

Content(2)

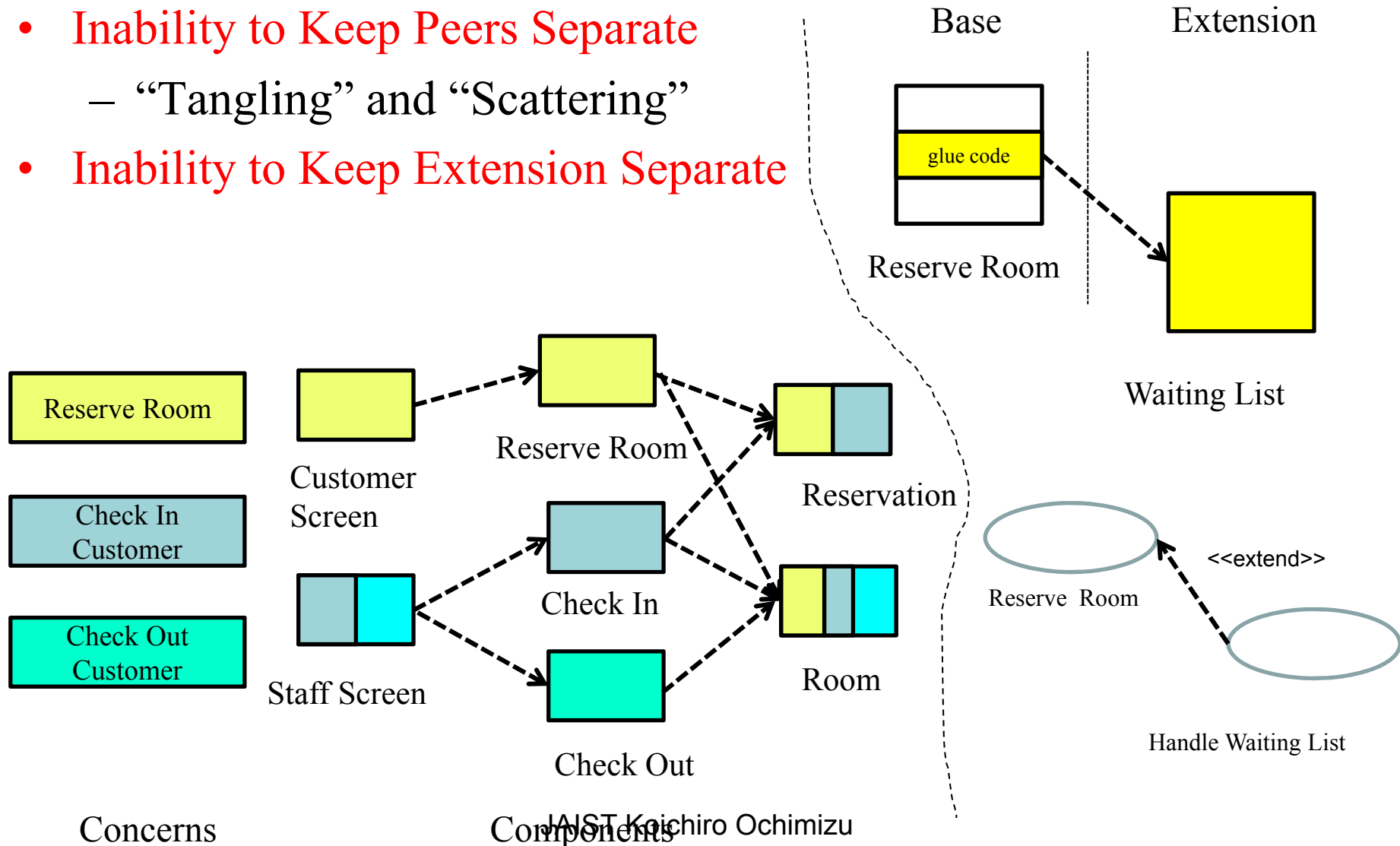
- **Object-oriented Software Development Methodology**
 - Outline of Unified Process and Use-case Driven Approach
 - Elevator Control System:
Problem Description and Use-case Model
 - Elevator Control System:
Finding of Problem Domain Objects
 - Elevator Control System:
Sub-System Design and Task Design
 - Elevator Control System:
Performance Evaluation
- **Product Line Technology**
 - Feature modeling
- **Aspect Oriented Software Design**
- **Contribution of OOT in Software Engineering**
 - History of SE Technologies and Contribution of OOT
in SE field

What is “Concerns”?

- **Concerns**
 - A concern is anything that is of interest to a stakeholder, whether an end user, project sponsor, or developer
 - A concern can be a functional requirement, a nonfunctional requirement(NFR), or a design constraint on the system.
- **Separation of Concerns**
 - Breaking down a problem into smaller parts is called “separation of concerns” in computer science.
 - Ideally, we want to be able to cleanly separate the different concerns into modules of some kind and explore and develop each in isolation, one at a time.
- **Crosscutting Concerns**
 - Although some concerns can be realized by distinct and separate components(class package, service), in general you can find many concerns for which components are not adequate. These are known as crosscutting concerns – **concerns that impact multiple components**.
 - Different kinds of crosscutting concerns
 - Concerns to meet FRs: **use case**
 - Infrastructure concerns to meet NFRs: logging, distribution, transaction management

Two types of crosscutting concerns discussed in the textbook !

- Inability to Keep Peers Separate
 - “Tangling” and “Scattering”
- Inability to Keep Extension Separate



Attacking the Problem with Aspects

- Aspect Orientation
 - Aspect orientation is a set of technologies aimed at providing better separation of crosscutting concerns.
- 1997
 - Research in aspect orientation has been going on for a relatively long time, but it started gaining a lot of recognition from mainstream observers in 1997 when Gregor Kiczales from Xerox Parc presented his keynote presentation on aspect-oriented programming(AOP) at OOPSLA'97
- 2001
 - AspectJ : intertype declaration, advices, pointcuts, joinpoints

**There are three basic concepts of
AOP**

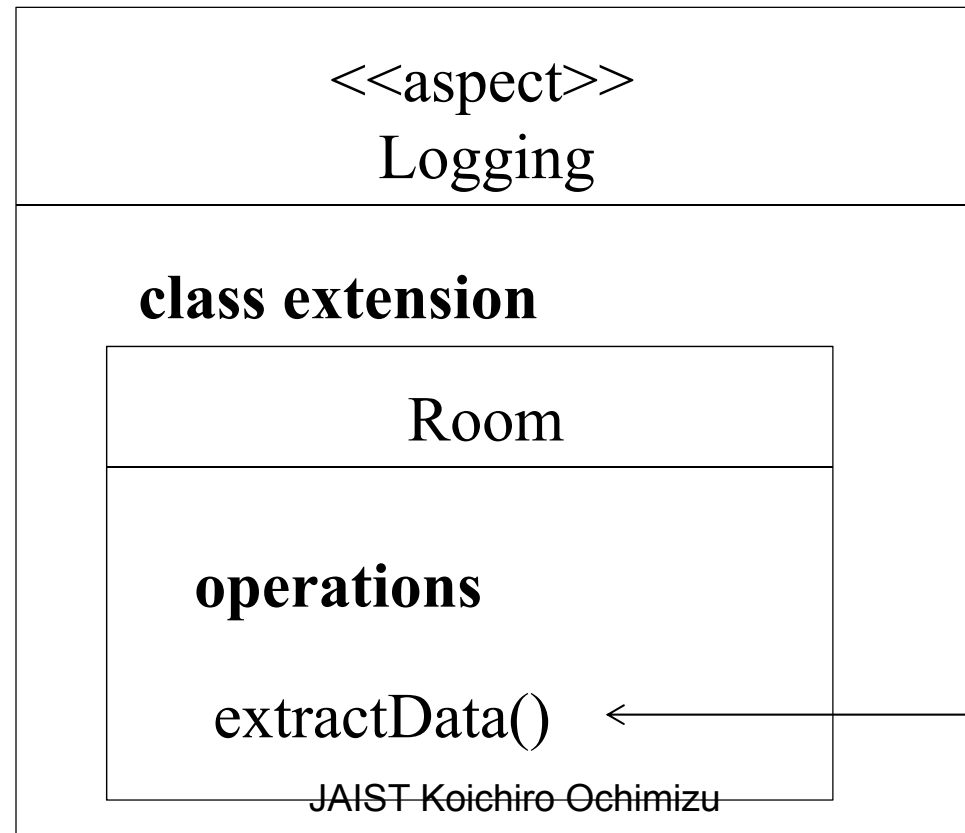
**They are basic concepts of AOSD
too**

- Intertype declaration
- Advices
- Pointcuts

Intertype declarations

allow us to compose new features (attributes, operations and relationships) into existing classes

1. `public aspect Logging{`
2. `public void Room.extractData(){`
3. `// code`
4. `}`
5. `}`



This operation tells aspect composition to add `extraData()` into `Room` class.

Advices

provide the means to extend existing operations at extension points designed by pointcuts in AOP

- an operations extension logData to the ReserveRoomHandler class
makeReservation() operation before the execution point when a call is made to Room.retrieve(),

Public **aspect** Logging {

1. **before () :**
2. **withincode(void ReservationRoomHandler.makeReservation())**
3. **&& call(void Room.retrieve()) {**
4. **// code** Change the behavior of Room. retrieve or makeReservation
5. **}** (actually change joinpoint)

6. } <<aspect>>
Logging

class extension

ReserveRoomHandler

operations

makeReservation() { before(call(Room.retrieve())) logData }

JAIST Koichiro Ochimizu

Pointcuts

- an operation extension logData
- to an existing operation within the ReserveRoomHandler class(This is parameterized as <roomAccessOp>, and it establishes the structural context of the operation extension.
- before a call to a Room operation(This is parameterized as <roomCall> and it establishes the behavioral context of the operation extension

<<aspect>>
Logging

pointcuts

roomAccessOp = *(..) name given to structural context
roomCall = call(Room.*(..)) name given to behavioral context

class extension

ReserveRoomHandler

operations

<roomAccessOp>(){ before (<roomCall>) logData}

JAIST Koichiro Ochimizu

Keeping Peers Separate with Aspects

To check in a customer, you assign him to a room and consume his reservation. At the same time, you create an initial bill for the customer.

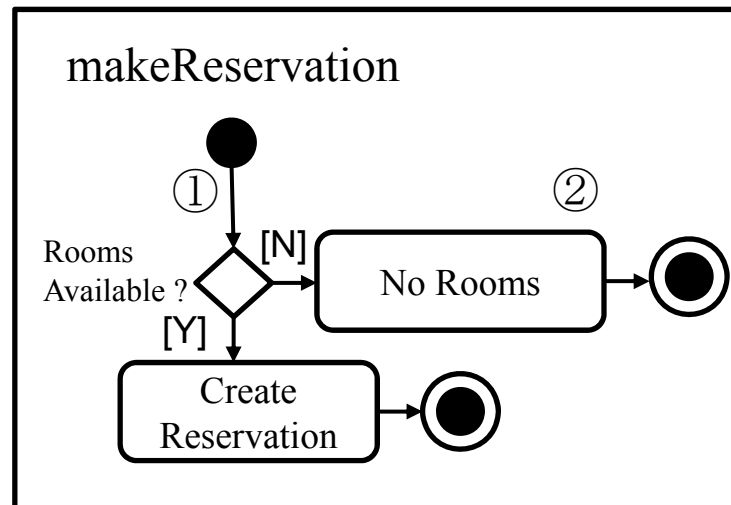
These functionalities cut across classes in the system

	Room	Reservation	Payment
Reserve Room	checkAvailability()	create()	
Check In Customer	assignCustomer()	consume()	createBill()
Check Out Customer	removeCustomer()		payBill()

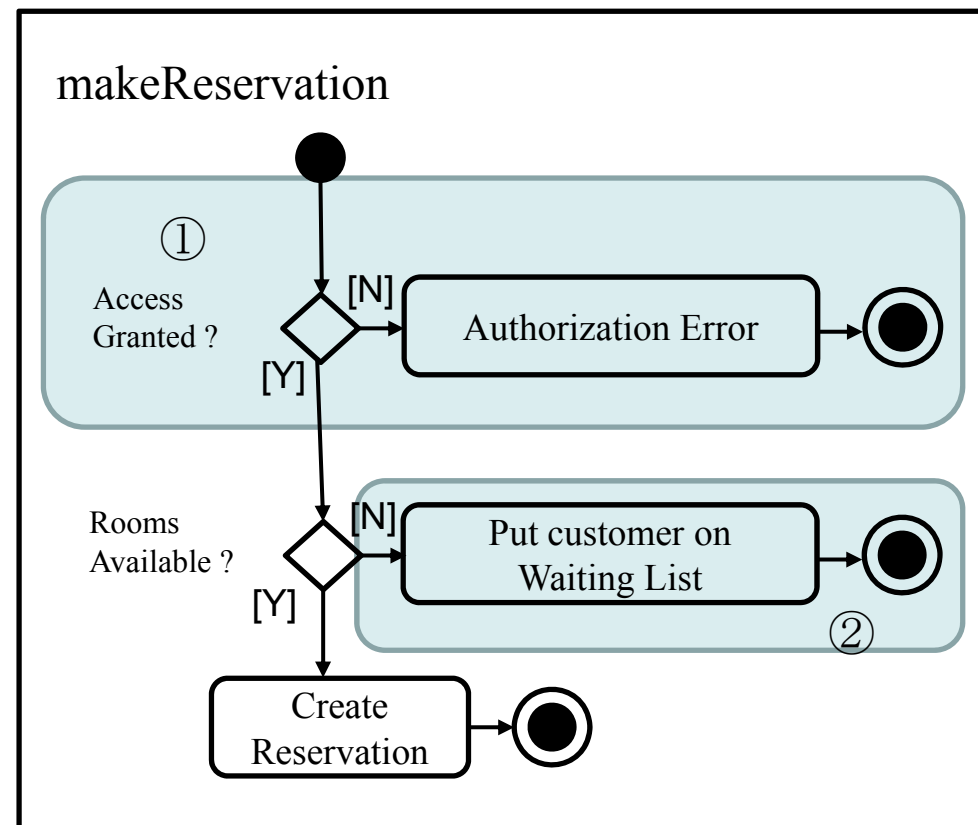
1. public **aspect** CheckInCustomer {
2. ...
3. public void Room.assignCustomer() { //code to check in costomer}
4. public void Reservation.consume() { //code to consume reservation}
5. public void Payment.createBill() { //code to generate an initial outstanding bill}
6. }

Keeping Extensions Separate with Aspects

1. Identify the extension points in the existing operation in which the behavior needs to be executed.
2. Define the additional behavior that will be used to extend the behavior at these extension points.



makeReservation operation



makeReservation modified with
authorization and waiting list.

Examples of Source Code for the second case

- Simplified Source Code for ReservationRoomHandle

```
1.    Class ReserveRoomHandler{
2.        ...
3.        public void makeReservation() throws NoRoomException {
4.            if( theRoom.getQuantityAvailable() <=0) { throw new NoRoomException(); }
5.            createReservation();
6.        }
7.        ...
8.    }
```

- Simplified Source Code to Handle Waiting List

```
1.    aspect HandleWaitingList {
2.        ...
3.        pointcut makingReservation();
4.        execution (void ReserveRoomHandler.makeReservation());
5.        ...
6.        after throwing (NoRoomException e) ; makingReservation() {
7.            //code to add customer to waiting list
8.        }
9.    }
```

Use Case Slice

- **Use-case slice:**

Each use-case slice keeps the specifics of a use-case realization in one model (e.g. analysis model, design model etc.). An use-case slice contains the specifics of a model in a single package. A use-case slice of the analysis or design model contains classes and aspects of classes specific to a use case. It also contains the collaboration that describes the realization of the use case in terms of interaction, communication, class diagrams, and so on

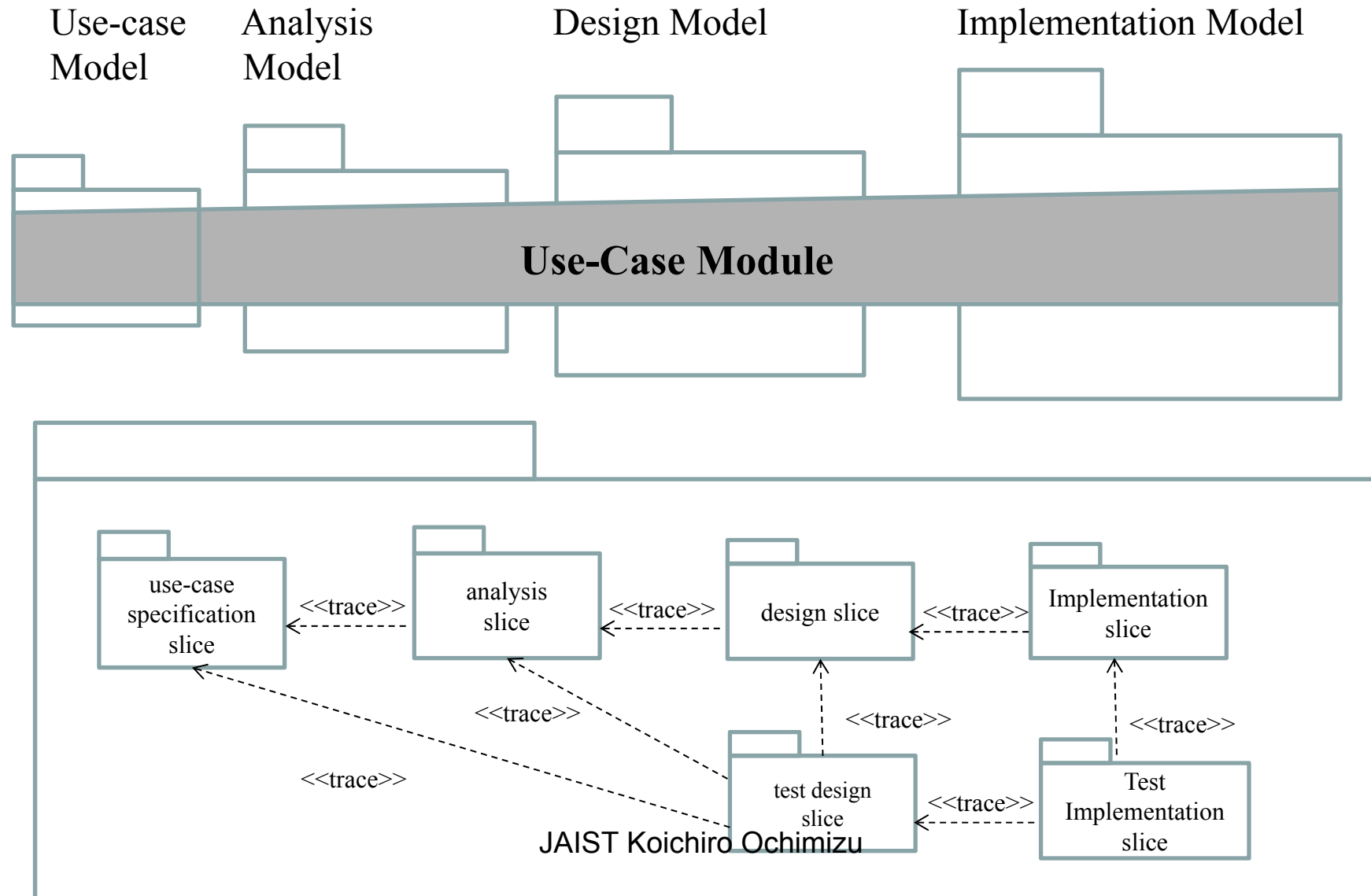
- **Non-use-case –specific slice:**

A use-case slice that adds only classes into the element structure of the system. A non-use-case-specific slice does not contain any aspects.

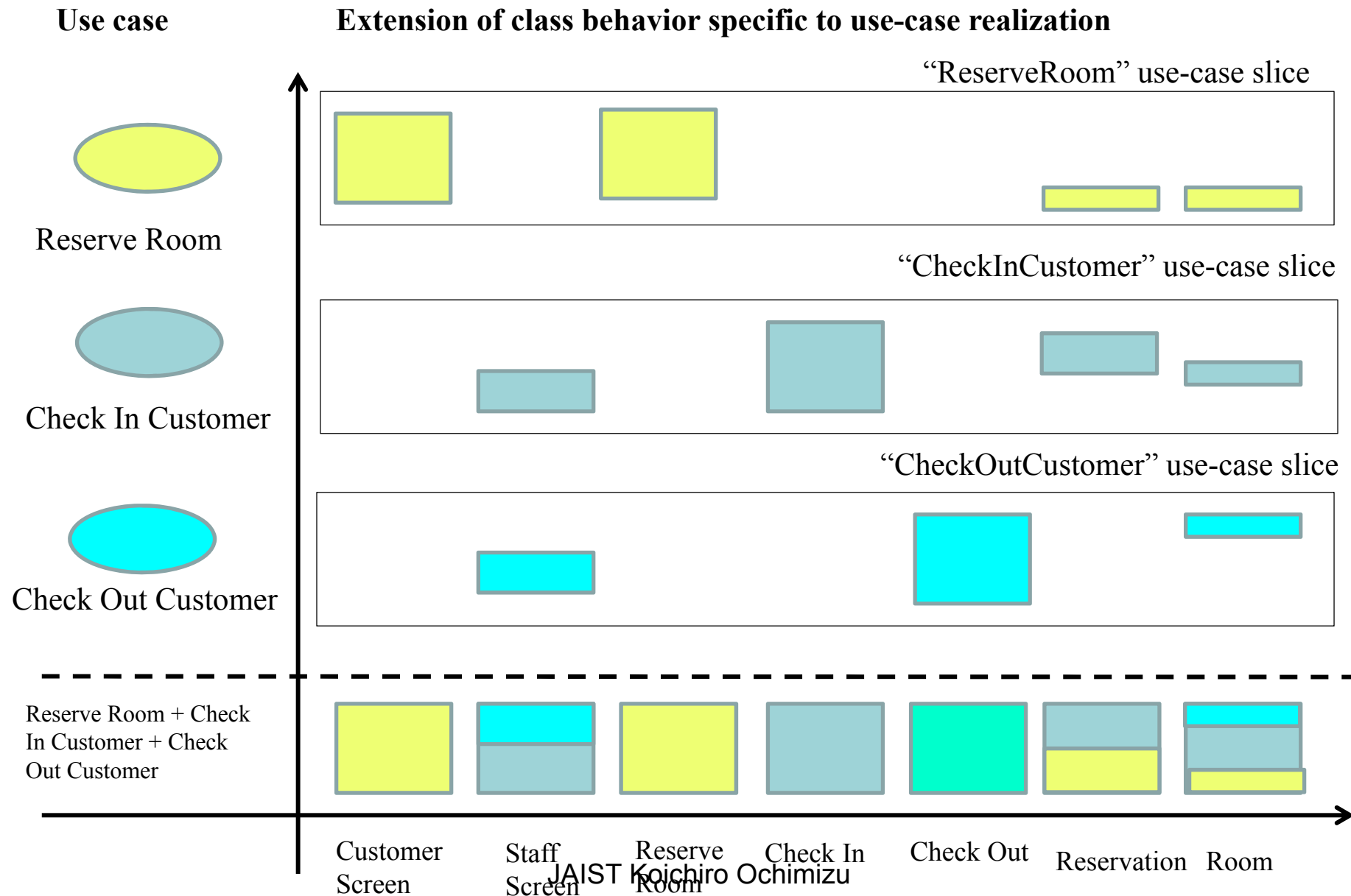
- **Use-case Module:**

The localization of everything about a use case within a single package. It contains many use-case slices.

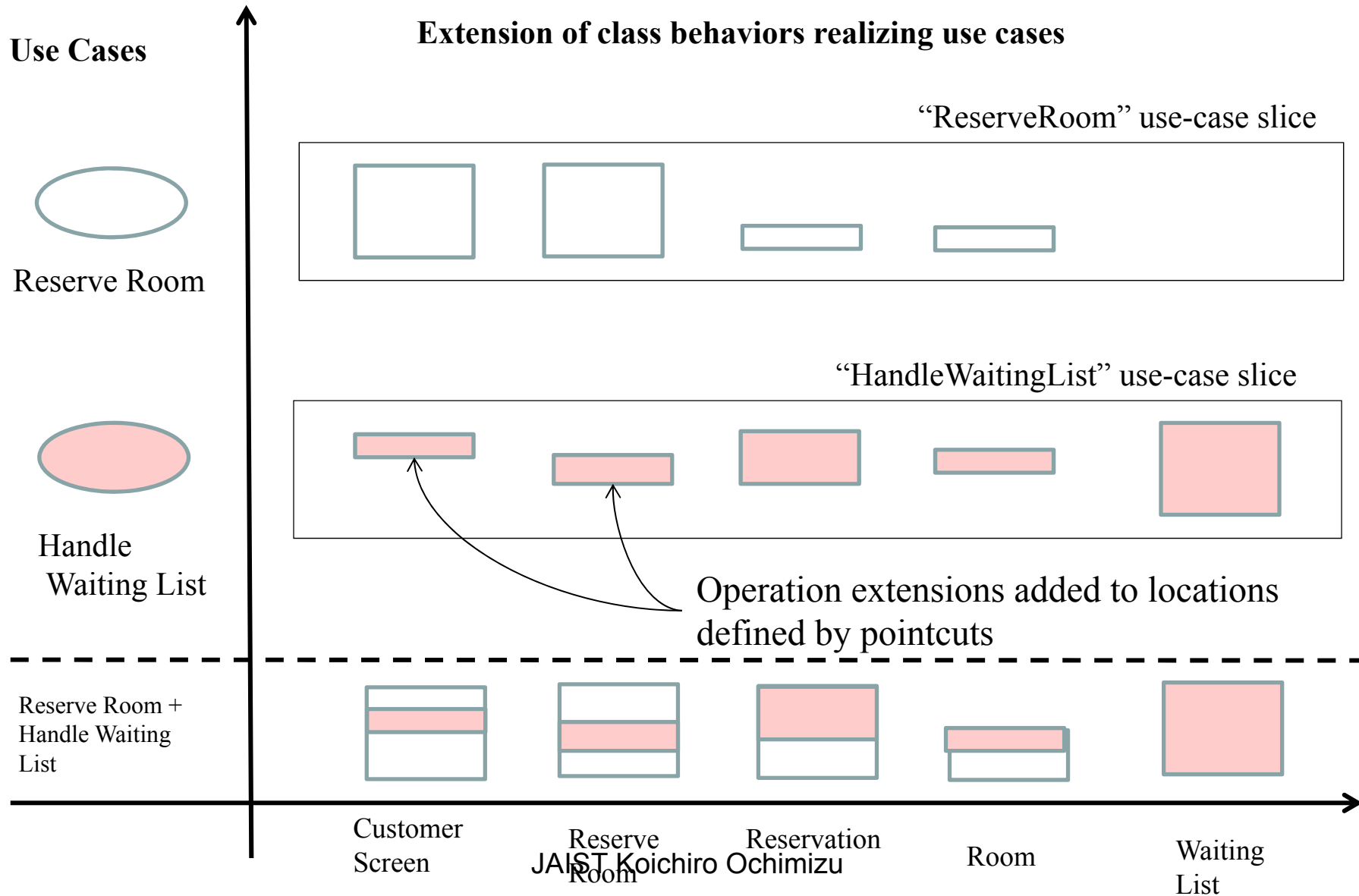
Use-Case Modules Cut Across Models



Composing peer use-case realization with use-case slices



Composing extension and base use-case realizations



Aspect Definition cooperating with use-case driven software development

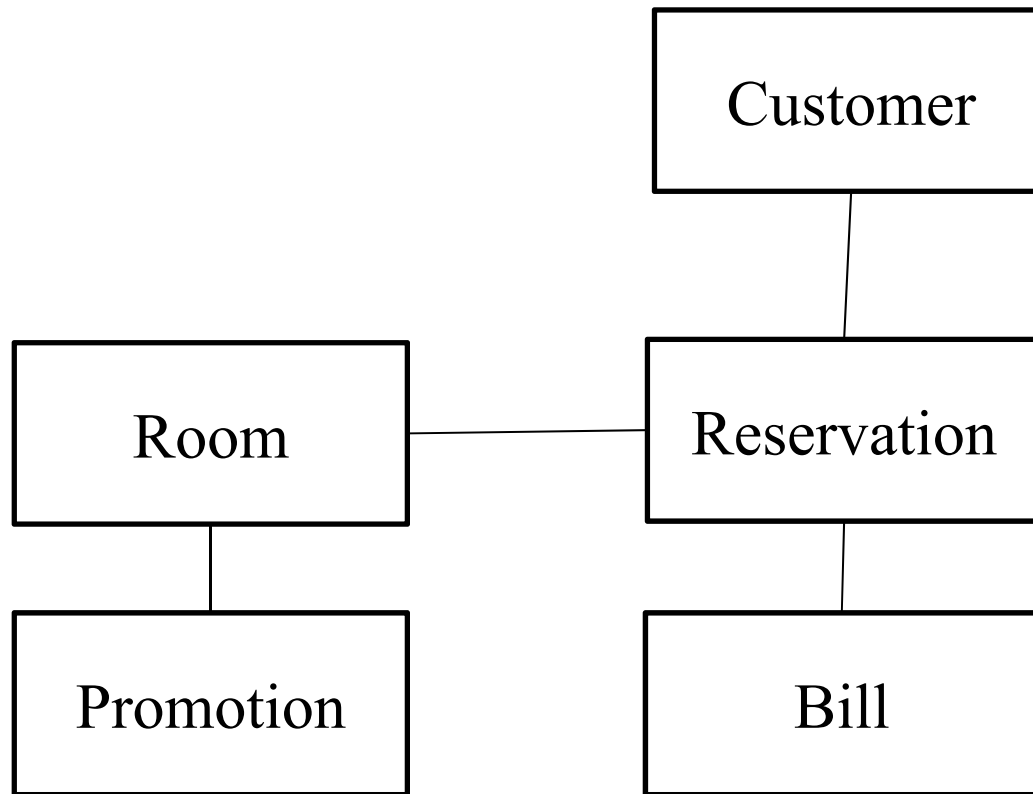
Combining Aspect-orientation with Use-case driven approach

- **Modeling and Capturing Concerns with Use Cases**
 - Capturing Application Use Cases
 - Capturing Infrastructure Use Cases
- **Keeping Concerns Separate with Use-Case Modules**
 - Keeping Peer Use-Case Realizations Separate with Aspects
 - Keeping Extensions Separate with Pointcuts
 - Building Systems with Use-Case Modules
- **Establishing an Architecture Based on Use Case and Aspect**
 - Road to a Resilient Architecture
 - Separating Functional Requirements with Application Peer Use Cases
 - Separating Nonfunctional Requirements with Infrastructure Use Cases
 - Separating Platform Specifics with Platform-Specific Use-Case Slice
 - Separating Tests with Use-Case Test Slices

Capturing Concerns with Use Cases

- **Understanding Stakeholder Concerns**
 - Understanding the Problem Domain(Domain Model)
 - Eliciting System Features
- **Capturing Application Use Cases**
 - Identifying Use-Case Variability
 - Handling Use-Case Variability
 - Dealing with Extension Use Cases
- **Capturing Infrastructure Use Cases**
 - The “Perform Transaction” Use Case

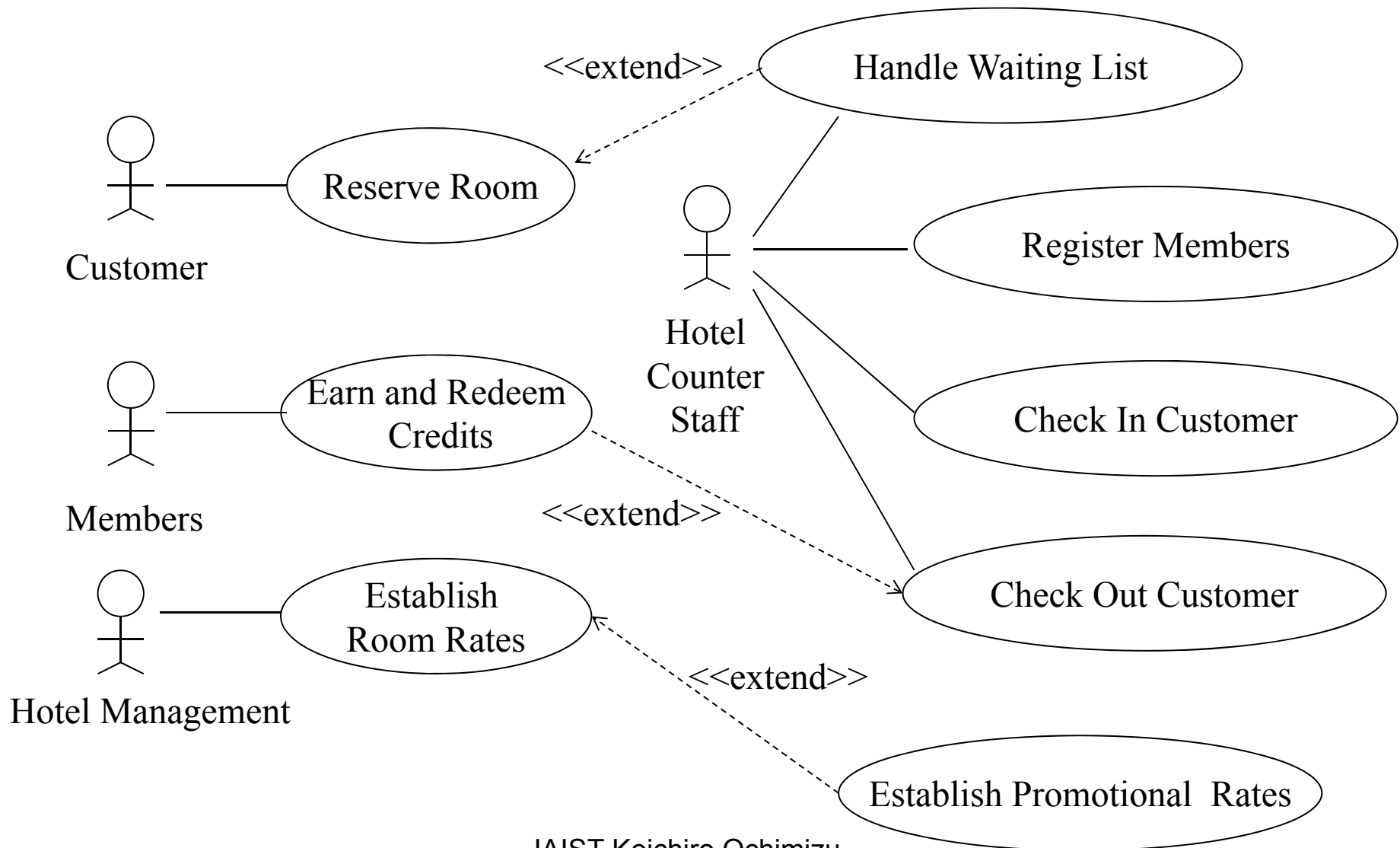
Domain classes for the Hotel Management System



Key Features of Hotel Management System

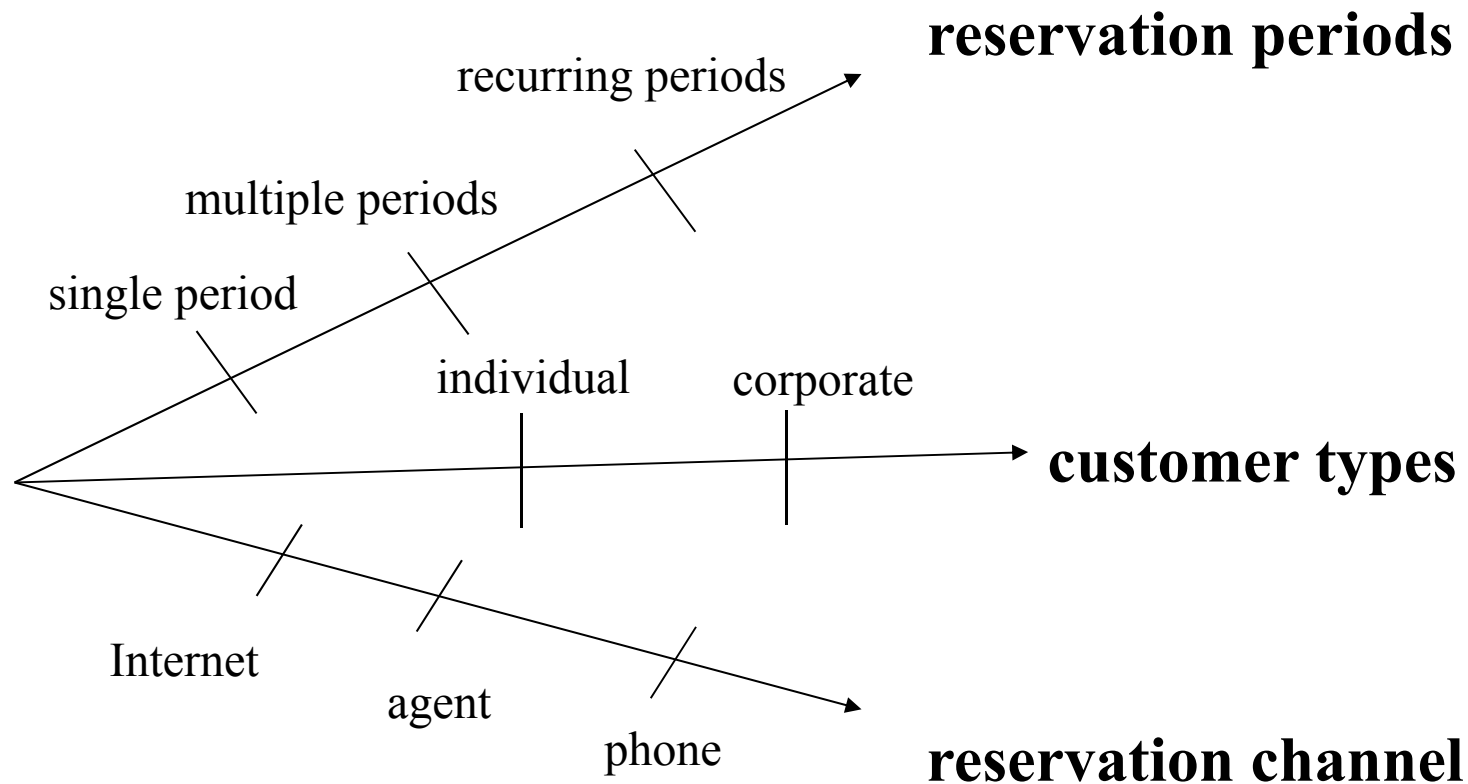
1. A customer can make a reservation for a hotel room.
2. Counter staffs can check in and check out customers.
3. Hotel management can define standard room rates and promotional offers when room rates are reduced for limited periods.
4. Members can accumulate loyalty points and use them for any payment.
5. There will be a waiting list in case the rooms are fully reserved.
6. Different types of customer(individual, corporate, members) must be handled.
7. The room reservation can be over different channels, such as through agents, internet, or via phone.
8. The system has to be Web-enabled.
9. The system will store all records in a relational databases.
10. For audit purposes, all transactions in the system have to be logged.
11. Only authorized personnel can perform their functions.
12. To promote ease of use, the system will track the users' preferences and use them as default.
13. All retrieval of record should take no longer than 2 seconds.

Application use cases for Hotel Management System




Identifying and Handling Use-Case Variability

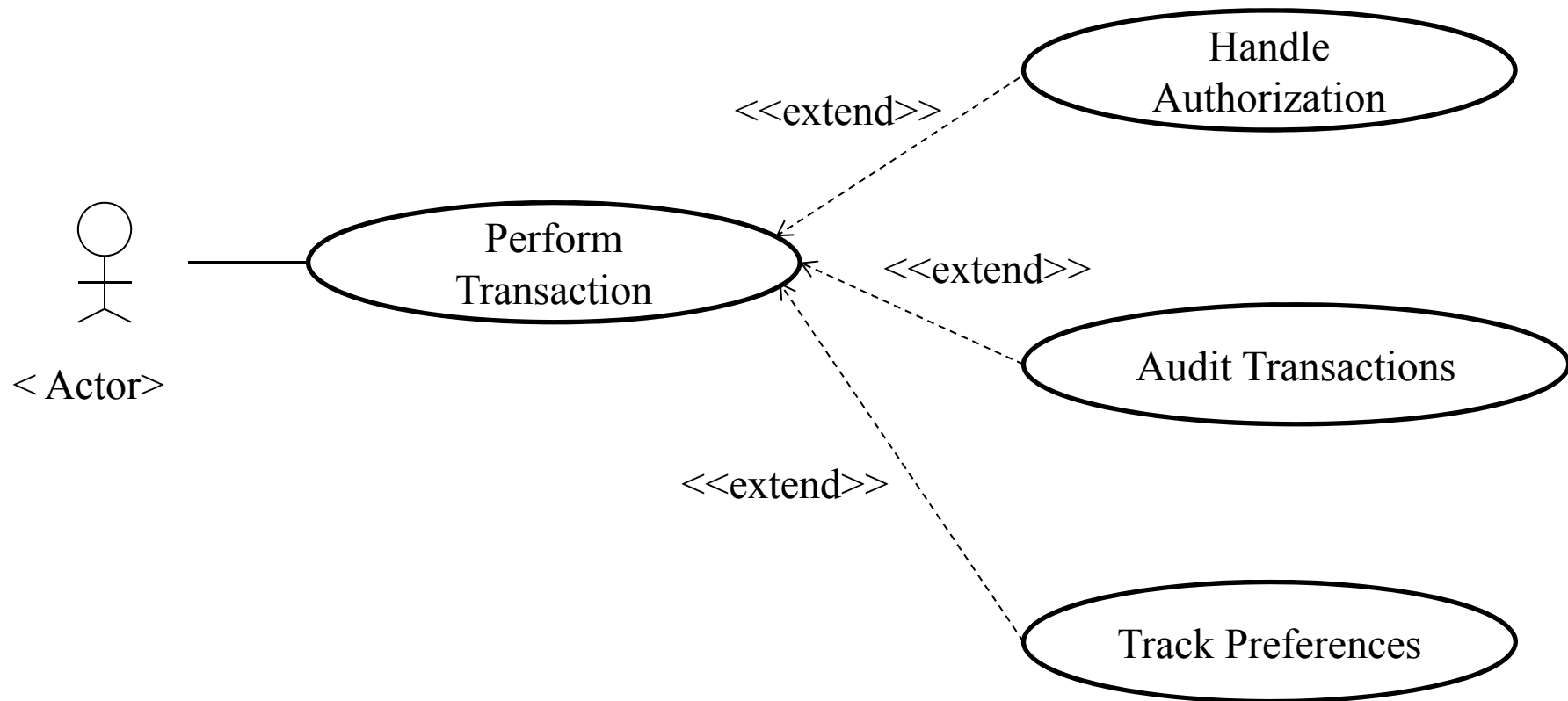
(this is important for test cases design)



Dealing with Extension Use Cases

Earn and Redeem Credits	
Flows {basic} View Credit Balance {basic} Redeem Credits {alt} Make Payment with Credits { around MakingPayment} {alt} Earn Credits { after MakingPayment}	
Extension Pointcuts MakingPayment = Check Out Customer. Collect Payment	

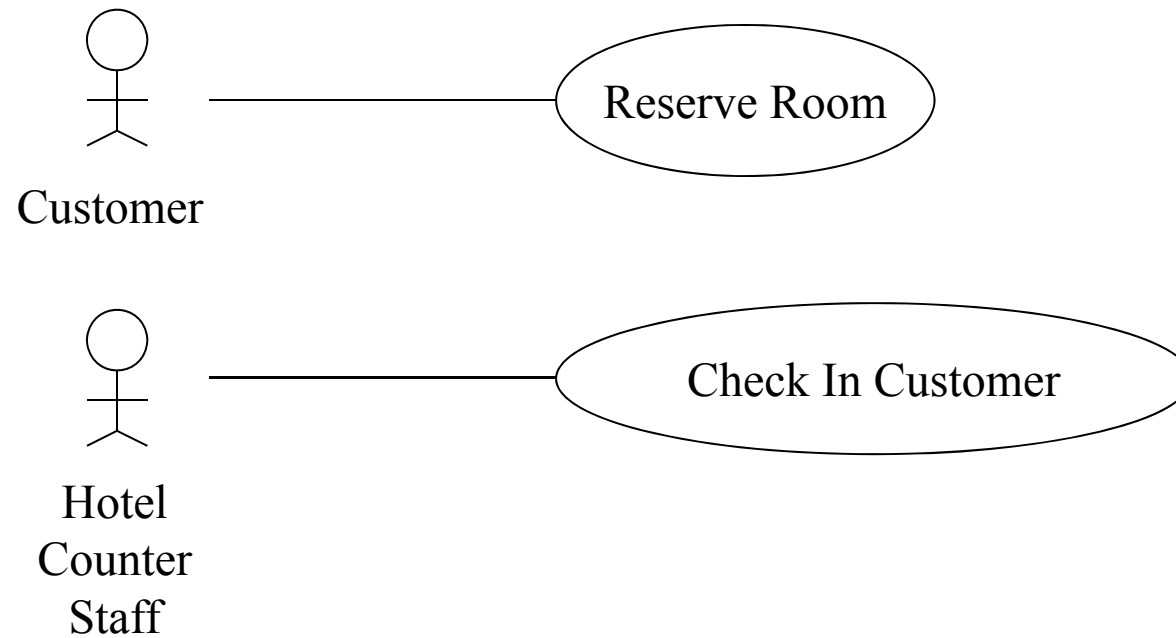
Capturing and Structuring Infrastructure Use Cases



Keeping Concerns Separate with Use-Case Modules

- **Keeping Peer Use-Case Realizations Separate with Aspects**
 - Realizing Peer Use Cases by Collaborations
 - Overlap between Peer Use-Case
 - Keeping Use-Case Specifics Separate
 - Composing Use-Case-Specific Classes
 - Dealing with Overlap
- **Keeping Extensions Separate with Pointcuts**
 - Realizing Extension Use Cases
 - Keeping Modularity of Extension Use-Case
- **Building Systems with Use-Case Modules**
 - Design and Implementation Model
 - Use-Case Modules Cut Across Models
 - A Use-Case Module Contains Use-Case Slices

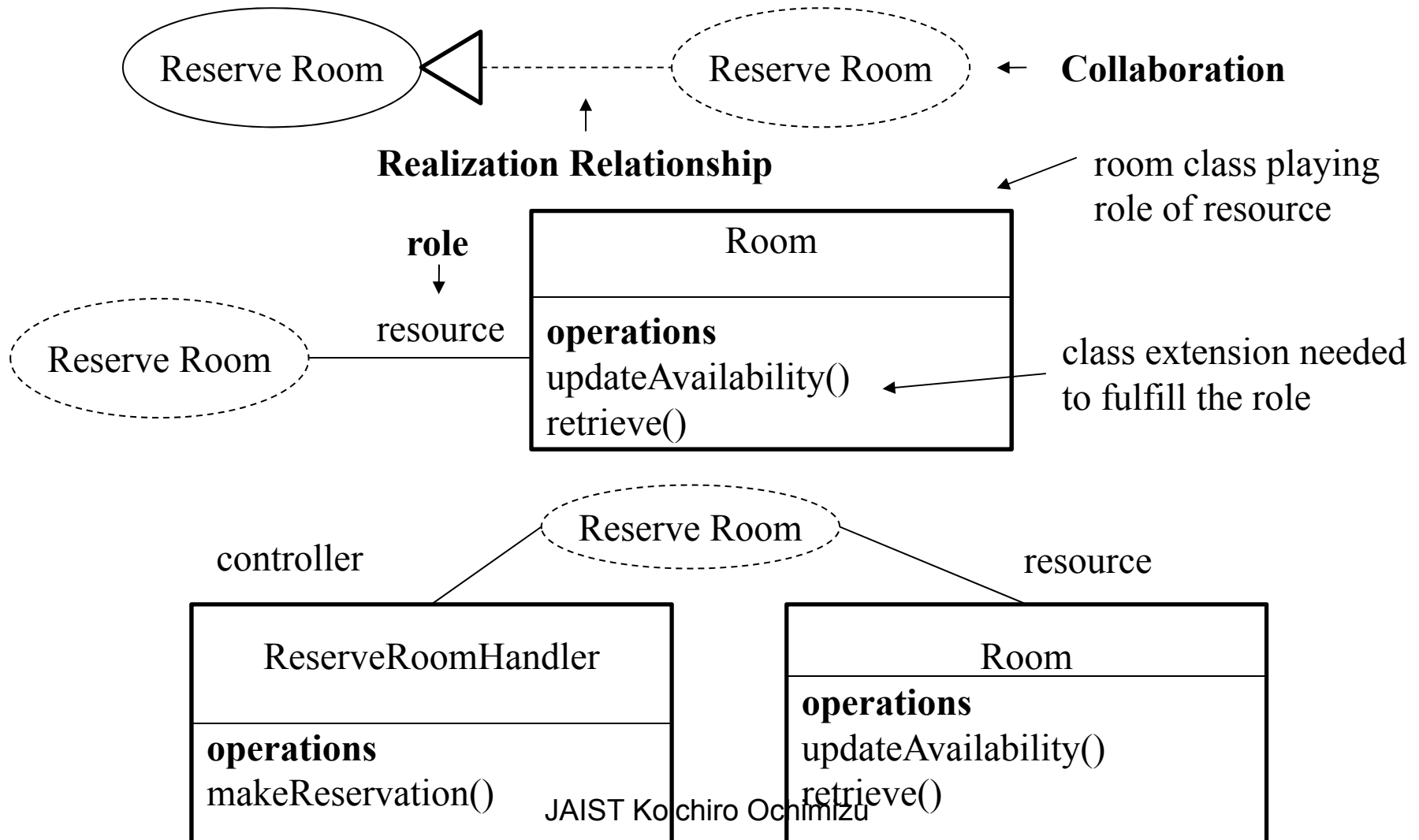
Peer Use Cases



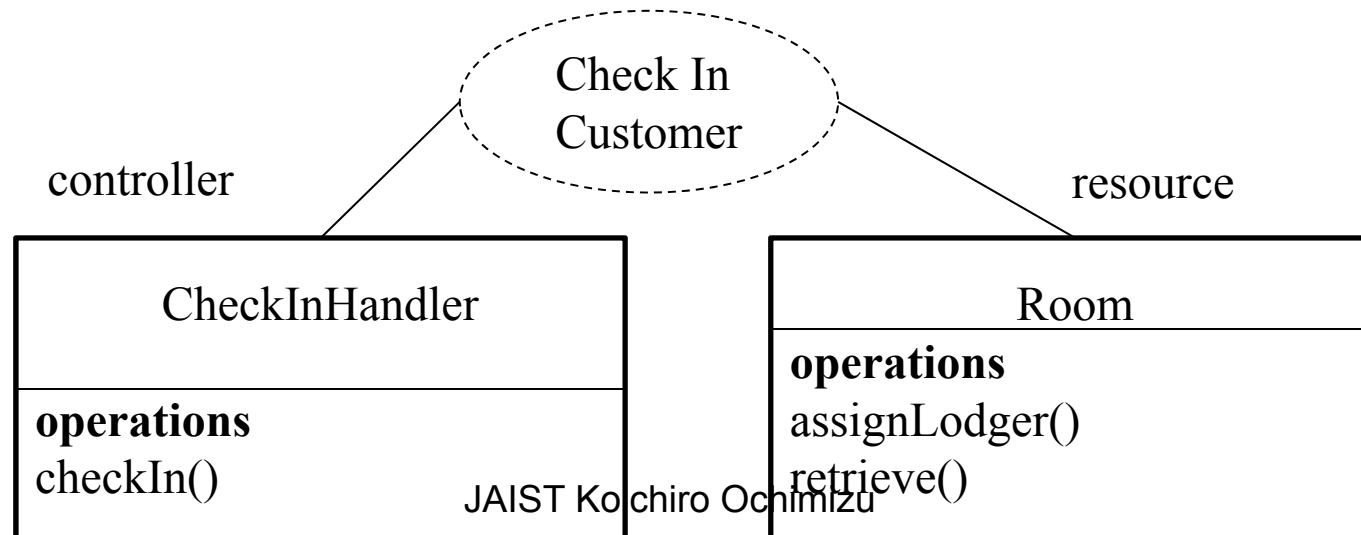
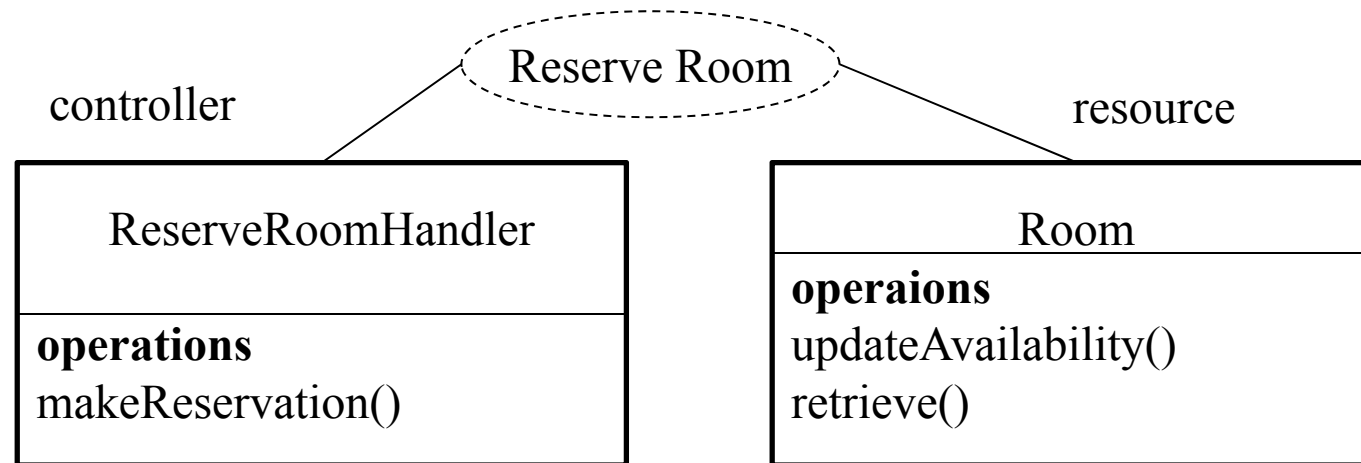
Use Case Realization by a Collaboration

External Perspectives of System

Internal Perspective of System

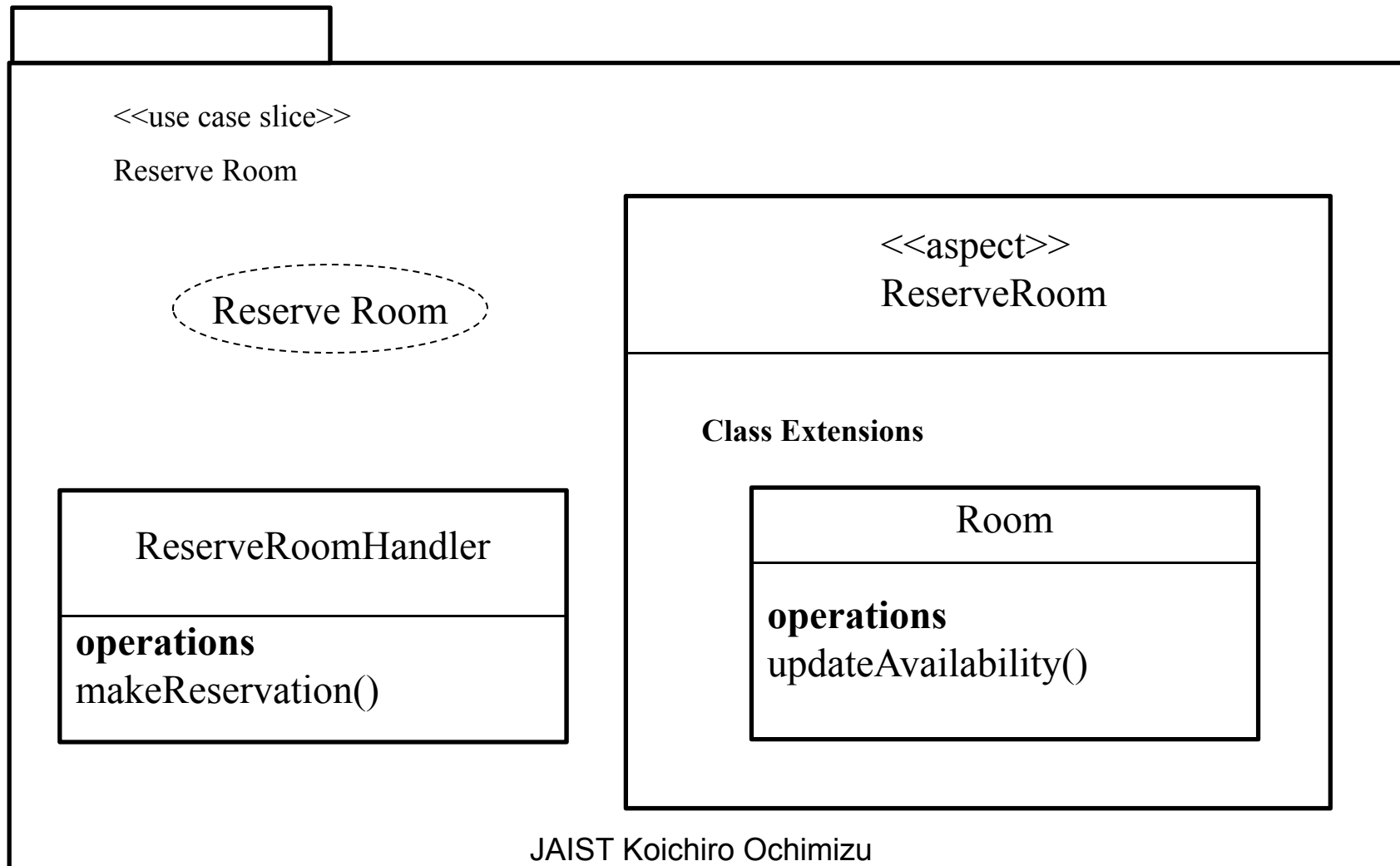


Overlap between Peer Use-Case



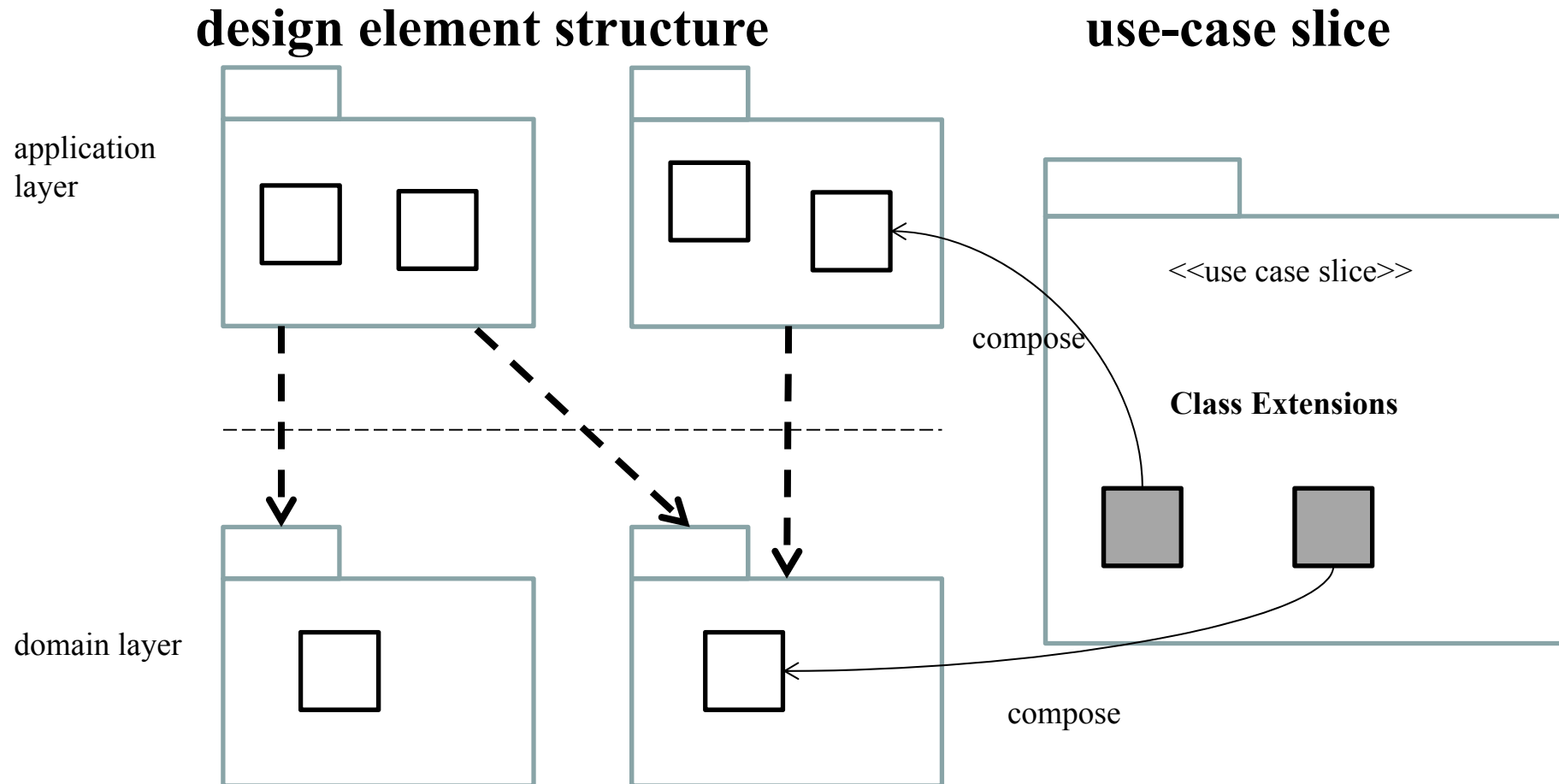
Keeping Use-Case Specifics Separate

Preserving use-case modularity with use-case slices and aspects



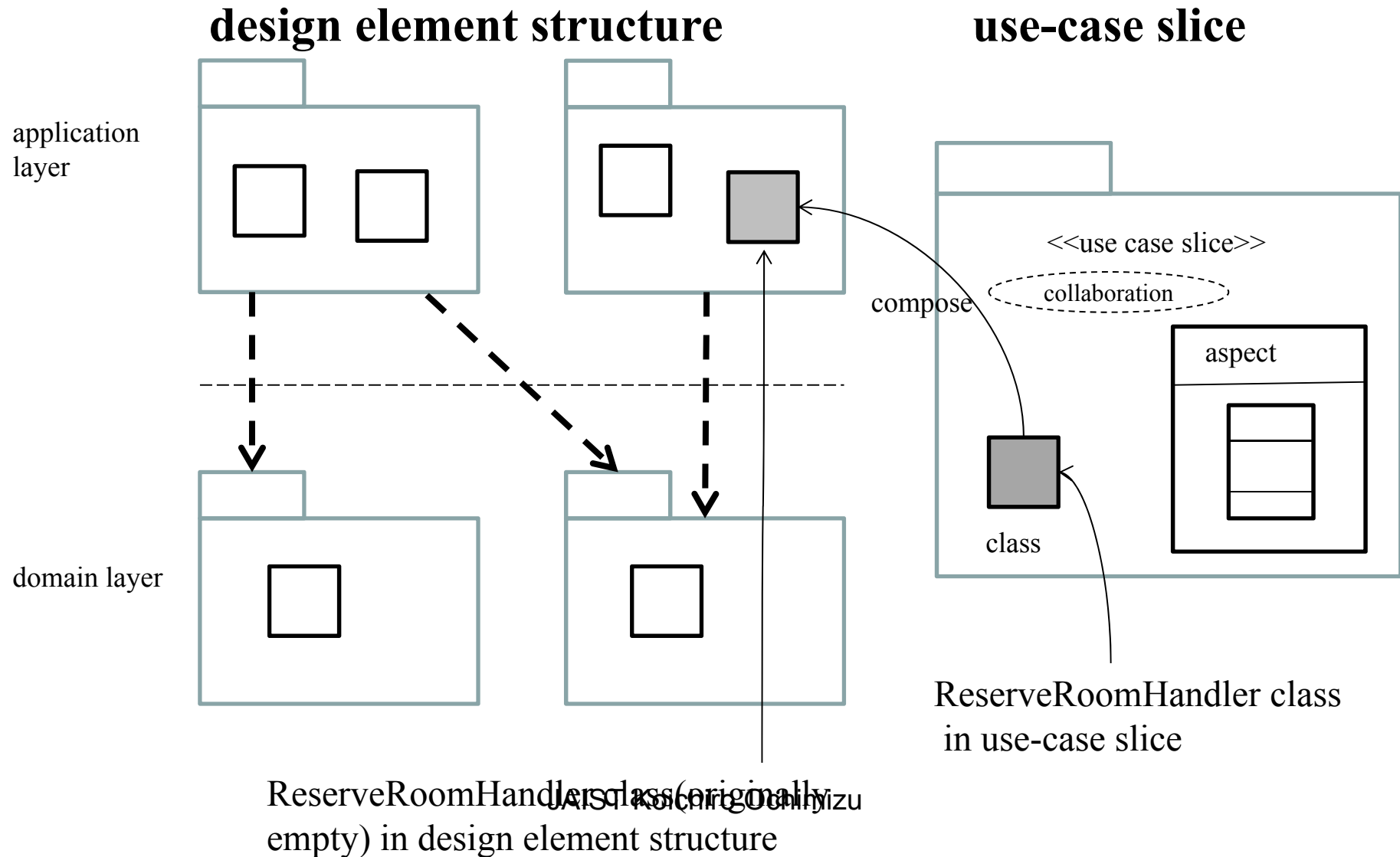
Composing Use-Case-Specifics Classes

overlaying classes within use-case slice onto design element structure

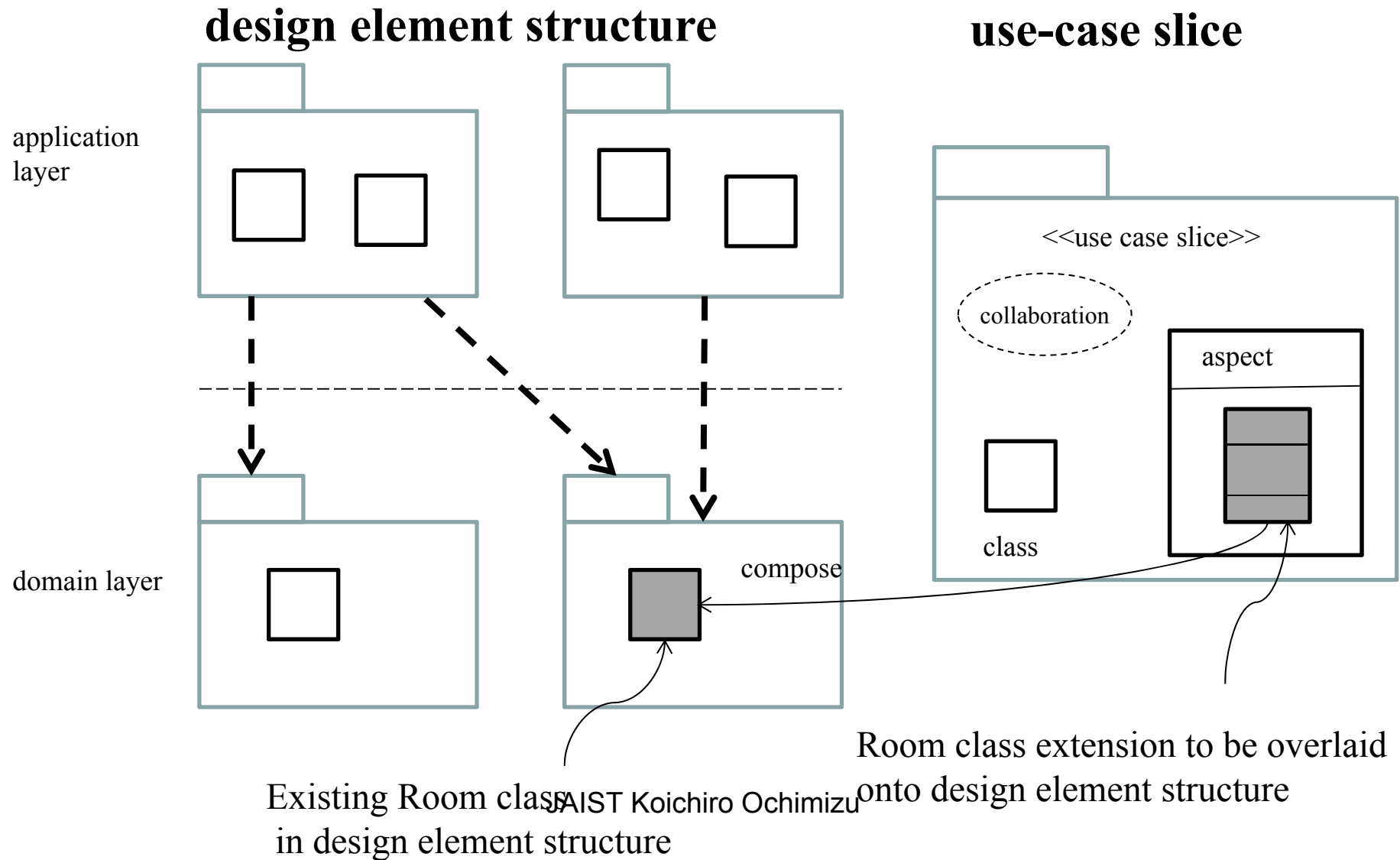


Need to identify the class uniquely in the design element structure. This is achieved by giving a fully qualified name in the design element structure. A compiler would compose the content in the use-case slice.

Overlaying classes within use-case slice onto design element structure

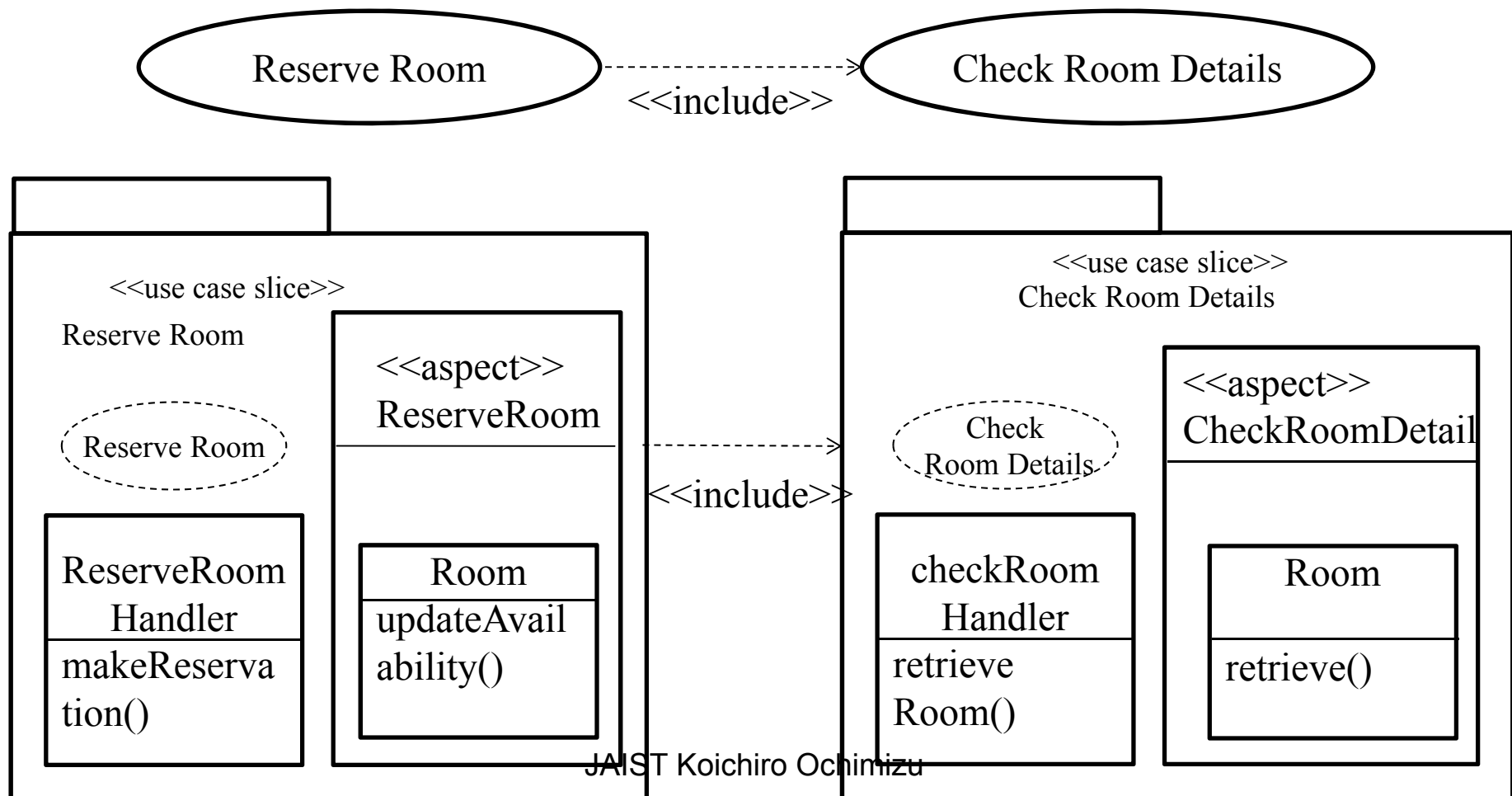


Overlaying classes within use-case slice onto design element structure



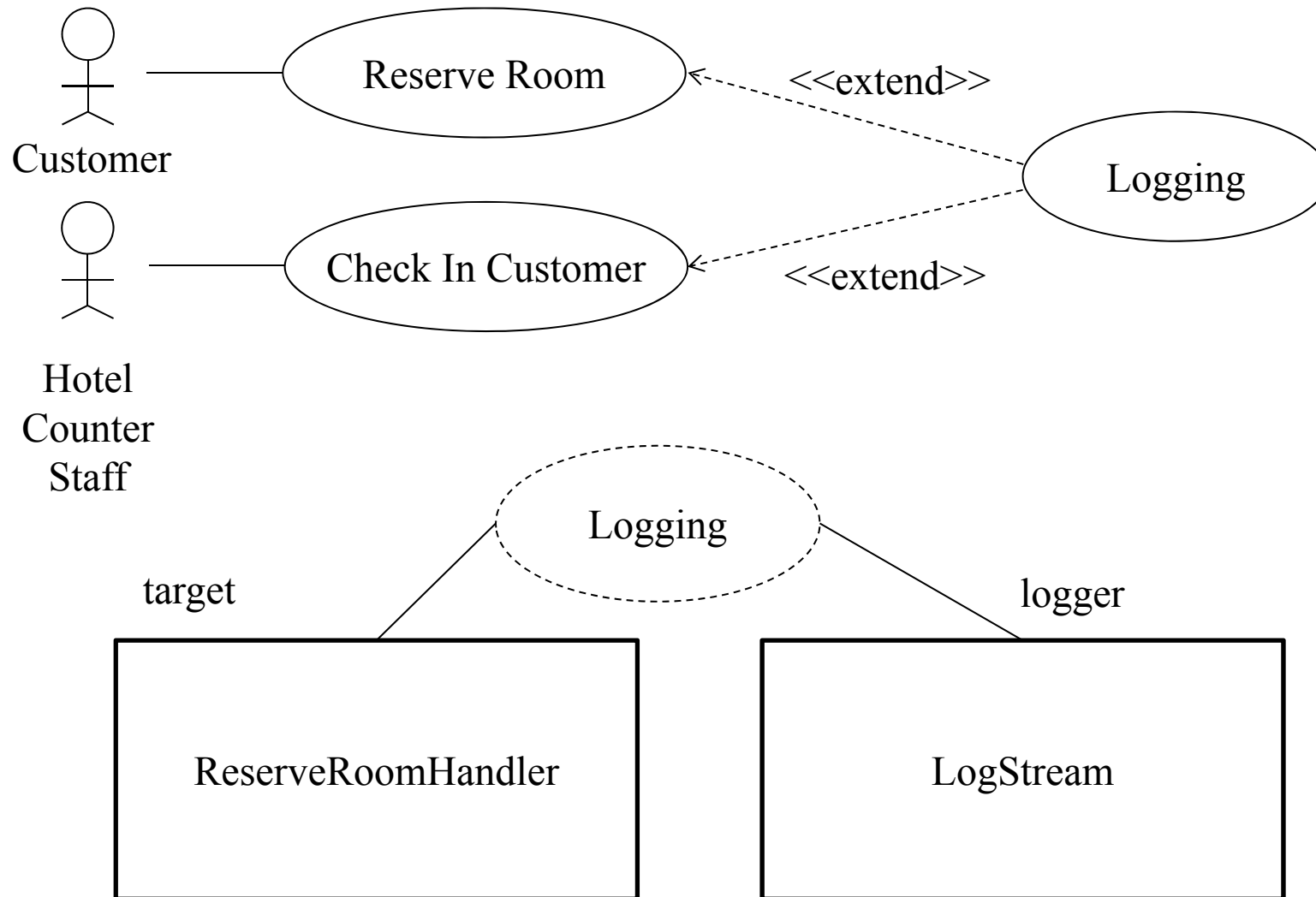
Dealing with Overlap

The retrieve() operation in the Room class is used in both “Reserve Room” and “Check In Customer” use-case slices.

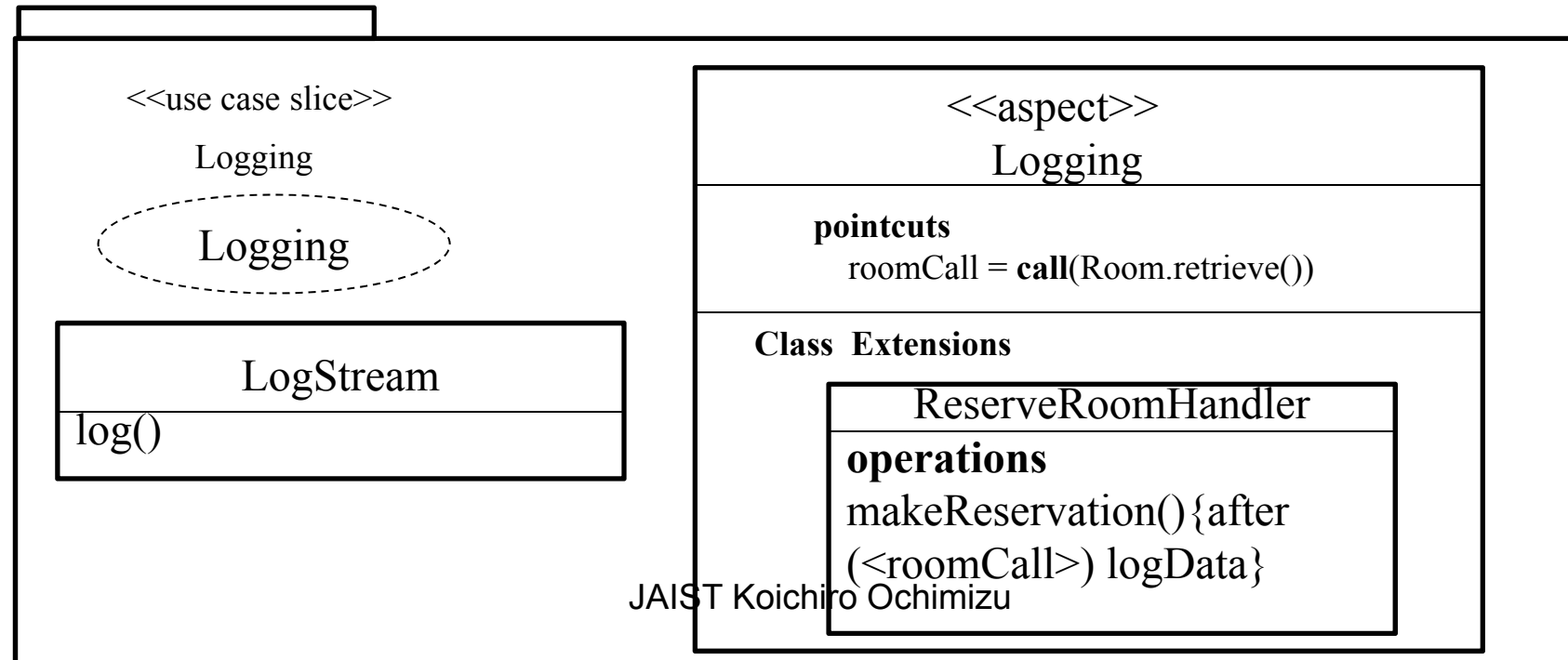
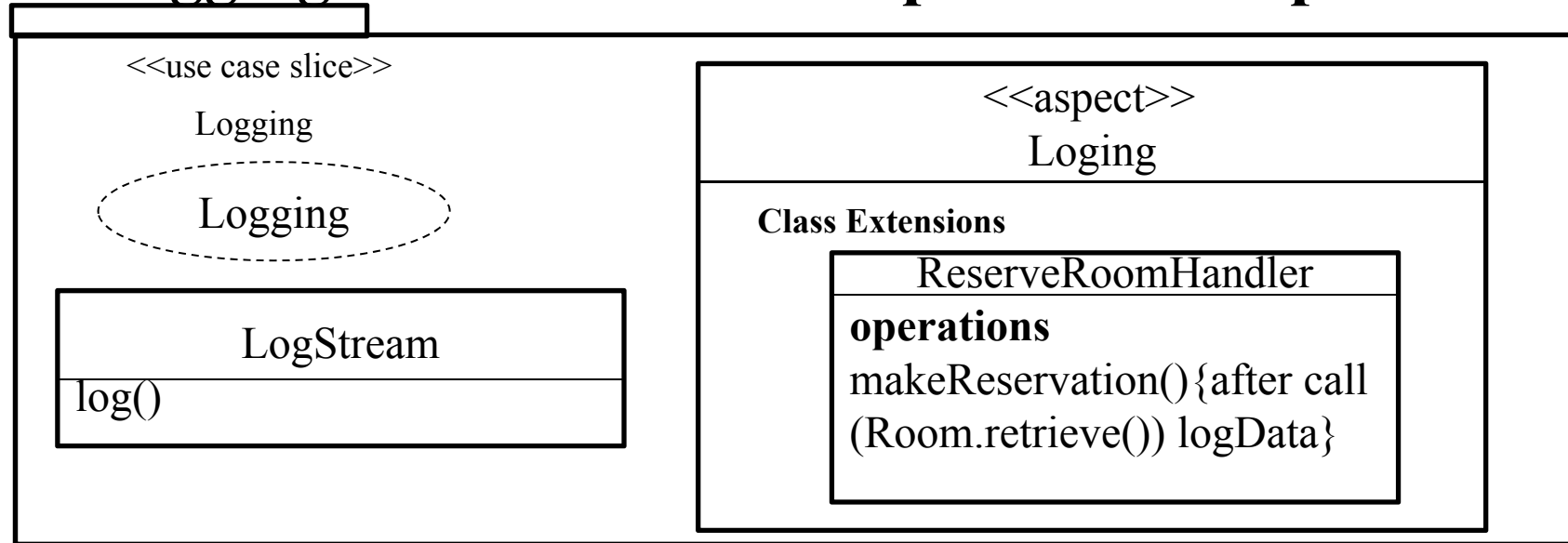


Keeping Extensions Separate with Pointcuts

Logging use case and Roles in Logging extension



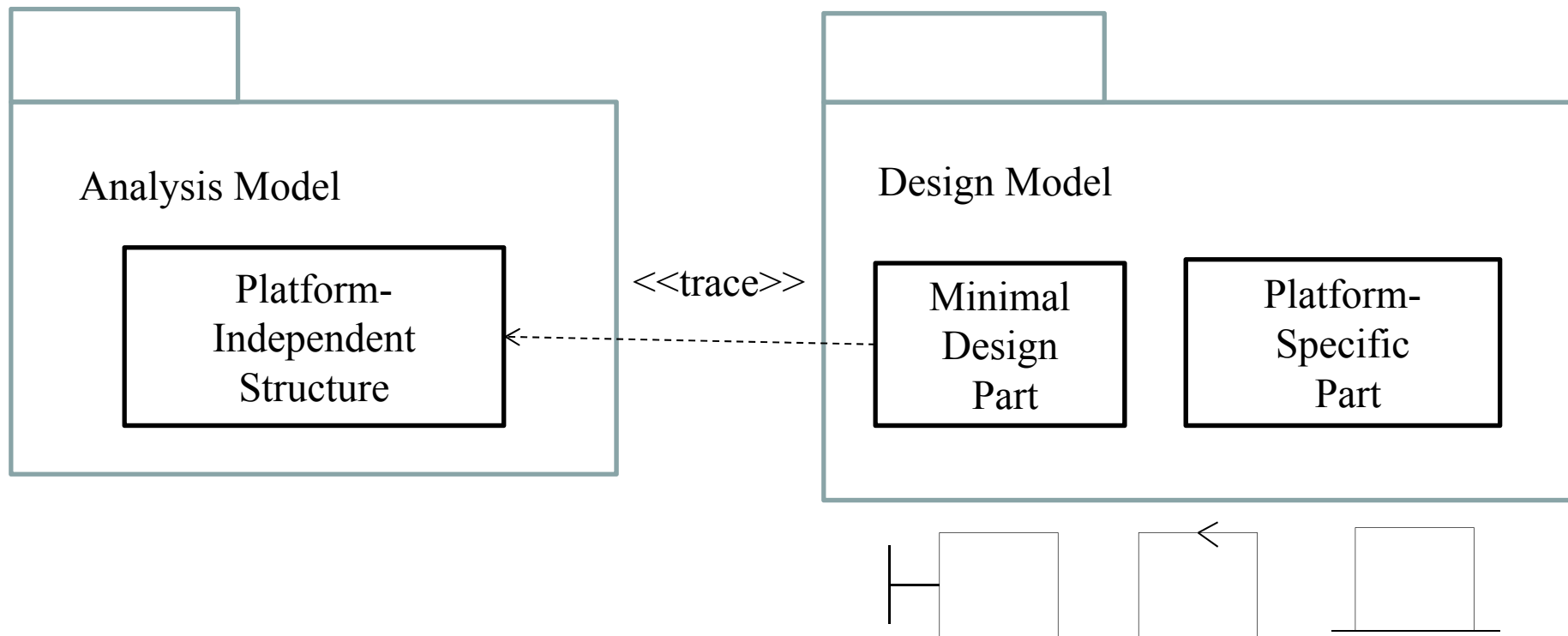
Logging use-case slice with pointcut compartment



Design and Implementation Models

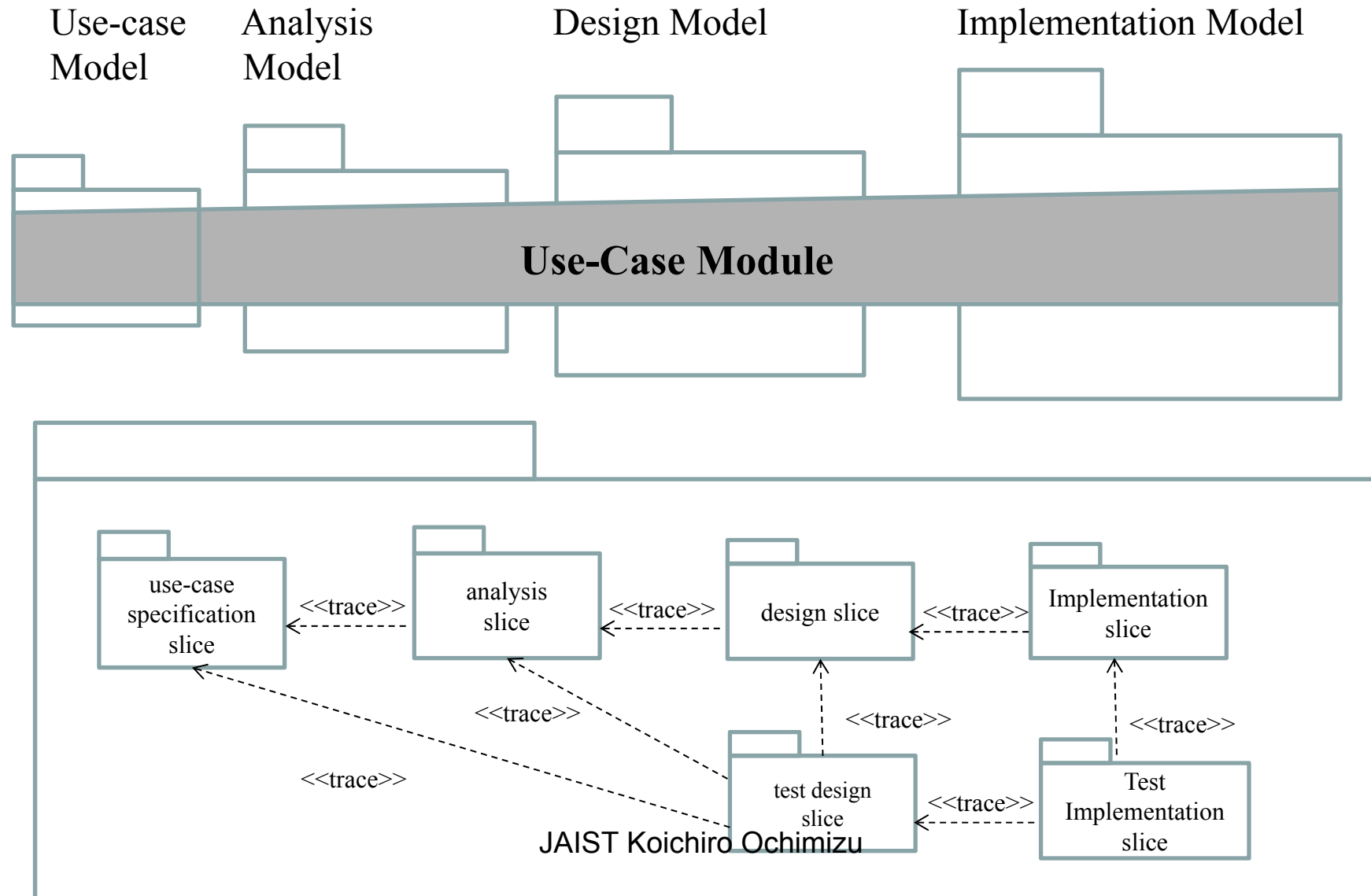
- To describe platform specifics, the design model must contain more constructs and, hence, more structures than the analysis model. It contains the following structures.
- **Deployment structure**
 - nodes and links
- **Process structure**
 - processes and threads
- **Design element structure**
 - Design classes organized into layers, subsystem, packages
- **Use-case design structure**
 - The use-case structure runs orthogonally across the design element structure. It comprises use-case slices, aspects, class extensions and so on.

Preserving the structure of analysis model in the design model

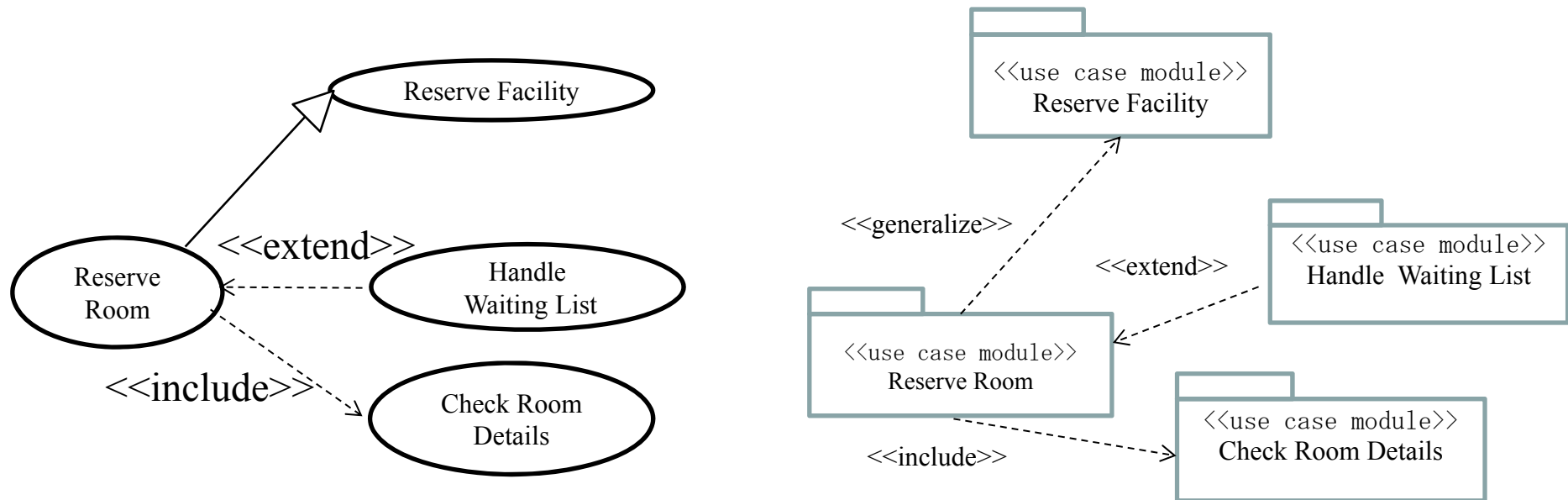


Stereotyped design classes

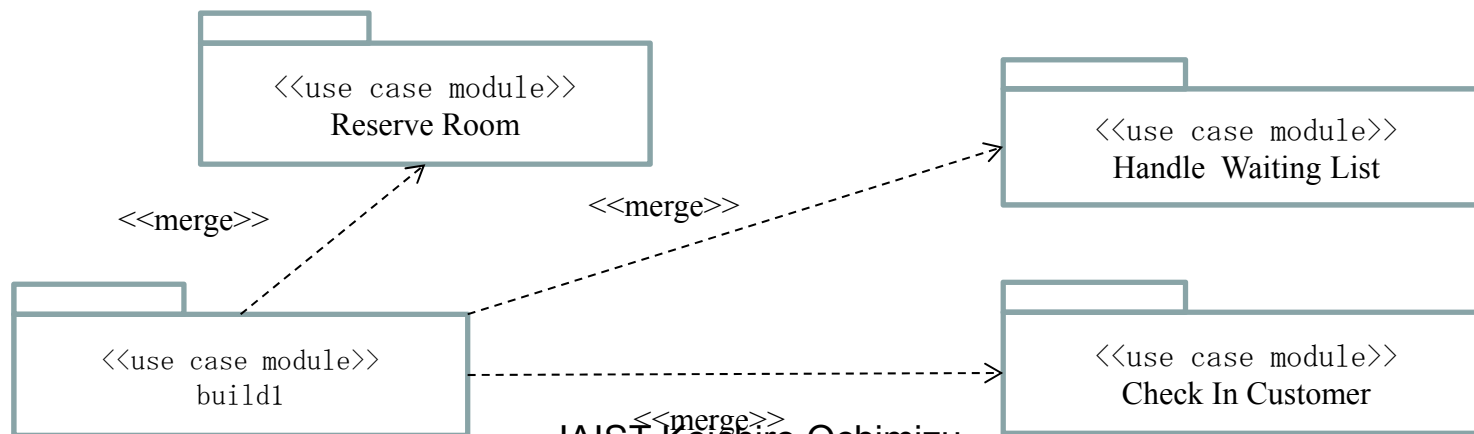
Use-Case Modules Cut Across Models



Deriving use-case-module relationships from use cases



Composing use-case module

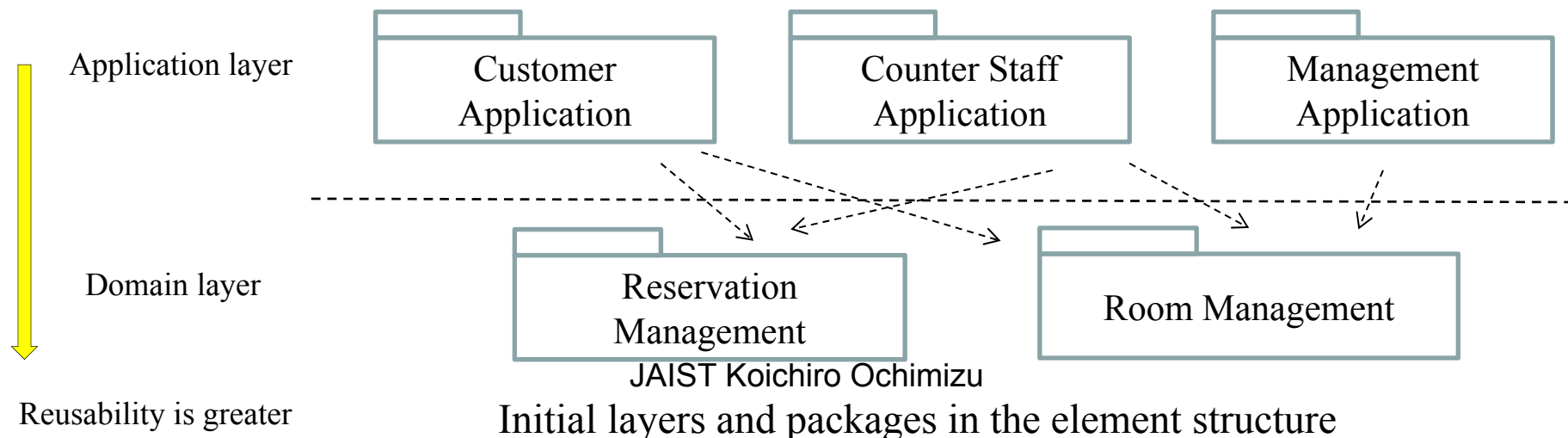


What is a Good Architecture?

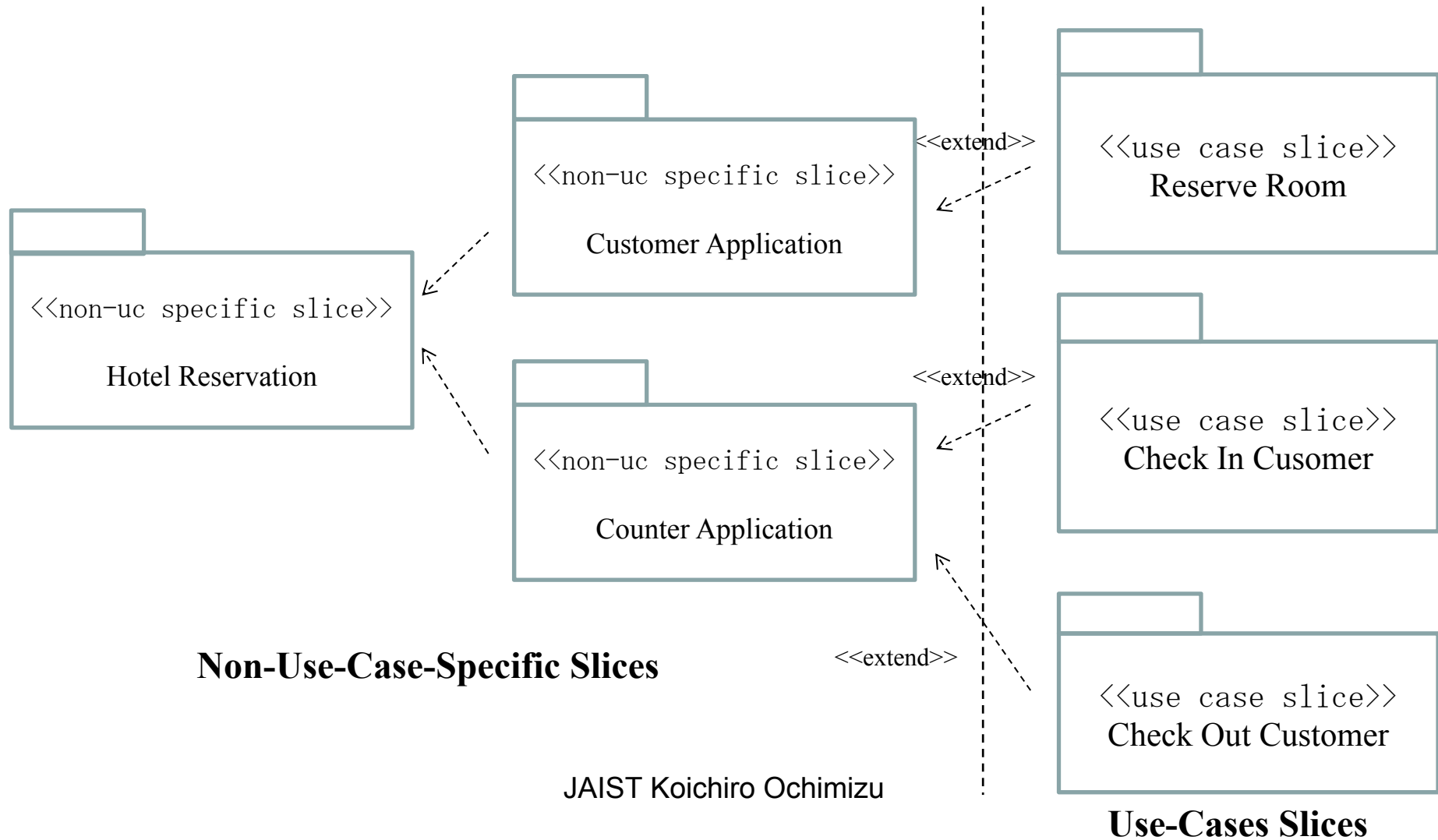
- **Separating Functional Requirements**
 - Functional requirements address different end-user concerns and will evolve separately
- **Separating Nonfunctional from Functional Requirements**
 - Nonfunctional requirements usually specify the desired quality attributes of the system. These are provided by some infrastructure mechanisms.
- **Separating Platform Specifics**
 - Even for a single infrastructure mechanism such as authorization, you still have many technologies(e.g. through HTTP cookies, session identifiers, etc.) to choose from. These technologies are often platform- and vendor-specific. Thus we need to keep platform specifics separate.
- **Separating Tests from the Units Being Tested**
 - As part of implementing a test, you must perform some control and instrumentation(e.g. debugging, tracing, logging, etc.) . Such control and instrumentation behavior have to be removed after the test is conducted.

Platform-Independent Structure

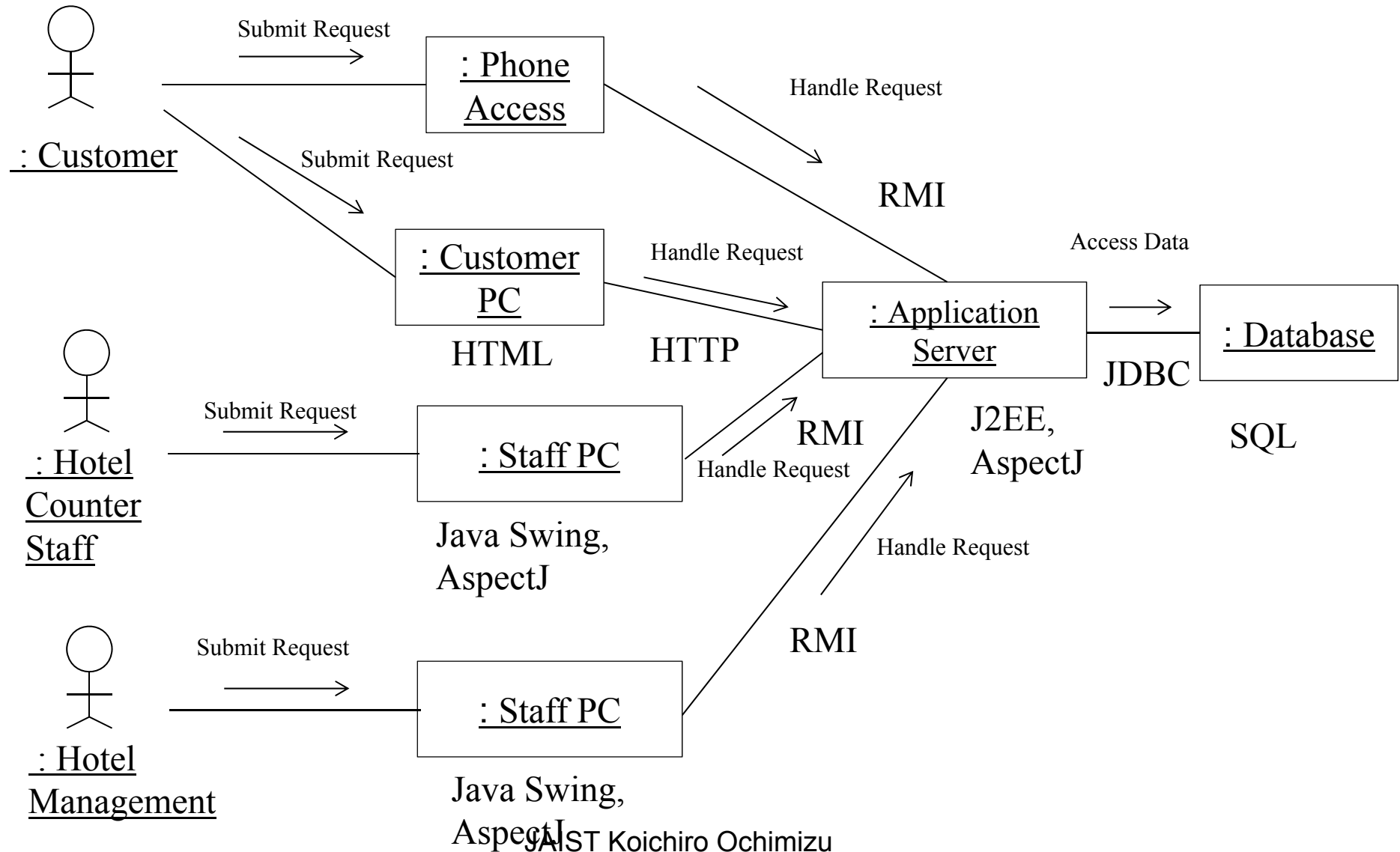
- **Element Structure:** identifies where elements of the system are located in the element namespace
 - **Layer:** used to group software elements that are on the same level of abstraction
 - **Application Layer:** contains elements that realize workflows in the use cases supporting the primary actors of the system
 - **Domain Layer:** contains elements representing significant domain concepts



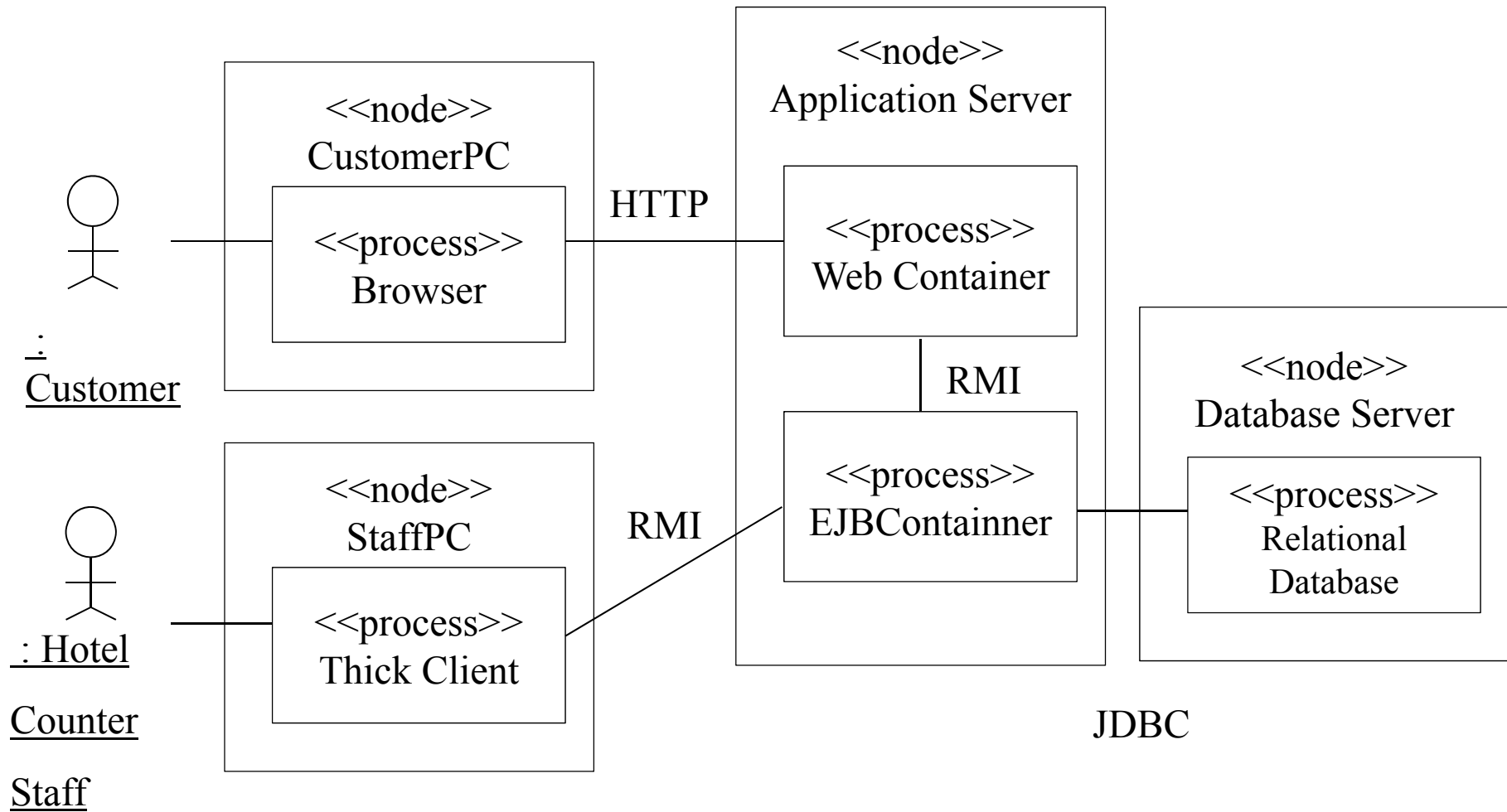
Use-case structure



Deployment structure for Hotel Management System design model



Process structure for Hotel Management System design model



```
graph LR; UCDS[Use-Case Design Slice] -- "Use-Case Presentation" --> UC_P[Use-Case Presentation]; UC_P -- "Use-Case Persistence" --> UC_Pers[Use-Case Persistence]; UC_Pers -- "Use-Case Distribution" --> UC_D[Use-Case Distribution]; UC_D -- "Use-Case Design" --> UC_Min[Minimal Use-Case Design]; UC_Min -- "Use-Case Presentation" --> UC_P; UC_Pers -- "Use-Case Persistence" --> UC_Pers; UC_D -- "Use-Case Distribution" --> UC_D; UC_Min -- "Use-Case Design" --> UC_Min;
```

The diagram illustrates the relationships between five use case slices, each represented by a box with a tab on top. The slices are: Use-Case Design Slice, Minimal Use-Case Design, Use-Case Presentation, Use-Case Persistence, and Use-Case Distribution. The relationships are as follows: Use-Case Design Slice is connected to Use-Case Presentation by a solid line labeled "Use-Case Presentation". Use-Case Presentation is connected to Use-Case Persistence by a solid line labeled "Use-Case Persistence". Use-Case Persistence is connected to Use-Case Distribution by a solid line labeled "Use-Case Distribution". Use-Case Distribution is connected to Minimal Use-Case Design by a solid line labeled "Use-Case Design". Additionally, there are dashed arrows labeled "Use-Case Presentation" from Use-Case Presentation to Minimal Use-Case Design, and "Use-Case Persistence" from Use-Case Persistence to Use-Case Persistence.