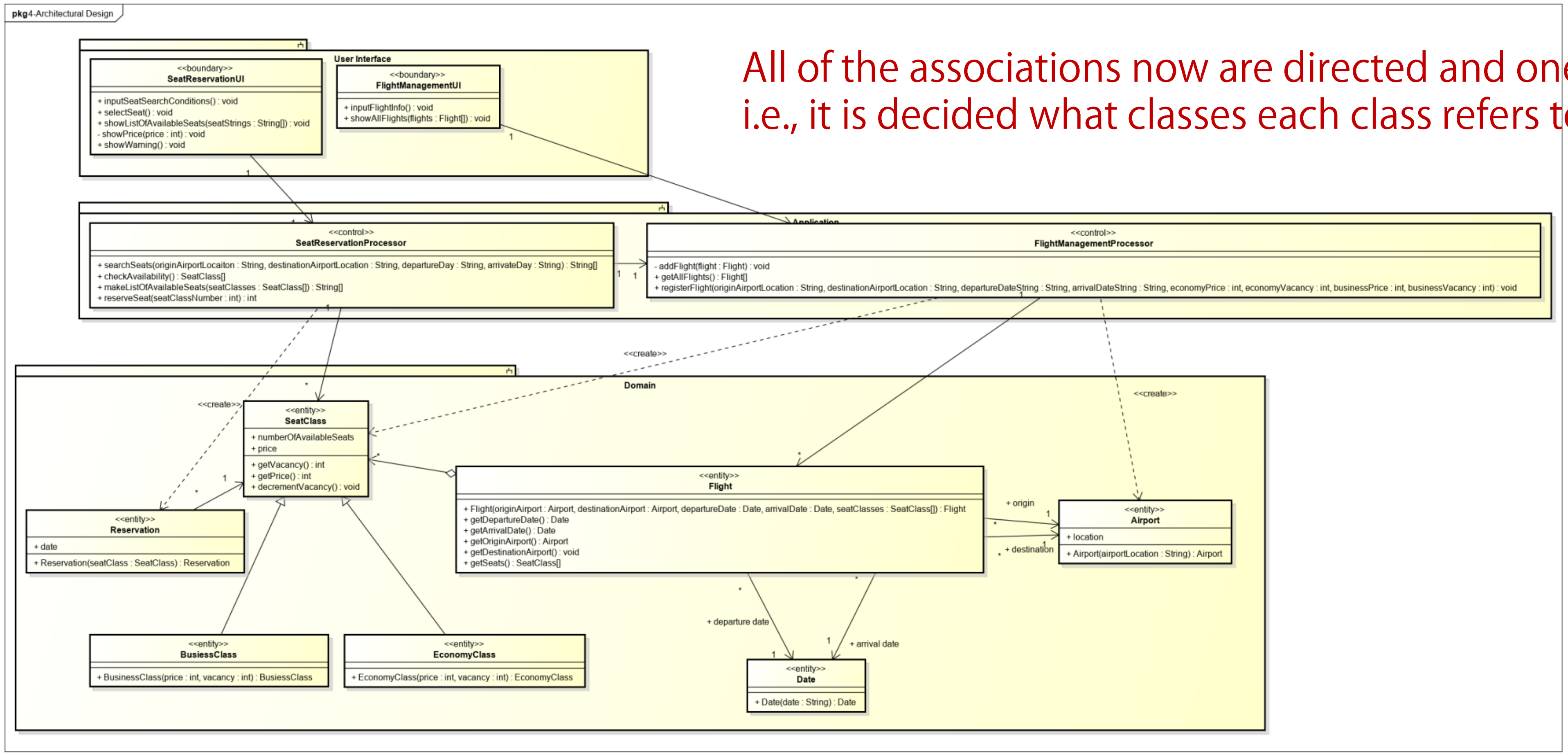


The "show price" operation was invoked by SeatReservationProcessor in the system model version. Now the trigger is just "return" from the "reserve seat" operation and the dependency between the UI and the Processor is one-way.

Change the operation name to comply with Java rules

Mutual invocation is removed
to just consider the return value

Example: Refinement of System Analysis Result

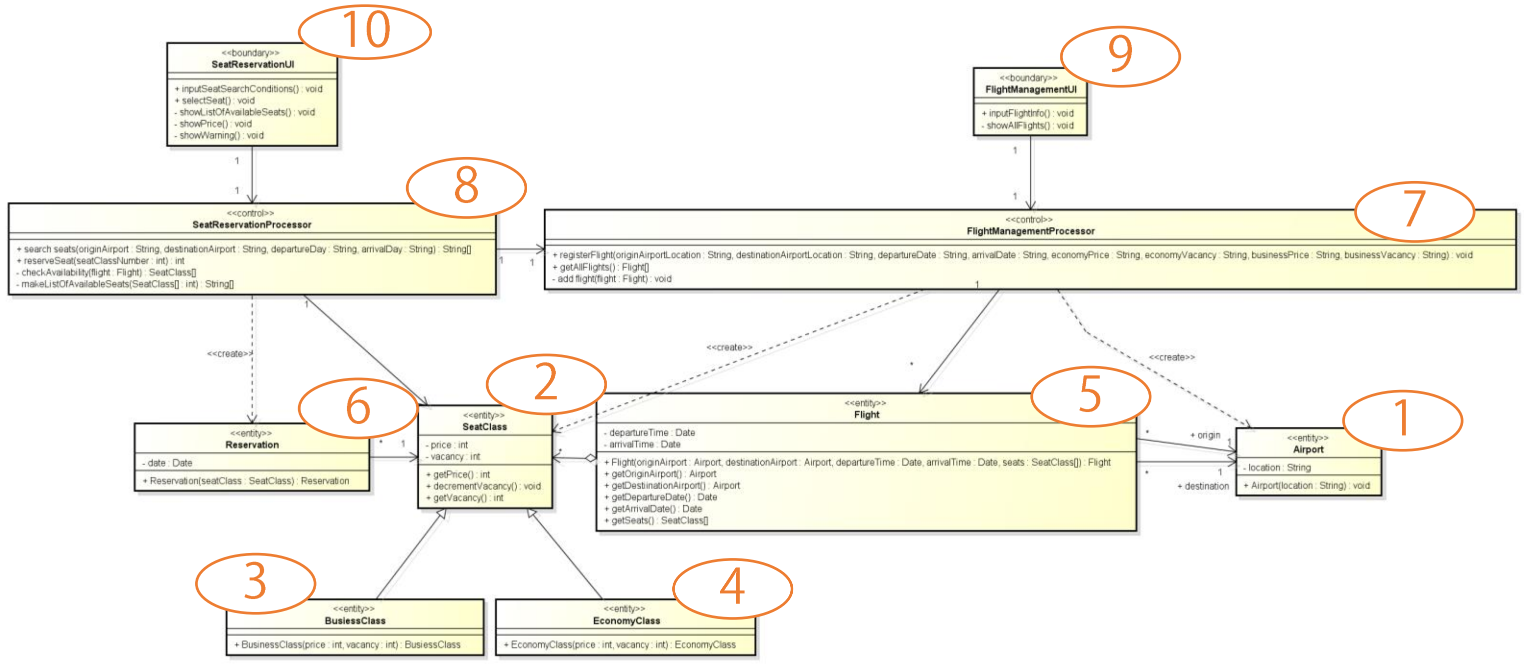

pkg4-Architectural Design
User Interface
<<boundary>>
SeatReservationUI
+ inputSeatSearchConditions() : void
+ selectSeat() : void
+ showListOfAvailableSeats(seatStrings : String[]) : void
- showPrice(price : int) : void
+ showWarning() : void
<<boundary>>
FlightManagementUI
+ inputFlightInfo() : void
+ showAllFlights(flights : Flight[]) : void
Application
<<control>>
SeatReservationProcessor
+ searchSeats(originAirportLocaiton : String, destinationAirportLocation : String, departureDay : String, arrivateDay : String) : String[]
+ checkAvailability() : SeatClass[]
+ makeListOfAvailableSeats(seatClasses : SeatClass[]) : String[]
+ reserveSeat(seatClassNumber : int) : int
<<control>>
FlightManagementProcessor
- addFlight(flight : Flight) : void
+ getAllFlights() : Flight[]
+ registerFlight(originAirportLocation : String, destinationAirportLocation : String, departureDateString : String, arrivalDateString : String, economyPrice : int, economyVacancy : int, businessPrice : int, businessVacancy : int) : void
<<create>>
Domain
<<create>>
<<create>>
<<entity>>
SeatClass
+ numberOfAvailableSeats
+ price
+ getVacancy() : int
+ getPrice() : int
+ decrementVacancy() : void
<<entity>>
Reservation
+ date
+ Reservation(seatClass : SeatClass) : Reservation
<<entity>>
Flight
+ Flight(originAirport : Airport, destinationAirport : Airport, departureDate : Date, arrivalDate : Date, seatClasses : SeatClass[]) : Flight
+ getDepartureDate() : Date
+ getArrivalDate() : Date
+ getOriginAirport() : Airport
+ getDestinationAirport() : void
+ getSeats() : SeatClass[]
+ origin
+ destination
<<entity>>
Airport
+ location
+ Airport(airportLocation : String) : Airport
<<entity>>
BusiessClass
+ BusinessClass(price : int, vacancy : int) : BusiessClass
<<entity>>
EconomyClass
+ EconomyClass(price : int, vacancy : int) : EconomyClass
+ departure date
+ arrival date
<<entity>>
Date
+ Date(date : String) : Date

All of the associations now are directed and one-way, i.e., it is decided what classes each class refers to/uses

TOC

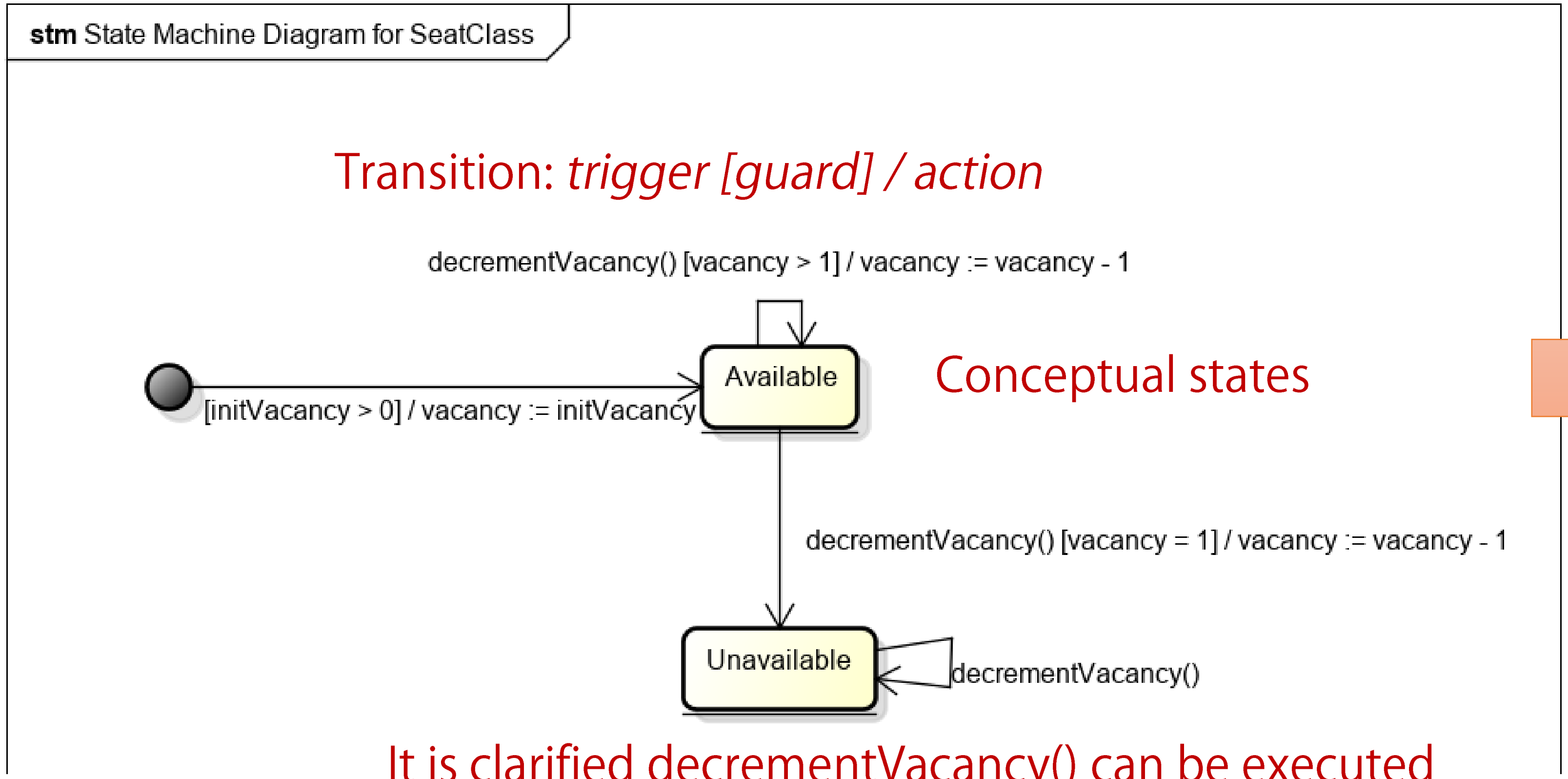
Detailed Design

- Component-level design after architecture-level design
  - Significant for classes with complex state transitions
  - Inside-outside Principle: we should start with refinement of more stable components, depended by other components


10
<<boundary>>
SeatReservationUI
+ inputSeatSearchConditions() : void
+ selectSeat() : void
- showListOfAvailableSeats() : void
- showPrice() : void
- showWarning() : void
9
<<boundary>>
FlightManagementUI
+ inputFlightInfo() : void
- showAllFlights() : void
8
<<control>>
SeatReservationProcessor
+ search seats(originAirport : String, destinationAirport : String, departureDay : String, arrivalDay : String) : String[]
+ reserveSeat(seatClassNumber : int) : int
- checkAvailability(flight : Flight) : SeatClass[]
- makeListOfAvailableSeats(SeatClass[] : int) : String[]
7
<<control>>
FlightManagementProcessor
+ registerFlight(originAirportLocation : String, destinationAirportLocation : String, departureDate : String, arrivalDate : String, economyPrice : String, economyVacancy : String, businessPrice : String, businessVacancy : String) : void
+ getAllFlights() : Flight[]
- add flight(flight : Flight) : void
<<create>>
<<create>>
<<create>>
6
<<entity>>
Reservation
- date : Date
+ Reservation(seatClass : SeatClass) : Reservation
2
<<entity>>
SeatClass
- price : int
- vacancy : int
+ getPrice() : int
+ decrementVacancy() : void
+ getVacancy() : int
5
<<entity>>
Flight
- departureTime : Date
- arrivalTime : Date
+ Flight(originAirport : Airport, destinationAirport : Airport, departureTime : Date, arrivalTime : Date, seats : SeatClass[]) : Flight
+ getOriginAirport() : Airport
+ getDestinationAirport() : Airport
+ getDepartureDate() : Date
+ getArrivalDate() : Date
+ getSeats() : SeatClass[]
+ origin
+ destination
1
<<entity>>
Airport
- location : String
+ Airport(location : String) : void
3
<<entity>>
BusiessClass
+ BusinessClass(price : int, vacancy : int) : BusiessClass
4
<<entity>>
EconomyClass
+ EconomyClass(price : int, vacancy : int) : EconomyClass

Example of State Transition

- UML State Machine Diagram
  - For the SeatClass class in the flight example


stm State Machine Diagram for SeatClass
Transition: *trigger [guard] / action*
decrementVacancy() [vacancy > 1] / vacancy := vacancy - 1
Available
[initVacancy > 0] / vacancy := initVacancy
Conceptual states
decrementVacancy() [vacancy = 1] / vacancy := vacancy - 1
Unavailable
decrementVacancy()

```
...
int vacancy=initVacancy;

decrementVacancy(){
  if vacancy >= 1
    vacancy -= 1;
}
...
```

It is clarified decrementVacancy() can be executed but do nothing when there is no available seat

TOC

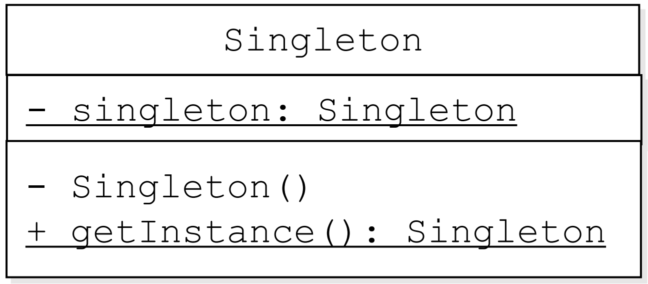
Design Patenrs

- Design Patterns
  - Obtained by generalizing solutions to reoccurring problems to reuse them
  - Often organized in the form of catalogue
    - Context: reoccurring situation
    - Problem: objective and constraints, force to motivate use of the pattern
    - Solution: principles and rules to solve the problem
- GoF Patterns: 23 patterns in object-orientation
  - Gang of Four: E. Gamma, R. Helm, R. Johnson, J. Vlissides

Example: Singleton Pattern

- We want to ensure there is at most one instance of a target class in the system
  - For configuration class, proper names, …

➡ Encapsulation of instance creation



The sole instance is kept in a static private variable

The default constructor is not accessible form outside

Instance creation is done only by a dedicated method that returns the instance if it already exists, otherwise creates a new one

Underline means static (class variables/methods), +/- mean public/private

Example: 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;
  }
  ...
}
```

The pattern is arranged to make
“only one instance for each airport name”
not “in the system”

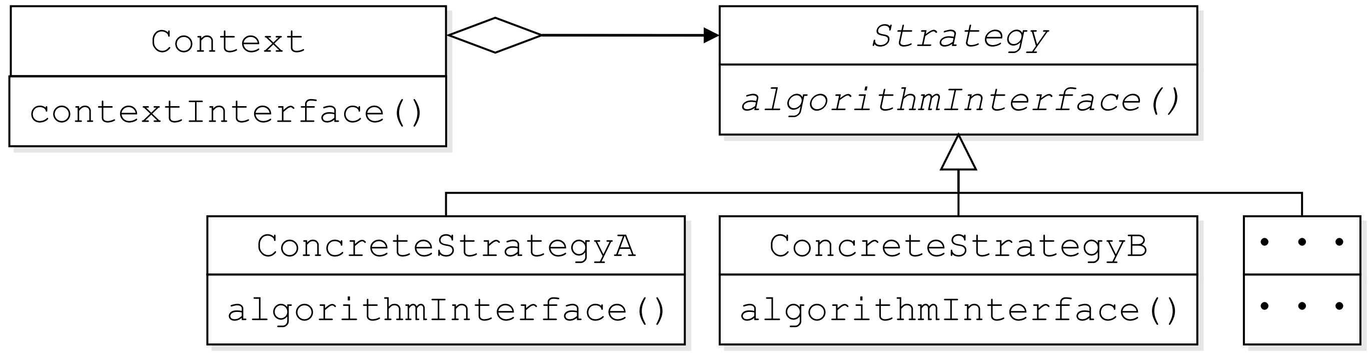
Example: Strategy Pattern

- There are multiple candidates for algorithms but we don't want embed them to the client

➡ Encapsulation and polymorphism

  - Trade-off: overhead with many classes

The client class Context owns Strategy objects

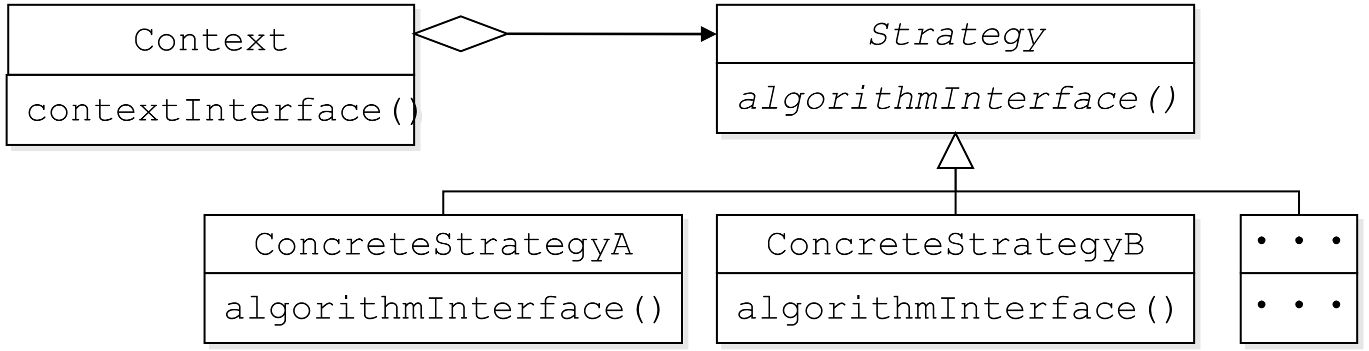


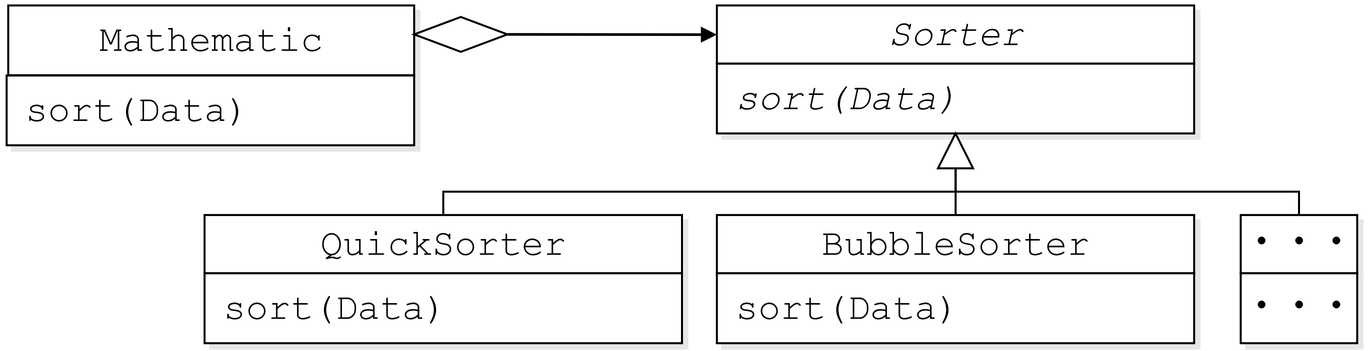
Implementations of different algorithms is provided as subclasses

Italic means abstract classes/methods

Example: Strategy Pattern

- Example in class diagrams





Example: Strategy Pattern

- Before/after of pattern application
  - What happens if we add new algorithms?
  - How about overhead?

```
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;
  }
}
```

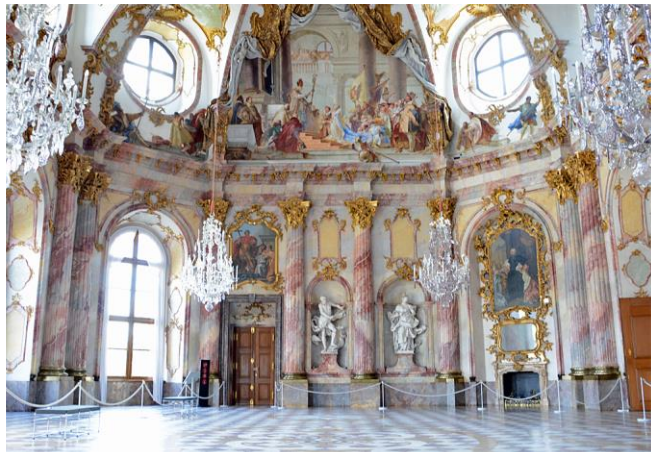
```
abstract class Sorter {
  public abstract Data sort(Data);
```

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

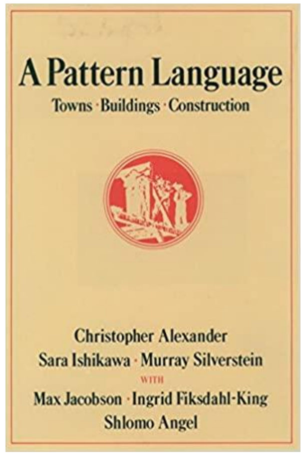
```
class BubbleSorter extends Sorter {
  public Data sort(Data) { ... }
```

Historical Notes

- Both “architecture” and “design patterns” came from the terminology about building/construction
  - Rococo Architecture

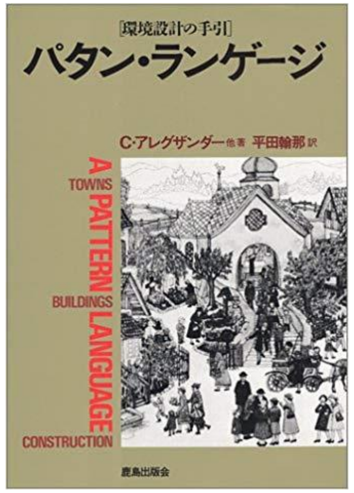


[https://en.wikipedia.org/wiki/Rococo#/media/File:Kaisersaal_W%C3%BCrzburg.jpg]



A Pattern Language
Towns · Buildings · Construction

Christopher Alexander
Sara Ishikawa · Murray Silverstein
WITH
Max Jacobson · Ingrid Fiksdahl-King
Shlomo Angel



[環境設計の手引]
パタン・ランゲージ
C·アレグザンダー他著 平田翰那訳
A PATTERN LANGUAGE
TOWNS
BUILDINGS
CONSTRUCTION
鹿島出版会

[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]

Summary

- Design
  - Define implementation strategies by considering various non-functional requirements from requirements and use cases
  - From architecture-level to component level