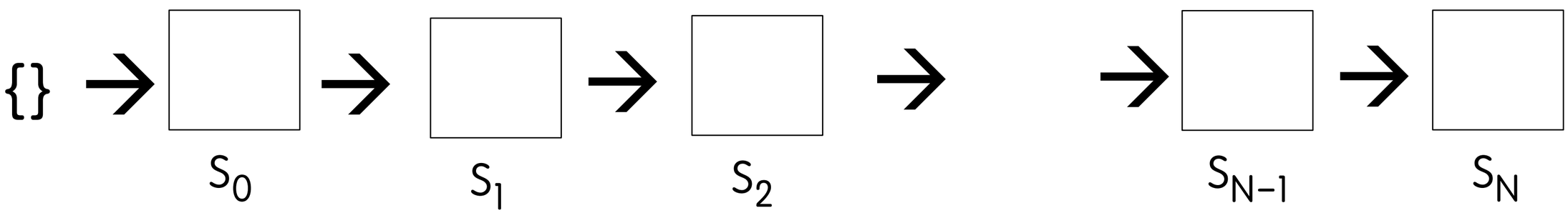
ThoughtWorks®

Domain Use Cases & Aspect Thinking End-to-End Evolutionary Design

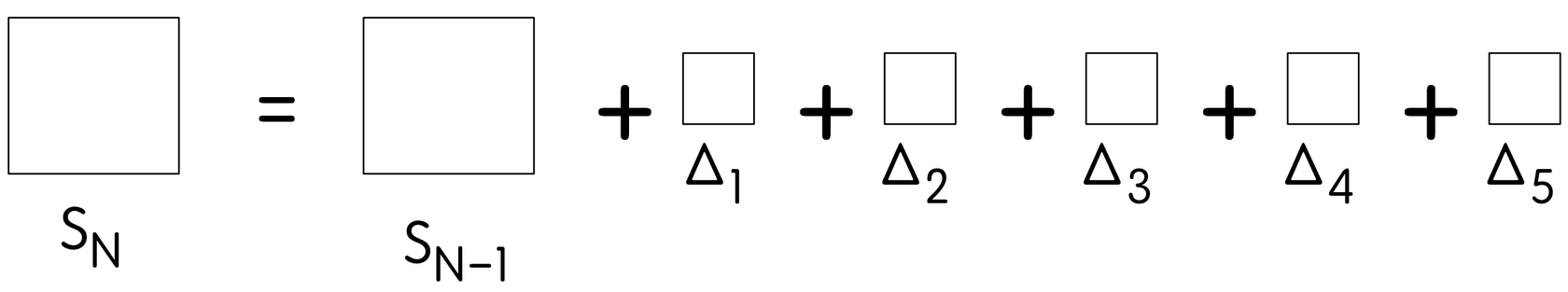
Pan-Wei Ng, Ph.D.
黄邦伟 博士
pwng@thoughtworks.com

On Software Evolution and Change Modularity

- Normal case for software evolution/development is from S_{N-1} to S_N
- with { } to S_0 as a special case

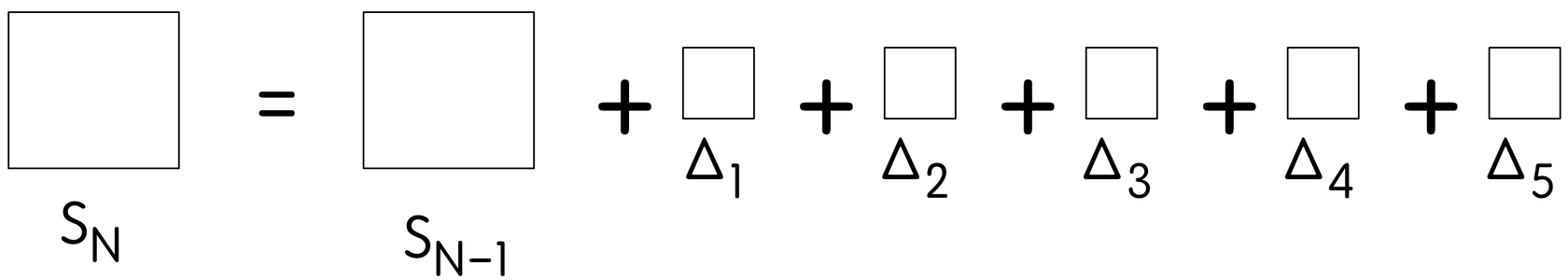


- A software evolution/development is a composition of changes



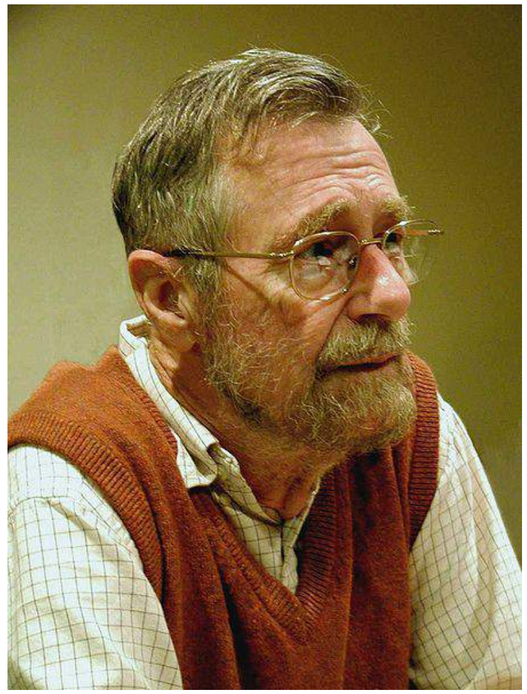
On Change Modularity

- A software evolution/development is a composition of changes



- Every release is a “composition” of ”changes”
  - Every change is an identifiable module
  - Every change has a natural and unique place in the system structure
    - Requirements architecture - The timeless beauty of good software
    - The art of moving from variability in time to variability in space
  - Every change has a lifecycle from idea “requirements” to “composition”

Separation and Modularity of Concerns ()



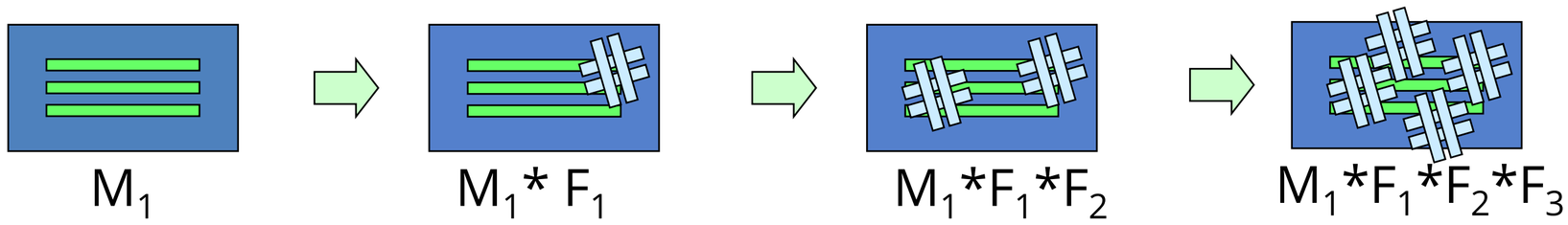
by Edsger W. Dijkstra in his 1974 paper "On the role of scientific thought

… We (*want to*) know that a program must be correct ….
.. we can study it from that viewpoint only …
But nothing is gained --on the contrary!-- by tackling these various aspects simultaneously

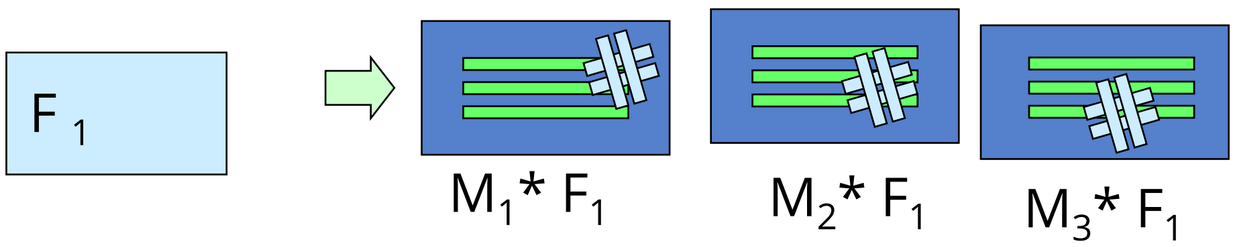
It is what I sometimes have called "the separation of concerns", which, even if not perfectly possible, is yet the only available technique for effective ordering of one's thoughts, that I know of.

Effects of Poor Separation of Concerns

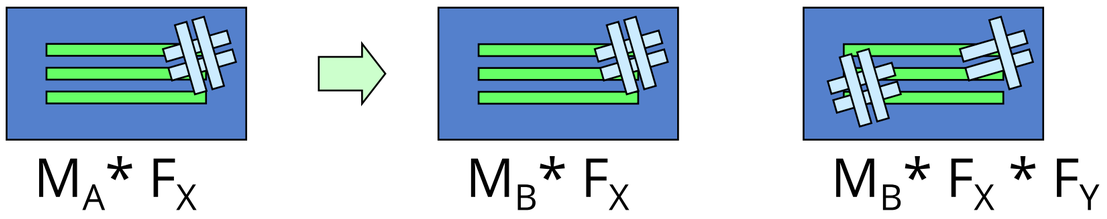
- Tangling



- Scattering

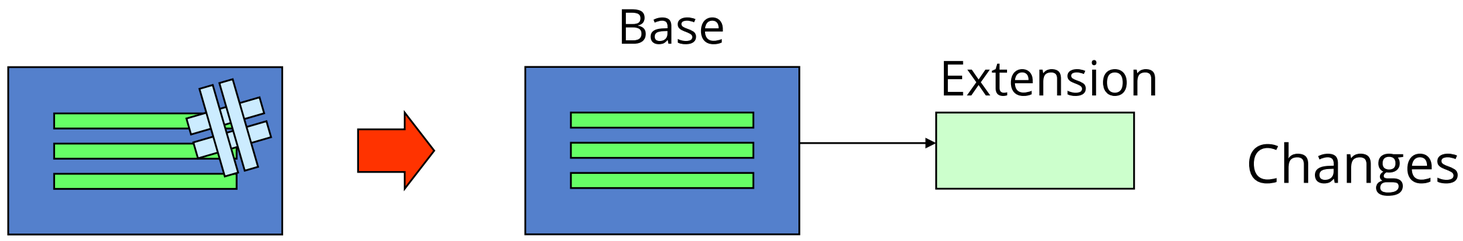


- Duplication

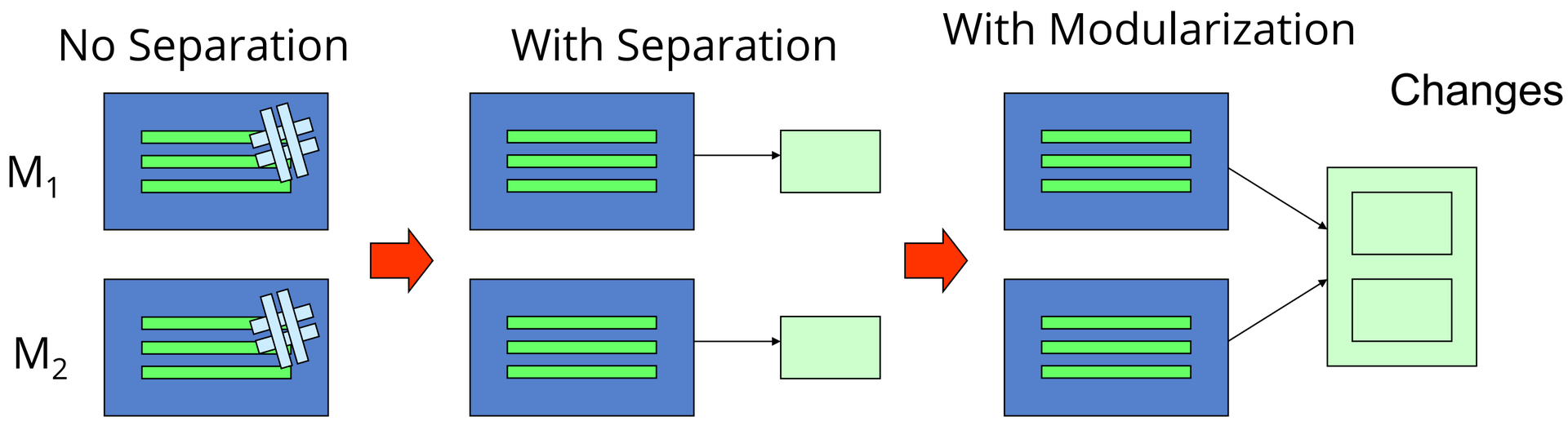


Effective Separation and Modularization of Changes

- Fighting Tangling: Separation of Extensions (Changes)

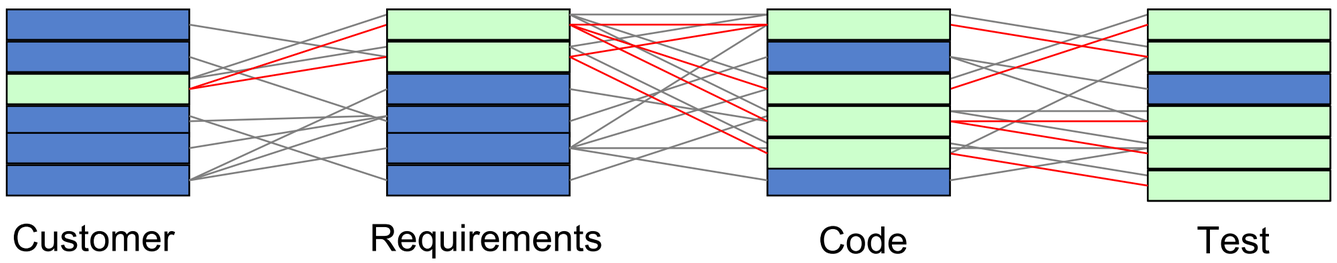


- Fighting Tangling: Modularization of Extensions (Changes)

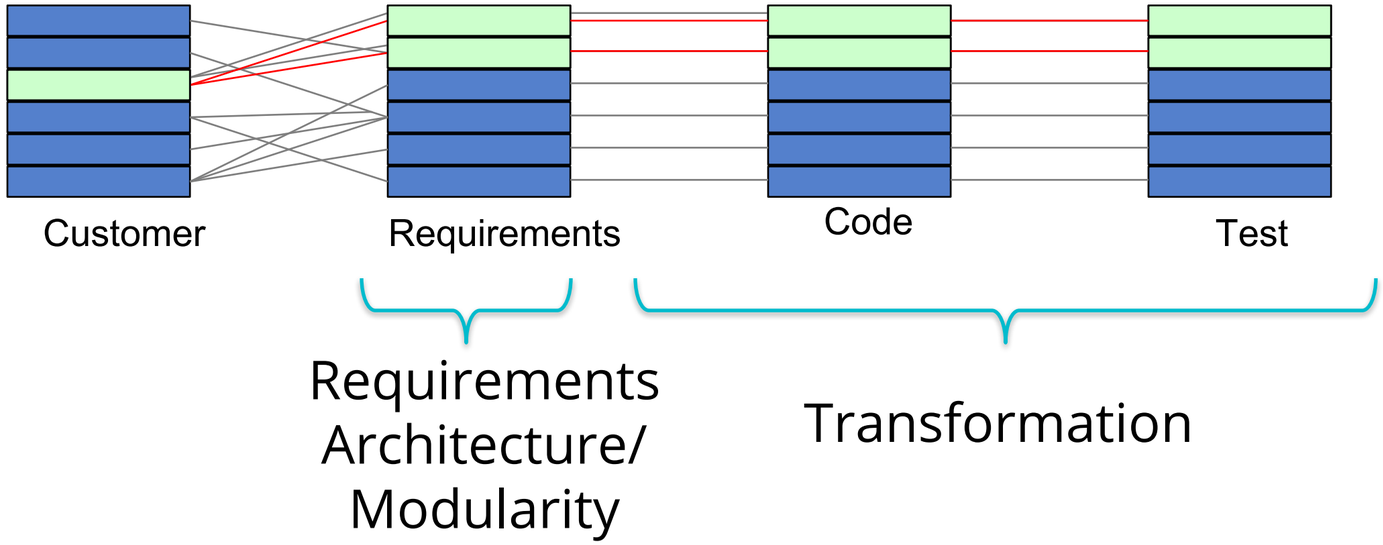


Towards Change Separation and Modularity

- Different Roles Different Perspectives



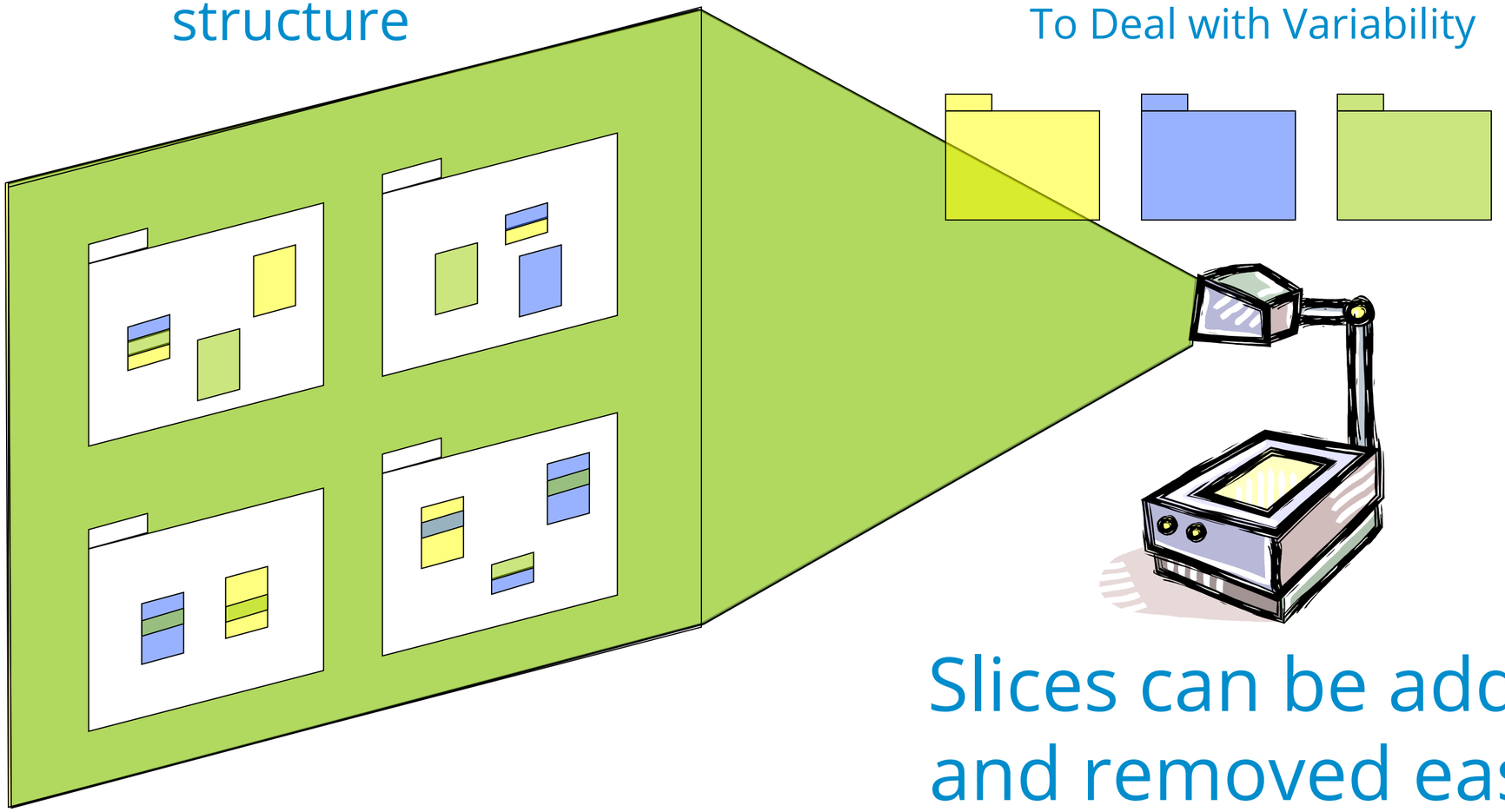
- Unified Perspective：Alignment from Requirements to Code and Test



From Domain Modularity to Change (Use Case & Aspect) Modularity

Object or Domain structure

Slices
To Deal with Variability



Slices can be added and removed easily

Most Methods Do Not Specifically Deal with Change Modularity

- Many design methods? Do they deal with evolution and change modularity?

Use Case Analysis and Design

Class Responsibilities Collaboration

UML in Color

Event Storming

Software Evolution?

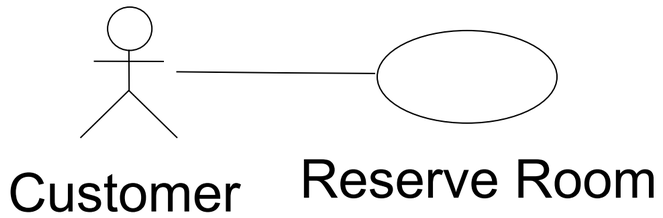
- $S_{N-1} \rightarrow S_N$

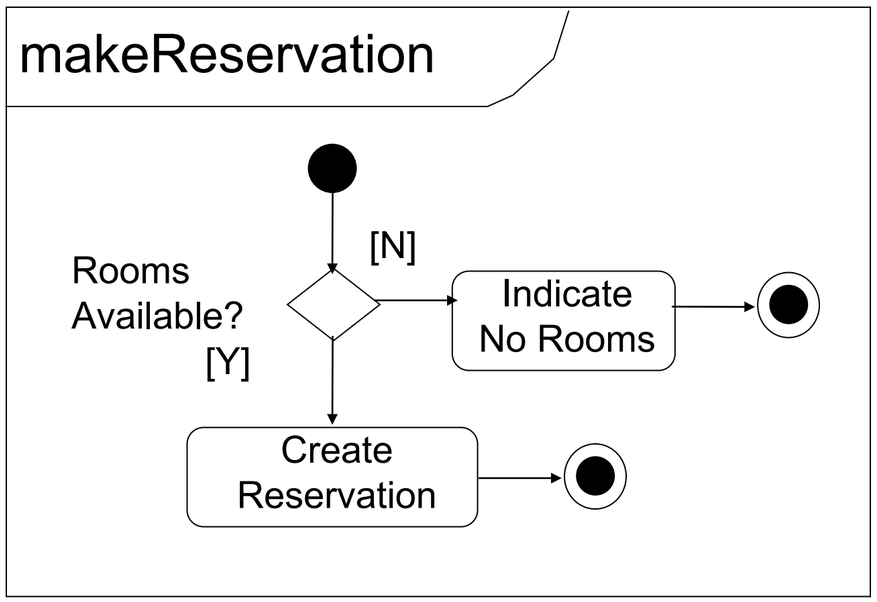
Change Modularity?

- $S_N = S_{N-1} + \Delta_1 + \Delta_2$

Let’s take a concrete example: hotel reservation

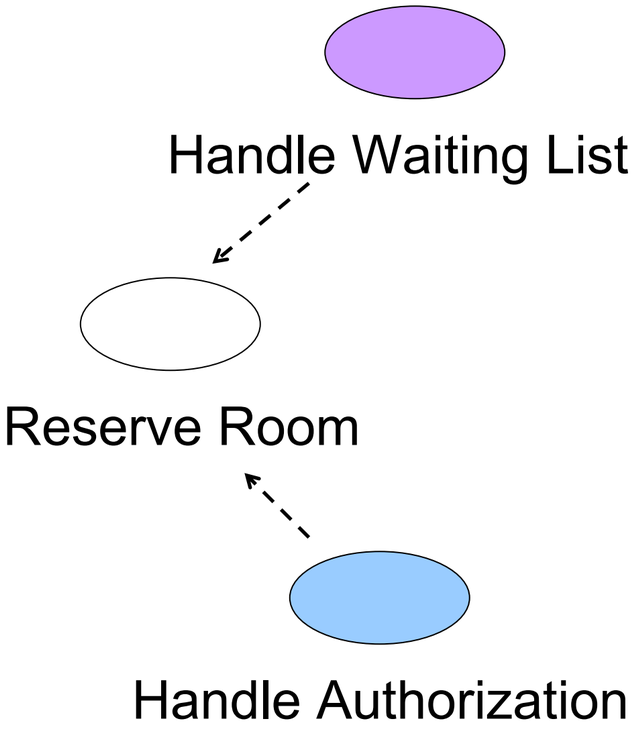
- Consider existing class with operation makeReservation

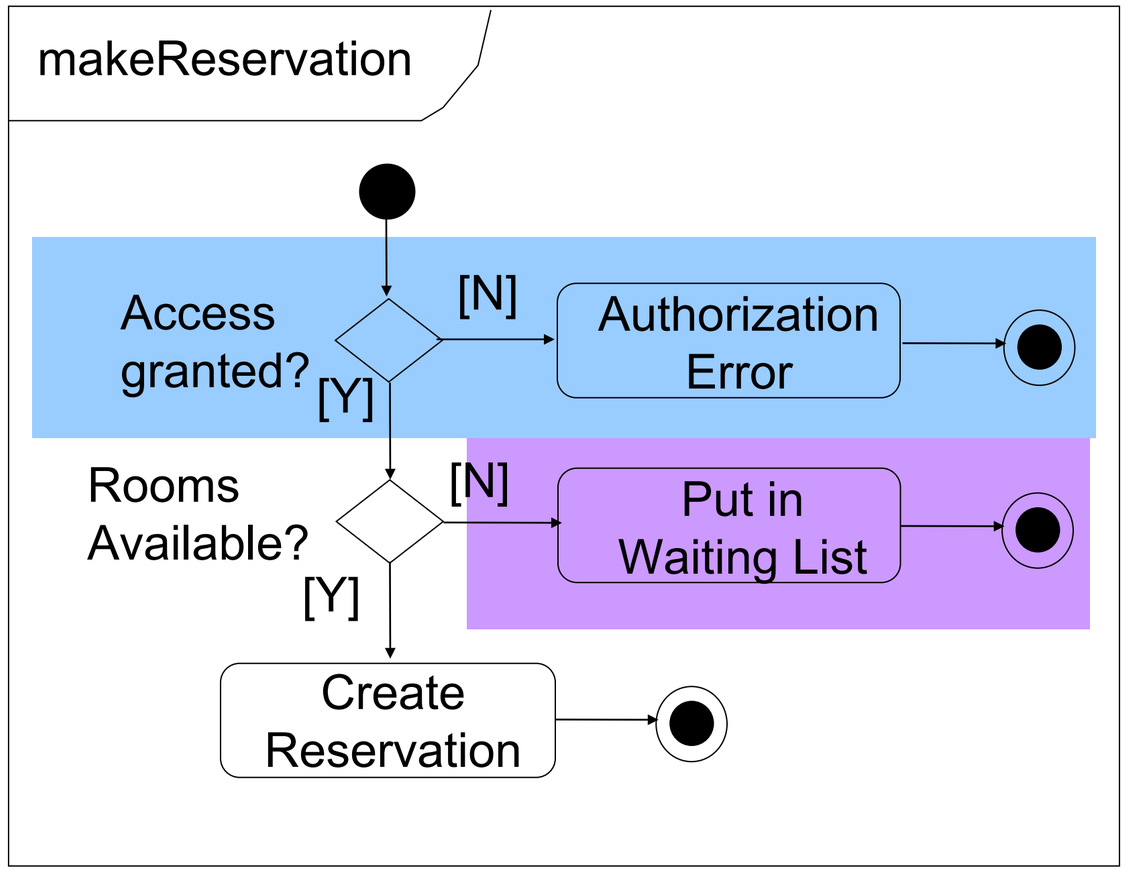




Effects of Tangling

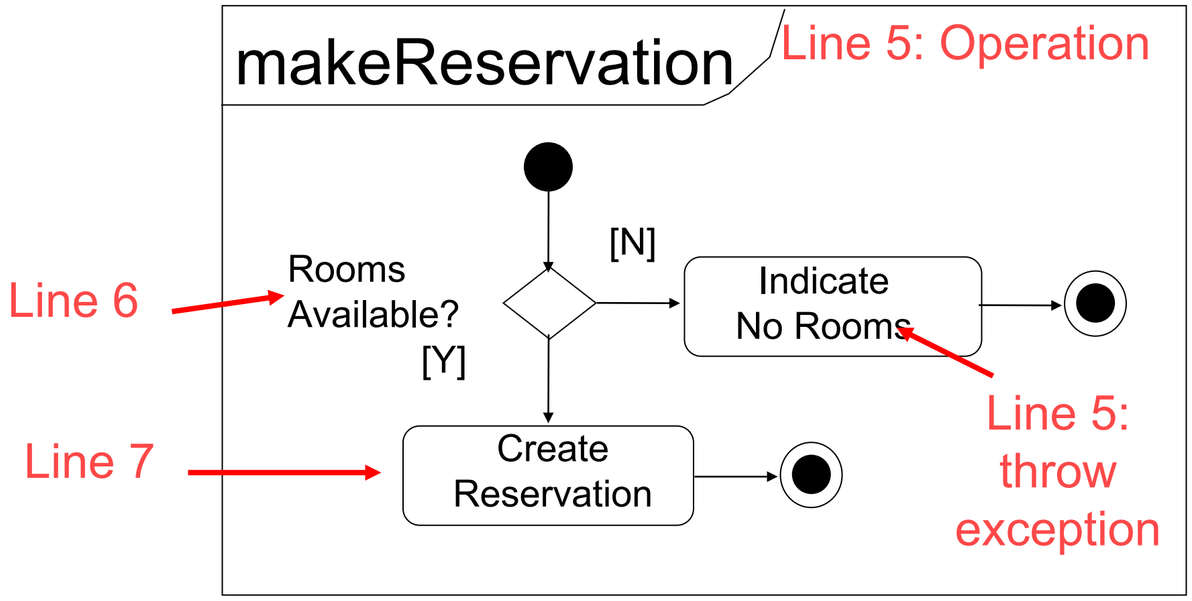
- What happens after you add more functionality?





Lets have some code examples

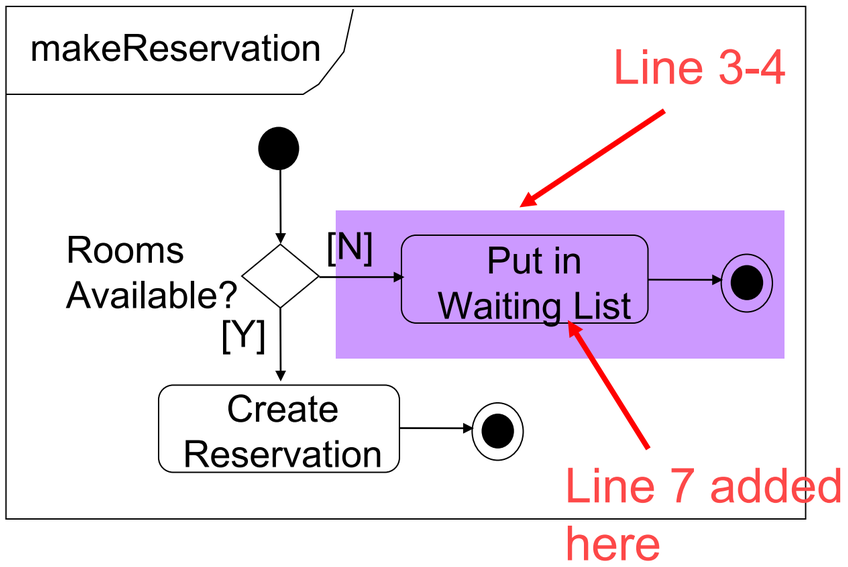
- Reserve Room Realization




```
1.  class Room {
2.    int quantityAvailable ;
3.  }
4.  class ReserveRoom {
5.    void makeReservation(Room theRoom) throws
  NoRoomException {
6.        checkRoomAvalability(theRoom) ;
7.       createReservation() ;
8.   }
9.}
```

Lets add a waiting list use-case

- Waiting List Realization



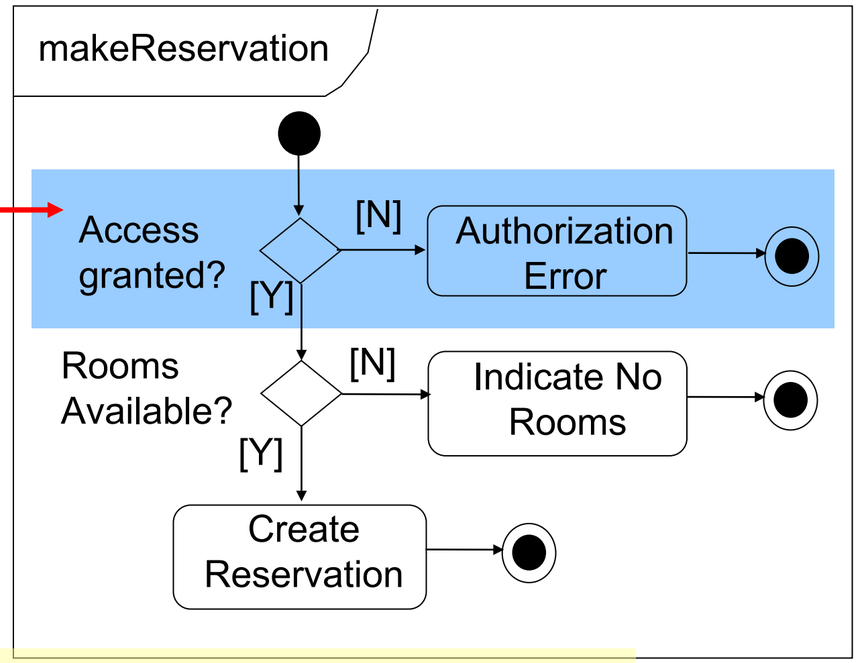

```
1.  aspect HandleWaitingList {
2.     WaitingList Room.queue ; // inter-type declaration
3.    pointcut reservingRoom():
4.             execution(ReserverRoomHandler.makeReservation()) ;
5.    // advice
5.    after throwing (NoRoomException e) : reservingRoom() {
7.       // behavior to add customer into queue
8.    }
9.  }
```

Note:
You need not use aspectJ,
You can apply other techniques.

Lets add authorization use-case

- Authorization Realization

Line 2-3



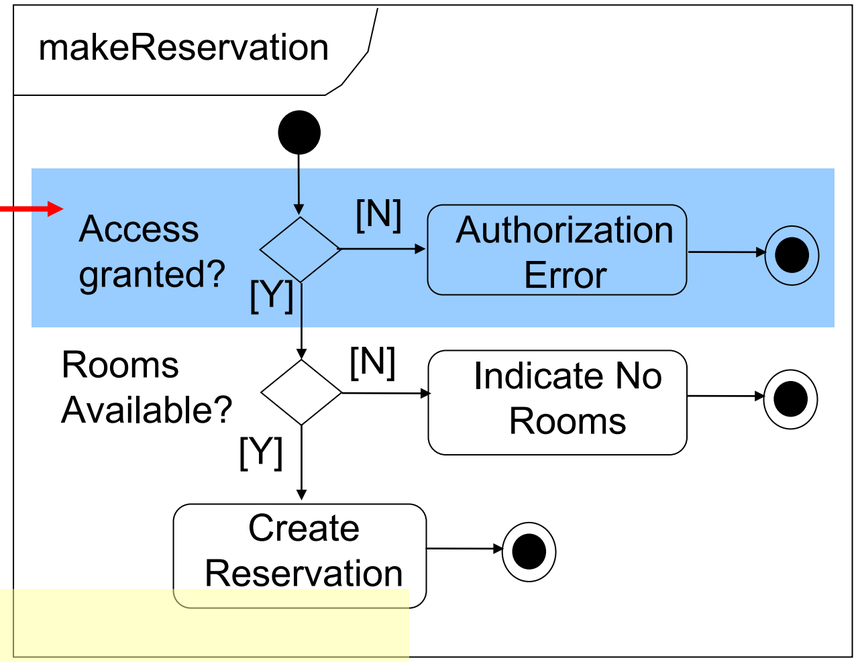

```
1.  aspect HandleAuthorization {
2.     pointcut requestHandling() :
3.           call(ReserveRoomHandler.makeReservation(..)) ;
3.
4.     around () : requestHandling() {
5.        if(AccessControlList.isAuthorized()) {
6.            proceed() ;
7.        }
8.     }
9. }
```

Note:
You need not use aspectJ,
You can apply other techniques.

Lets add authorization (crosscutting)

- Authorization Realization

Line 2-4




```
1.  aspect HandleAuthorization {
2.     pointcut requestHandling() :
3.           call(ReserveRoomHandler.*(..))
4.         || call(CheckInhandler.*(..));
3.
4.     around () : requestHandling() {
5.        if(AccessControlList.isAuthorized()) {
6.            proceed() ;
7.        }
8.     }
9. }
```

Note:
You need not use aspectJ,
You can apply other techniques.

The Use Case + Aspect Advantage

Requirements Modularity

Basic Flow – Reserve Room

1. Customer select room
2. Customer submit reservation
3. System update room availability
4. System create new reservation

Alternate Flow – Waiting List

1. This alternate flow occurs at step 3 of basic flow when no rooms are available
2. The system puts customer in waiting list

Implementation Modularity

```
1. class ReserveRoom {
2.   void makeReservation() {
3.      checkRoomAvalability(theRoom);
4.      createReservation() ;
5.   `}
6. }
```

```
1. aspect HandleWaitingList {
2.   pointcut reservingRoom()
3.     : execution(makeReservation());
4.   after thowing (..)
5.     : reservingRoom() {
6.       // add customer into queue
7.   }
8. }
```

Different Aspect Technology

AspectJ – Aspect Oriented Programming

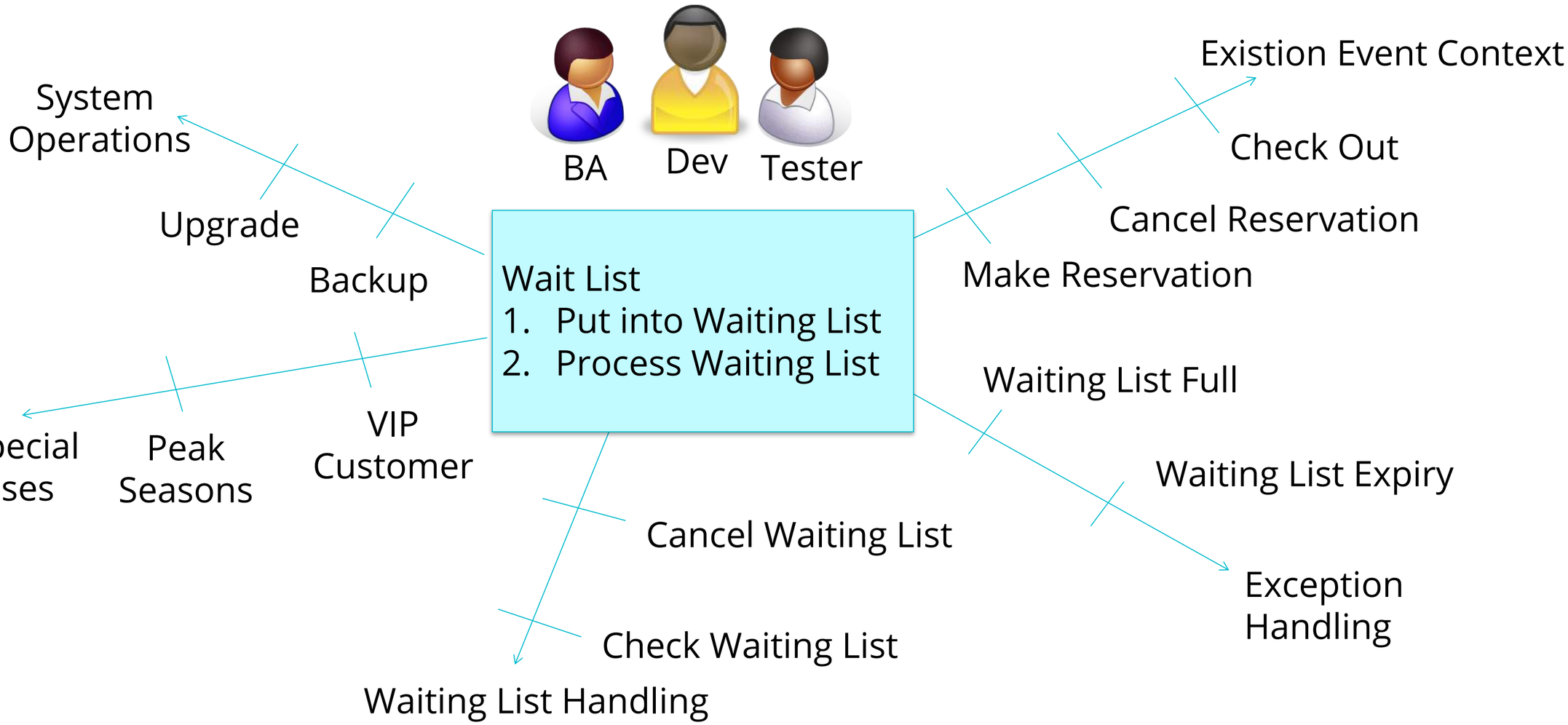
Hyper/J – Multi-Dimensional Separation of Concerns

DemeterJ – Adaptive Programming

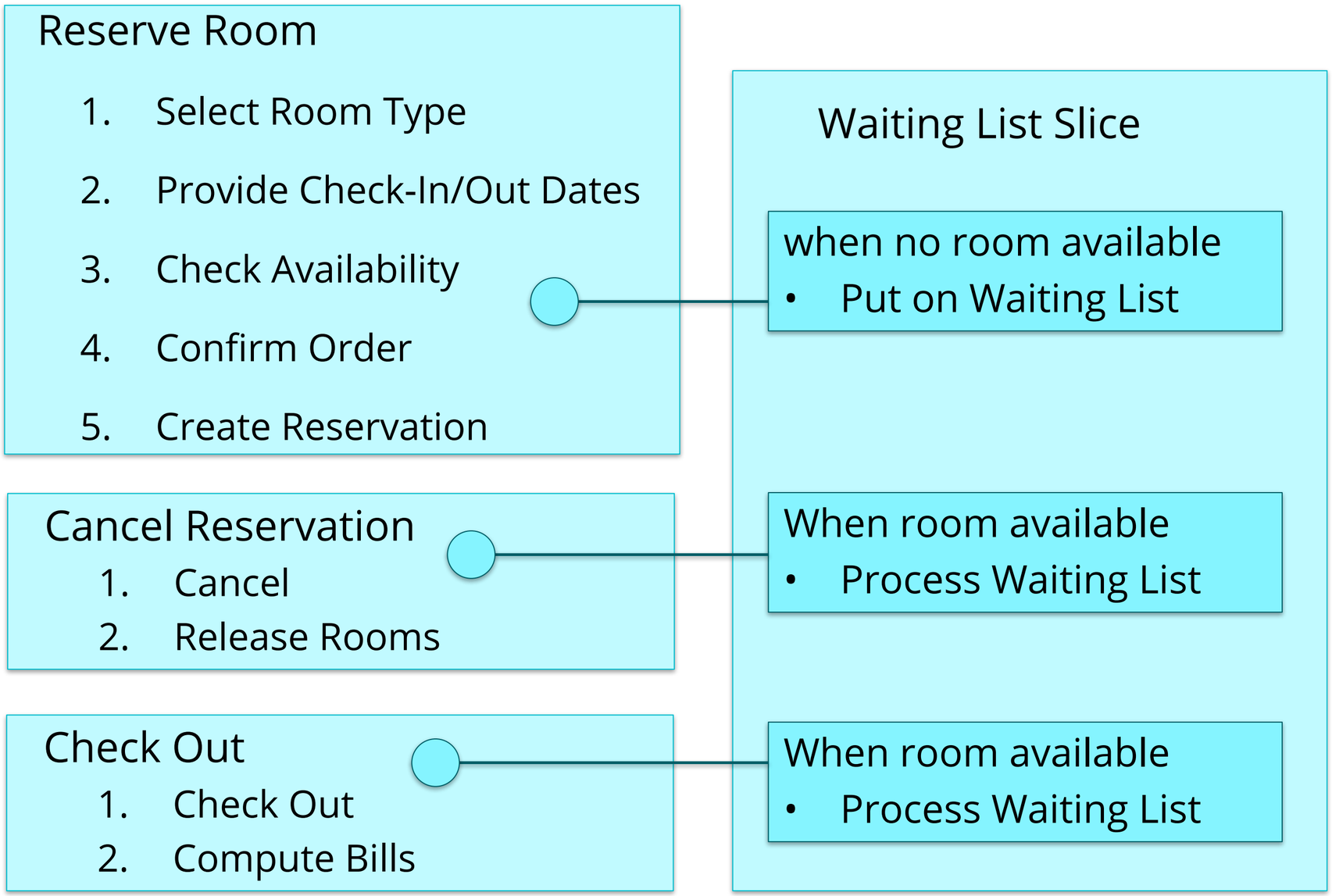
Operationalizing Software Evolution Modularity

1 Change Storming	2 Change Modularization	3 Change Homing
Brainstorm Dimensions Of Change/ Variability	Clarify Semantics and Boundaries of Changes	Find the Natural Home in the Universe

Change Storming: Identifying Different Dimensions of Variability



1. Change Modularization: Extension Semantics (Requirements)

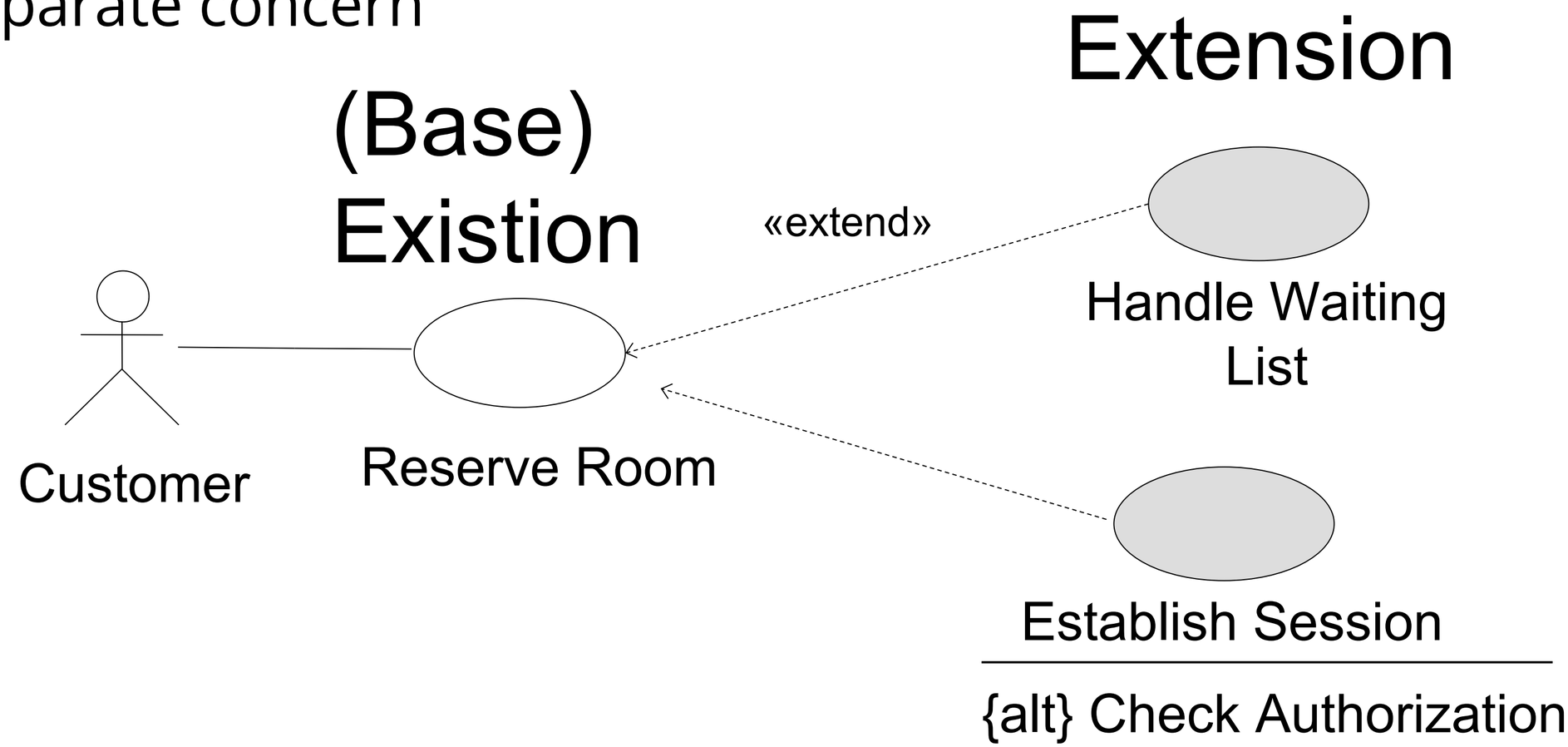


Use Case Extension Semantics

- Event driven or object traversal existion
- Before, after, around
- Singular, multiple, cross-cutting
- Extension / Feature interaction
- 1st order, second order, Nth order interactions

Use-case extend

- Use case extension is used when you want to add additional behaviors on top of existing use cases
- Additional behavior can be
  - enhancement of the use case
  - a separate concern



Use-case specification for extension (current approach)

Use Case: Reserve Room
Basic Flow

1. The use case begins when a customer wants to reserve a room(s).
2. …
3. The system displays the types of rooms in the hotel and the their rates
4. The customer Choose Rooms.
5. …
6. …
7. …
8. The system displays the reservation confirmation number and check in instructions.
9. The use case terminates.

Alternate Flows

…

Extension Points

E1 Update Room Availability

Basic Flow – Step 5

Use Case: Handle Waiting List
Basic Flow

…

Extension Flows

EF1.Queue For Room

This extension flow occurs at the extension point "Update Room Availability" in the Reserve Room use case when there are no rooms of the selected type available.

1. The system creates a pending reservation with a unique identifier for the selected room type.
2. The system puts the pending reservation into a waiting list
3. The system displays unique identifier of the pending reservation to the customer.
4. The base use case terminates

Extension User Story

When …
As a …
I want to ….
So that …..

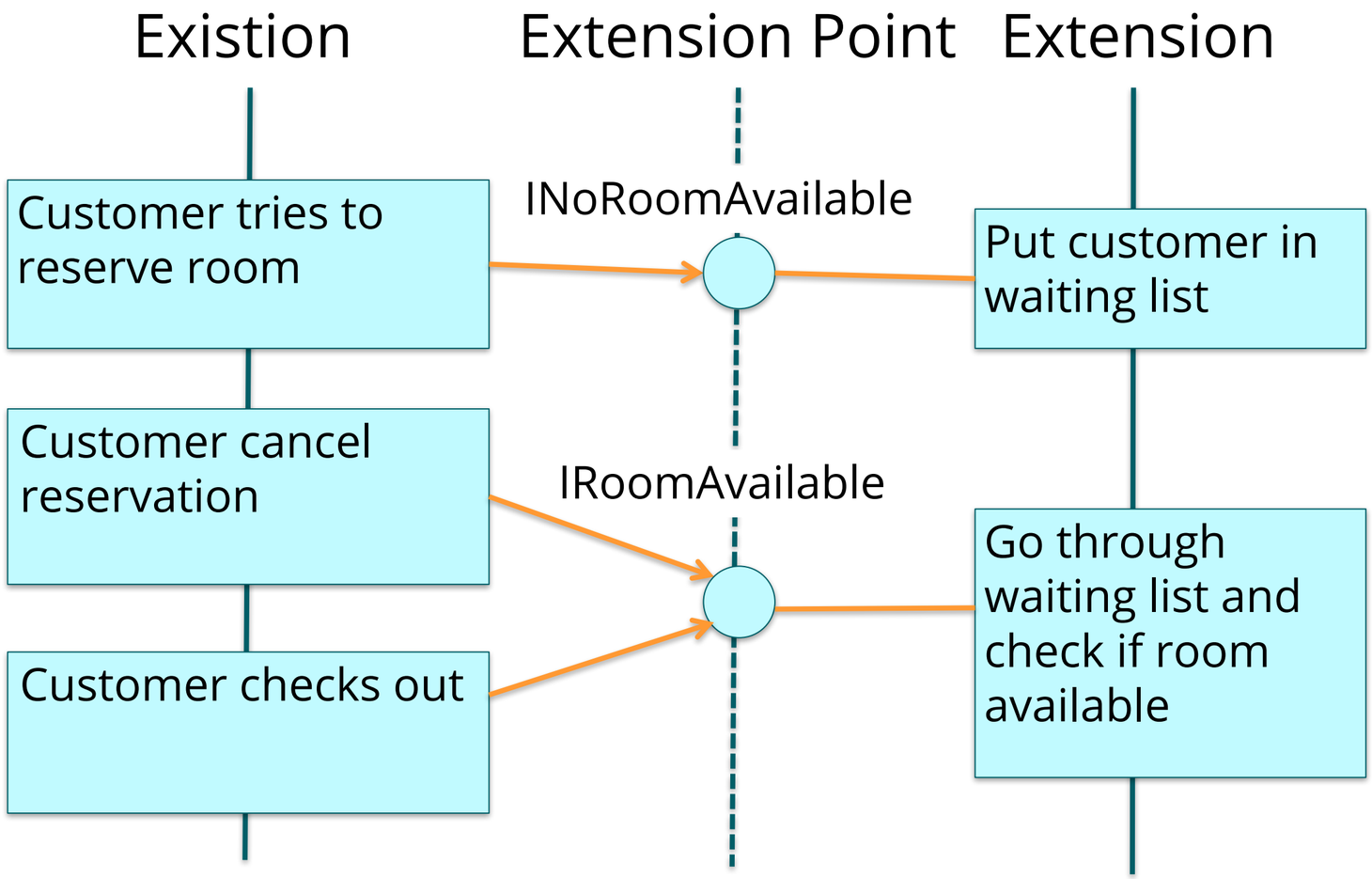
When there is no rooms during reservation

As a traveler
I want to be put on a waiting list so that I get chance to stay in the hotel I like.

Extension Context

Before
After
Around

2. Change Modularization: Crisp Boundaries

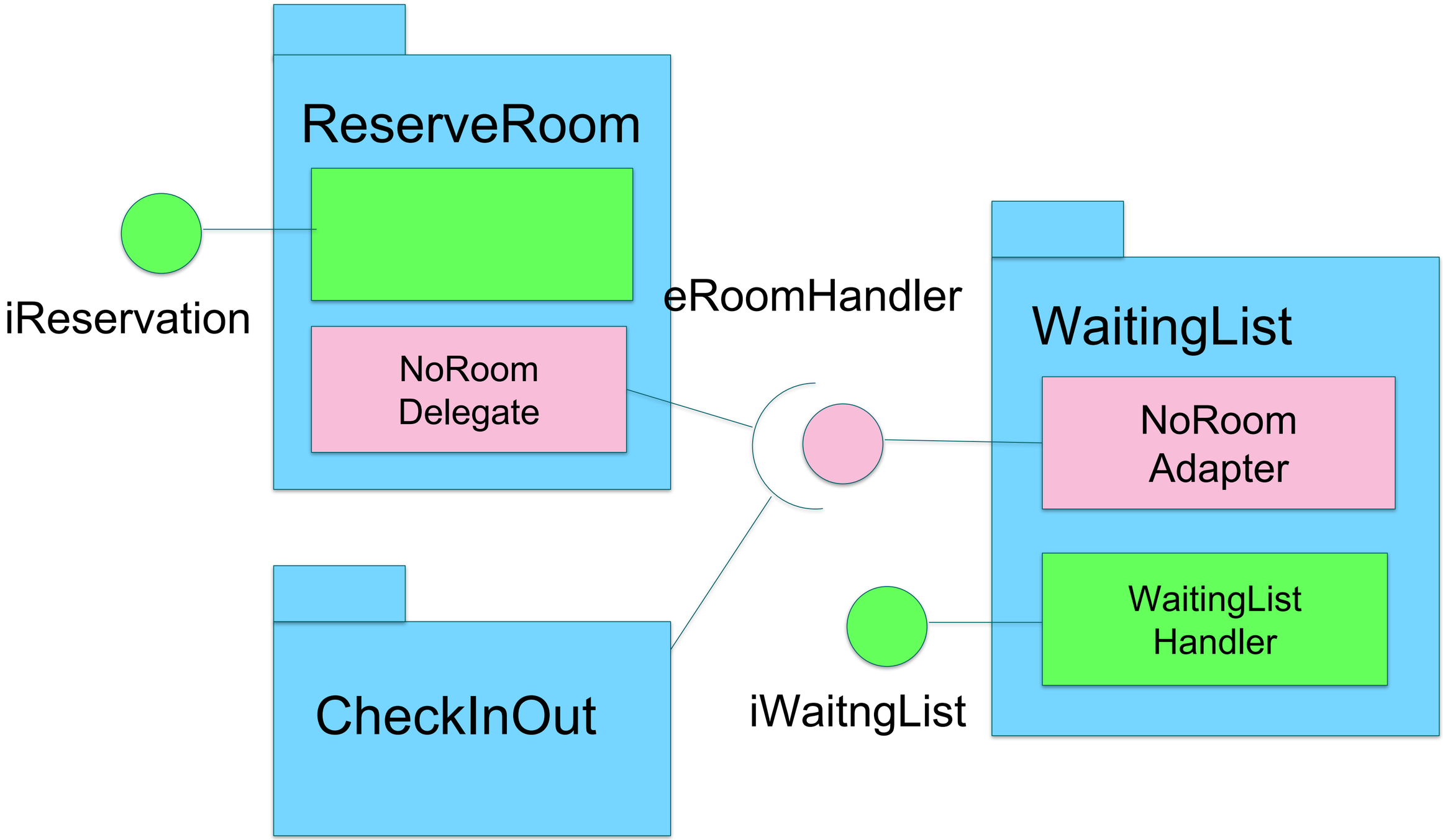


Different and many implementation mechanisms available

- Design frameworks (e.g. Spring)
- Design patterns – decorator, adapter, observer, strategy and visitor, etc.
- Service notification, service mesh

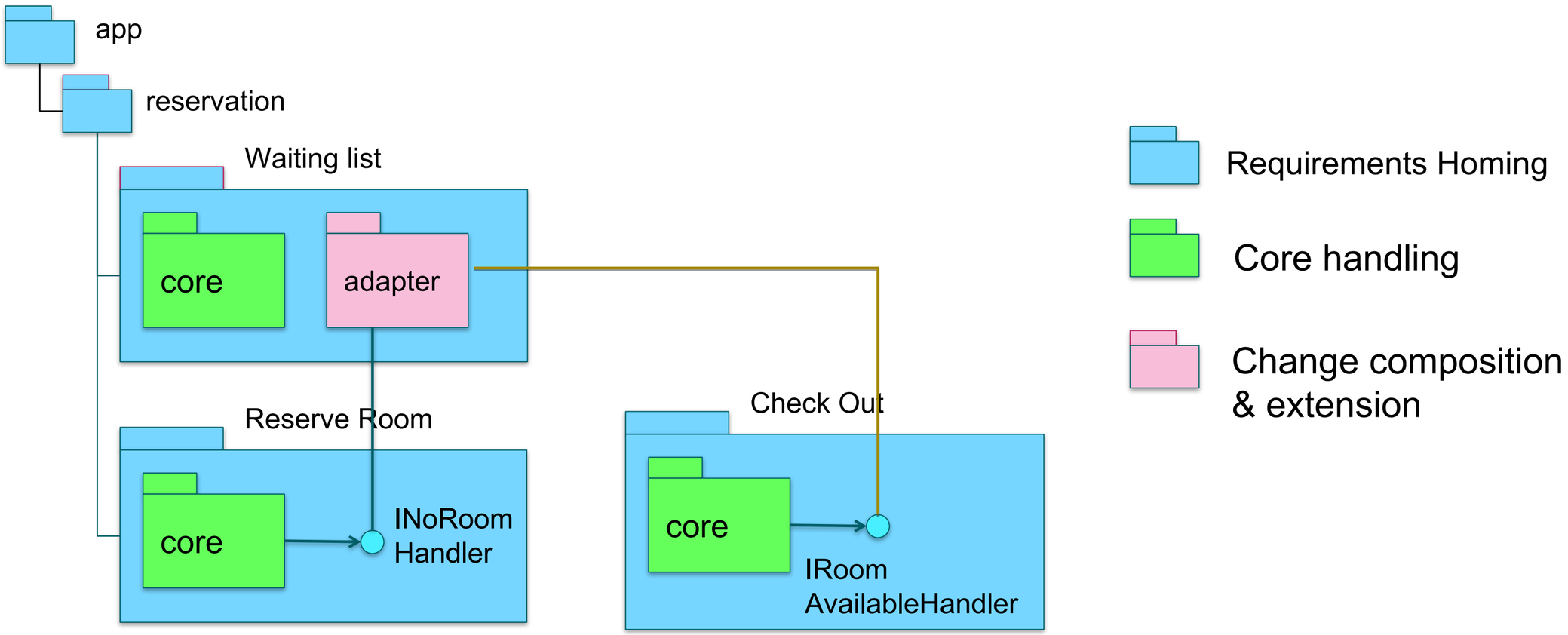
The key is still modularity

2. Change Modularization: Crisp Boundaries

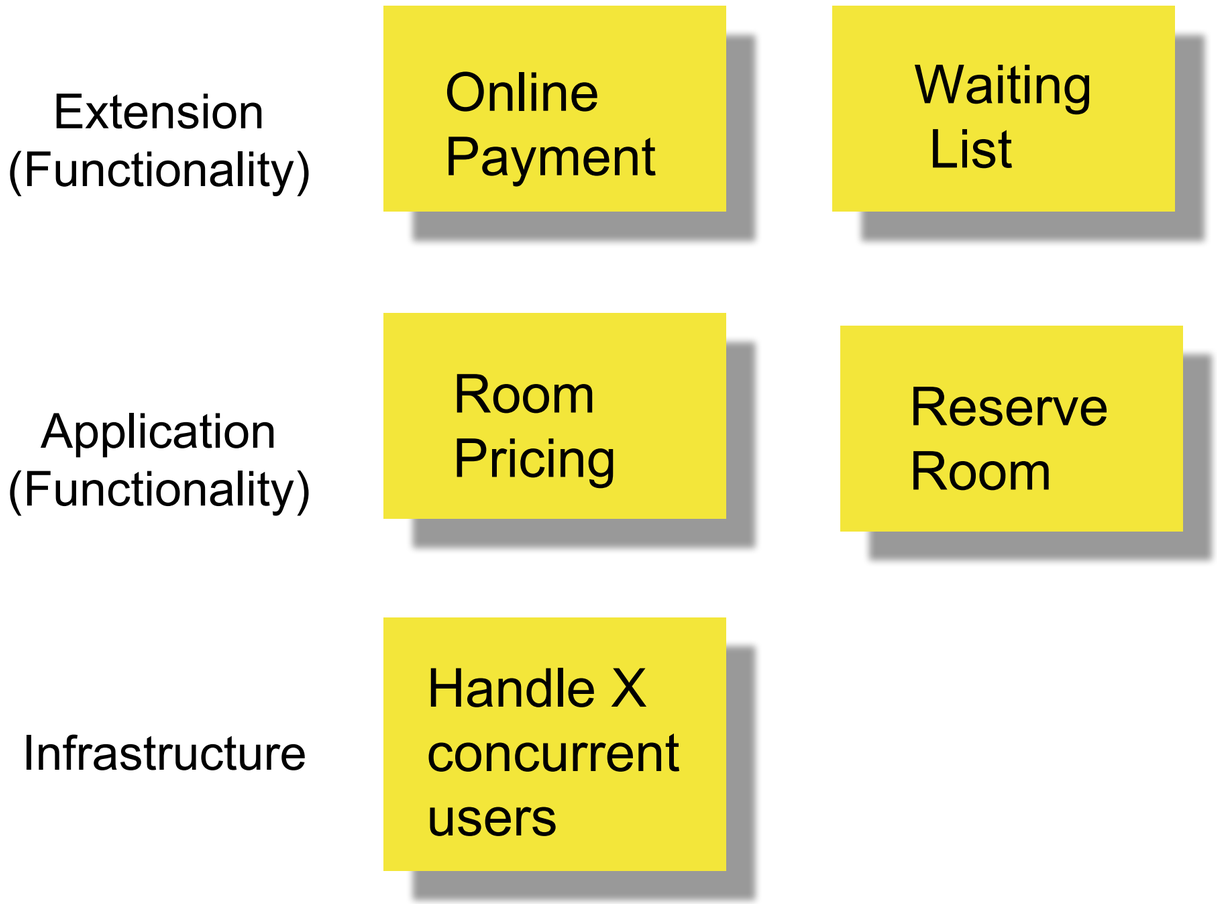


3. Change Homing: Finding the natural home

- Law of Demeter, Principle of Least Knowledge
- Structure according to change / requirements modularity



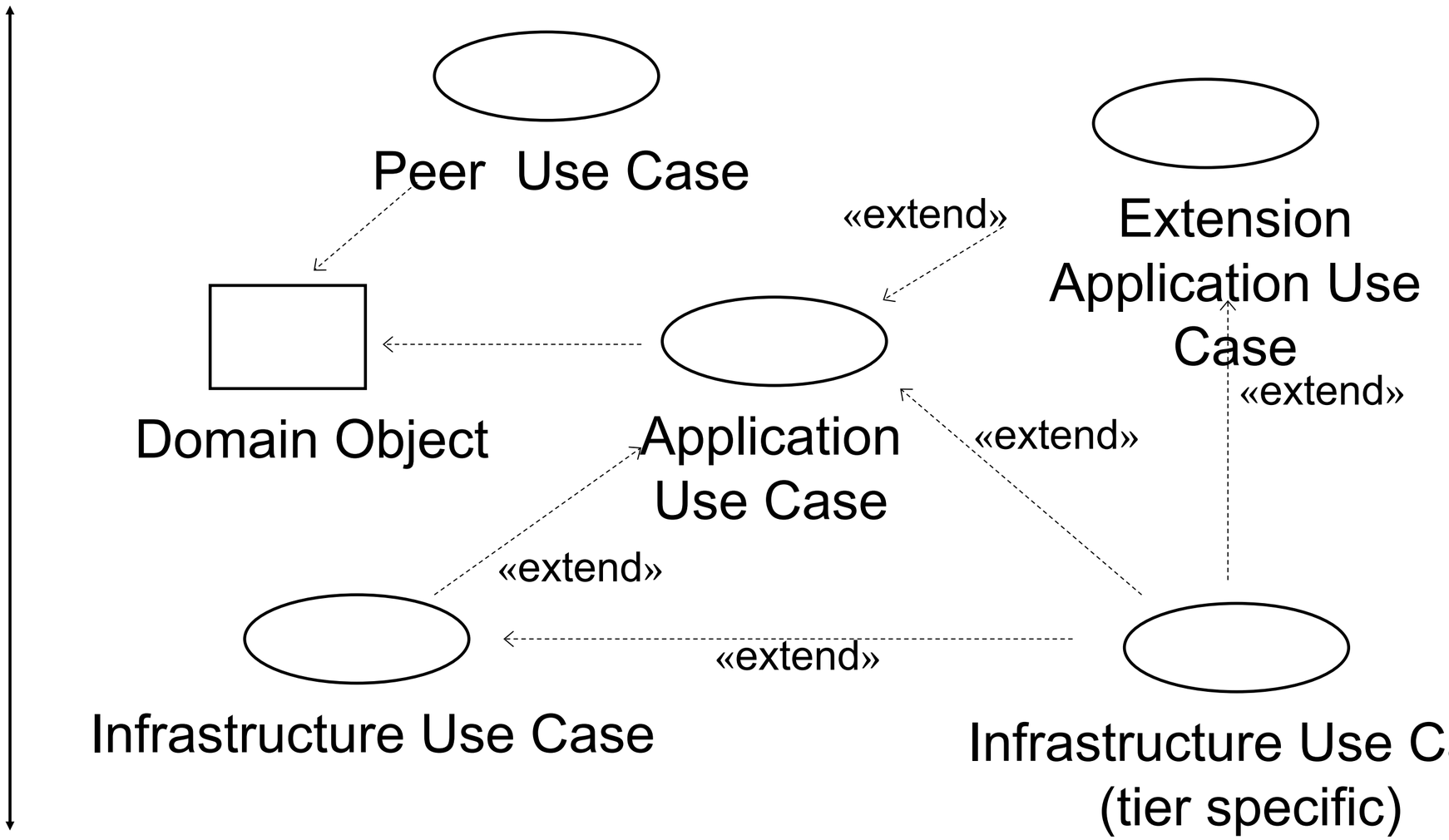
Change Modularity in the Large



- Every Release adds new capability to the system
  - New functionality
  - Extension (Aspect)
  - Infrastructure mechanism

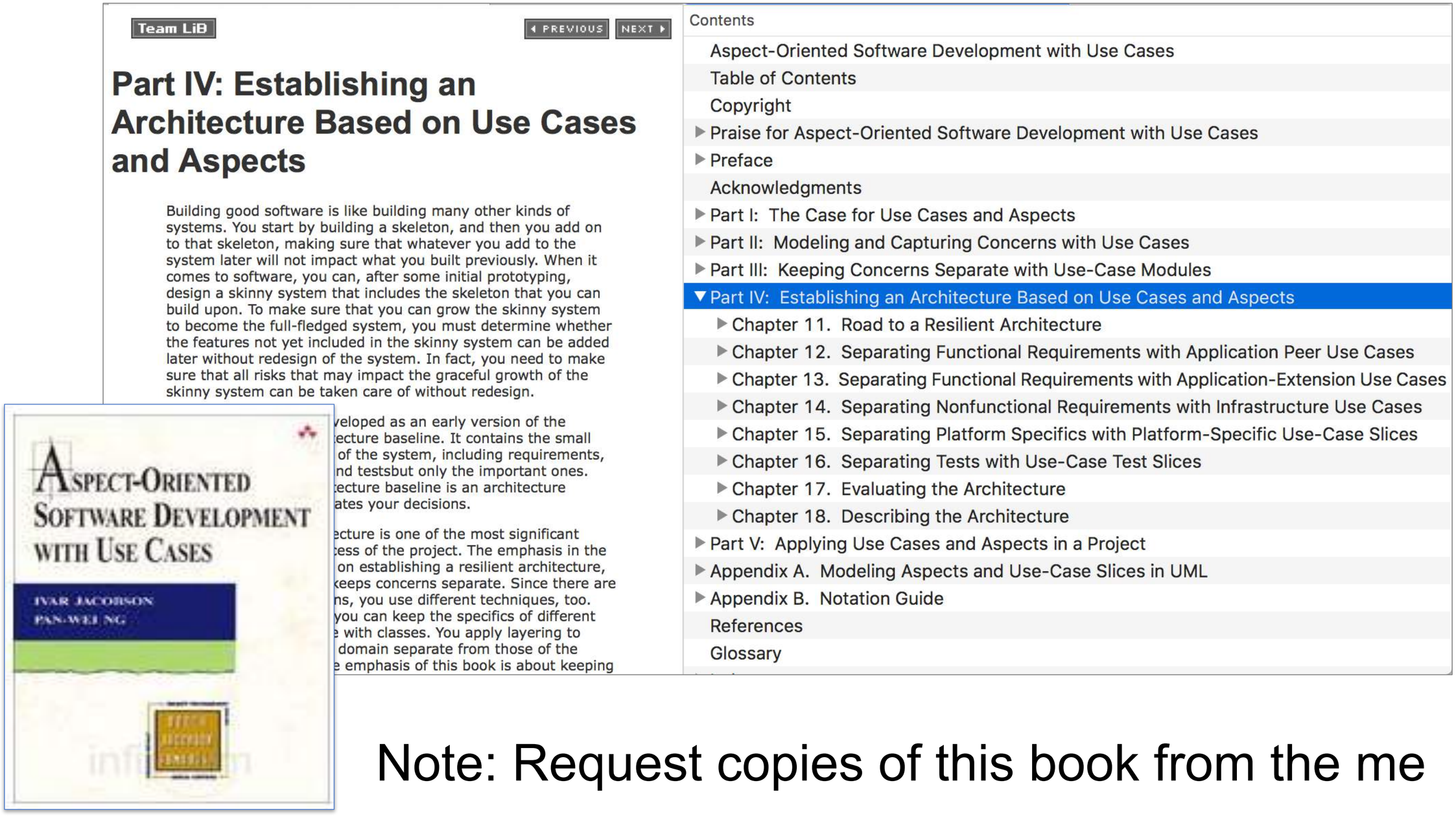
Use Case as Context and Change Modularity Constructs

Functional Requirements



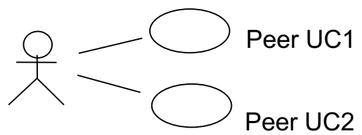
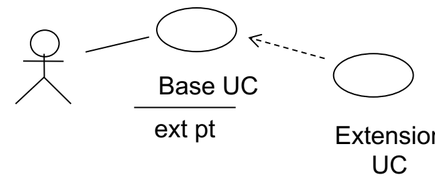
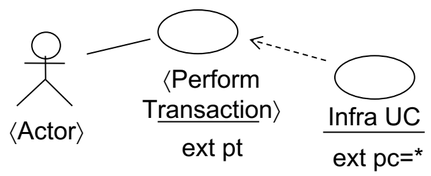
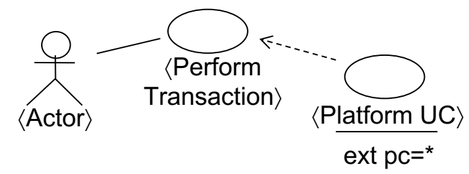
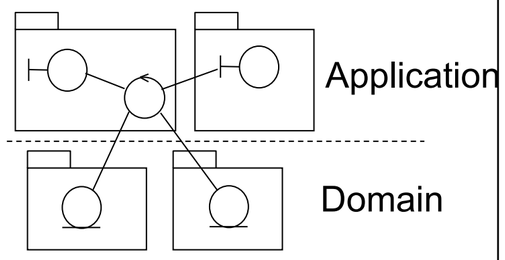
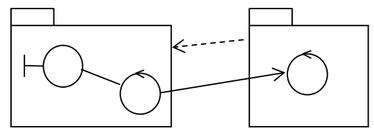
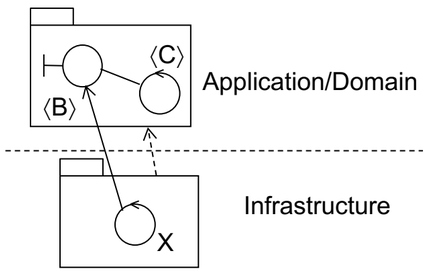
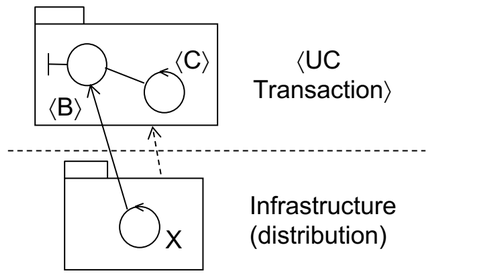
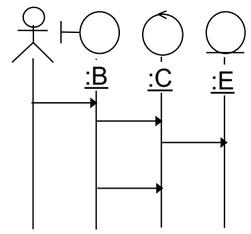
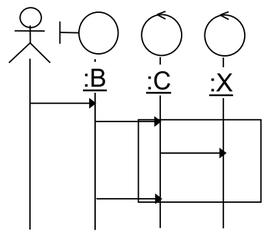
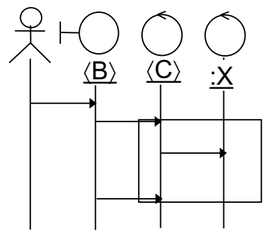
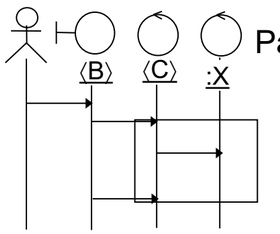
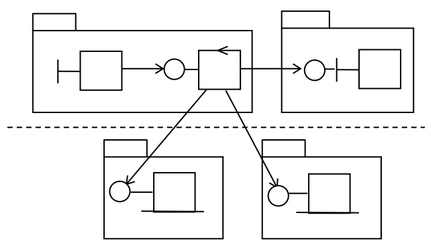
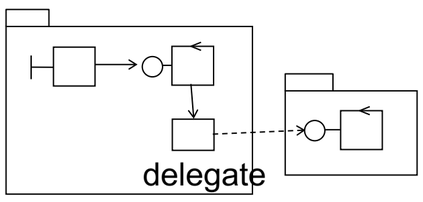
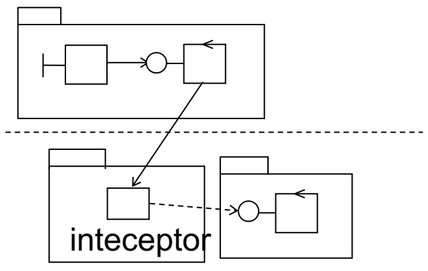
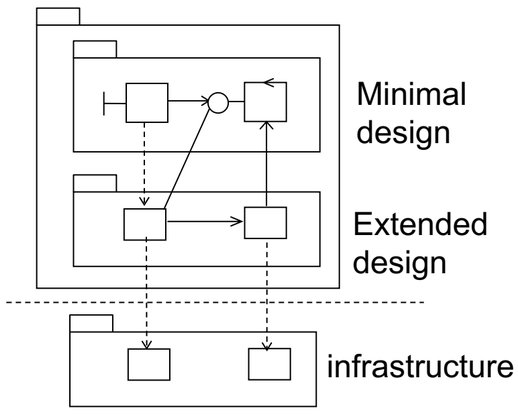
Non-Functional Requirements

A Mature Process to deal with Extensions



Note: Request copies of this book from the me

Use Case driven development

	Peer	Extension	Infrastructure	Platform
Use Case Model				
Analysis Model				
				
Design Model				

We will not discuss this in detail

Summary

- Software development = software evolution
- Modularize change
- Change modularity starts with requirements modularity
- Good requirements modularity leads to good design
- Good requirements modularity is change modularity
  - Limit impact of change to a single requirements module
  - Use cases provide constructs for requirements modularity
  - Aspect and related technology helps implement change modulairty

Thank You


黄药师
Singapore
Scan the QR Code to add me on WeChat
DOMAIN
领域驱动设计
中国峰会2017
DESIGN