Component Level Design

목차

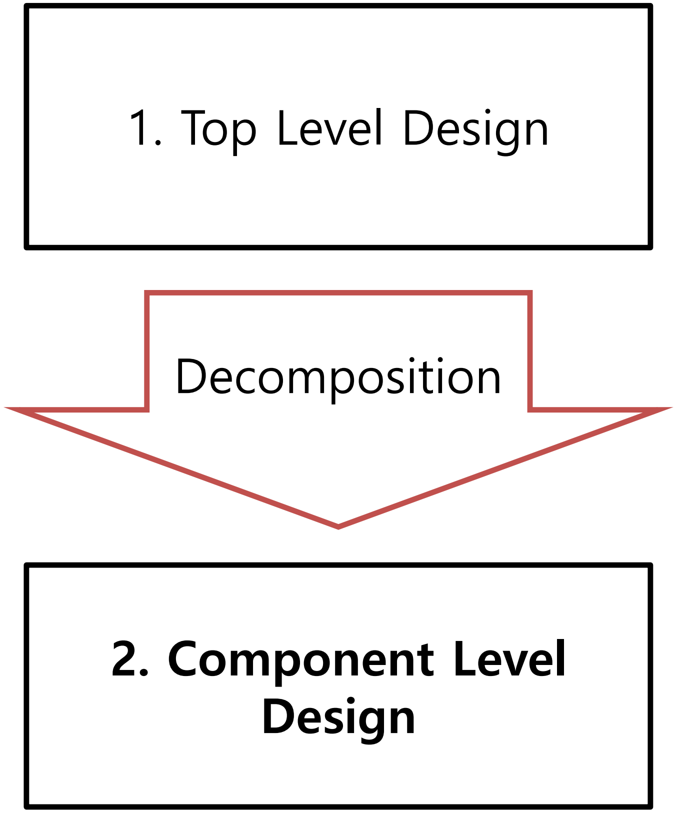
- Component Level Design
  - Structure View

- Design Principles
  - Cohesion/Coupling/Complexity
  - SOLID
  - Package Cohesion and Package Coupling

- Design Techniques
  - Design Patterns
  - Variability Design with Patterns

COMPONENT LEVEL DESIGN

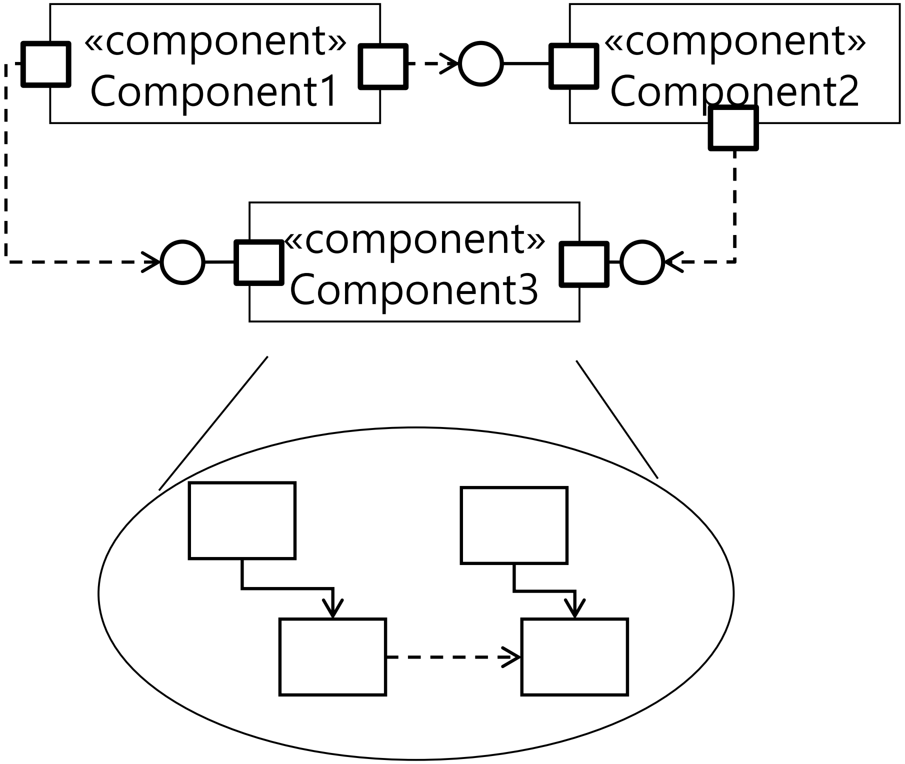
Component Level Design



1. Top Level Design

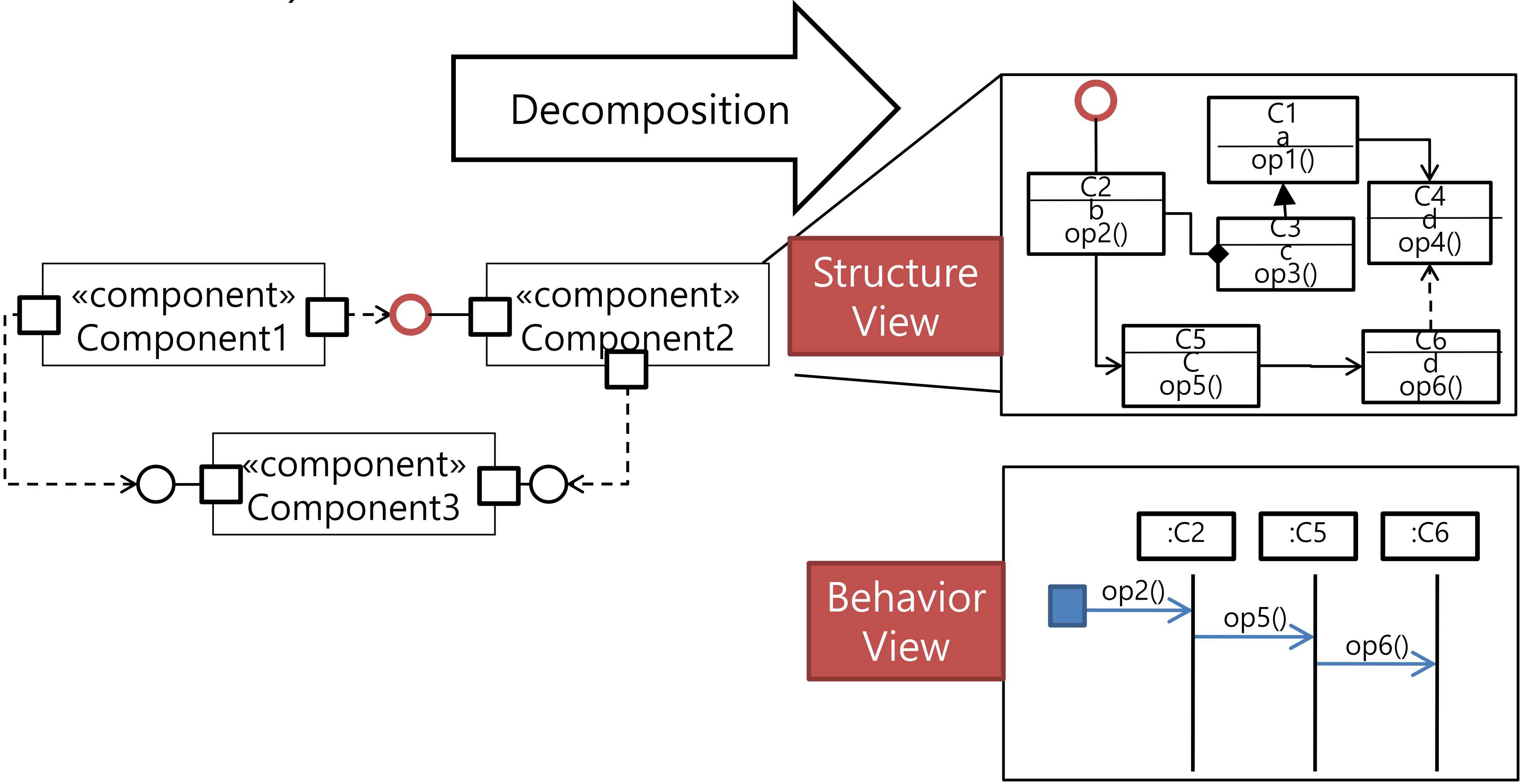
Decomposition

2. Component Level Design



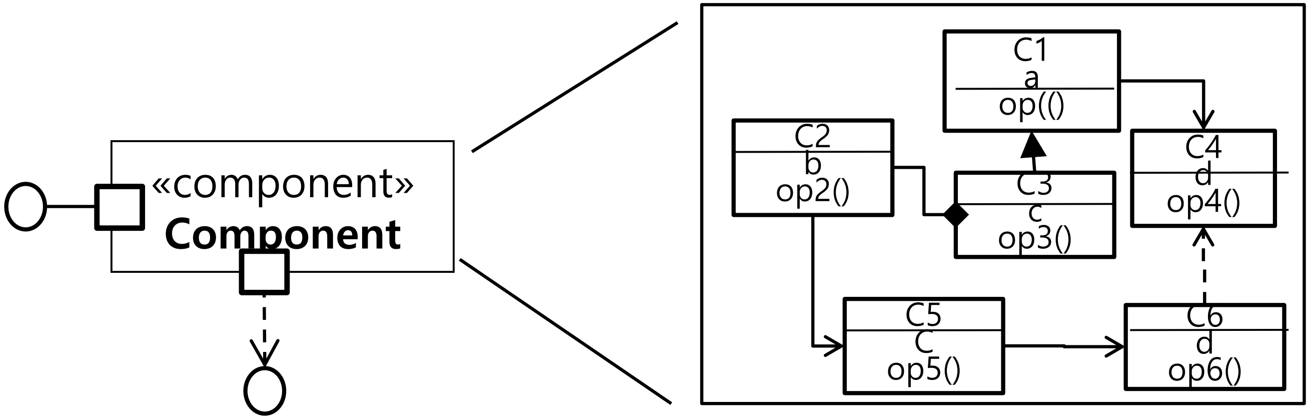
Component Level Design

- ❖ Decompose each component into fine-grained elements(i.e., classes)



Component Description

❖ Represent the decomposition of the component

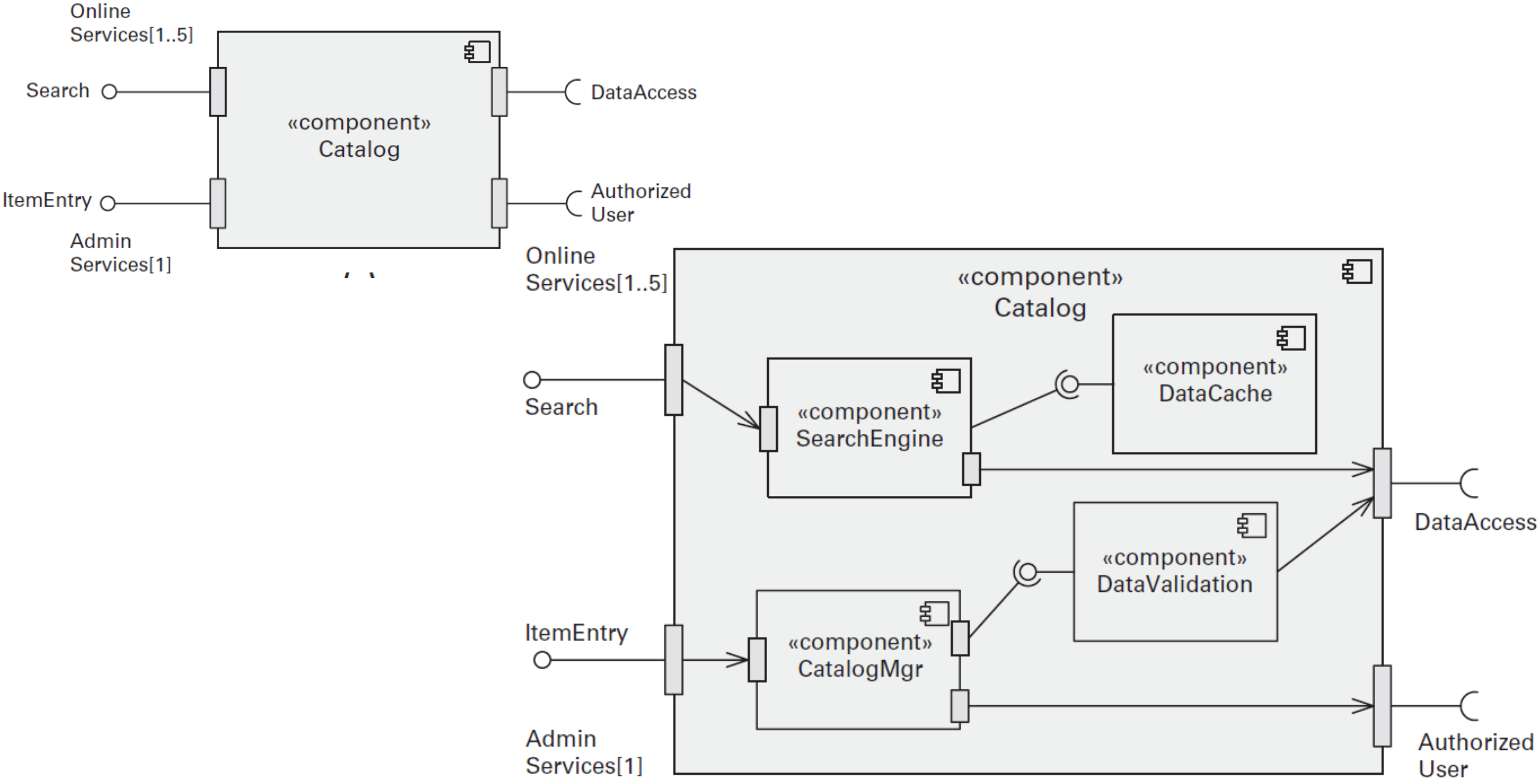


❖ Describe the followings for each component

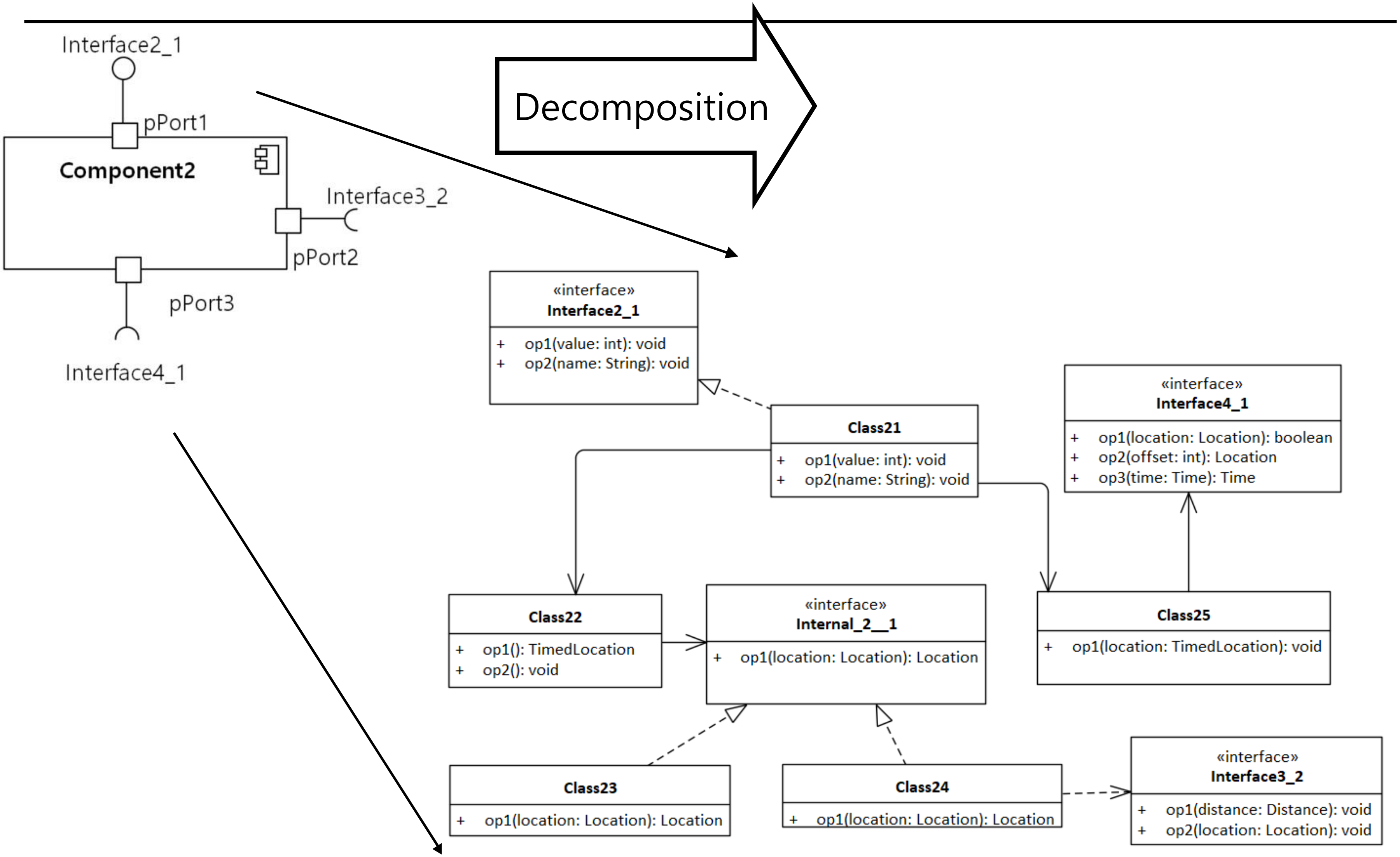
- Static Structure Diagram
- Element List
- Design Rationale

Static Structure Diagram

- ❖ A component consists of several fine-grained elements including smaller components and/or classes

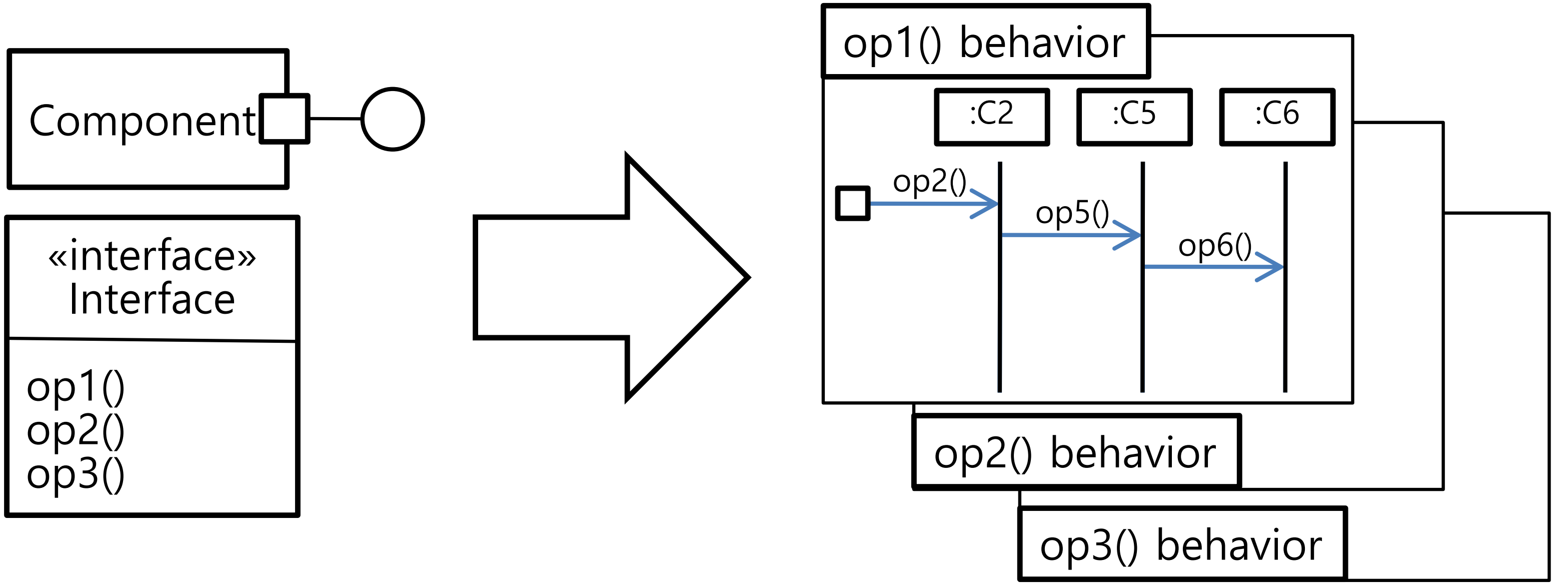


Static Structure Diagram


Interface2_1
pPort1
Component2
Interface3_2
pPort2
pPort3
Interface4_1
Decomposition
«interface»
Interface2_1
+ op1(value: int): void
+ op2(name: String): void
Class21
+ op1(value: int): void
+ op2(name: String): void
«interface»
Interface4_1
+ op1(location: Location): boolean
+ op2(offset: int): Location
+ op3(time: Time): Time
Class22
+ op1(): TimedLocation
+ op2(): void
«interface»
Internal_2__1
+ op1(location: Location): Location
Class25
+ op1(location: TimedLocation): void
Class23
+ op1(location: Location): Location
Class24
+ op1(location: Location): Location
«interface»
Interface3_2
+ op1(distance: Distance): void
+ op2(location: Location): void

Component Behavior View

- Describe how each operation of the provided interface can be realized



- For each provided interface of the component
  - Operation Behavior Model for each operation
  - Design Rationale

Decomposition Strategies

- Functionality
  - Decomposing a system based on functionality is perhaps the most obvious strategy
  - You inventory the required functionality and clump together related functions.

- Archetypes
  - Archetypes / core types are salient types from the domain, such as a Contact, Advertisement, User, or Email
  - Characteristics of an archetype include having an independent existence and having few mandatory associations to other types

- Pattern
  - A system can be decomposed so that its components are elements defined by an architectural pattern and design pattern
  - Choosing an architectural pattern is highly effective at achieving quality attribute goals because each style has known qualities that it promotes

Just enough software architecture: A risk-driven approach(2010)

Decomposition Strategies

- Achievement of certain quality attributes
  - For example, to support modifiability, impact of any one change is localized

- Build-versus-buy decisions
  - Some modules may be bought in the commercial marketplace, reused intact from a previous project, or obtained as open-source software

- Product line implementation
  - it is essential to distinguish between common components, used in every or most products, and variable components, which differ across products

- Team allocation
  - To allow implementation of different responsibilities in parallel, separate components that can be allocated to different teams should be defined

Documenting software architecture: Views and beyond, 2nd edition(2010)

Design Techniques

❖ Tactics

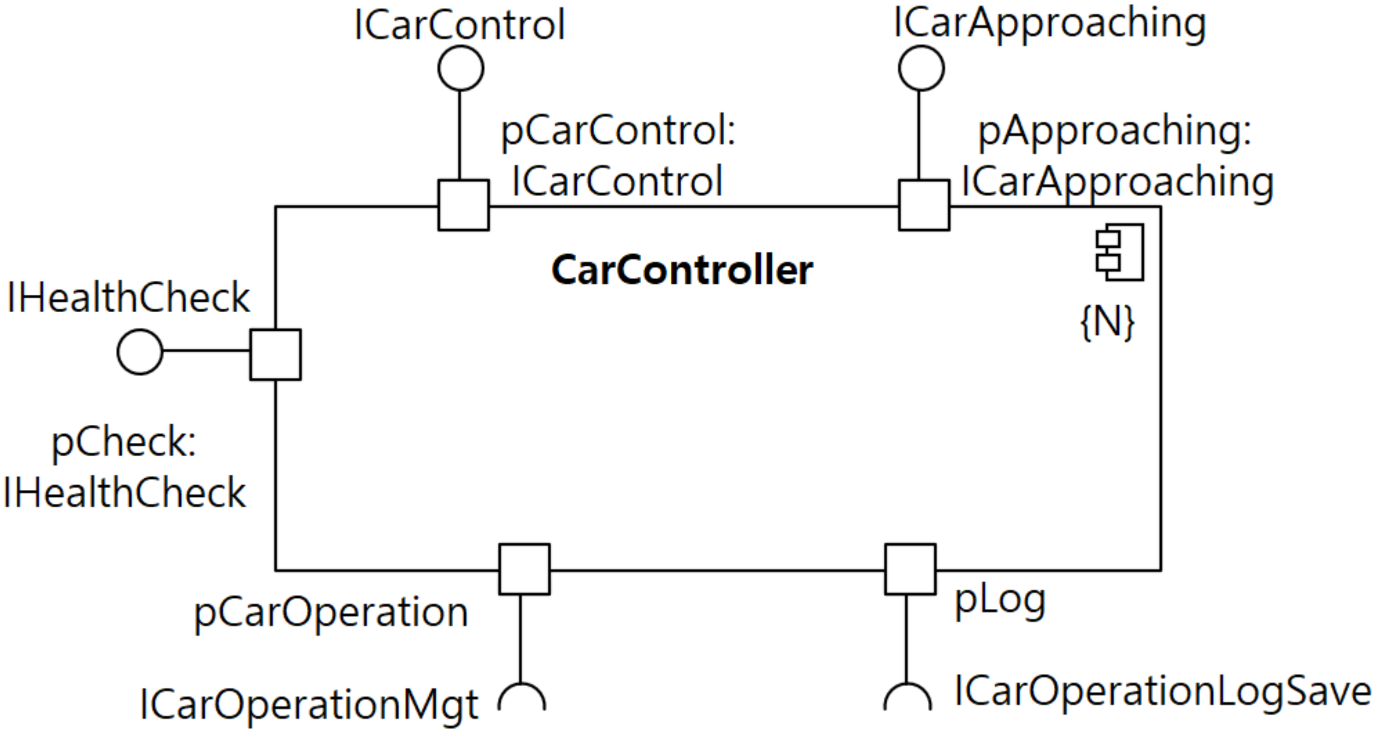
- 성능을 고려하면 multi-threading, thread-safe queue 등이 필요함
- 유지보수성을 고려하면 응집도, SOLID 등을 적용해서 세분화 필요가 있음

❖ Design Patterns

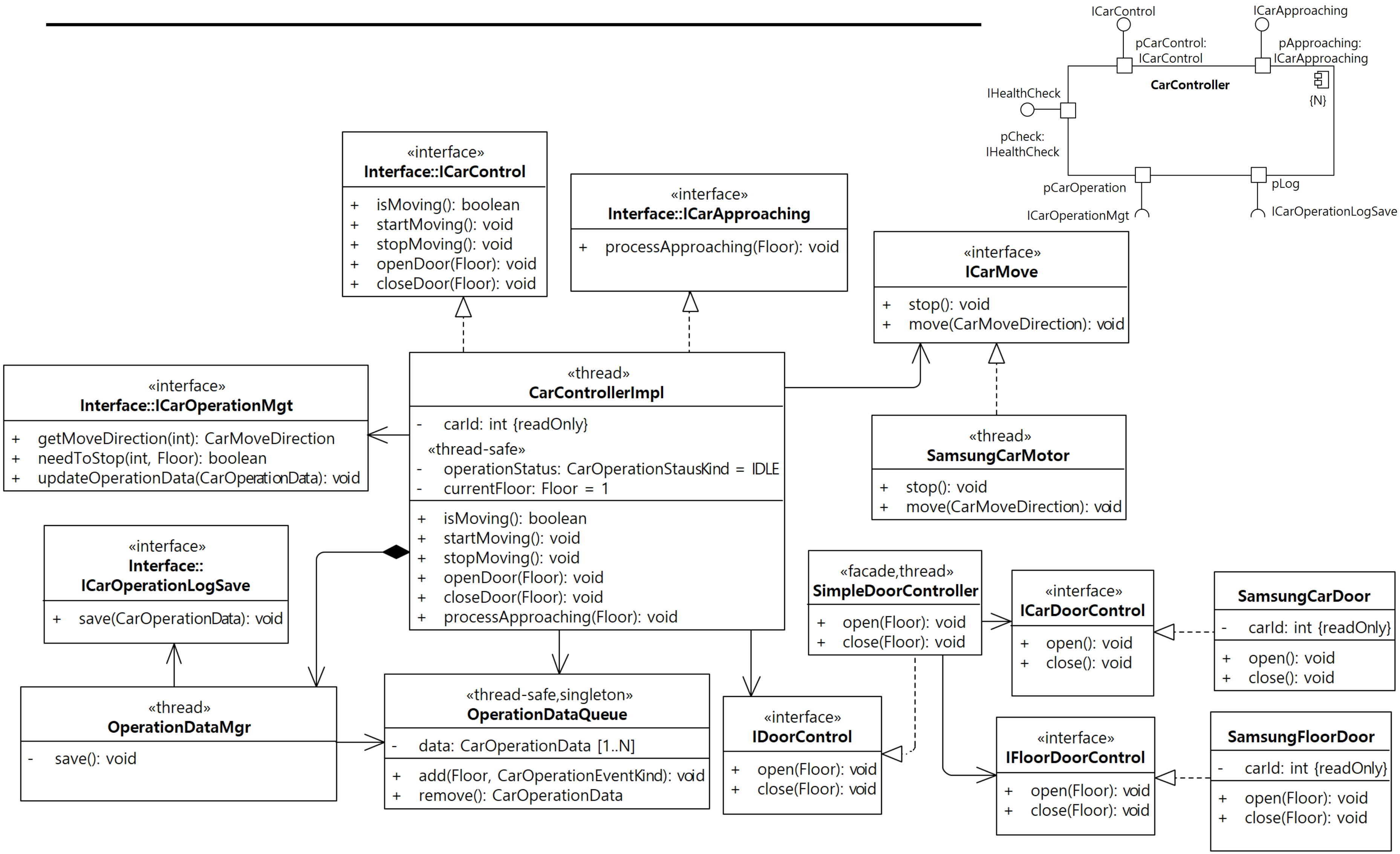
- Strategy, Template method, Façade, …
- Factory method, Abstract factory, …

Component Structure View

❖ CarController Component



Component Structure View: CarController



ICarControl
pCarControl:
ICarControl
ICarApproaching
pApproaching:
ICarApproaching
CarController
{N}
IHealthCheck
pCheck:
IHealthCheck
pCarOperation
ICarOperationMgt
pLog
ICarOperationLogSave

«interface»
Interface::ICarControl
+ isMoving(): boolean
+ startMoving(): void
+ stopMoving(): void
+ openDoor(Floor): void
+ closeDoor(Floor): void

«interface»
Interface::ICarApproaching
+ processApproaching(Floor): void

«interface»
ICarMove
+ stop(): void
+ move(CarMoveDirection): void

«interface»
Interface::ICarOperationMgt
+ getMoveDirection(int): CarMoveDirection
+ needToStop(int, Floor): boolean
+ updateOperationData(CarOperationData): void

«thread»
CarControllerImpl
- carId: int {readOnly}
«thread-safe»
- operationStatus: CarOperationStausKind = IDLE
- currentFloor: Floor = 1
+ isMoving(): boolean
+ startMoving(): void
+ stopMoving(): void
+ openDoor(Floor): void
+ closeDoor(Floor): void
+ processApproaching(Floor): void

«thread»
SamsungCarMotor
+ stop(): void
+ move(CarMoveDirection): void

«interface»
Interface::
ICarOperationLogSave
+ save(CarOperationData): void

«facade,thread»
SimpleDoorController
+ open(Floor): void
+ close(Floor): void

«interface»
ICarDoorControl
+ open(): void
+ close(): void

SamsungCarDoor
- carId: int {readOnly}
+ open(): void
+ close(): void

«thread»
OperationDataMgr
- save(): void

«thread-safe,singleton»
OperationDataQueue
- data: CarOperationData [1..N]
+ add(Floor, CarOperationEventKind): void
+ remove(): CarOperationData

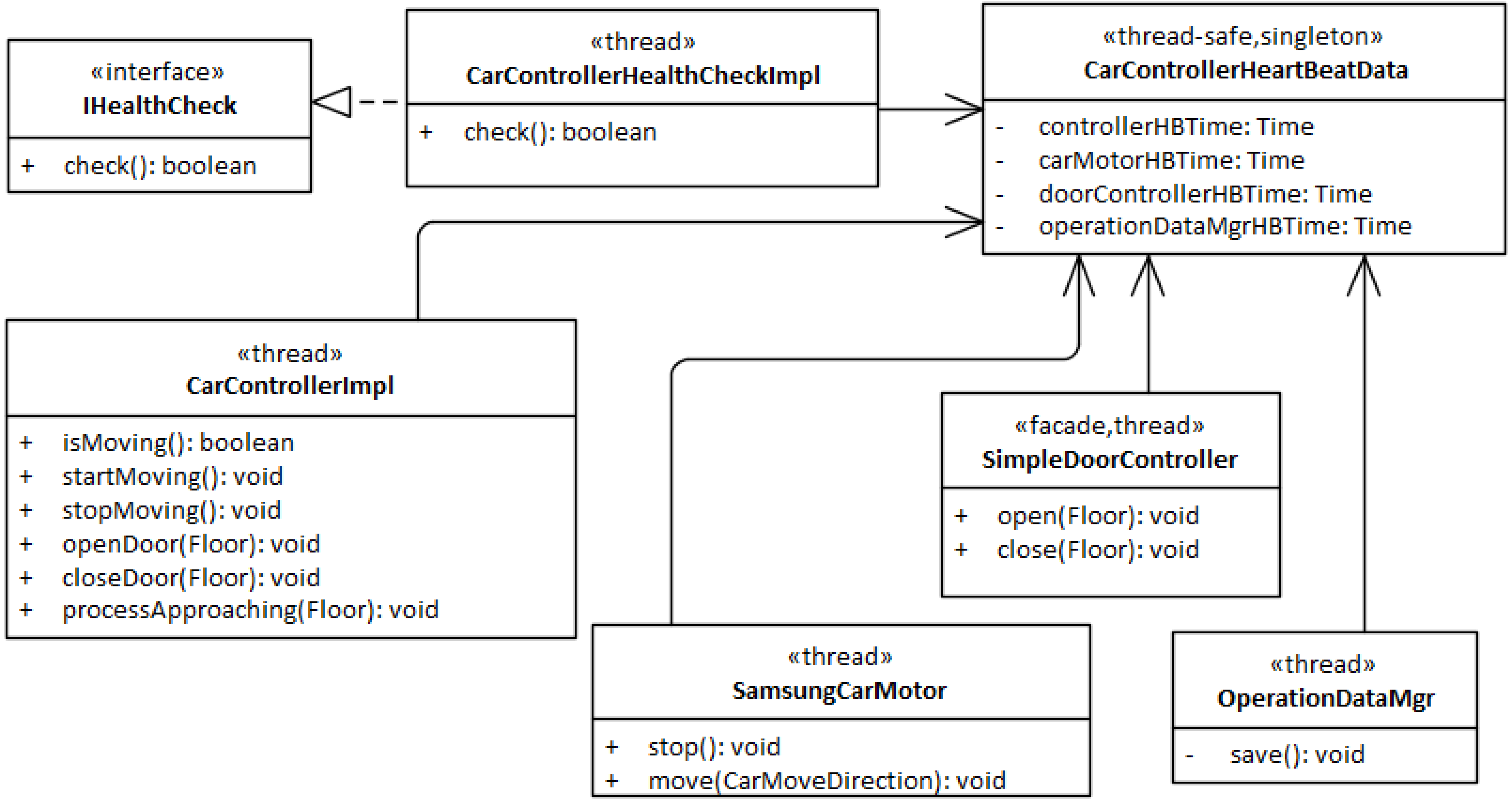
«interface»
IDoorControl
+ open(Floor): void
+ close(Floor): void

«interface»
IFloorDoorControl
+ open(Floor): void
+ close(Floor): void

SamsungFloorDoor
- carId: int {readOnly}
+ open(Floor): void
+ close(Floor): void

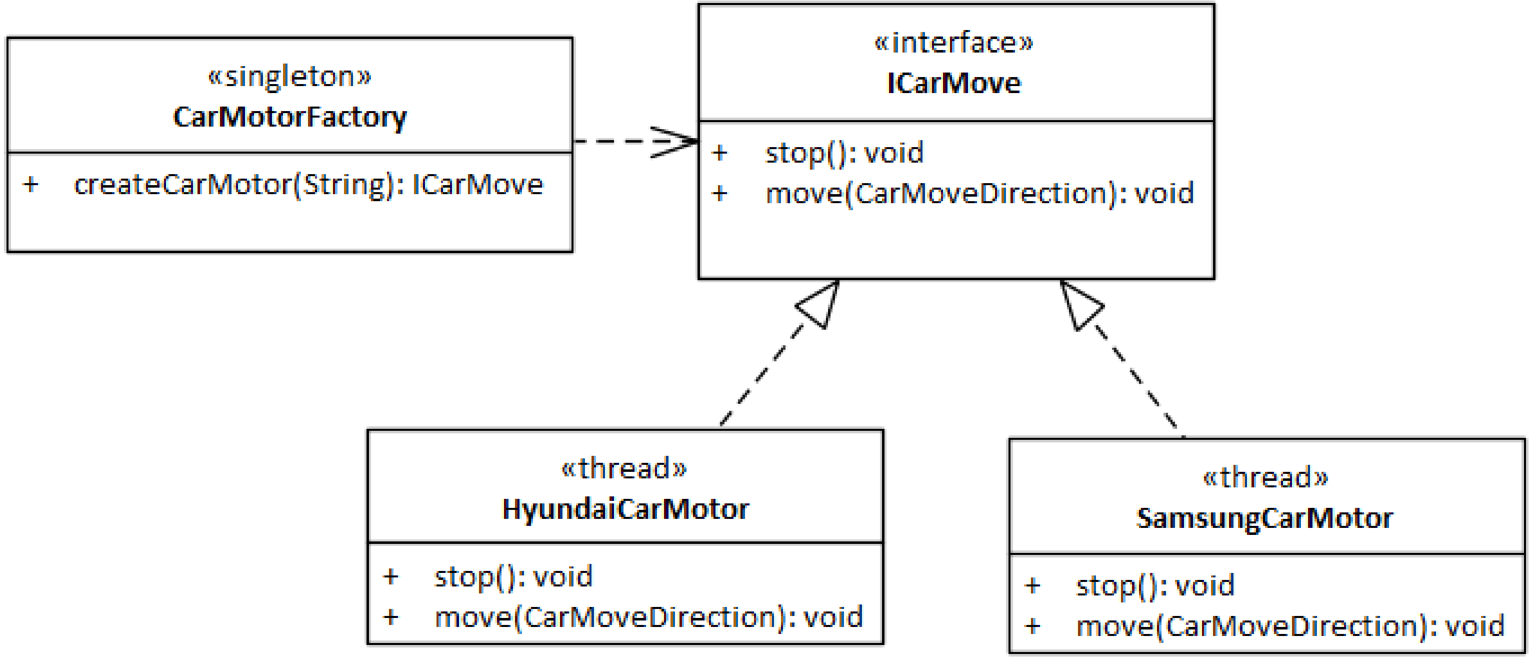
Component Structure View: CarController

Health Monitoring



Component Structure View: CarController

Factory method pattern



«singleton»
CarMotorFactory

+ createCarMotor(String): ICarMove

«interface»
ICarMove

+ stop(): void
+ move(CarMoveDirection): void

«thread»
HyundaiCarMotor

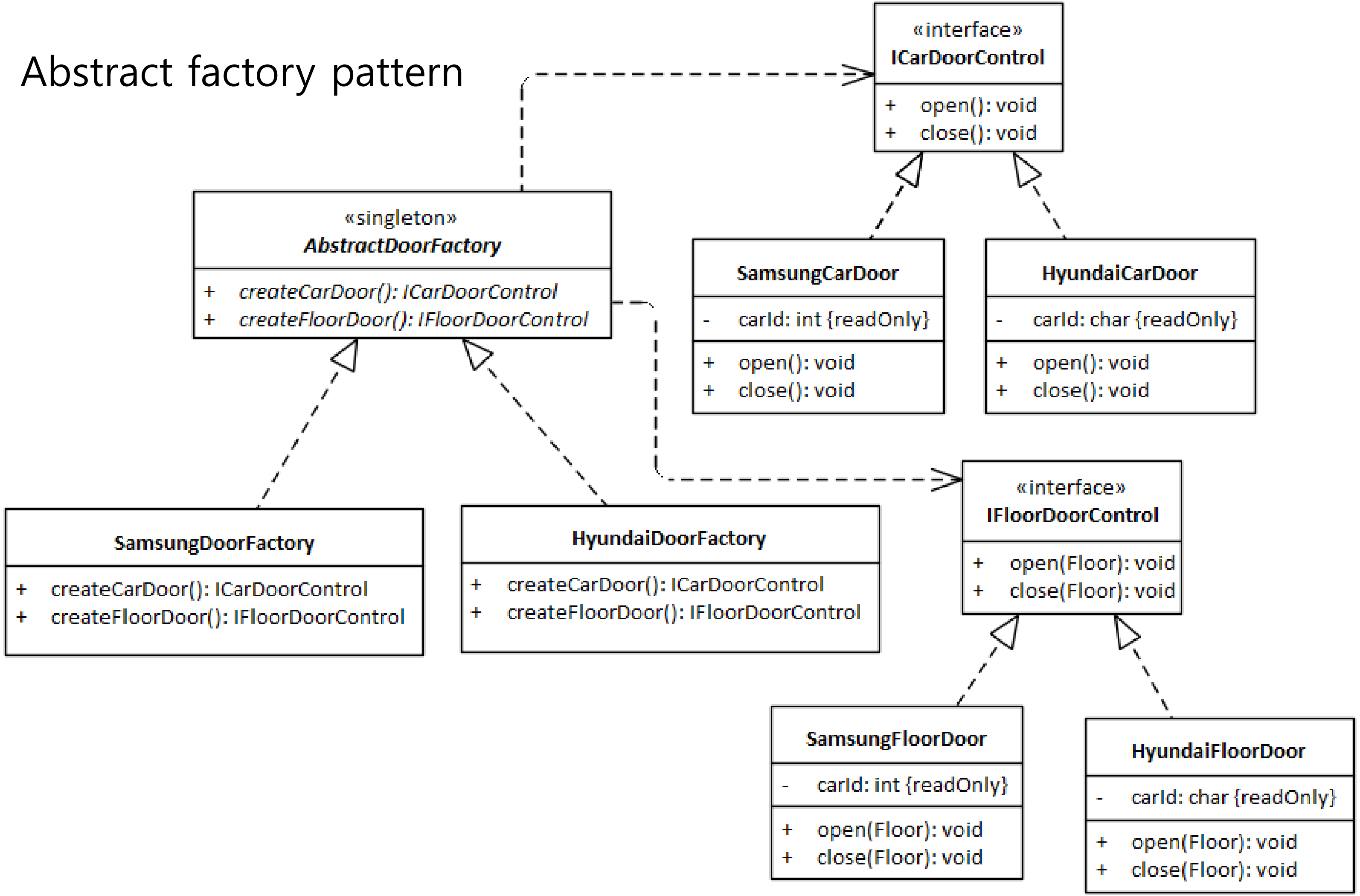
+ stop(): void
+ move(CarMoveDirection): void

«thread»
SamsungCarMotor

+ stop(): void
+ move(CarMoveDirection): void

Component Structure View: CarController

Abstract factory pattern



«interface»
ICarDoorControl

+ open(): void
+ close(): void

«singleton»
AbstractDoorFactory

+ *createCarDoor(): ICarDoorControl*
+ *createFloorDoor(): IFloorDoorControl*

SamsungCarDoor

- carId: int {readOnly}

+ open(): void
+ close(): void

HyundaiCarDoor

- carId: char {readOnly}

+ open(): void
+ close(): void

SamsungDoorFactory

+ createCarDoor(): ICarDoorControl
+ createFloorDoor(): IFloorDoorControl

HyundaiDoorFactory

+ createCarDoor(): ICarDoorControl
+ createFloorDoor(): IFloorDoorControl

«interface»
IFloorDoorControl

+ open(Floor): void
+ close(Floor): void

SamsungFloorDoor

- carId: int {readOnly}

+ open(Floor): void
+ close(Floor): void

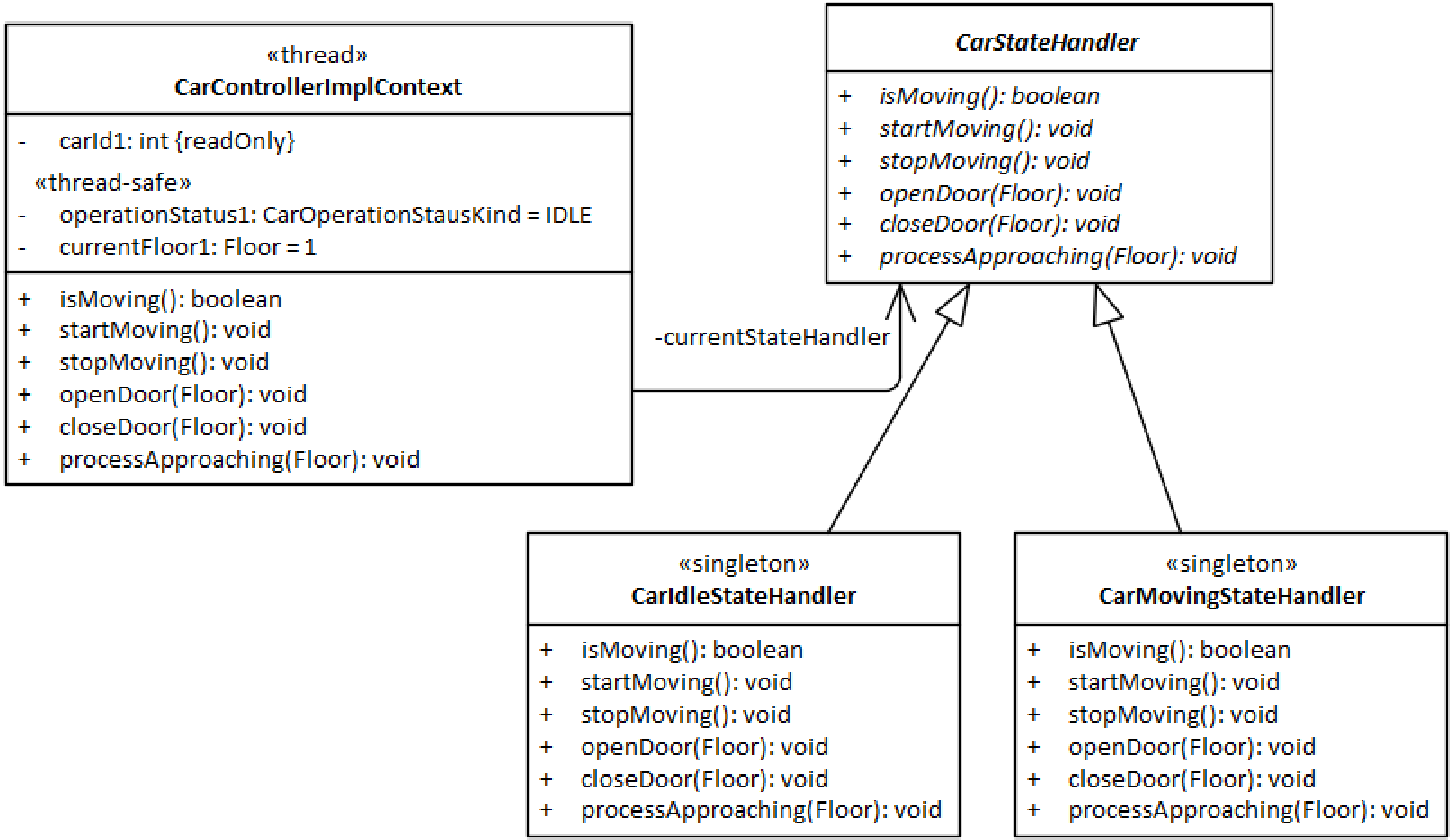
HyundaiFloorDoor

- carId: char {readOnly}

+ open(Floor): void
+ close(Floor): void

Component Structure View: CarController

State pattern



«thread»
CarControllerImplContext

- carId1: int {readOnly}

«thread-safe»

- operationStatus1: CarOperationStausKind = IDLE
- currentFloor1: Floor = 1

+ isMoving(): boolean
+ startMoving(): void
+ stopMoving(): void
+ openDoor(Floor): void
+ closeDoor(Floor): void
+ processApproaching(Floor): void

-currentStateHandler

CarStateHandler

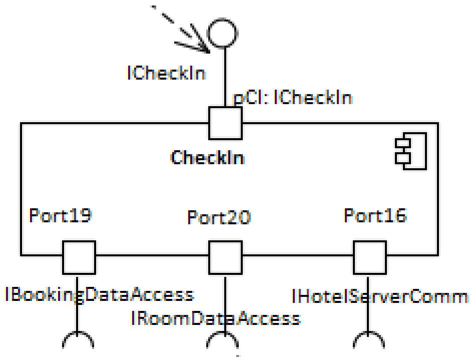
+ *isMoving(): boolean*
+ *startMoving(): void*
+ *stopMoving(): void*
+ *openDoor(Floor): void*
+ *closeDoor(Floor): void*
+ *processApproaching(Floor): void*

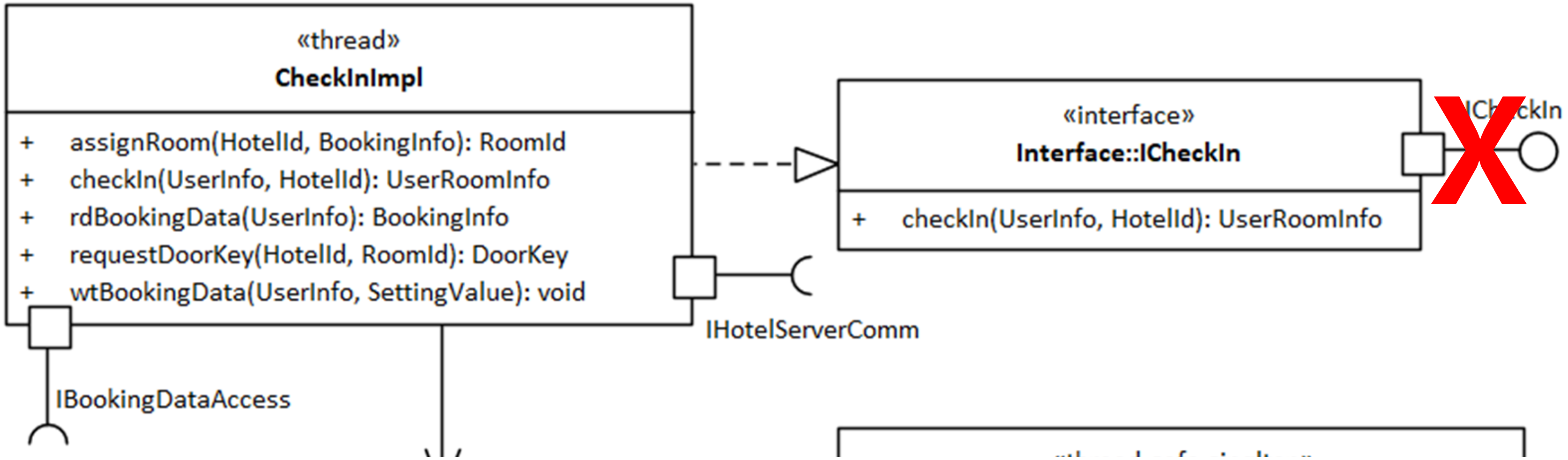
«singleton»
CarIdleStateHandler

+ isMoving(): boolean
+ startMoving(): void
+ stopMoving(): void
+ openDoor(Floor): void
+ closeDoor(Floor): void
+ processApproaching(Floor): void

«singleton»
CarMovingStateHandler

+ isMoving(): boolean
+ startMoving(): void
+ stopMoving(): void
+ openDoor(Floor): void
+ closeDoor(Floor): void
+ processApproaching(Floor): void

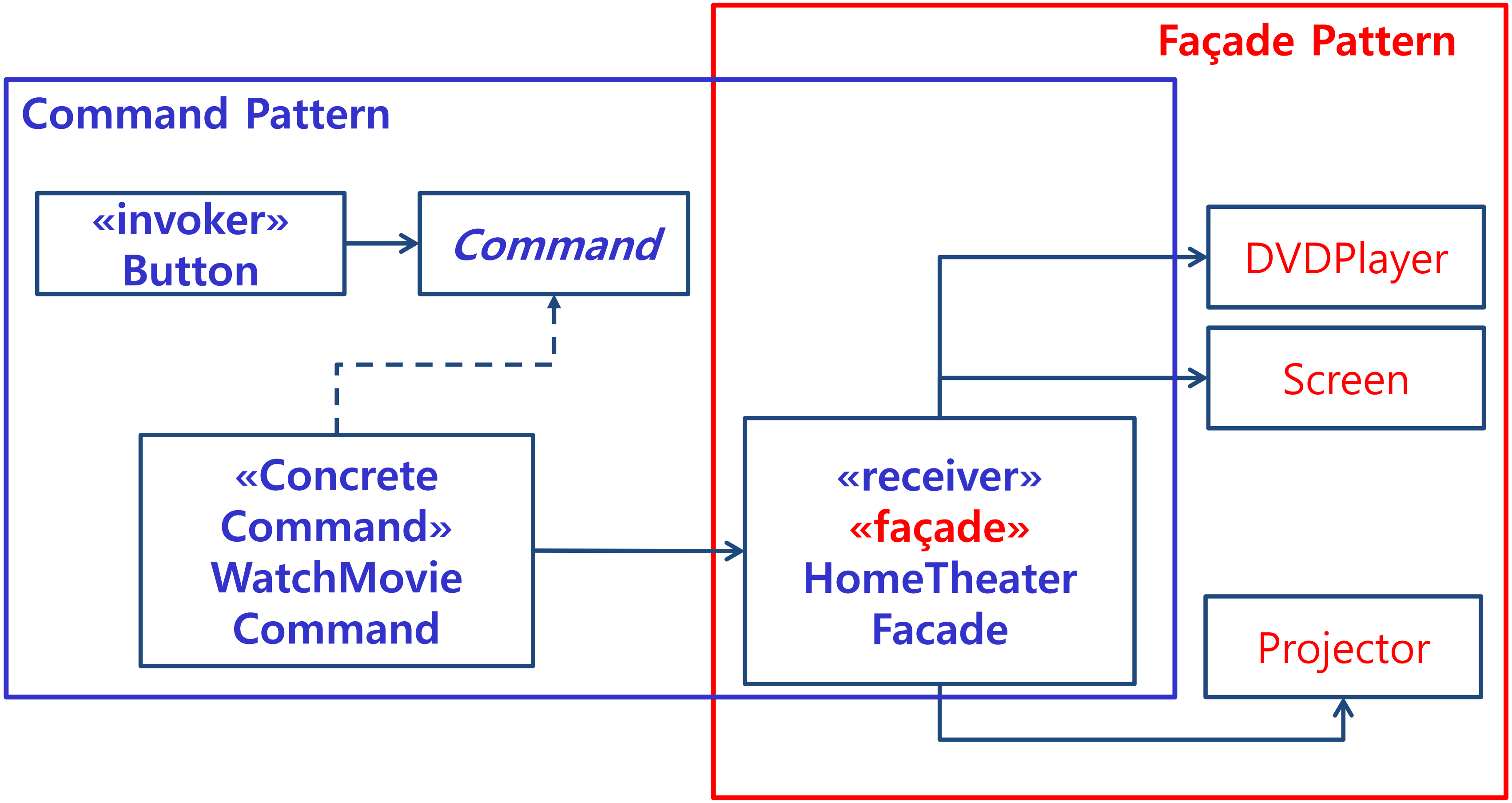

ICheckIn
pCI: ICheckIn
CheckIn
Port19
Port20
Port16
IBookingDataAccess
IRoomDataAccess
IHotelServerComm


«thread»
CheckInImpl
+ assignRoom(HotelId, BookingInfo): RoomId
+ checkIn(UserInfo, HotelId): UserRoomInfo
+ rdBookingData(UserInfo): BookingInfo
+ requestDoorKey(HotelId, RoomId): DoorKey
+ wtBookingData(UserInfo, SettingValue): void
«interface»
Interface::ICheckIn
+ checkIn(UserInfo, HotelId): UserRoomInfo
ICheckIn
X
IHotelServerComm
IBookingDataAccess

GoF Pattern Catalogue

	Creational	**Structural**	**Behavioral**
Class-level	Factory Method	Adapter (class)	Interpreter Template Method
Object-level	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Facade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

Combination of Patterns



Element List

- Describe each element comprising the component with its responsibility.

Element Name	Responsibility
Class21	
Class22	
Internal_2_1	
Class23	
Class24	
Class25	

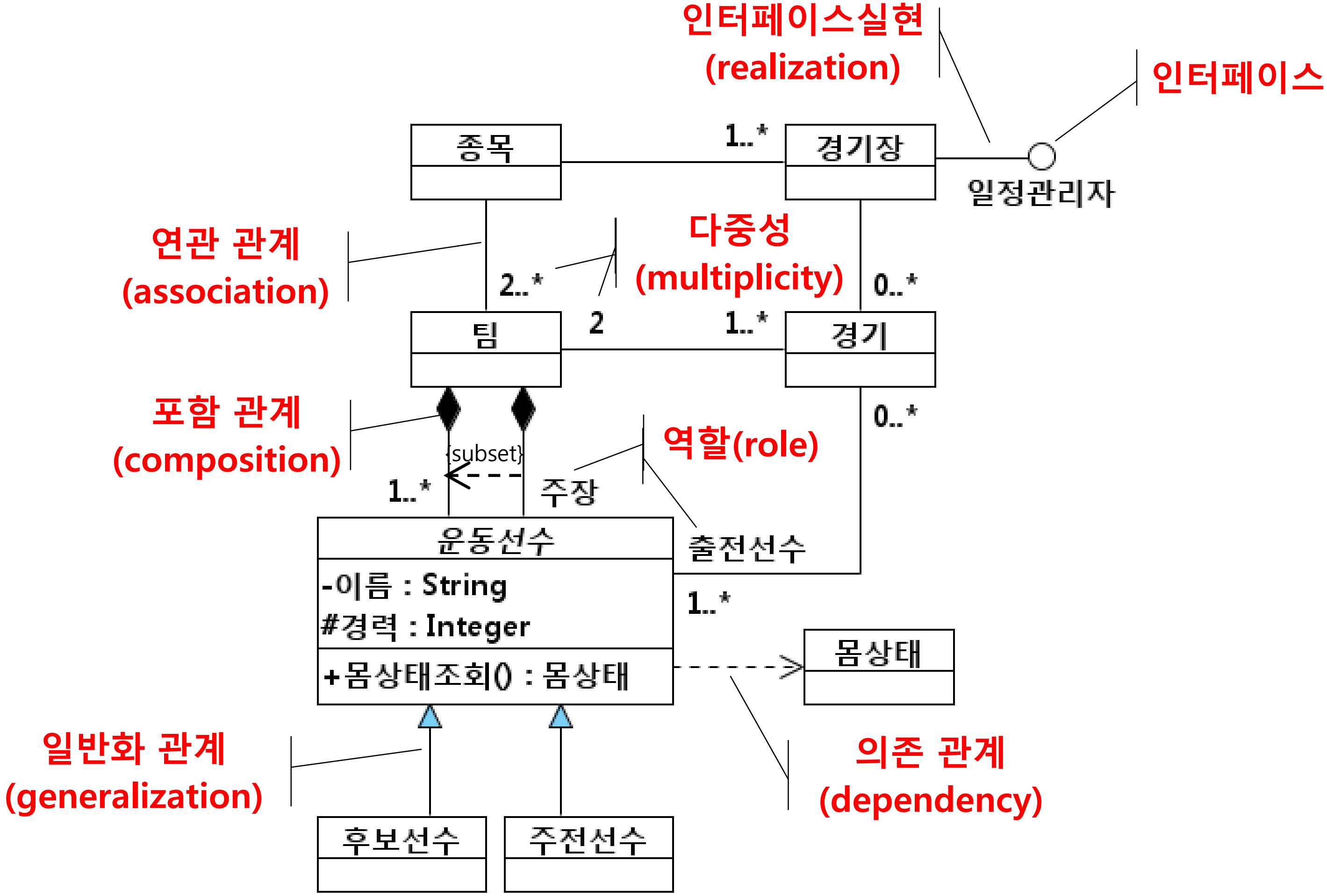
Design Rationale

- Describe the rationale for the decomposition.
- Relate the design decisions to the quality requirement by describing how each quality requirement are promoted by the decomposition.

QA	Relevant Elements	Description
QA-01		
QA-02		

CLASS DIAGRAM

Class Diagram - Overview



인터페이스실현
(realization)

인터페이스

연관 관계
(association)

다중성
(multiplicity)

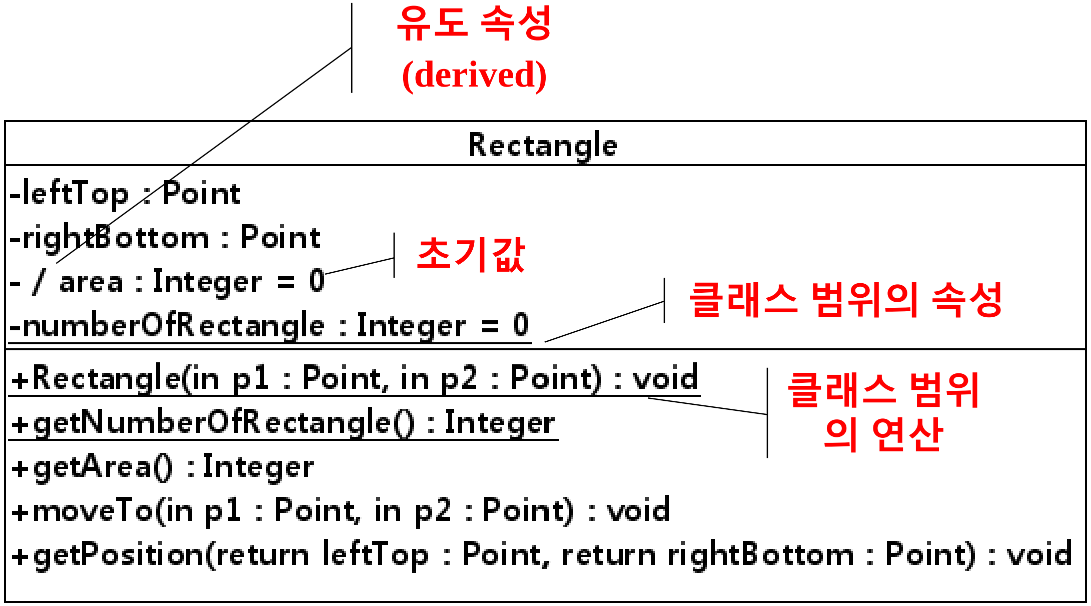
포함 관계
(composition)

역할(role)

일반화 관계
(generalization)

의존 관계
(dependency)

Class



유도 속성
(derived)

Rectangle

-leftTop : Point
-rightBottom : Point
- / area : Integer = 0
-numberOfRectangle : Integer = 0

초기값

클래스 범위의 속성

+Rectangle(in p1 : Point, in p2 : Point) : void
+getNumberOfRectangle() : Integer
+getArea() : Integer
+moveTo(in p1 : Point, in p2 : Point) : void
+getPosition(return leftTop : Point, return rightBottom : Point) : void

클래스 범위
의 연산

```
class Rectangle {
private:
  Point leftTop ;
  Point rightBottom ;
  int area{0} ;
  static int numberOfRectanlge{0} ;
public:
  Rectangle(Point p1, Point p2) ;
  static int getNumberOfRectangle() ;
  int getArea() ;
  void moveTo(Point p1, Point p2) ;
  void getPosition(Point& leftTop,
     Point& rightBottom) ;
} ;
```

Operation Property

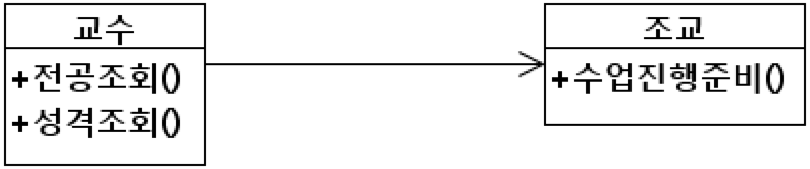
Property	Description
query	the operation does not change the state of the system.
ordered	when there is a multi-valued return Parameter and means that its values are ordered.
unordered	when there is a multi-valued return Parameter and means that its values are not ordered
unique	when there is a multi-valued return Parameter and means that its values have no duplicates.
nonunique	when there is a multi-valued return Parameter and means that its values may have duplicates.

Operation Concurrency Property

- specifies the semantics of concurrent calls to the same instance.

Kind	Description
sequential (default)	No concurrency management mechanism is associated with the operation and, therefore, concurrency conflicts may occur. Instances that invoke an operation need to coordinate so that only one invocation to a target on any operation occurs at once
concurrent	Multiple invocations that overlap in time may occur to one instance and all of them may proceed concurrently
guarded	Multiple invocations that overlap in time may occur to one instance, but only one is allowed to commence. The others are blocked until the performance of the currently executing operation is complete.

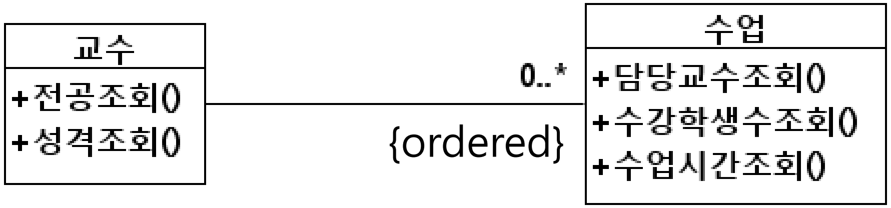
Association



class 교수 {
  private 조교 a조교 ;
  public void 전공조회() { **a조교**....() ; }
  public void 성격조회() { **a조교**....() ; }
}
	class 조교 {
	  ~~**private 교수 a교수 ;**~~
	  public void 수업진행준비() {~~**a교수....() ;**~~}
	}

class 교수 {
  private: 조교* a조교 ;
  public: void 전공조회() { **a조교->**...() ; }
  public: void 성격조회() { **a조교->**...() ; }
}
	class 조교 {
	  ~~**private 교수* a교수 ;**~~
	  public: void 수업진행준비() {~~**a교수->...() ;**~~}
	}

Association Multiplicity



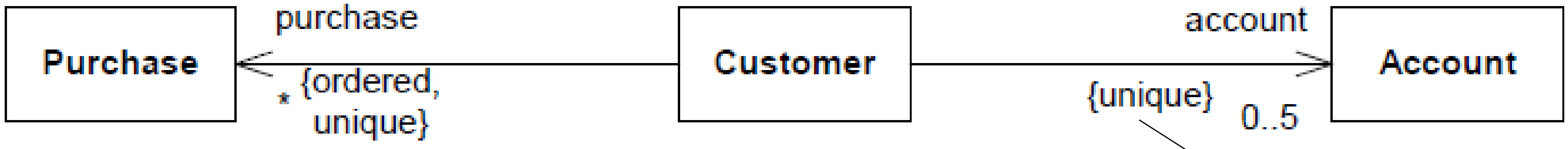
```
class 교수 {
  private List<수업> a수업 ;
  public void 전공조회() { … }
  public void 성격조회() { … }
}
```

```
class 수업 {
  private 교수 a교수 ;
  public void 담당교수조회() { … }
  public void 수강학생수조회() { … }
  public void 수업시간조회() { … }
}
```

```
class 교수 {
  private: vector<수업*> a수업 ;
  public: void 전공조회() { … }
  public: void 성격조회() { … }
}
```

```
class 수업 {
  private: 교수* a교수 ;
  public: void 담당교수조회() { … }
  public: void 수강학생수조회() { … }
  public: void 수업시간조회() { … }
}
```

Association Multiplicity

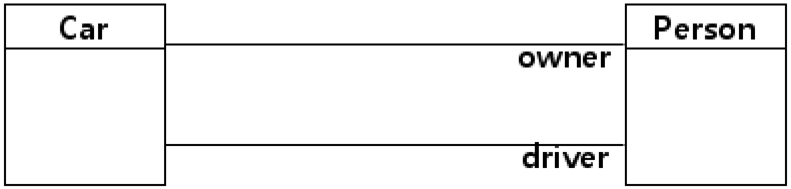


ordered
unordered (default)

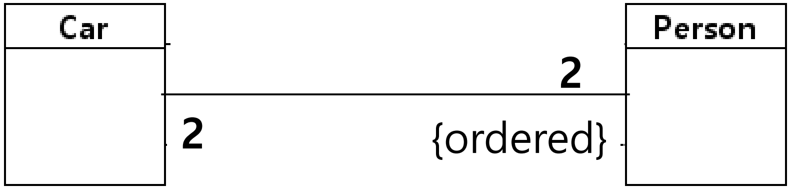
unique (default)
nonunique

ordered	unique	Collection Type
false	**true**	**Set**
true	**true**	**OrderedSet**
false	false	**Bag**
true	false	**Sequence**

Multiple Association and Association Multiplicity

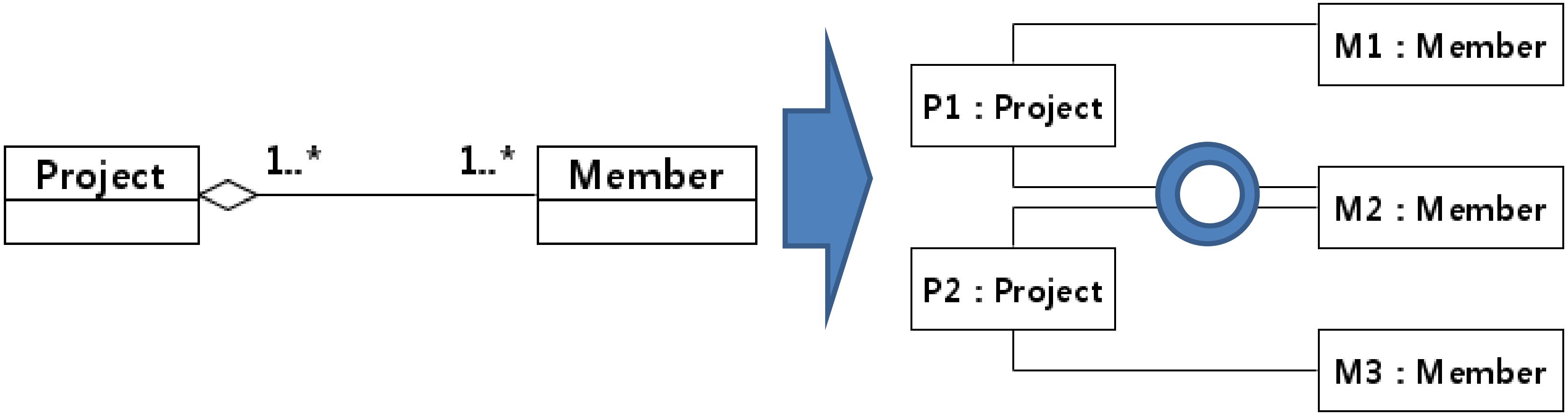


```
class Car {
  private Person owner ;
  private Person driver ;

  void setOwner(Person p) {
    owner = p ;
  }
  void setDriver(Person p) {
    driver = p ;
}
```



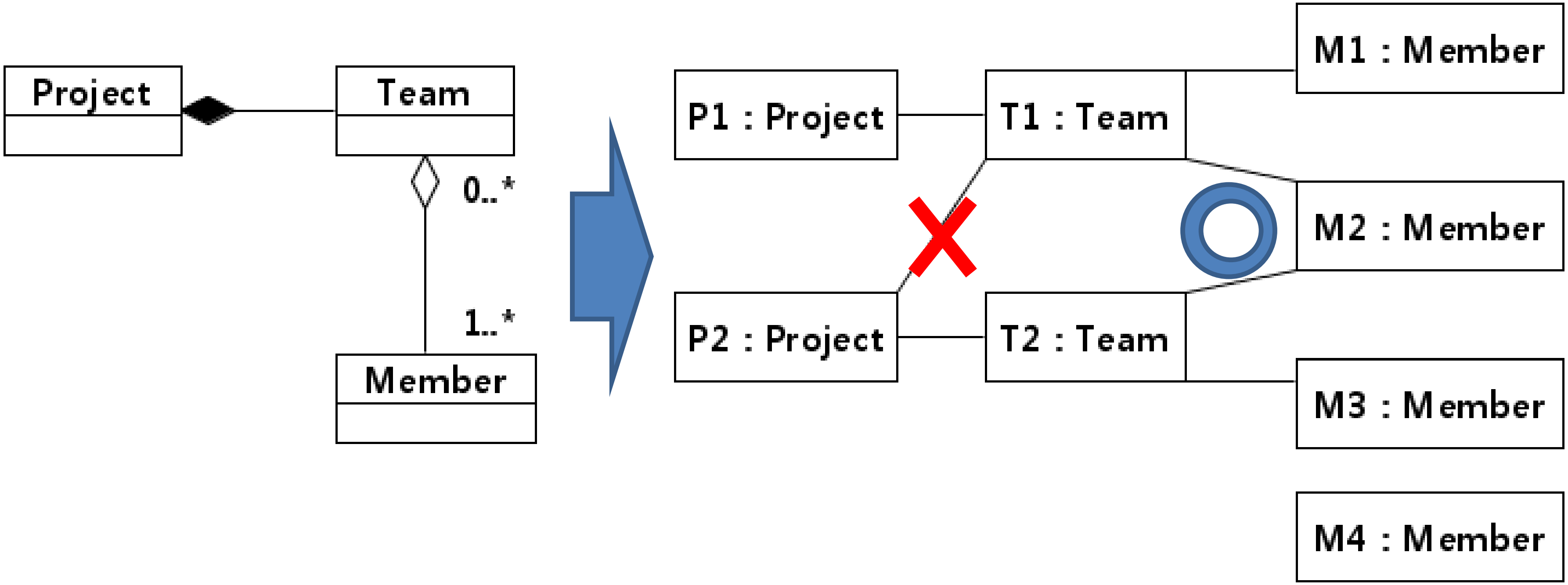
```
class Car {
  private Person[] persons  ;

  void setPerson(int i, Person p) {
    persons[i] = p ;
}
```

Aggregation

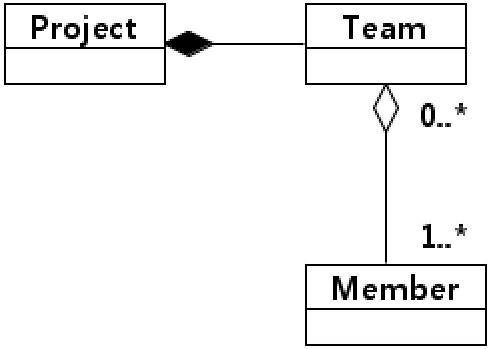


```
int main() {
   Project p1 ;
   Project p2 ;
   Member m1, m2, m3 ;

   p1.addMember(&m1) ;
   p1.addMember(&m2) ;

   p2.addMember(&m2) ;
   p2.addMember(&m3) ;
}
```

Composition



Aggregation/Composition in C++

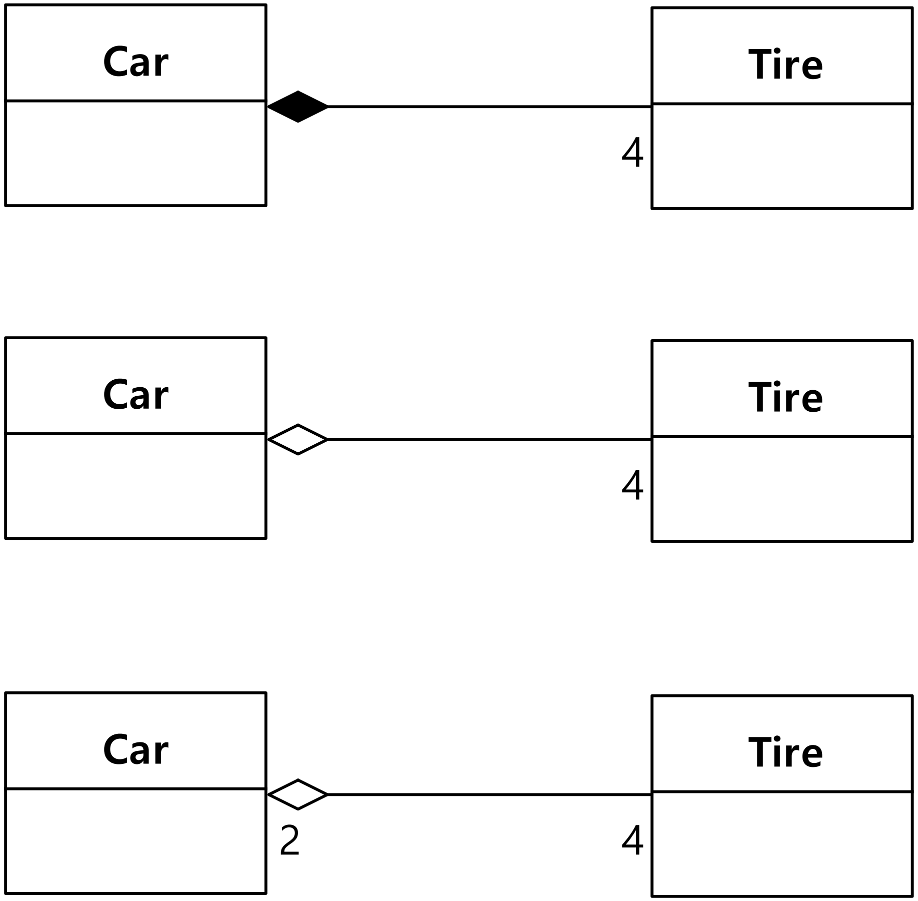


```
class Project {
  Team theTeam ;
public:
  Project() : theTeam(*this) {}
  Team& getTeam() { return theTeam ; }
} ;
```

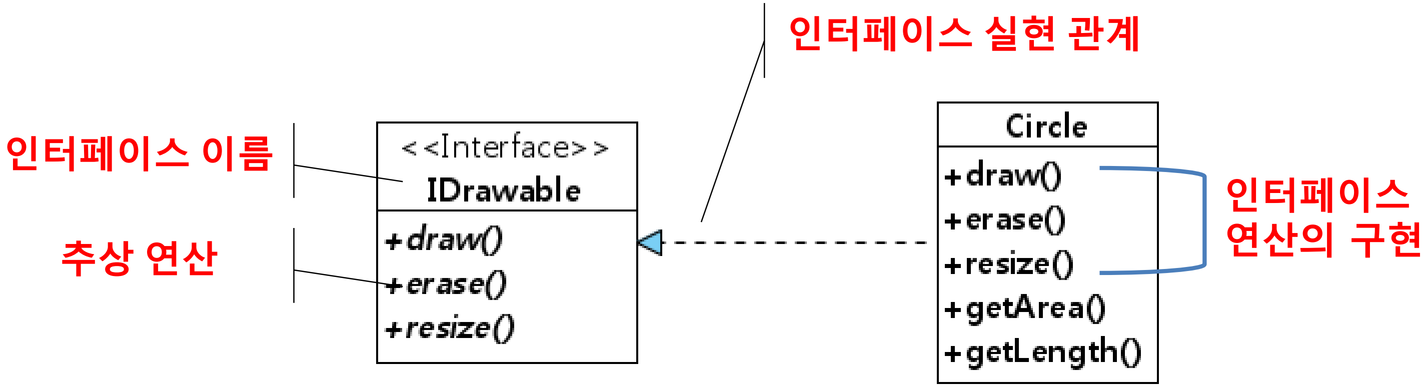
```
class Team {
  Project& theProject ;
  vector<Member*> members ;
public:
 Team(Project& project) : theProject(project) {}
  void addMember(Member* m) {
    members->push_back(m) ;
    m->addTeam(this) ;
  }
  void removeMember(Member* m) {
    members->remove(m) ;
    m->removeTeam(this) ;
  }
} ;
```

```
class Member {
  vector<Team*> teams ;
public:
  void addTeam(Team* t) {
     teams ->push_back(t) ;
  }
  void removeTeam(Team* t) {
     teams ->remove(t) ;
  }
} ;
```

Composition vs Aggregation



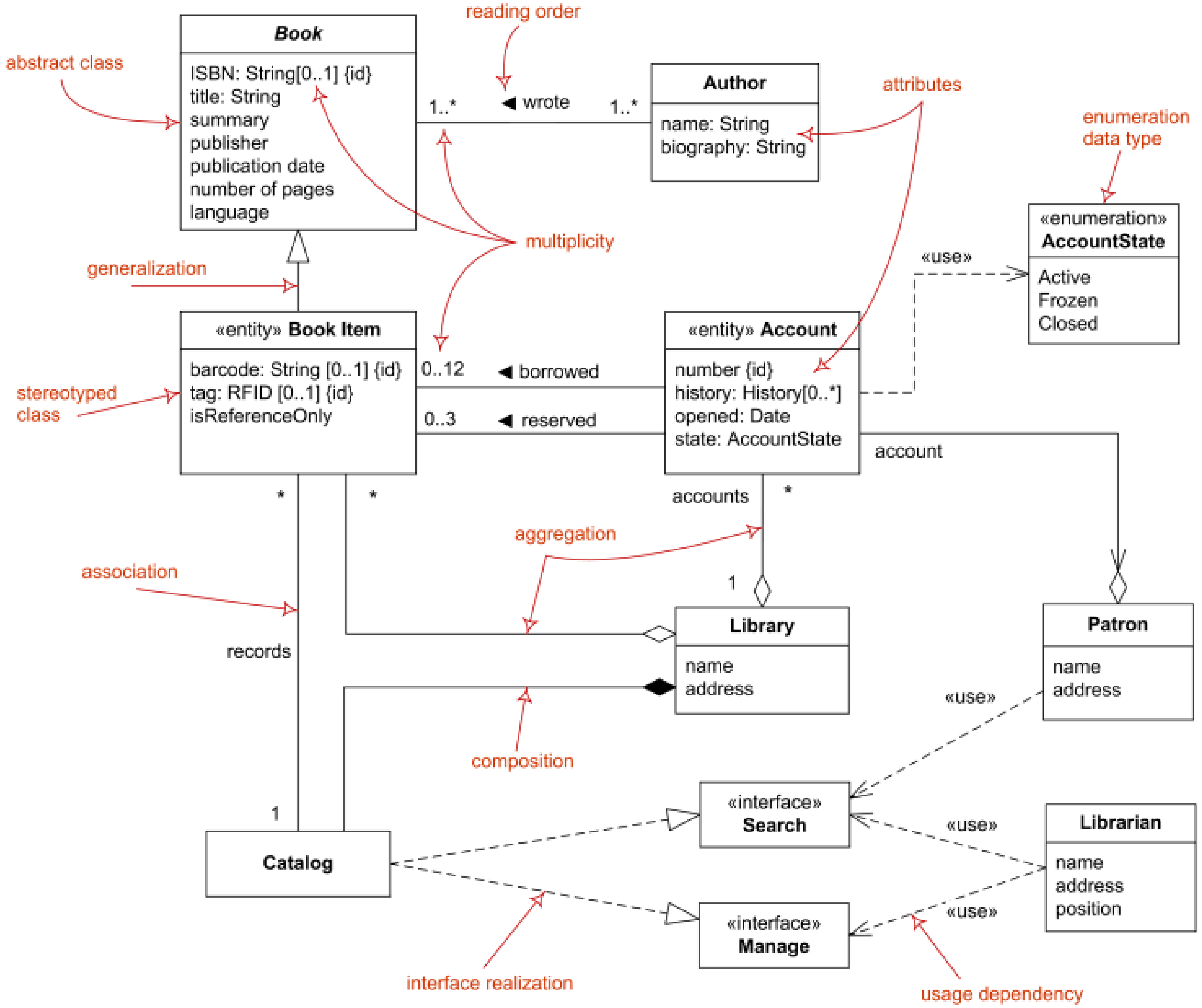
Interfaces



```
class IDrawable {
public:
 virtual void draw()  = 0 ;
 virtual void erase()  = 0 ;
 virtual void resize()  = 0 ;
} ;
```

```
class Circle : public IDrawable {
public:
 void draw() { … }
 void erase() { … }
 void resize() { … }
 void getArea() ;
 void getLength() ;
} ;
```

Class Diagram - Summary



Book
ISBN: String[0..1] {id}
title: String
summary
publisher
publication date
number of pages
language

Author
name: String
biography: String

«enumeration»
AccountState
Active
Frozen
Closed

«entity» **Book Item**
barcode: String [0..1] {id}
tag: RFID [0..1] {id}
isReferenceOnly

«entity» **Account**
number {id}
history: History[0..*]
opened: Date
state: AccountState

Library
name
address

Patron
name
address

«interface»
Search

«interface»
Manage

Librarian
name
address
position

Catalog

DESIGN PRINCIPLES

Design Principles

- Cohesion
- Coupling
- Complexity

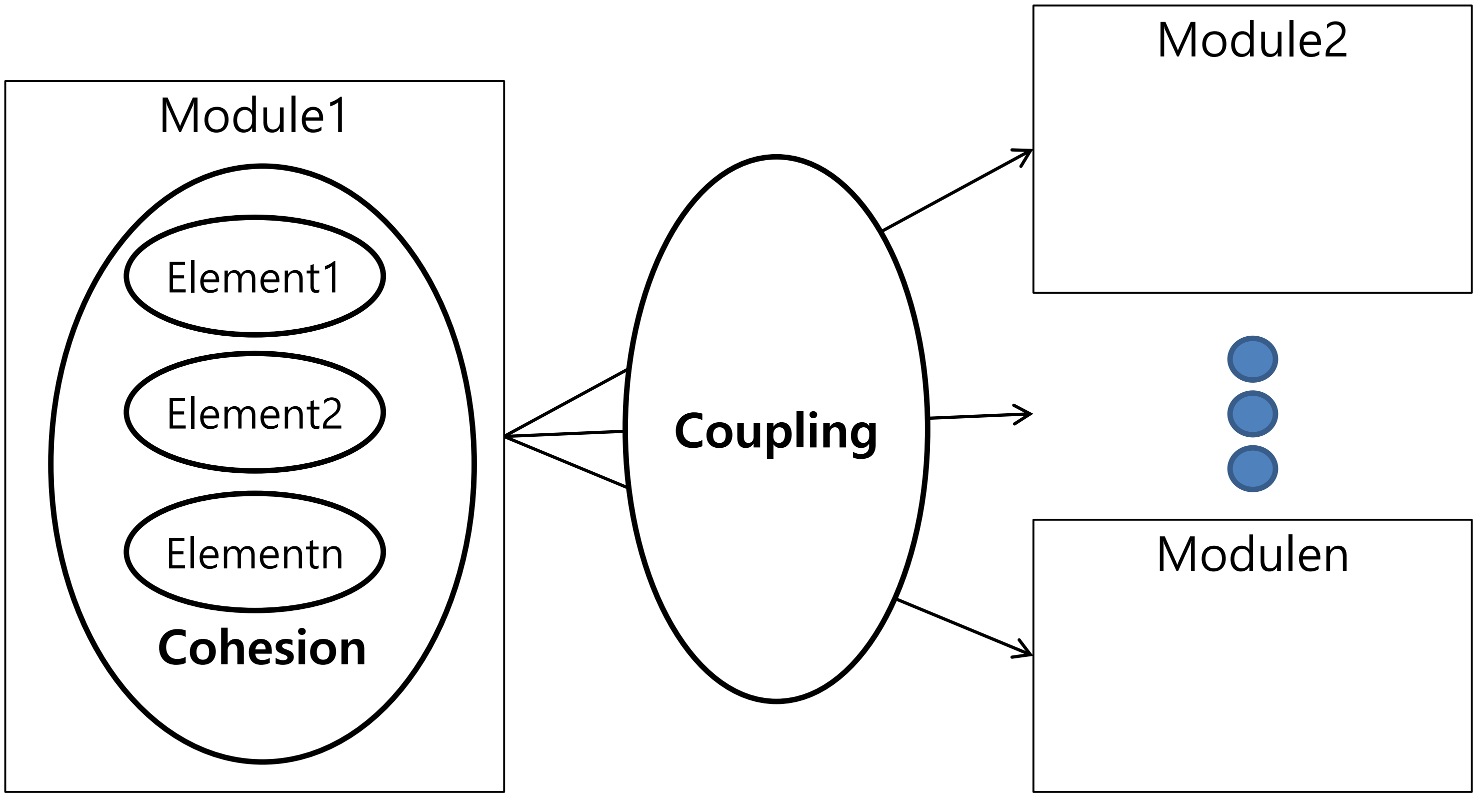
- SOLID

- Package Cohesion and Package Coupling

COHESION AND COUPLING

Cohesion vs Coupling

- Coupling: Degree of interdependence between two modules.
- Cohesion: strength of functional relatedness of elements within a module

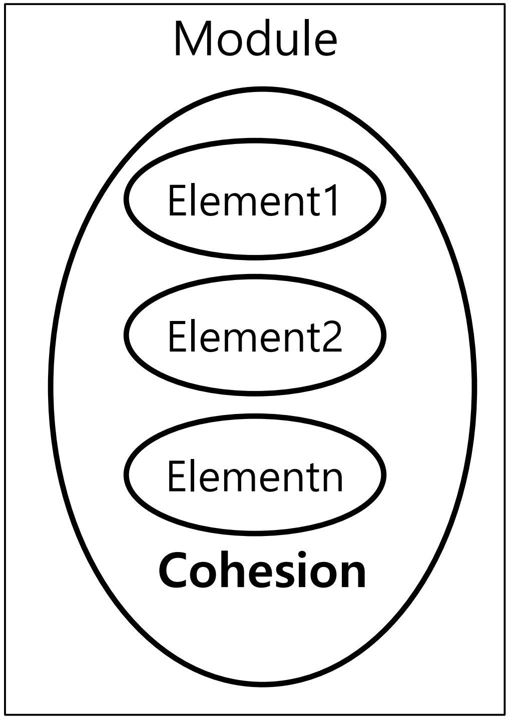


Cohesion

- Strength of functional relatedness of elements within a module

- Cohesion is a universal concept
  - Function cohesion
  - Class cohesion
  - Package cohesion
  - Component cohesion

- Cohesion metrics
  - LCOM
  - LCC/TCC



Function Cohesion

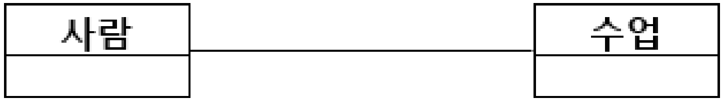
❖ More cohesive function is easier to understand

```
int sumAndProduct0(int flag, int* values, int size) {
    int result = (flag ==0) ? 0 : 1;
    for (unsigned int i = 0; i < size; i++) {
        if (flag == 0) {
           result += values[i];
        else
           result *= values[i];
    }
    return result;
}
```

Can you evaluate the design in terms of SOLID?

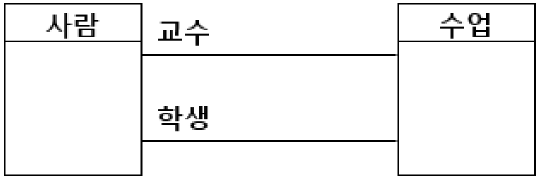
Class Cohesion

❖ Another examples of less cohesive classes



사람

수업

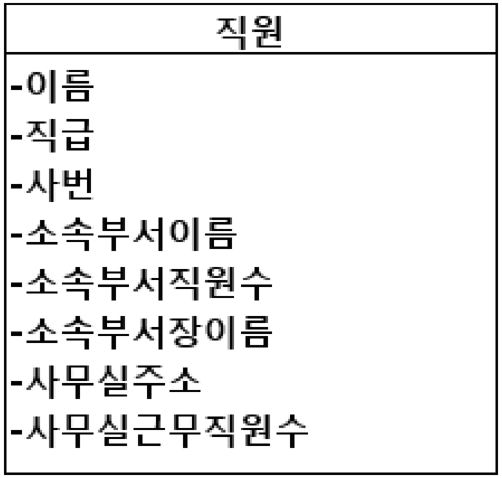


사람

교수

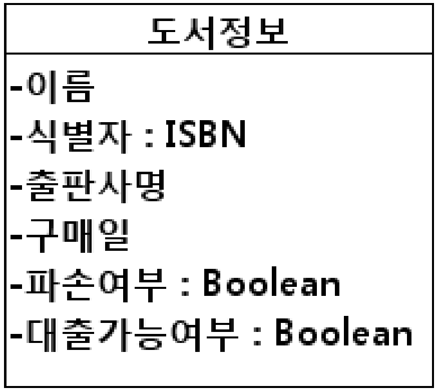
학생

수업



직원

-이름
-직급
-사번
-소속부서이름
-소속부서직원수
-소속부서장이름
-사무실주소
-사무실근무직원수



도서정보

-이름
-식별자 : ISBN
-출판사명
-구매일
-파손여부 : Boolean
-대출가능여부 : Boolean

Class Cohesion Metric - LCOM

❖ LCOM (Lack of Cohesion of Methods)

- count of the number of method pairs whose similarity is zero

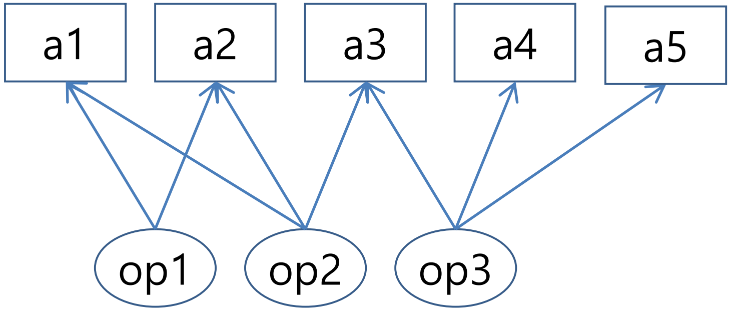
Given n methods $M_1, M_2, \ldots, M_n$ contained in a class and I_i is the set of instance variables referenced by M$_i$

Then for any method M_i we can define

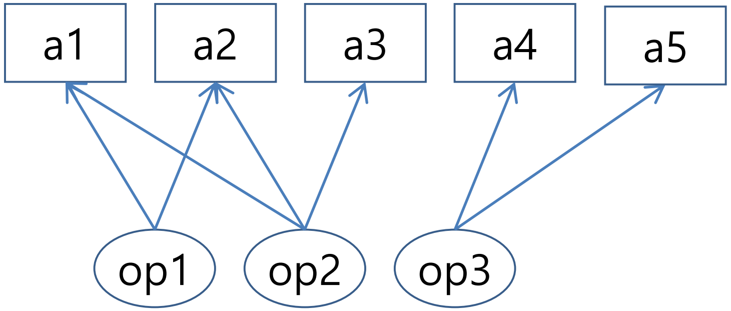
$$P = \{(I_i, I_j) \mid I_i \cap I_j = \varphi\} \text{ and } Q = \{(I_i, I_j) \mid I_i \cap I_j \neq \varphi\}$$

then **LCOM** = **$|P| - |Q|$, if $|P| > |Q|$**

= **0 otherwise**



LCOM = 0



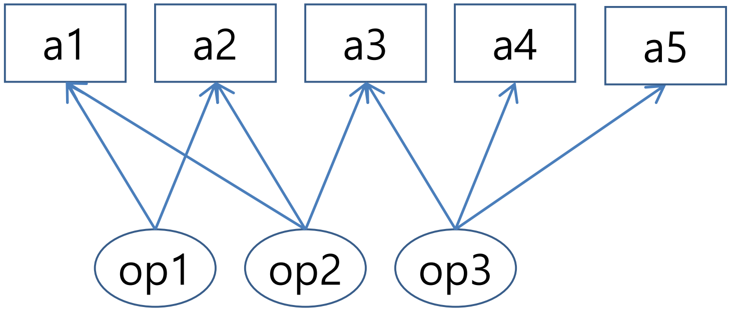
LCOM = 2 - 1

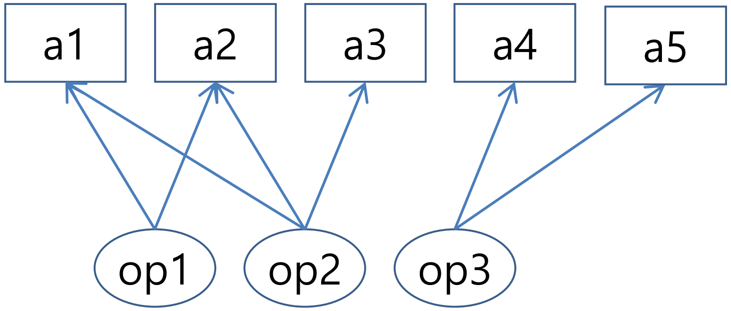
Cohesion Metrics - LCOM

❖ LCOM: Another Definition

- LCOM = 1 – (sum(MF)/M*F) : [0..1]
- LCOM HS(Hendersons-Seller) = [M – sum(MF)/F] / (M-1) [0..2]

- M is the number of methods in class
- F is the number of instance fields in the class.
- MF is the number of methods of the class accessing a particular instance field.
- Sum(MF) is the sum of MF over all instance fields of the class





LCOM = 1 – (8/15) = 0.47
LCOM HS = (3-8/5) / 2 = 0.7

LCOM = 1 – (7/15) = 0.53
LCOM HS = (3-7/5) / 2 = 0.8

Cohesion Metrics – LCC and TCC

❖ TCC(Tight Class Cohesion) and LCC(Loose Class Cohesion)

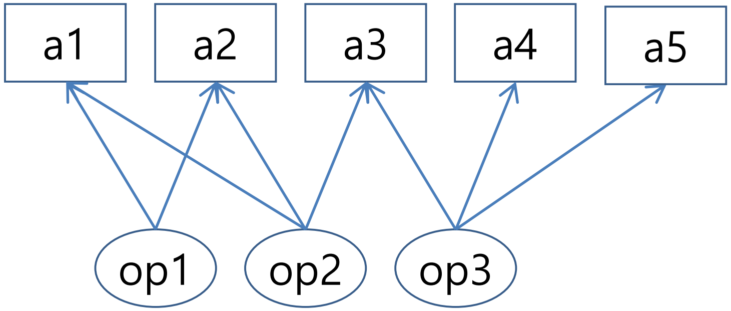
NP = maximum number of method pairs
= N * (N-1) / 2 where N is the number of methods

NDC = number of method pairs with direct connections

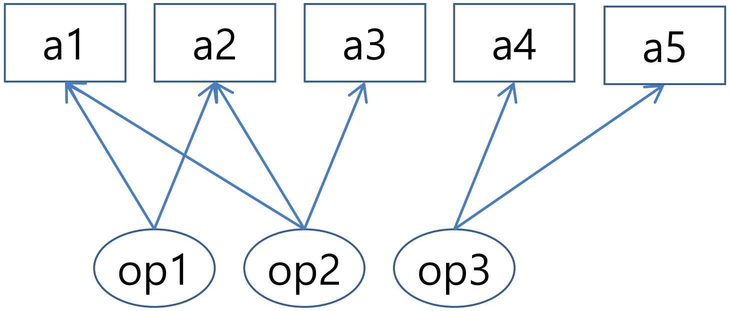
NIC = number of method pairs with indirect connections

TCC = NDC / NP

LCC = (NDC+NIC) / NP



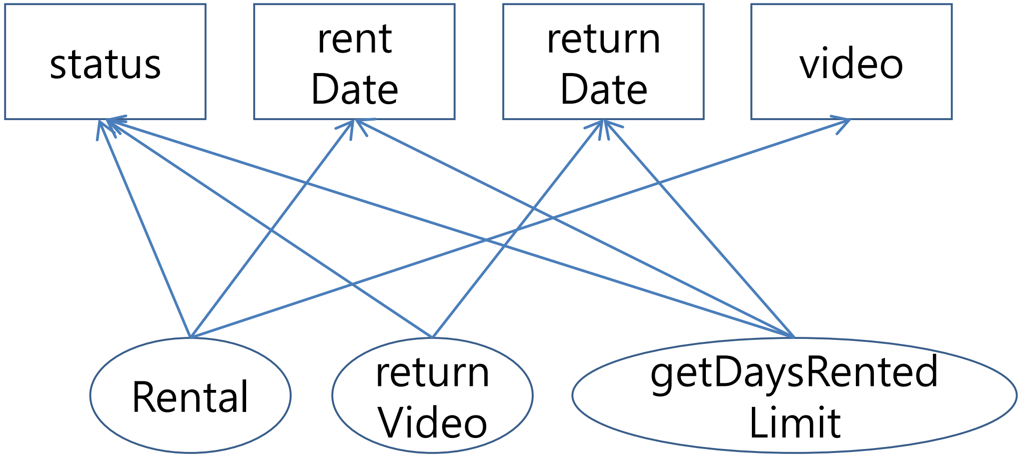
TCC = 2 / 3
LCC = 3 / 3



TCC = 1 / 3
LCC = 1 / 3

Cohesion Metrics: Example

❖ Cohesion measures for class Rental



Metric	Definition	Value	Description
LCOM	|P | - |Q |, if |P | > |Q |	0	|P| = 0, |Q| = 3
LCOM'	1 – (sum(MF)/M*F)	0.33	1 – 8 / 12
LCOM_HS	[M – sum(MF)/F] / (M-1)	0.5	[3 – 8/4] / (3-1)
TCC	NDC / NP	1	3/3
LCC	(NDC+NIC) / NP	1	3/3

How to Improve Cohesion

- Refactorings
  - Replace method with method object
  - Extract class
  - Extract subclass

  - Separate domain from presentation
  - Boundary, Control, Entity Pattern

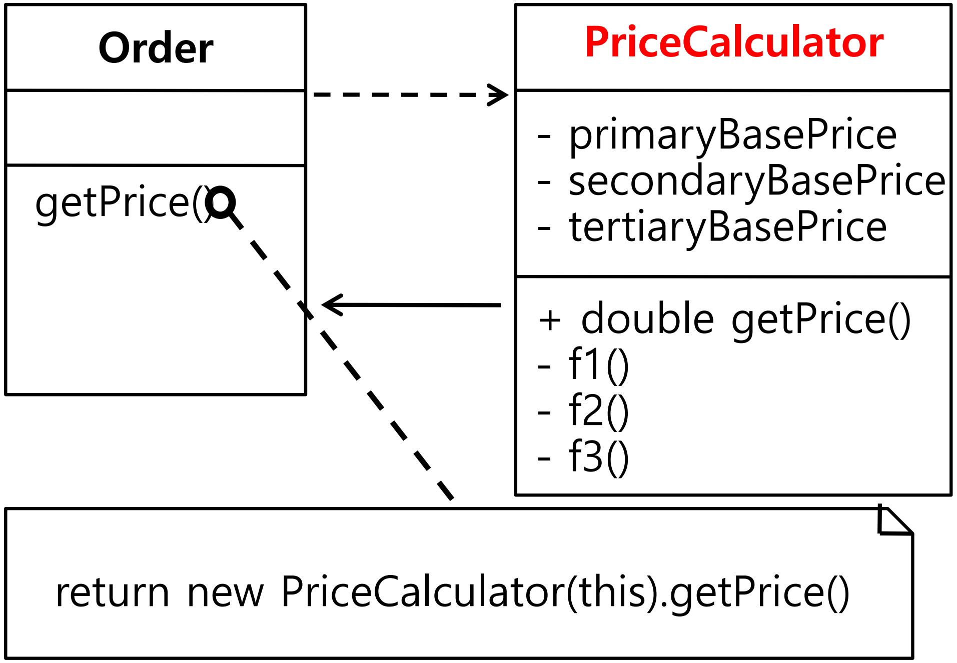
  - Replace type code with subclasses
  - Replace type code with state/strategy

Replace Method with Method Object

❖ Turn the method into its own object so that all the local variables become fields on that object.

You can then decompose the method into other methods on the same object

```
class Order {
 …
double getPrice() {
   double primaryBasePrice;
   double secondaryBasePrice;
   double tertiaryBasePrice;

   // long computation;
   …
   // f1
   // f2
   // f3
 }
}
```



Order

getPrice()

PriceCalculator

- primaryBasePrice
- secondaryBasePrice
- tertiaryBasePrice

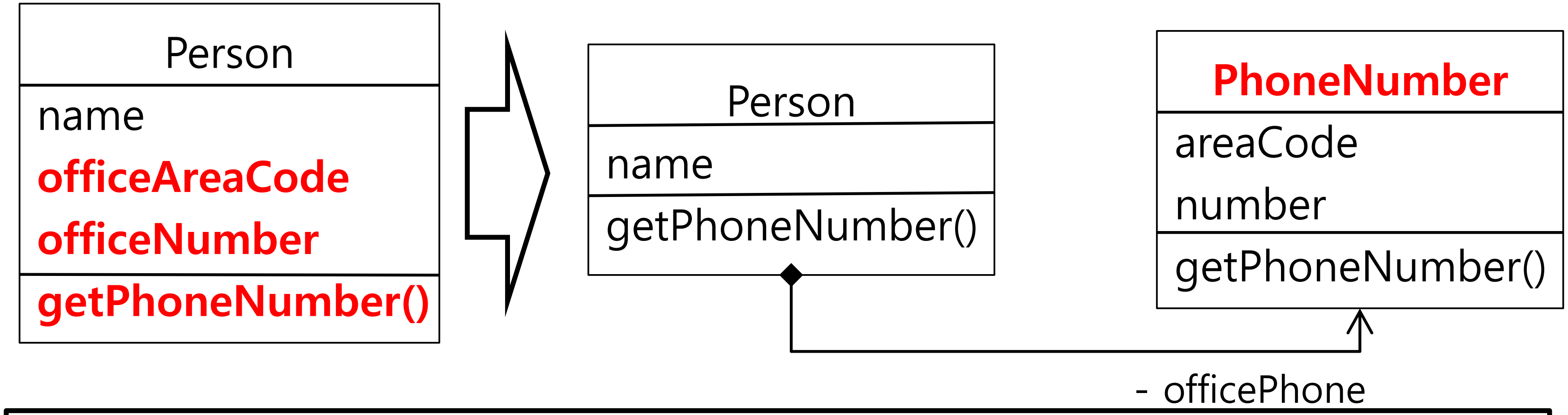
+ double getPrice()
- f1()
- f2()
- f3()

return new PriceCalculator(this).getPrice()

Then, strategy pattern can be applied to the method object

Extract Class

- **Large class**: You have one class doing work that should be done by two
- Create a new class and move the relevant fields and methods from the old class into the new class



Person

name

officeAreaCode

officeNumber

getPhoneNumber()

Person

name

getPhoneNumber()

PhoneNumber

areaCode

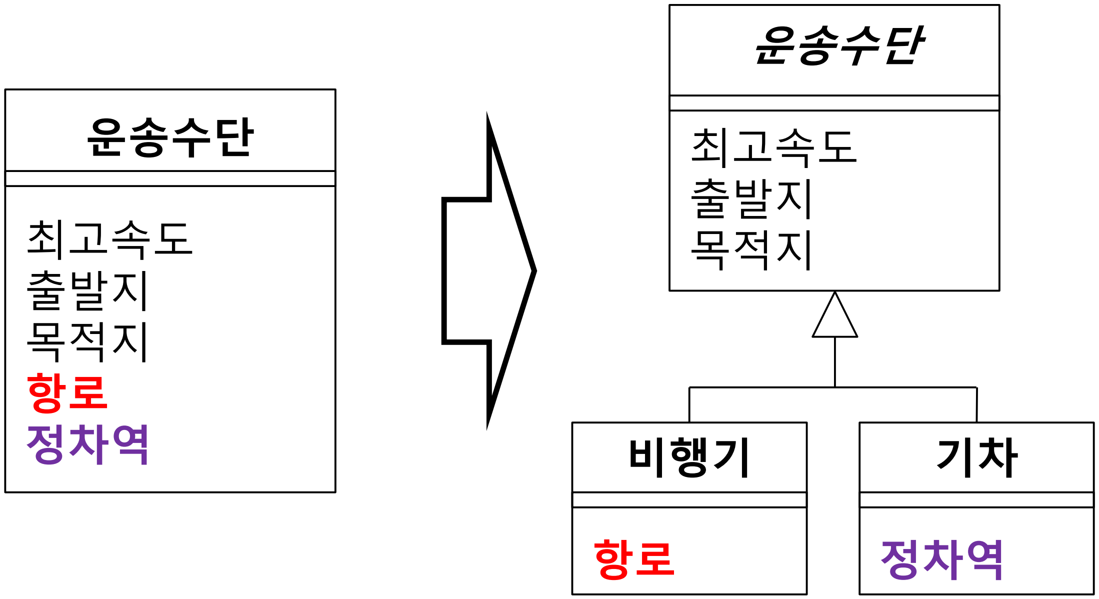
number

getPhoneNumber()

- officePhone

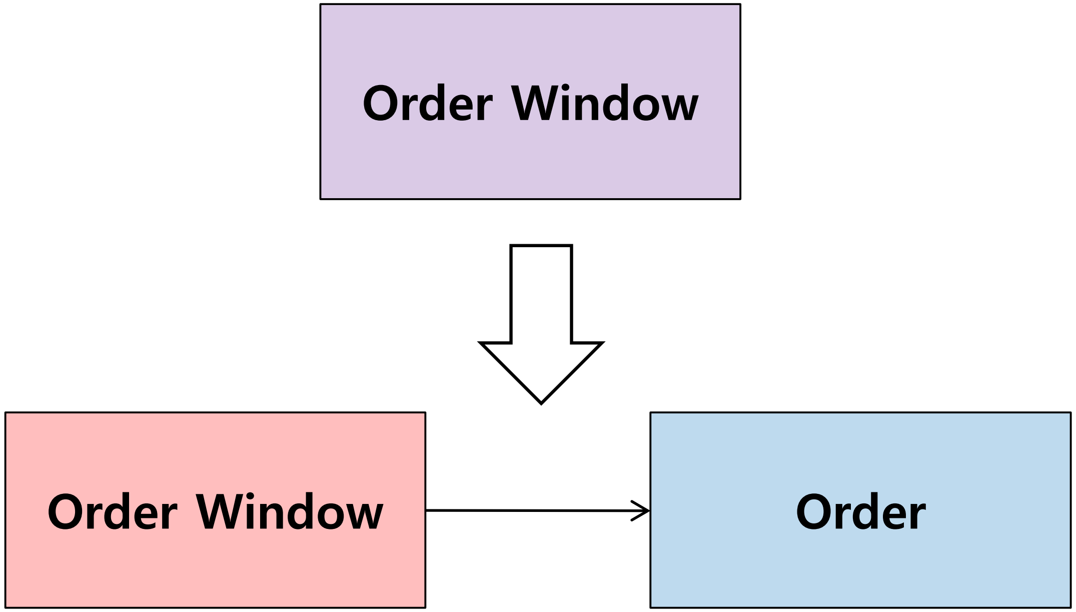
Extract Subclass

- A class has features that are used only in some instances
- Create a subclass for that subset of features



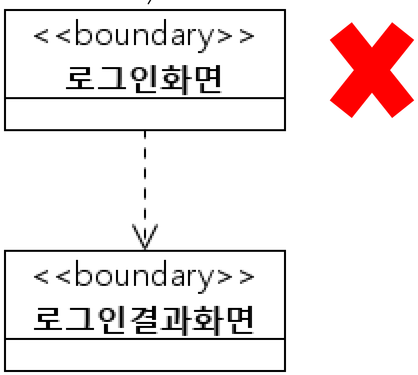
Separate Domain From Presentation

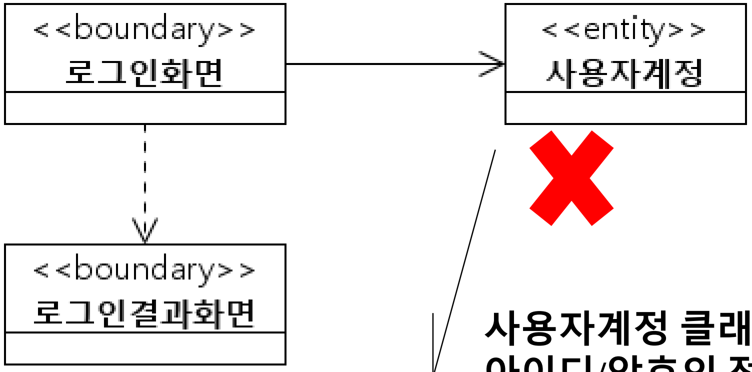
- **Divergent change**: You have GUI classes that contain domain logic
- Separate the domain logic into separate domain classes



Boundary, Control, Entity Pattern

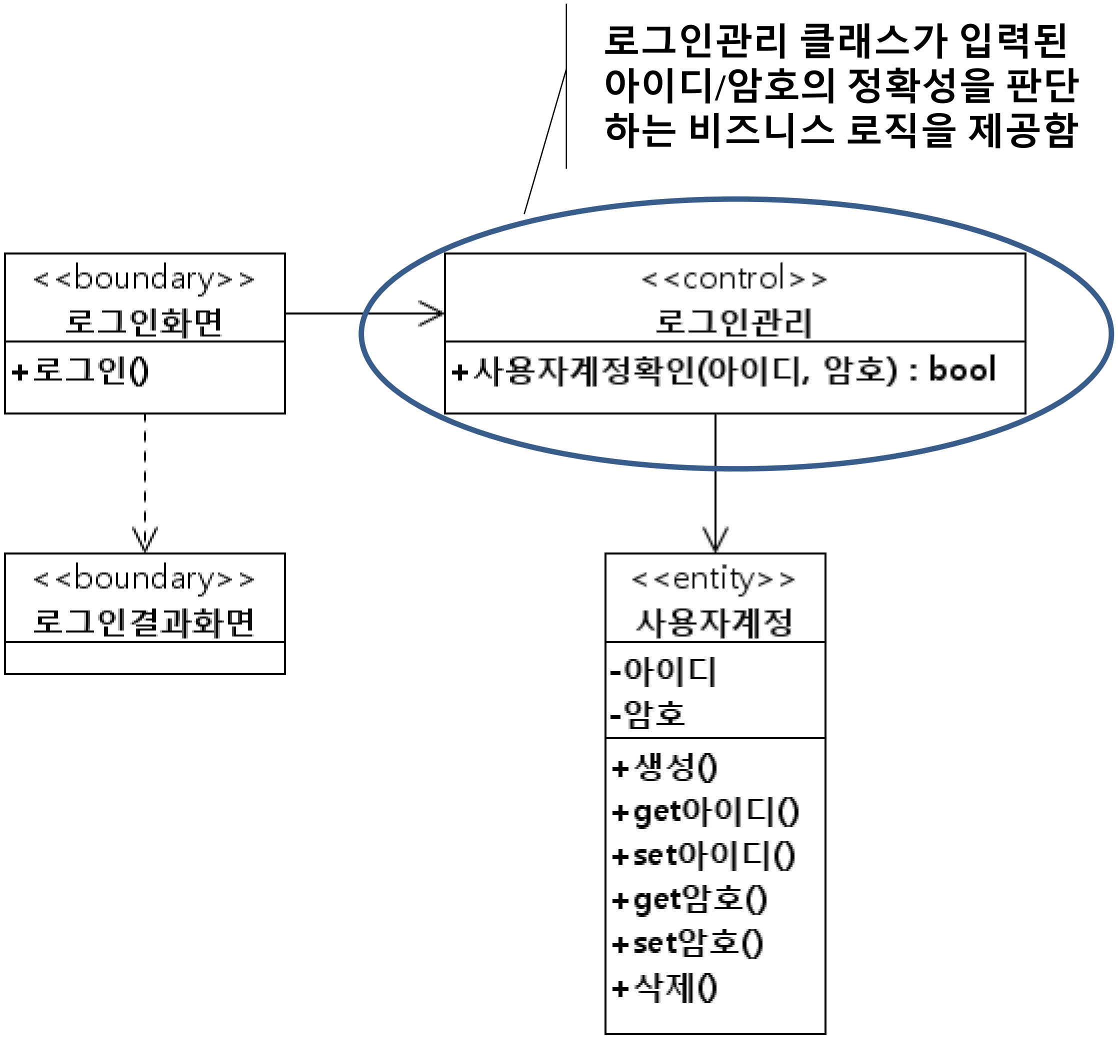
로그인화면 클래스가 입력된 아이디/암호의 정확성을 판단하는 비즈니스 로직을 제공하고 있음



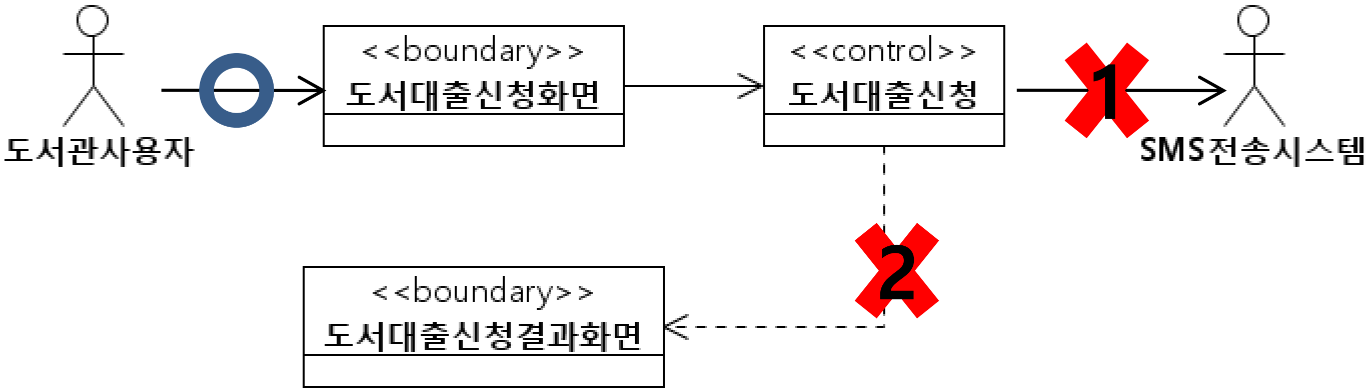


사용자계정 클래스가 입력된 아이디/암호의 정확성을 판단하는 비즈니스 로직을 제공하고 있음

Boundary, Control, Entity Pattern



Boundary, Control, Entity Pattern



도서관사용자

<<boundary>>
도서대출신청화면

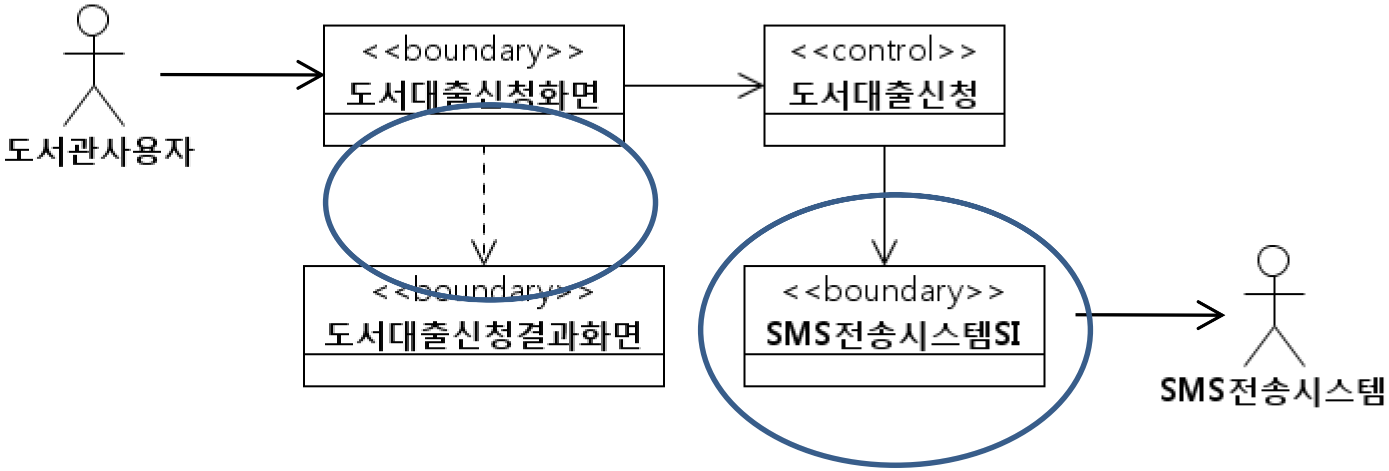
<<control>>
도서대출신청

1

SMS전송시스템

2

<<boundary>>
도서대출신청결과화면



도서관사용자

<<boundary>>
도서대출신청화면

<<control>>
도서대출신청

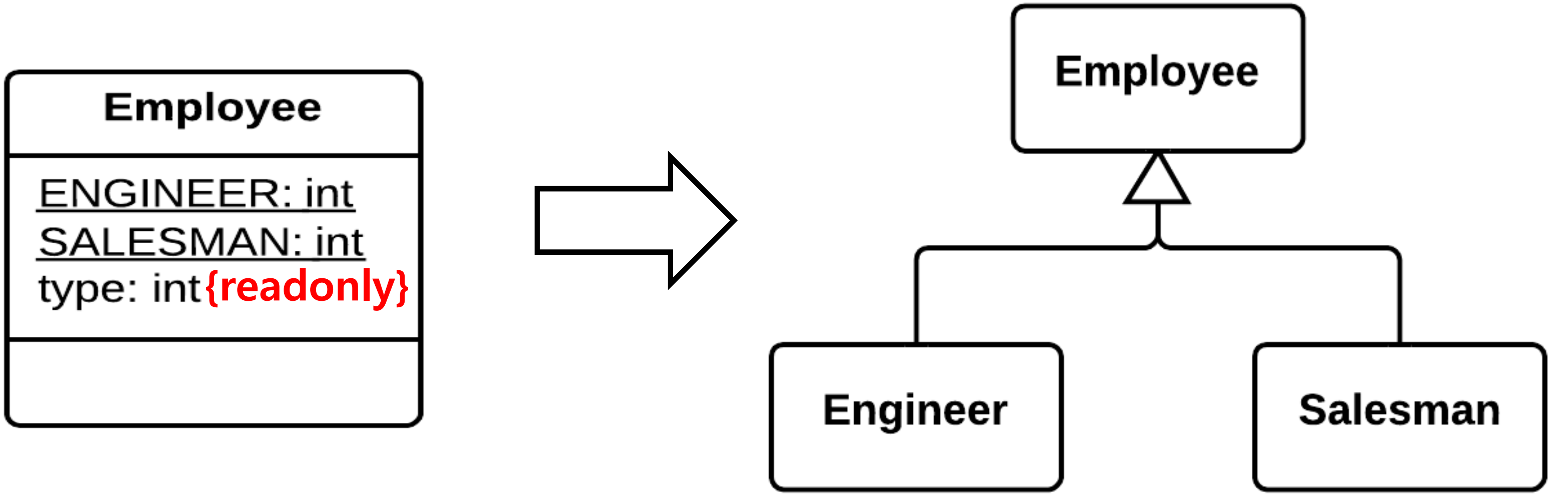
<<boundary>>
도서대출신청결과화면

<<boundary>>
SMS전송시스템SI

SMS전송시스템

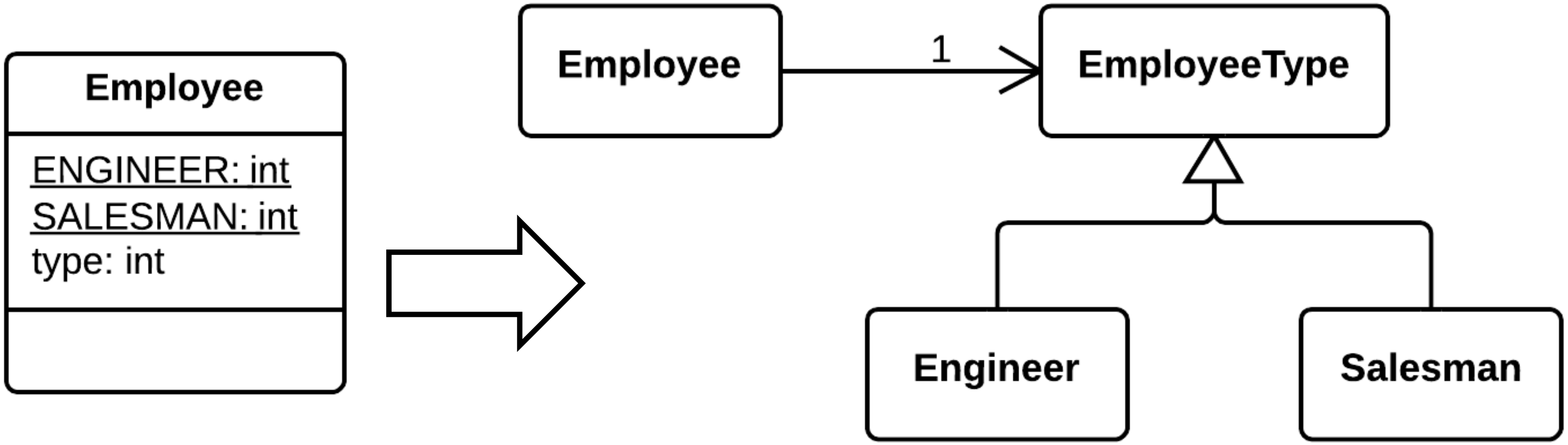
Replace Type Code with Subclasses

- You have a type code that affects the behavior of a class; but the **type code is immutable**
- Replace the type code with subclasses



Replace Type Code with State/Strategy

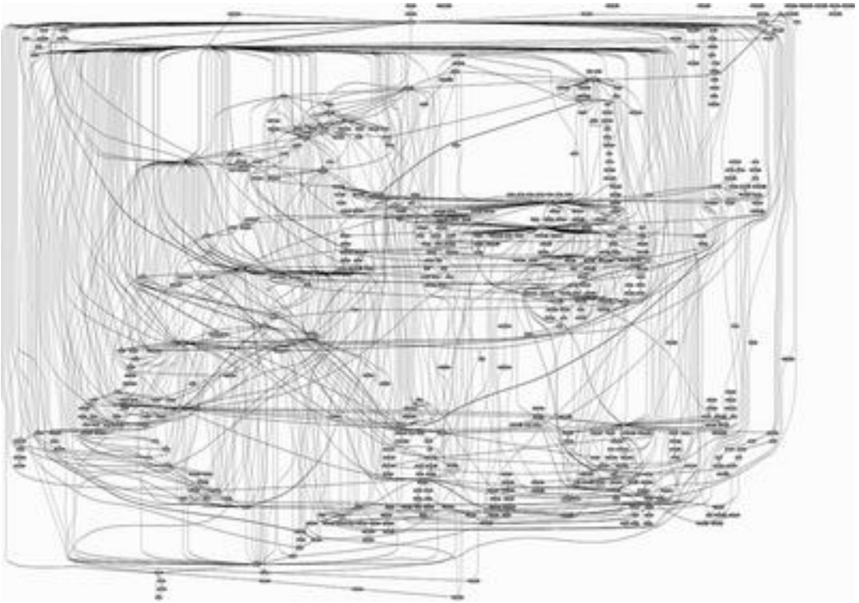
- ❖ You have a type code that affects behavior
  - ● you cannot use subclasses. or
  - ● mutable type code: the **values of type code can change** after the creation



- ❖ Strategy: to split a conditional that controls the selection of algorithms
- ❖ State: each value of the coded type is responsible not only for selecting an algorithm but for the whole condition of the class

Coupling

- Highly coupled systems are harder to understand and maintain.



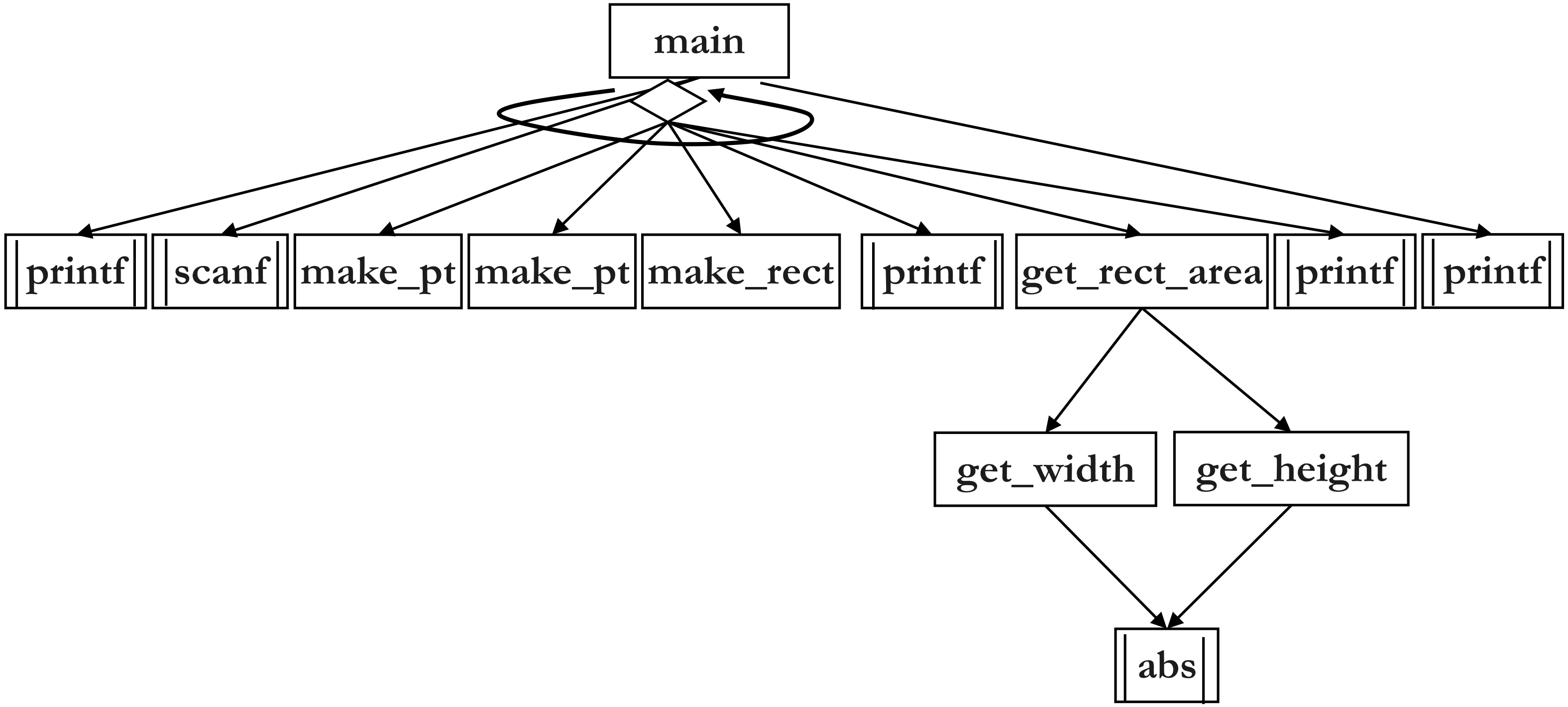
How to achieve low coupling

- Eliminate unnecessary relationships
- Minimize dependency on implementations(DIP)

- Coupling metrics
  - Fan-out, Fan-in
  - CBO, RFC

Coupling

- ❖ Fan out = the number of called modules
- ❖ Fan in = the number of calling modules



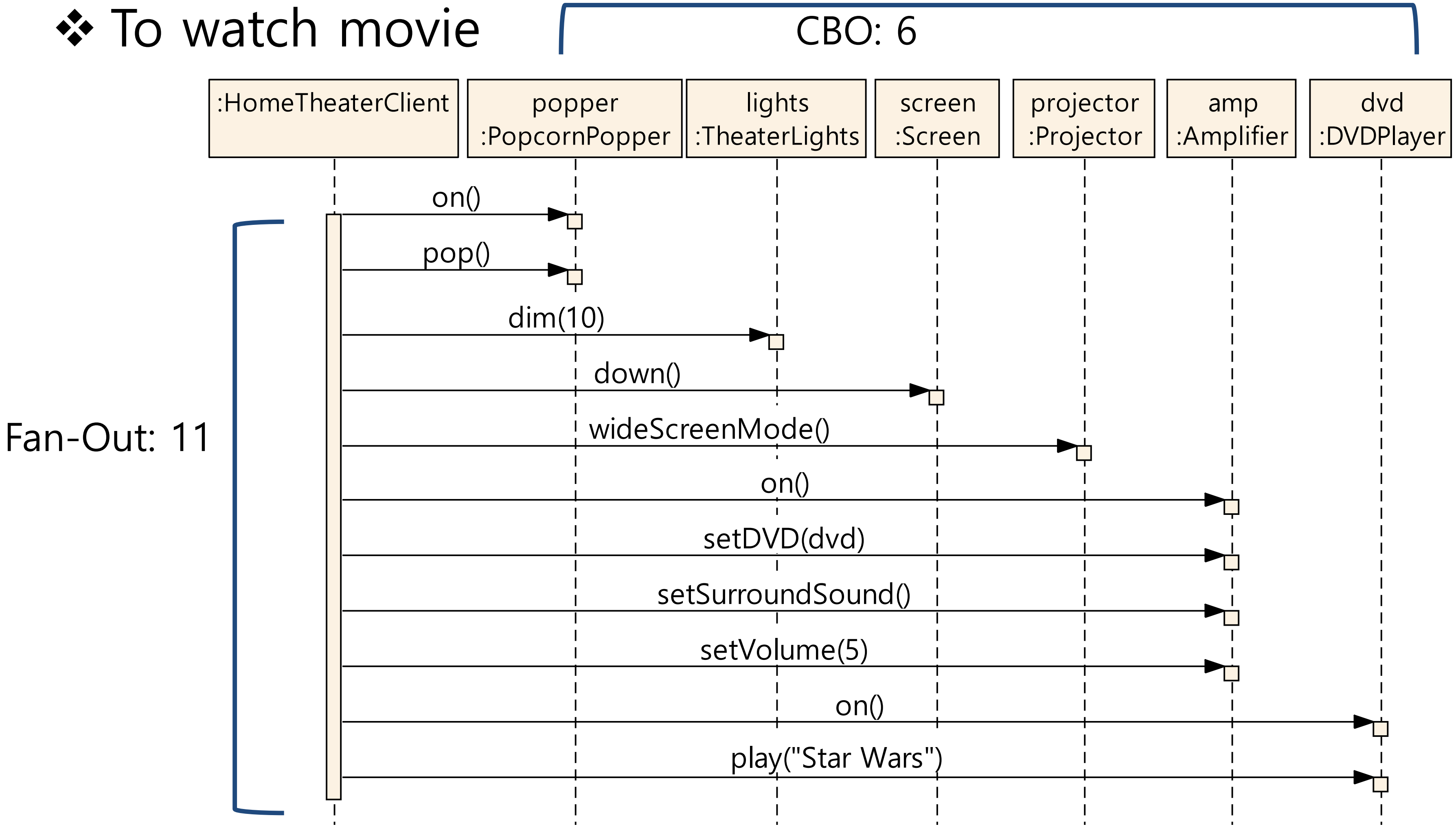
Coupling Metrics for Class

- RFC(Response For Class) measures coupling in terms of method calls
  - the number of methods in the class (not including inherited methods) +
  - the number of distinct method calls made by the methods in the class(Fan Out)

- CBO(Coupling Between Objects) measures coupling in terms of classes
  - the number of classes that a class referenced(Fan Out) +
  - the number of classes that referenced the class(Fan In)

Fan out in Sequence Diagram

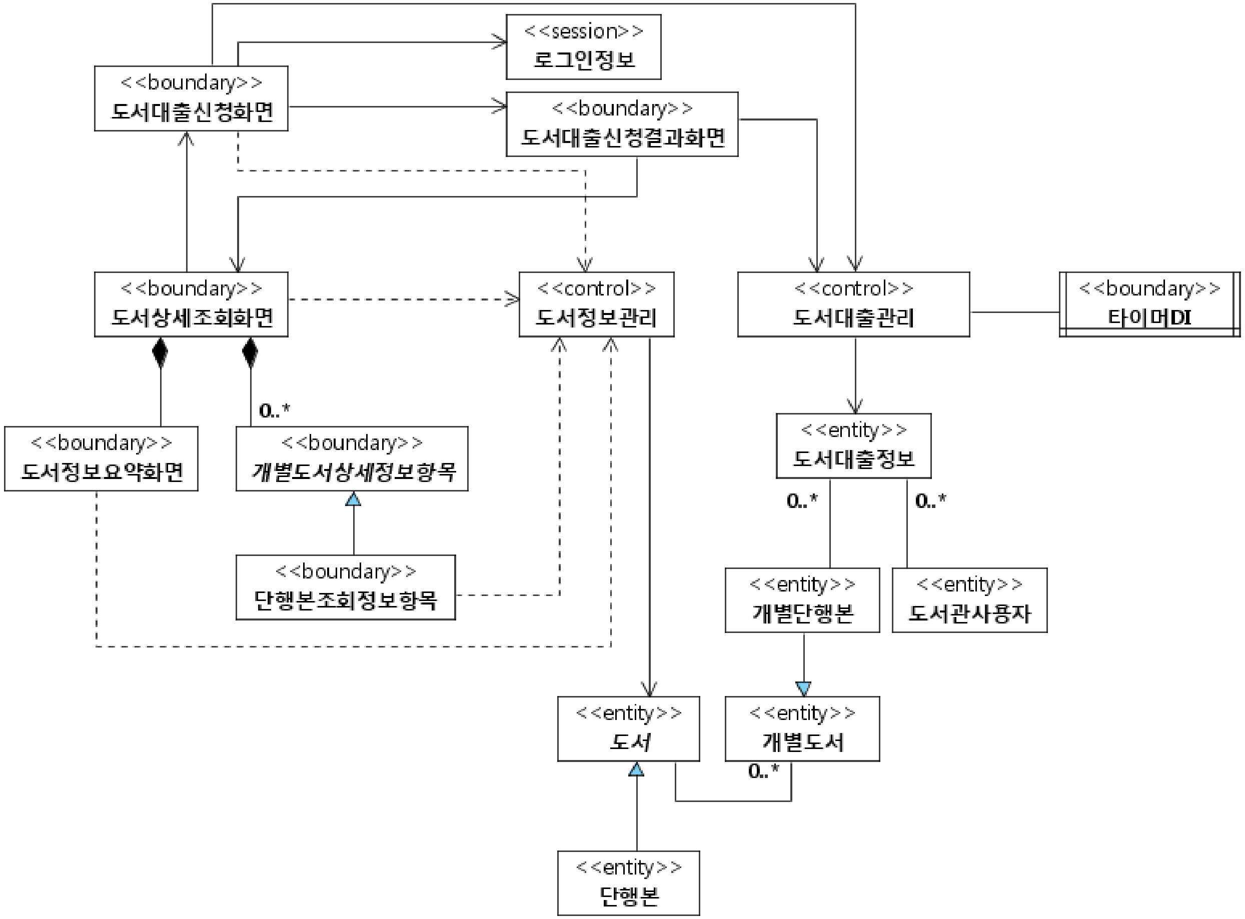
❖ To watch movie



CBO: Coupling between Object Classes

Coupling Metrics for Class

❖ CBO of 도서정보관리?



How to Reduce Coupling

- Refactorings
  - Replace Parameter with Method
  - Preserve Whole Object

  - Hide Delegate (Law of Demeter, TDA)
  - Encapsulate Collection

  - Introduce Façade Object
  - Introduce Façade Method

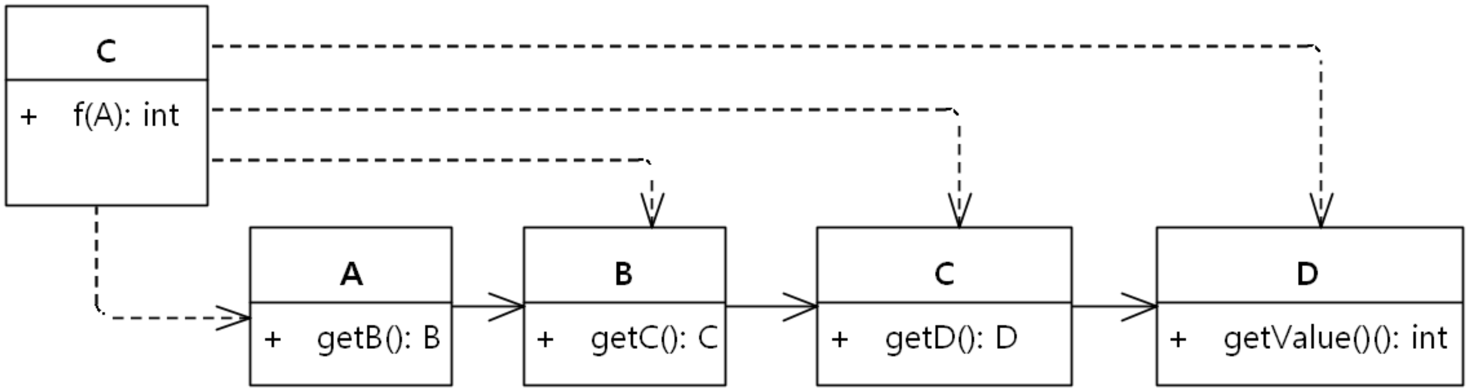
Hide Delegate

❖ TDA Principle: Tell, Don’t Ask

- Each unit should have only limited knowledge about other units: only units "closely" related to the current unit.
- Each unit should only talk to its friends; don't talk to strangers.
- Only talk to your immediate friends.

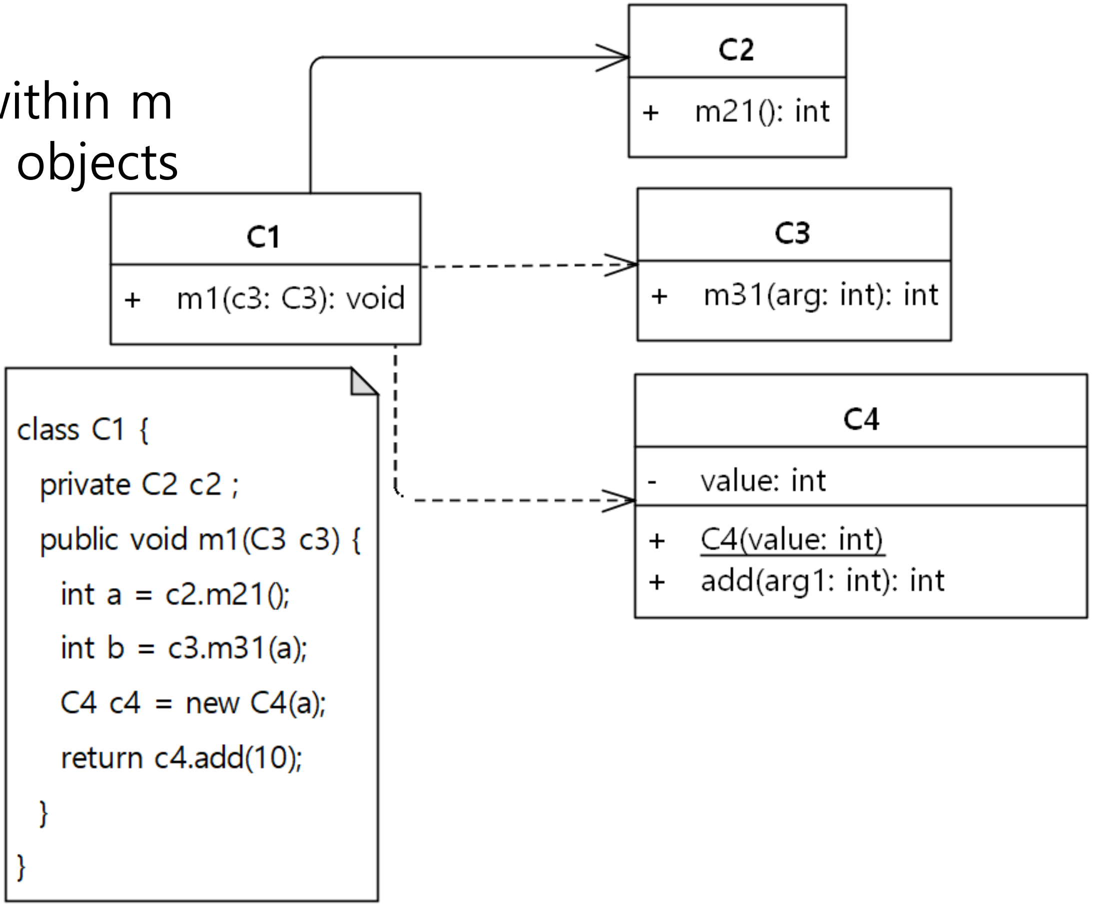
```
class C {
  public int f(A a) {
    return a.getB().getC().getD().getValue() ;
  }
}
```

“Message chain” smell



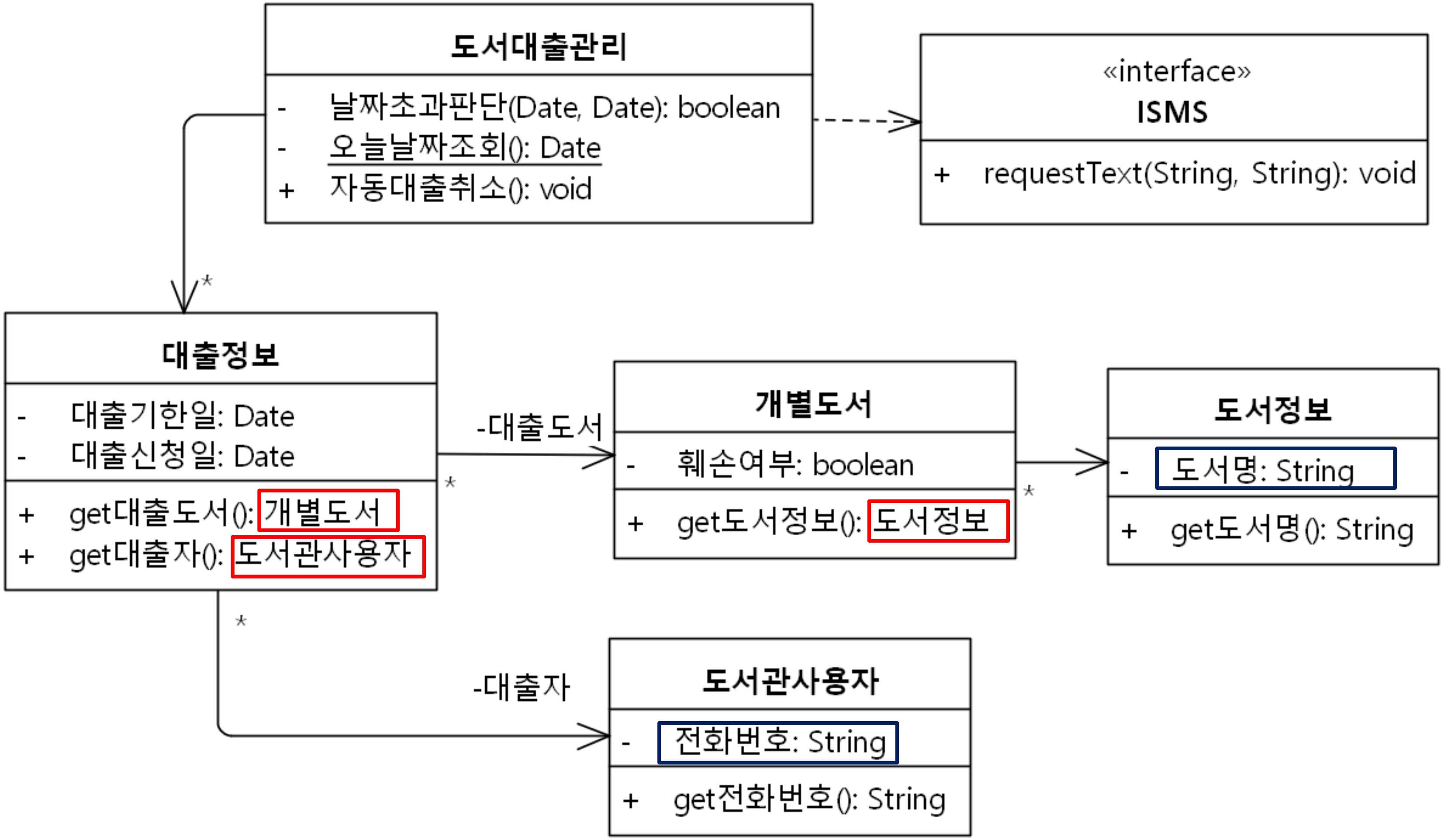
Law of Demeter

- Law of Demeter for functions requires that a method m of an object O may only invoke the methods of the following kinds of objects
  - m's parameters
  - Any objects created within m
  - O's direct component objects
  - A global variable



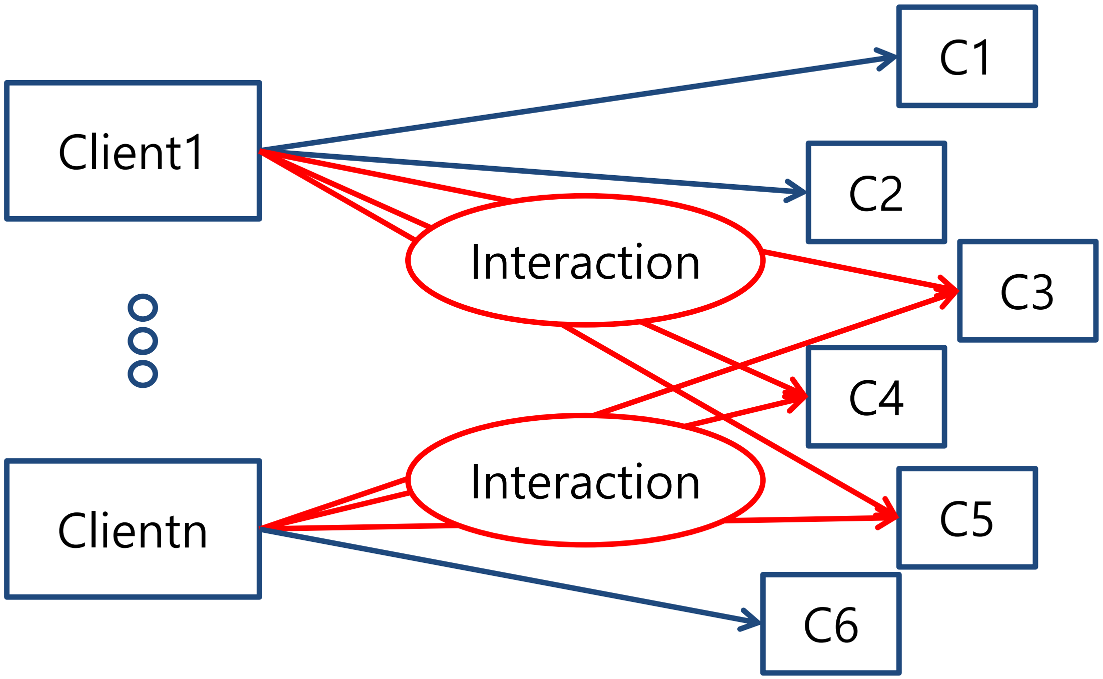
```
class C1 {
  private C2 c2 ;
  public void m1(C3 c3) {
    int a = c2.m21();
    int b = c3.m31(a);
    C4 c4 = new C4(a);
    return c4.add(10);
  }
}
```

Law of Demeter: Example



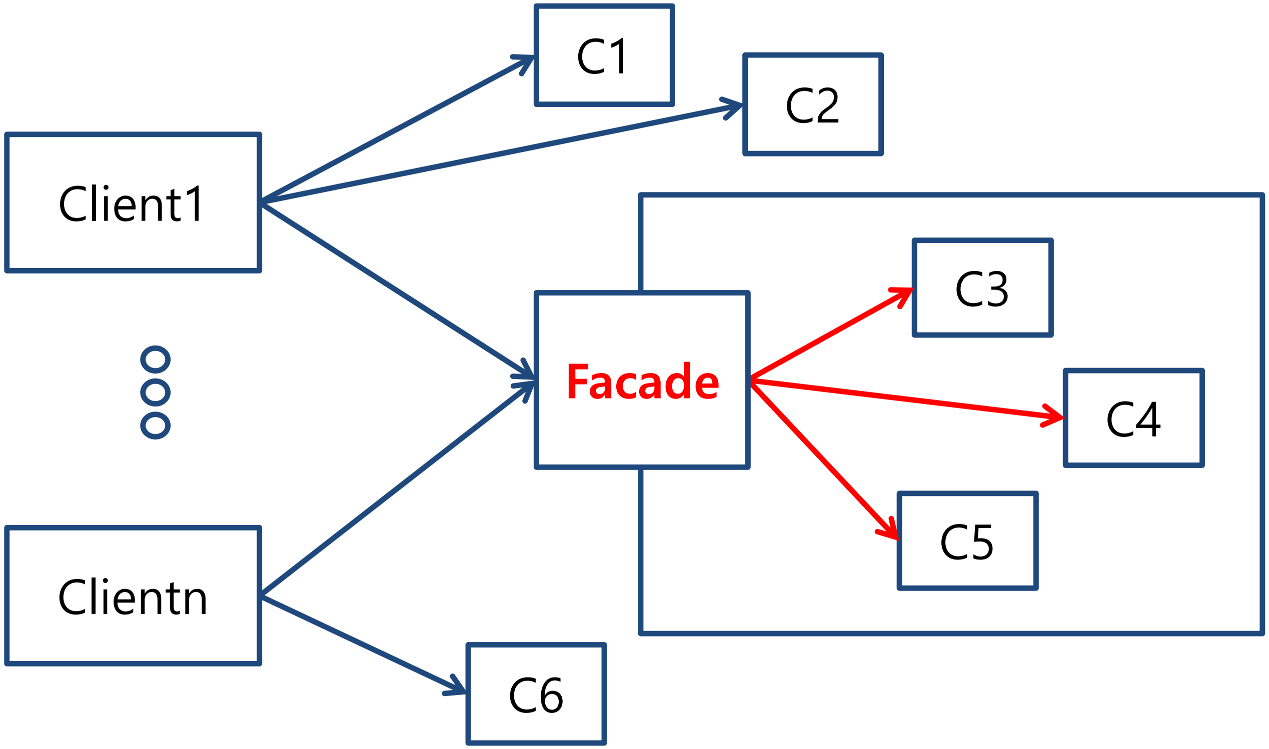
Introduce Façade Object

- Client has complex interaction with many components
- The code for interaction can be duplicated and will not be reused
- Thus, the interaction is not easy to extend



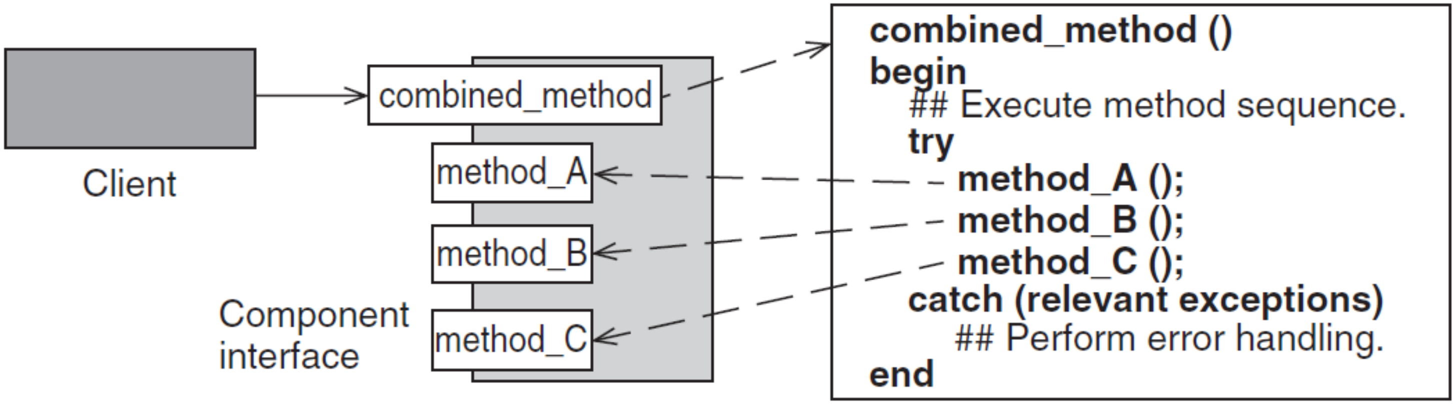
Introduce Façade Object

- Facade defines a higher-level interface that makes the subsystem easier to use.
- Encapsulate a complicated subsystem with a high-level interface.



Introduce Façade(Combined) Method

- Clients often must invoke multiple methods on a component in the same order to perform a specific task.
  - From a client's perspective, however, it is tedious and error-prone to call the method sequence explicitly each time it wants to execute the task on the component.
- Combine methods that must be, or commonly are, executed together on a component into a single(façade) method.



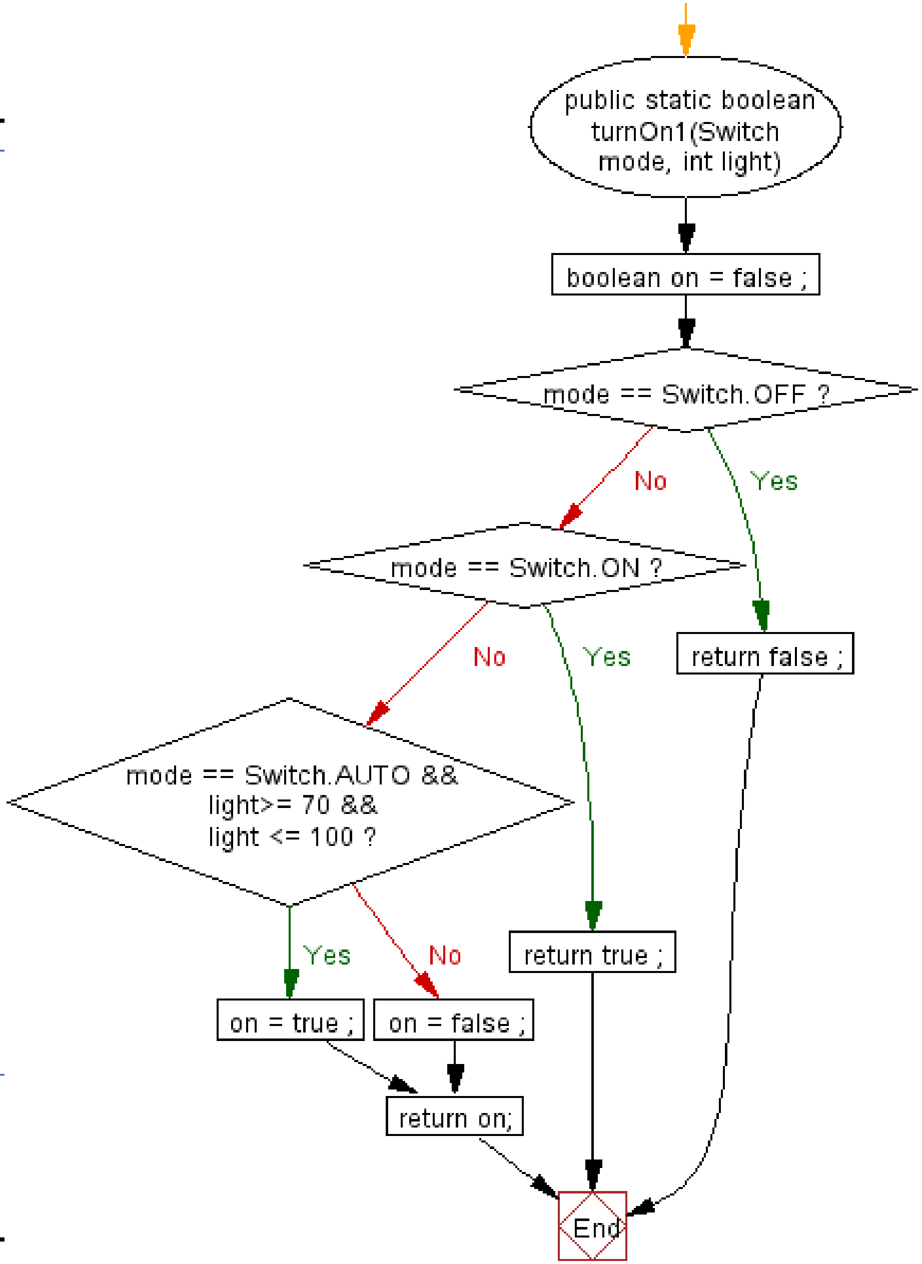
```
combined_method ()
begin
   ## Execute method sequence.
   try
      method_A ();
      method_B ();
      method_C ();
   catch (relevant exceptions)
      ## Perform error handling.
end
```

Pattern-oriented software architecture Vol. 4(2007)

COMPLEXITY

복잡도 - CC

```
public static boolean turnOn1(
  Switch mode, int light) {
  boolean on = false ;
  if ( mode == Switch.OFF )
    return false ;
  if ( mode == Switch.ON )
    return true ;
  if ( mode == Switch.AUTO &&
    light>= 70  && light <= 100 )
    on = false ;
  else
    on = true ;
  return on;
}
```



복잡도 - CC

❖ CC and Defect Risk

CC	Description	Risk
1-4	A simple procedure	Low
5-10	A well structured and stable procedure	Low
11-20	A more complex procedure	Moderate
21-50	A complex procedure, **alarming**	High
＞50	An error-prone, extremely troublesome procedure	Very High

http://www.aivosto.com/project/help/pm-complexity.html

Nesting Depth

❖ Number of Structuring Levels

```
public void function1(int i) {
  // …
  if ( i >= 0 )
    // …
  else
    // …
}
```

Nesting Depth = 1

```
int function2(int x) {
  int y = 0 ;
  for ( int  i = 0 ; i < x ; i ++ ) {
    if ( i >= 10 )
      // …
    else
      // y = ..
  }
  if ( y >= 0 ) return y ;
  else return –y ;
}
```

Nesting Depth = 2

NPath

❖ Number of (Static) Execution Paths

```
public void function1(int i) {
  // …
  if ( i >= 0 )
    // …
  else
    // …
}
```

NPath = 2

```
int function2(int x) {
  int y = 0 ;
  for ( int  i = 0 ; i < x ; i ++ ) {
    if ( i >= 10 )
      // …
    else
      // y = ..
  }
  if ( y >= 0 ) return y ;
  else return –y ;
}
```

NPath = 3 * 2 = 6

산업체 표준

유형	메트릭	허용 최대값				
		근거				
		MISRA	SCR-G	JPL	JSF	HIS
크기	Method Lines of Code(LOC)	80	200	60	200	50
	Comment Frequency	50%	30%	-	-	-
복잡도	Cyclomatic Complexity(CC)	15	20	-	20	10
	Number of Execution Paths(NPath)	75	-	-	-	80
	Number of Structuring Levels	6	6	-	-	4
결합도/ 모듈화	Number of Parameters	-	8	6	6	5
	Fan In	-	8	-	-	5
	Fan Out	-	10	-	-	7
	Number of Calling Levels	8	-	-	-	4

* MISRA: MISRA Report 5, Software Metrics
* SCR-G: 무기체계 소프트웨어 개발 및 관리 매뉴얼, 소프트웨어 신뢰성/보안성 시험 절차
* JPL: JPL(Jet Propulsion Lab.) Coding Standard for the C
* JSF: Joint Strike Fighter Air Vehicle C++ Coding Standards
* HIS: HIS(Audi, BMW 등 5개 자동차 업체 그룹) Source Code Metrics

SOLID

Principles of OOD

Five Basic Principles Object-Oriented Design for Maintainable and Extensible System

- **S**ingle **R**esponsibility **P**rinciple
- **O**pen **C**losed **P**rinciple
- **L**iskov **S**ubstitution **P**rinciple
- **I**nterface **S**egregation **P**rinciple
- **D**ependency **I**nversion **P**rinciple

By Robert Martin

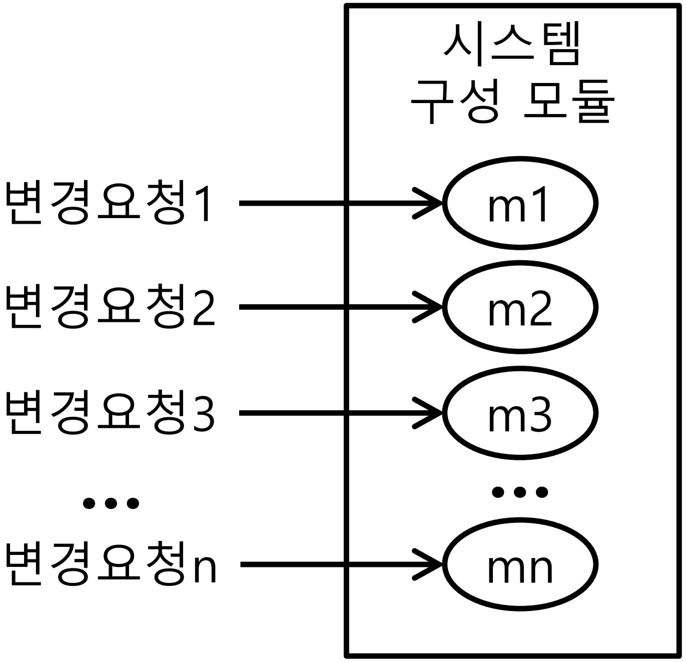
Single Responsibility Principle

There should never be more than **one reason** for a class to **change**.

A class should have **only one reason to change**

SRP(Single Responsibility Principle)

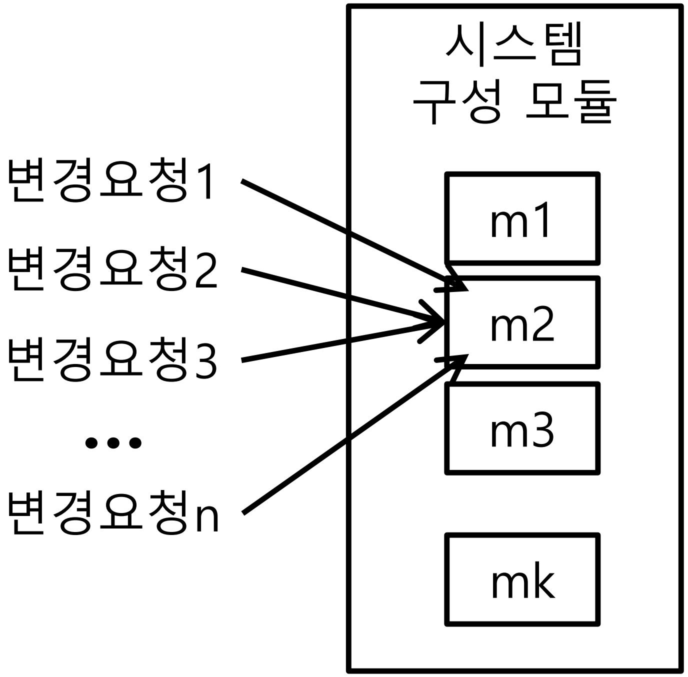
THERE SHOULD NEVER BE MORE THAN **ONE REASON** FOR A CLASS TO CHANGE.

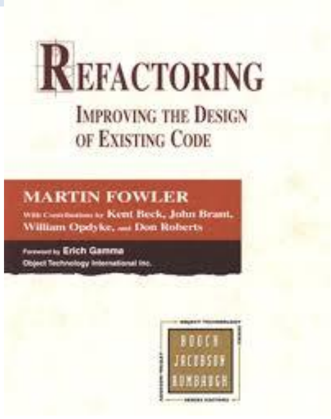


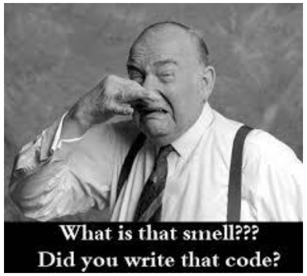
Bad Smell: Divergent Change

❖ Refactoring's definition:

Divergent change occurs when one class is commonly changed in different ways for different reasons.

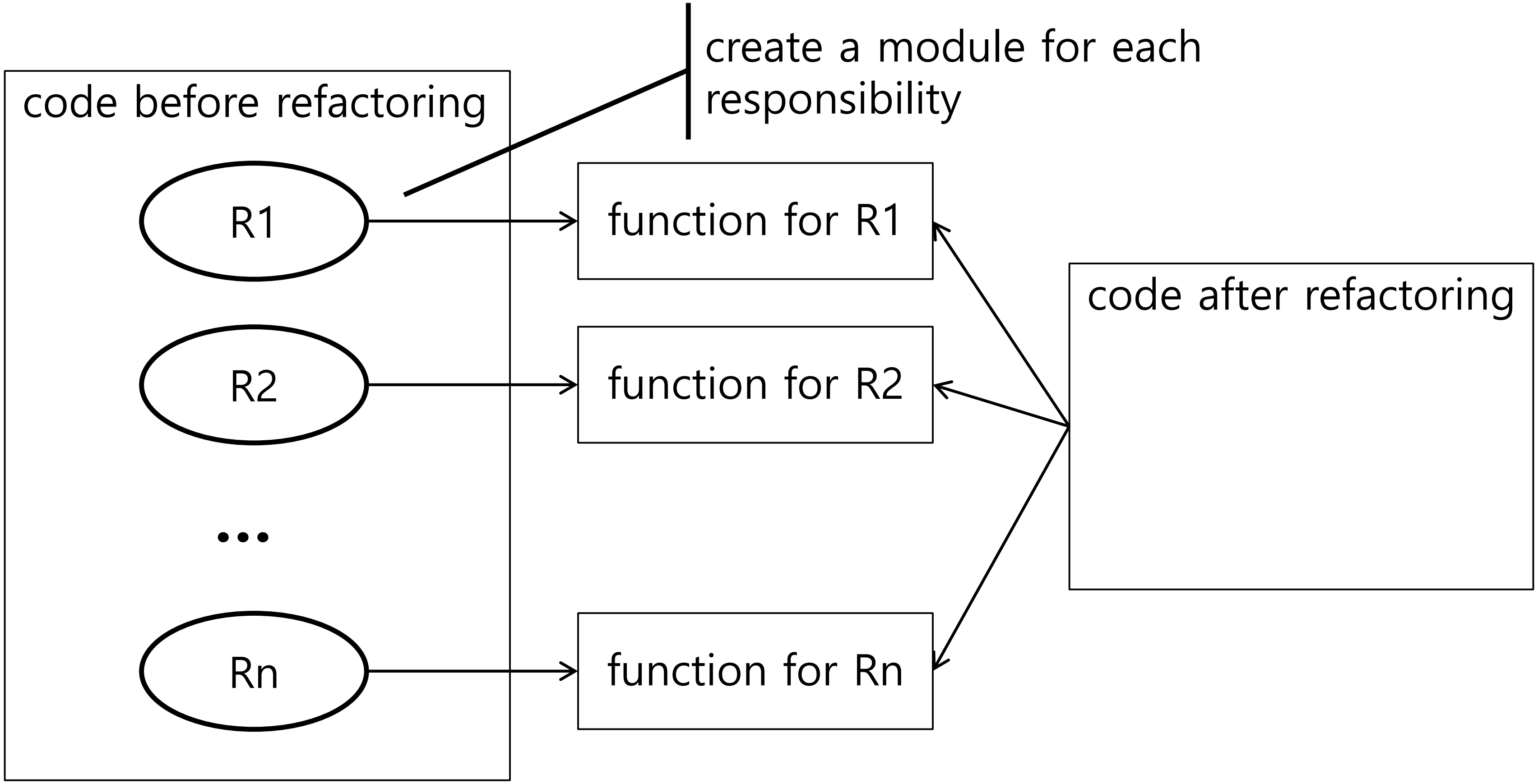



REFACTORING
IMPROVING THE DESIGN
OF EXISTING CODE
MARTIN FOWLER
With Contributions by Kent Beck, John Brant, William Opdyke, and Don Roberts
Foreword by Erich Gamma
Object Technology International Inc.
BOOCH
JACOBSON
RUMBAUGH



Refactoring for SRP

- ❖ Solution: Separate each responsibility into a function so that each of them conforms to SRP.



Open-Closed Principle - Definition

> Software Entities (Classes, Modules, Functions, etc.) should be open for extension, but closed for modification.
>
> by Bertrand Meyer
>
> Object-Oriented Software Construction. Prentice Hall, 1988

- **Open for Extension**
  - The behavior of the module can be extended.
  - We can make the module behave in new and different ways as the requirements change, or to meet new requirements.
- **Closed for Modification**
  - The source code of a module is inviolate.
  - No one is allowed to make source code changes to it.

- ➔ We can extend the behavior of a module by adding new code, not by changing its code.

OCP – 예제 코드

```
int getSum(const int values[], const int size) {
    int sum = 0;
    for (unsigned int i = 0; i < size; i++)
        if (values[i] > 0 )
          sum += values[i];
    return sum;
}
```

```
int getSum(const int values[], const int size, bool(*select)(const int) ) {
    int sum = 0;
    for (unsigned int i = 0; i < size; i++)
        if ( select(values[i]) )
            sum += values[i];
    return sum;
}
```

OCP – Static Binding Approach

```
# include “select.h”

int getSum(const int values[], const int size) {
    int sum = 0;
    for (unsigned int i = 0; i < size; i++)
        if ( select(values[i]) )
            sum += values[i];
    return sum;
}
```

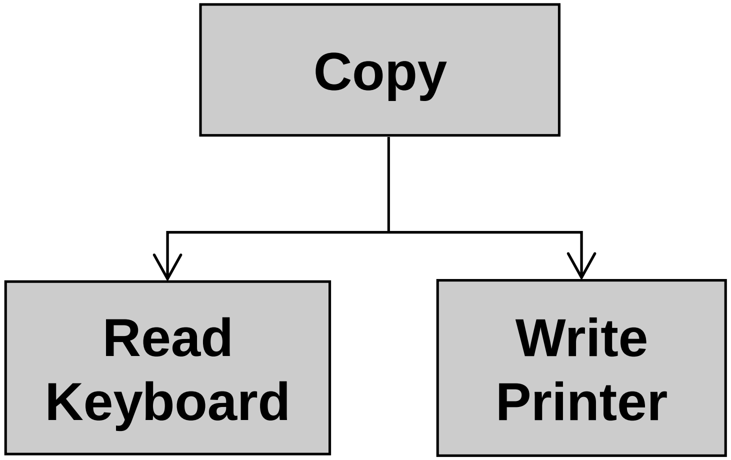
```
/* select_isPositive.c */
# include “select.h”

bool select(const int v) {
    return v >= 0;
}
```

```
/* select_isEven.c */
# include “select.h”

bool select(const int v) {
     return ( v % 2 ) ==  0 ;
}
```

Dependency Inversion Principle

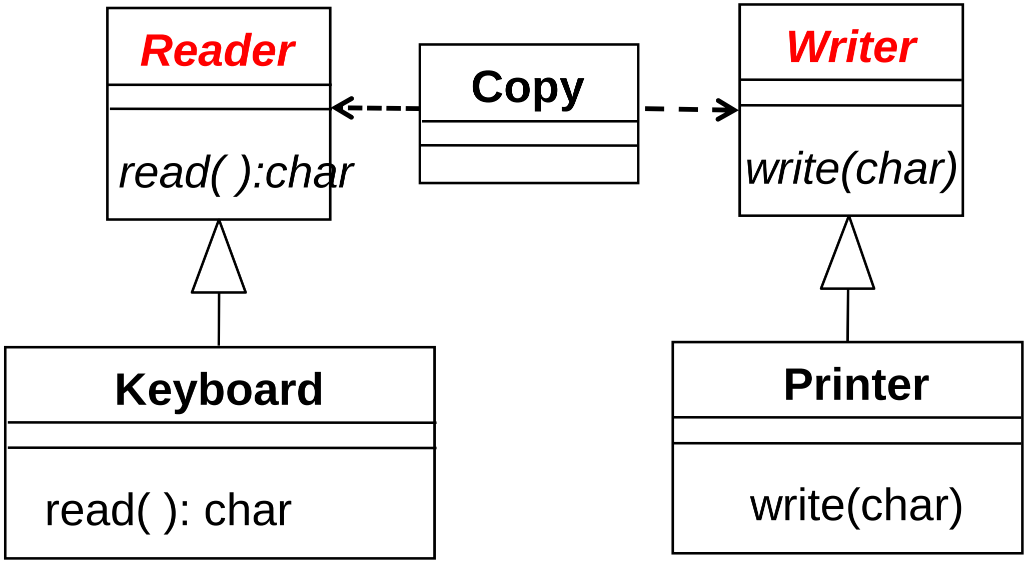
- ❖ Dependency is the key obstacle to maintaining software.
- ❖ Copy Program
  - ● copy characters typed on a keyboard to a printer

```
void Copy() {
  int c;
  while ( ( c = ReadKeyboard() ) != EOF )
    WritePrinter(c);
}
```



Dependency Inversion Principle

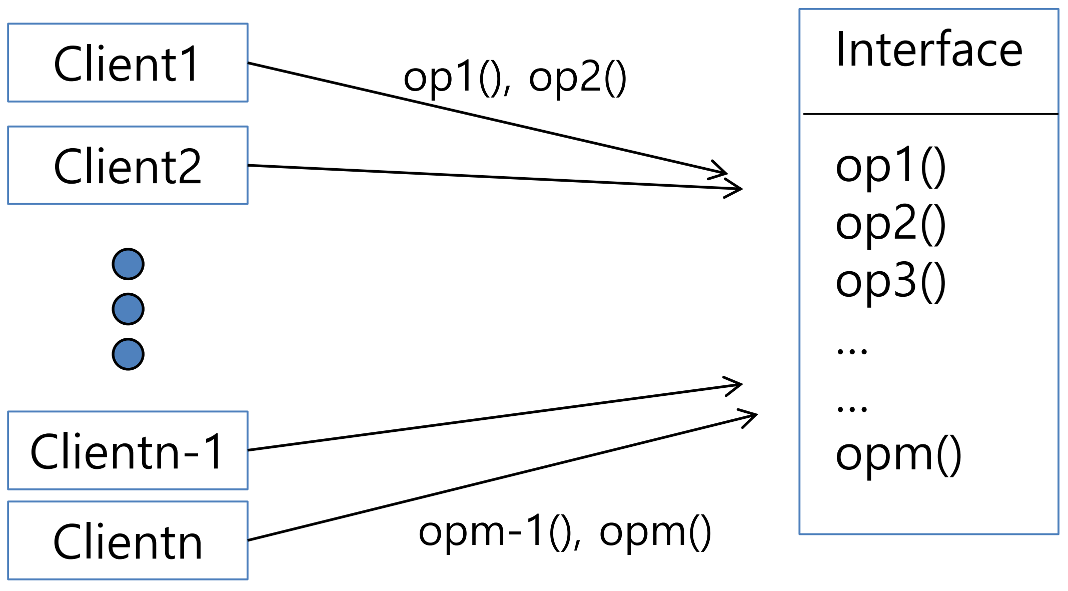
- Introduce an abstraction for communicating Copy and its low-level modules and communicate only with it.
- Now, Copy no longer depends on the details that it controls.



```
void Copy (
   char(*Reader_read)(),
   void(*Writer_write)(char) )
{
   int c;
   while ((c = Reader_read()) != EOF)
              Writer_write(c);
}
```

ISP: Interface Segregation Principle

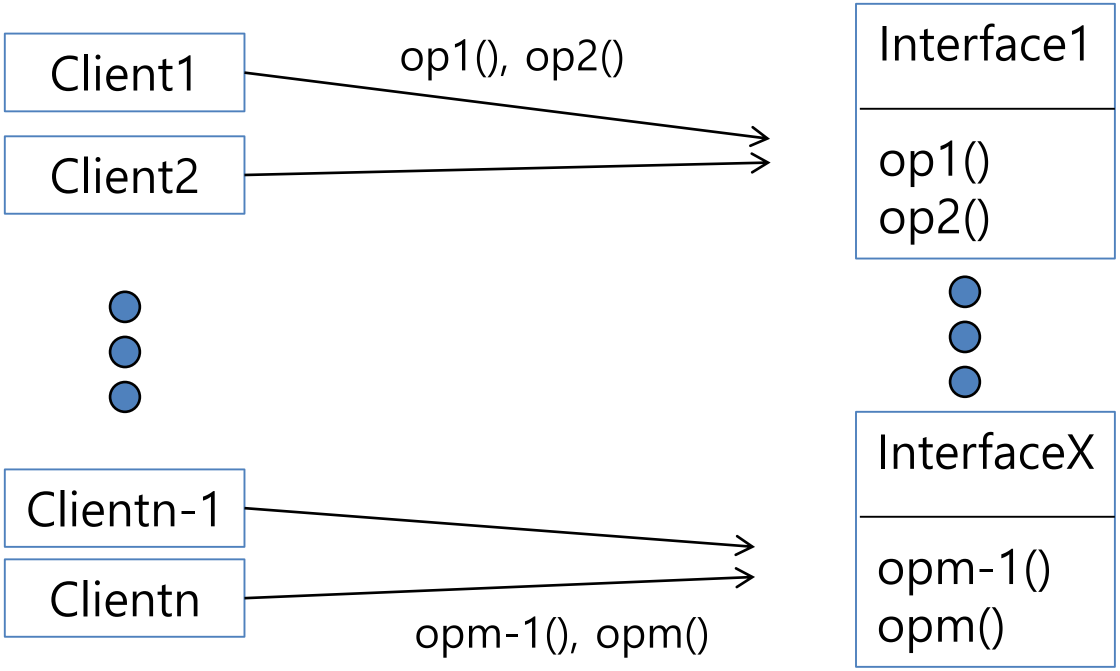
- Fat interface or polluted interface or non-cohesive interface
- Part of the interface is exclusively used by different clients.



ISP(Interface Segregation Principle)

❖ Fat interface should be partitioned into several small interface of high cohesion.

Clients should not be forced to depend upon interfaces that they do not use.



Liskov Substitution Principle(LSP)

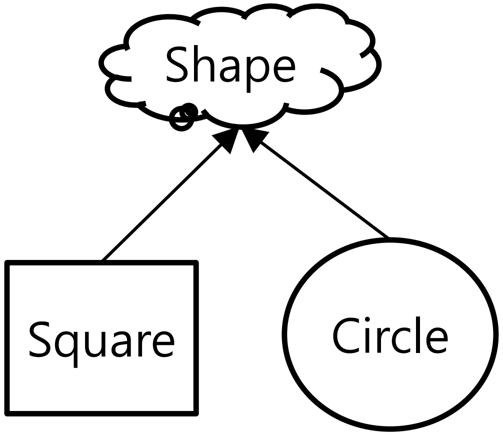
- ❖ Functions that use pointers or references to base classes must be able to use objects of derived classes **without knowing it**.
- ❖ Derived classes must be usable through the base class interface **without the need for the user to know the difference**.

Barbara Liskov, 1987

Liskov Substitution Principle(LSP)

- ❖ drawShape() must know every subclass of Shape. ➔ violates LSP.

```
void drawShape(Shape& s) {
    if (typeid(s) == typeid(Square))
        dynamic_cast<Square&>(s)->drawSquare() ;
    else if (typeid(s) == typeid(Circle))
        dynamic_cast<Circle&>(s)->drawCircle() ;
}
```

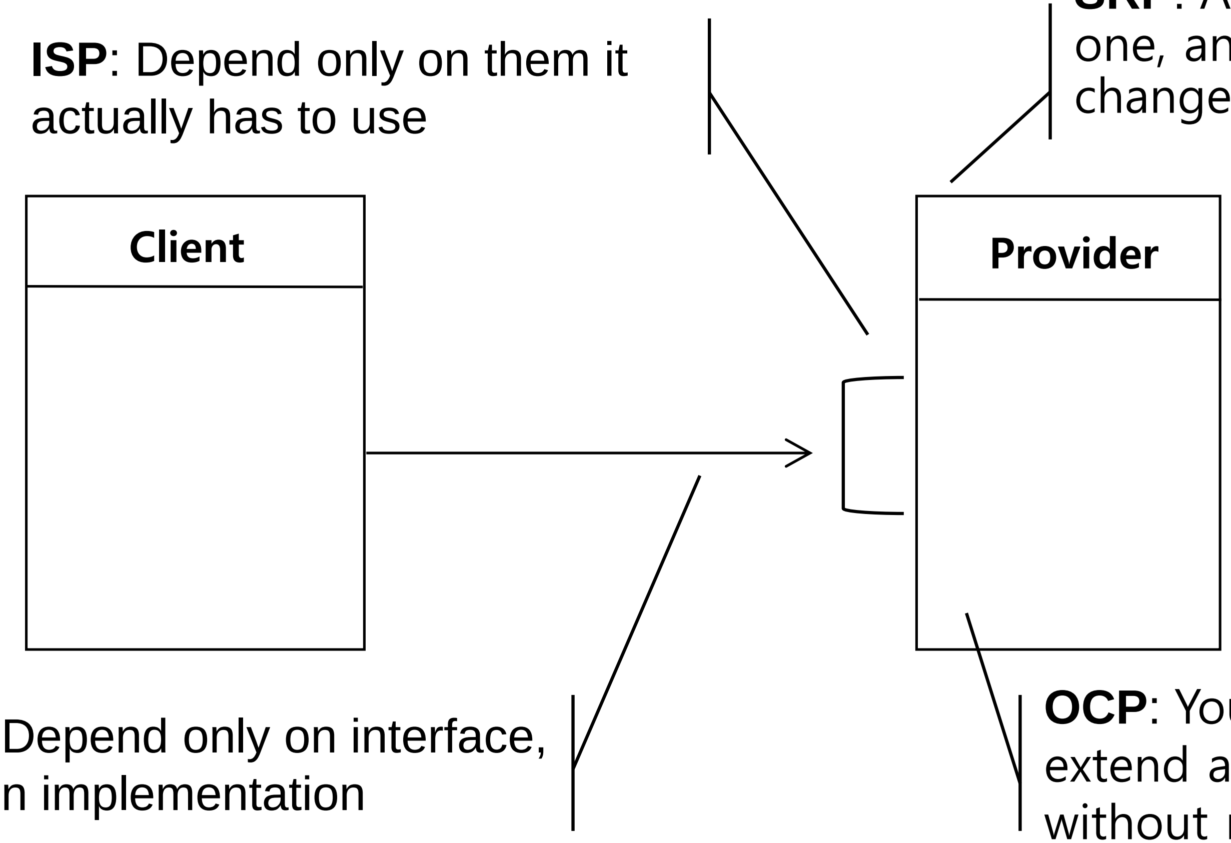


- ❖ In addition, drawShape() must be changed whenever new derivatives of the Shape class are added. ➔ violate OCP.
- ❖ Violation of LSP leads to violation of OCP.

SOLID - Summary

SRP	Single Responsibility Principle	A module should have one, and only one, reason to change.	Separate the module into multiple ones for each reason.
ISP	Interface Segregation Principle	Client should not be affected by the interface it does not use.	Make fine grained interfaces that are client specific.
OCP	Open Closed Principle	You should be able to extend a module behavior, without modifying it.	Provide extension points for any possible change.
LSP	Liskov Substitution Principle	Derived modules must be substitutable for their base classes.	Subclasses should conform to pre/post condition of its superclass
DIP	Dependency Inversion Principle	Do not depend on what are prone to change	Depend on interface, not on implementation.

SOLID - Summary



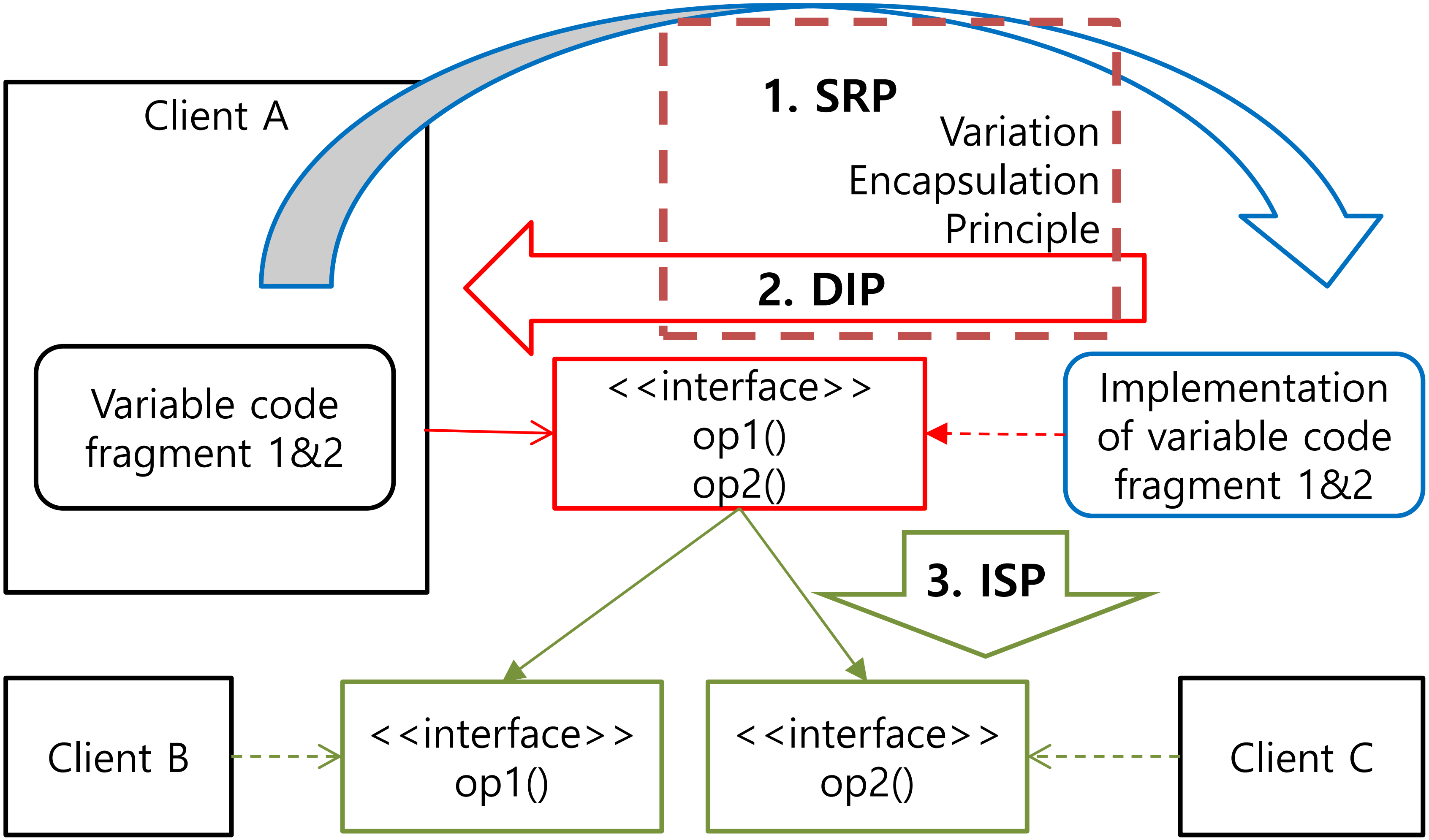
ISP: Depend only on them it actually has to use

SRP: A module should have one, and only one, reason to change.

DIP: Depend only on interface, not on implementation

OCP: You should be able to extend a module behavior, without modifying it.

Refactoring Procedure for OCP



Client A

Variable code fragment 1&2

1. SRP

Variation Encapsulation Principle

2. DIP

<<interface>>
op1()
op2()

Implementation of variable code fragment 1&2

3. ISP

Client B

<<interface>>
op1()

<<interface>>
op2()

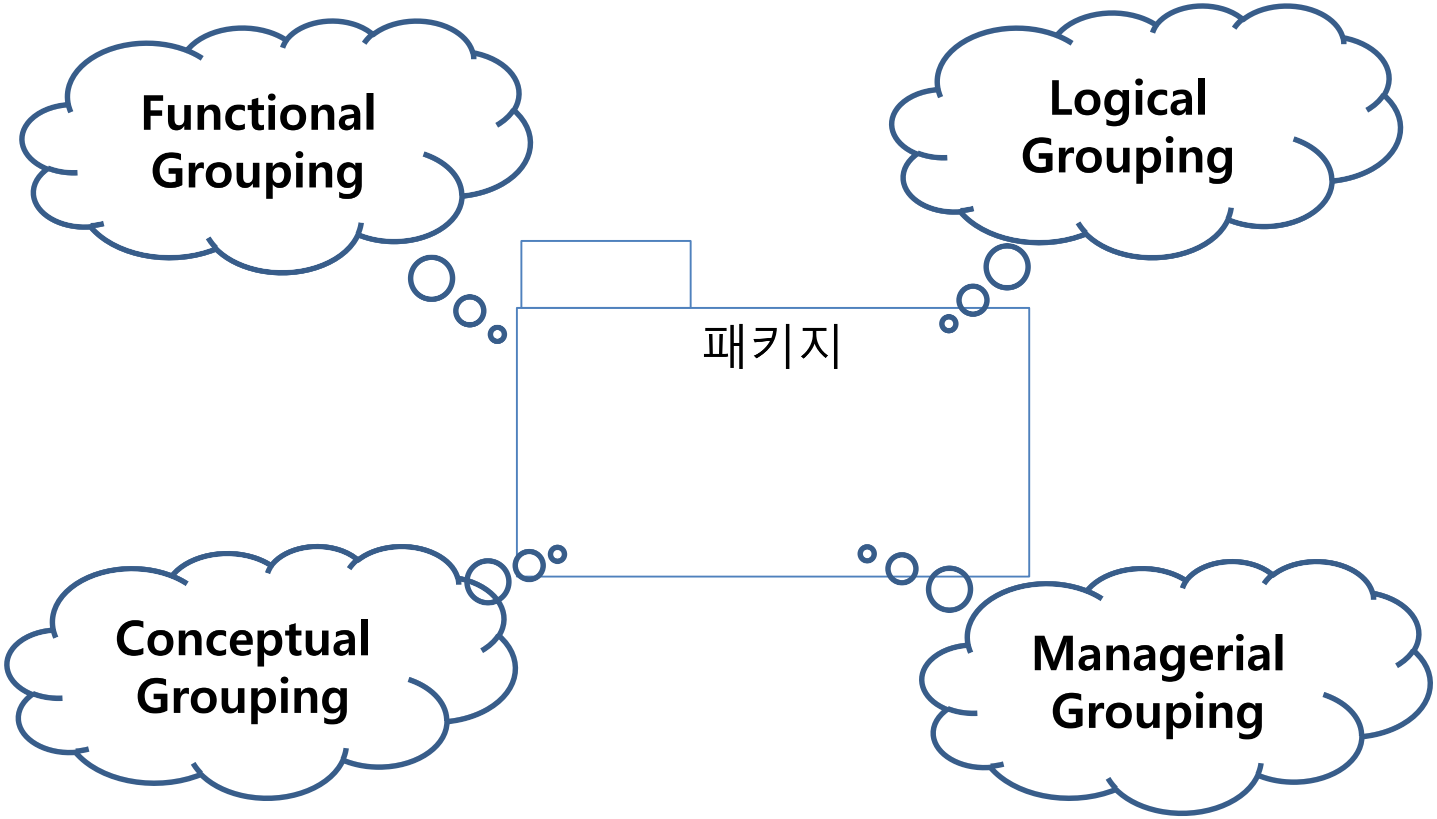
Client C

PACKAGE COHESION & COUPLING

Package

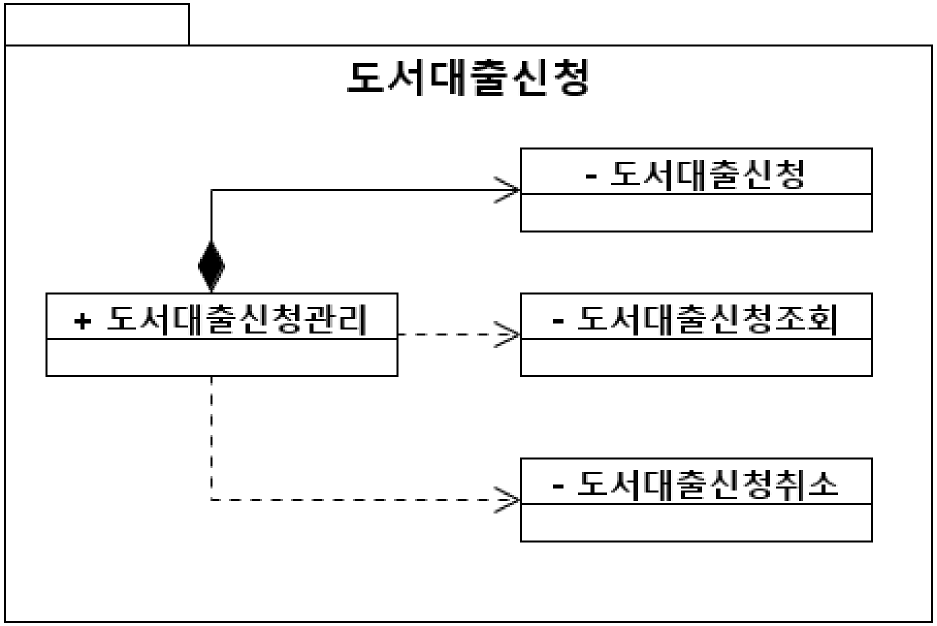
- ❖ Package is a namespace used to group together elements that are semantically related and might change together

https://www.uml-diagrams.org/package-diagrams.html



Package: Functional Grouping

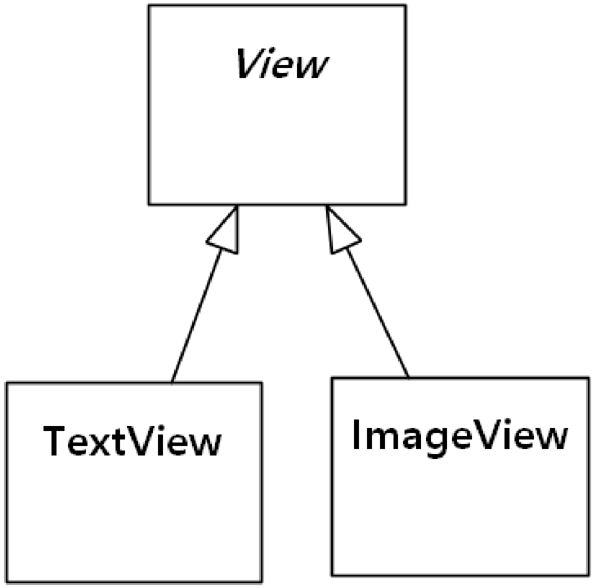
- ❖ Classes in a package are closely related to provide single functionality
- ❖ A façade class is declared as public, the others hidden from outside of the package
- ❖ An interface for the façade class can be defined



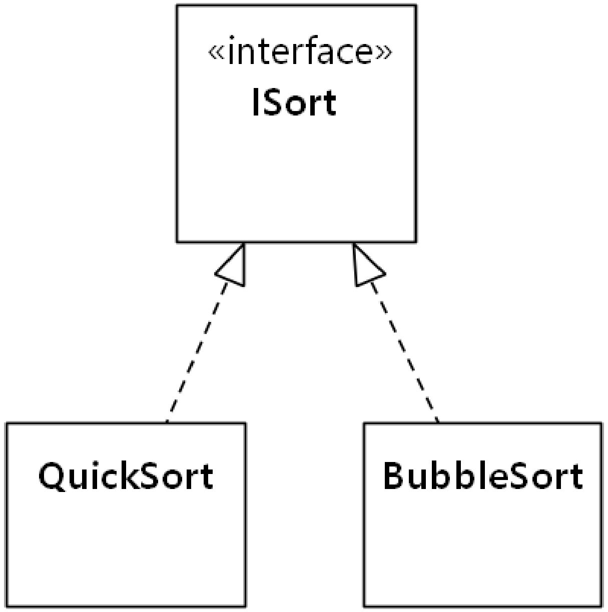
Package: Logical Grouping

❖ Logically related interfaces and classes are grouped into a package

Package for views



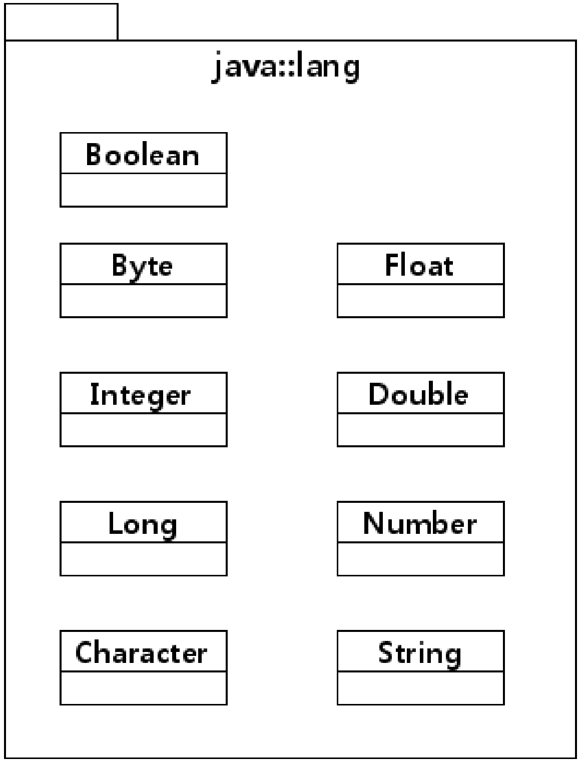
Package for soring algorithms



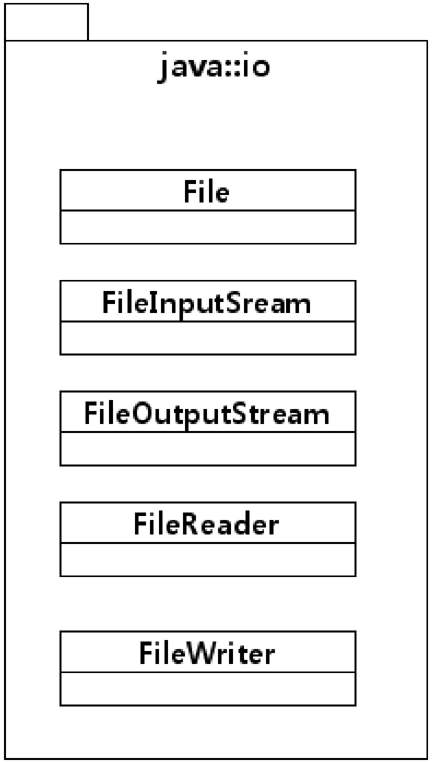
Package: Conceptual Grouping

- Classes in a package are conceptually related.
- For example, types and data structures are organized into individual package

Package for primitive types

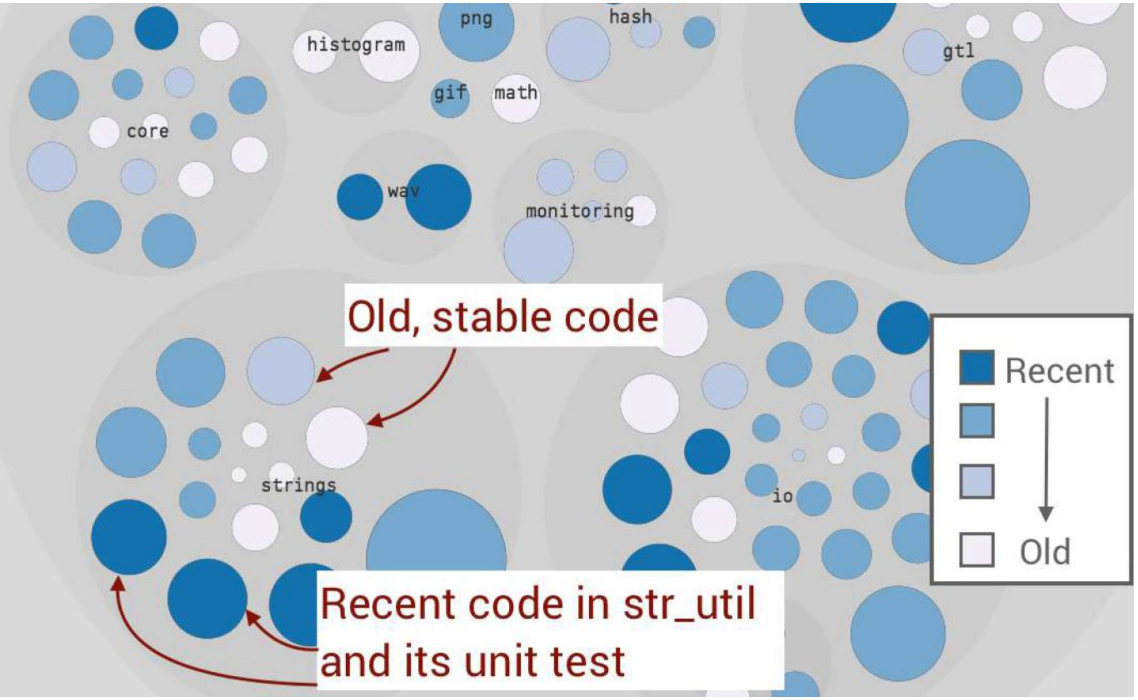


Package for IO Streams



Package: Co-change Grouping

- Classes in a package are changed together due to the same reason.
- Age-based package structure



Software design X-rays: Fix technical debt with behavioral code analysis, 2018

Package Cohesion Principles

❖ R. C. Martin’s cohesion principles on package

REP	Release Reuse Equivalency Principle	The granule of reuse is the granule of release.
CCP	Common Closure Principle	Classes that change together are packaged together.
CRP	Common Reuse Principle	Classes that are used together are packaged together.

Robert C. Martin, Clean architecture: A craftsman’s guide to software structure and design, 1st edition

REP: Reuse/Release Equivalence Principle

- We are now living in the age of software reuse: a huge number of reusable packages
- Reuse is fulfillment of one of the oldest promises of the object-oriented model
- REP means that classes and modules in a package should be releasable together.
  - They share the same version number and the same release tracking, and
  - are included under the same release documentation

Package is a unit of RELEASE

The granule of reuse is the granule of release.

CCP: Common Closure Principle

- For most applications, maintainability is more important than reusability

- Gather together in one place all the classes that are likely to change for the same reason

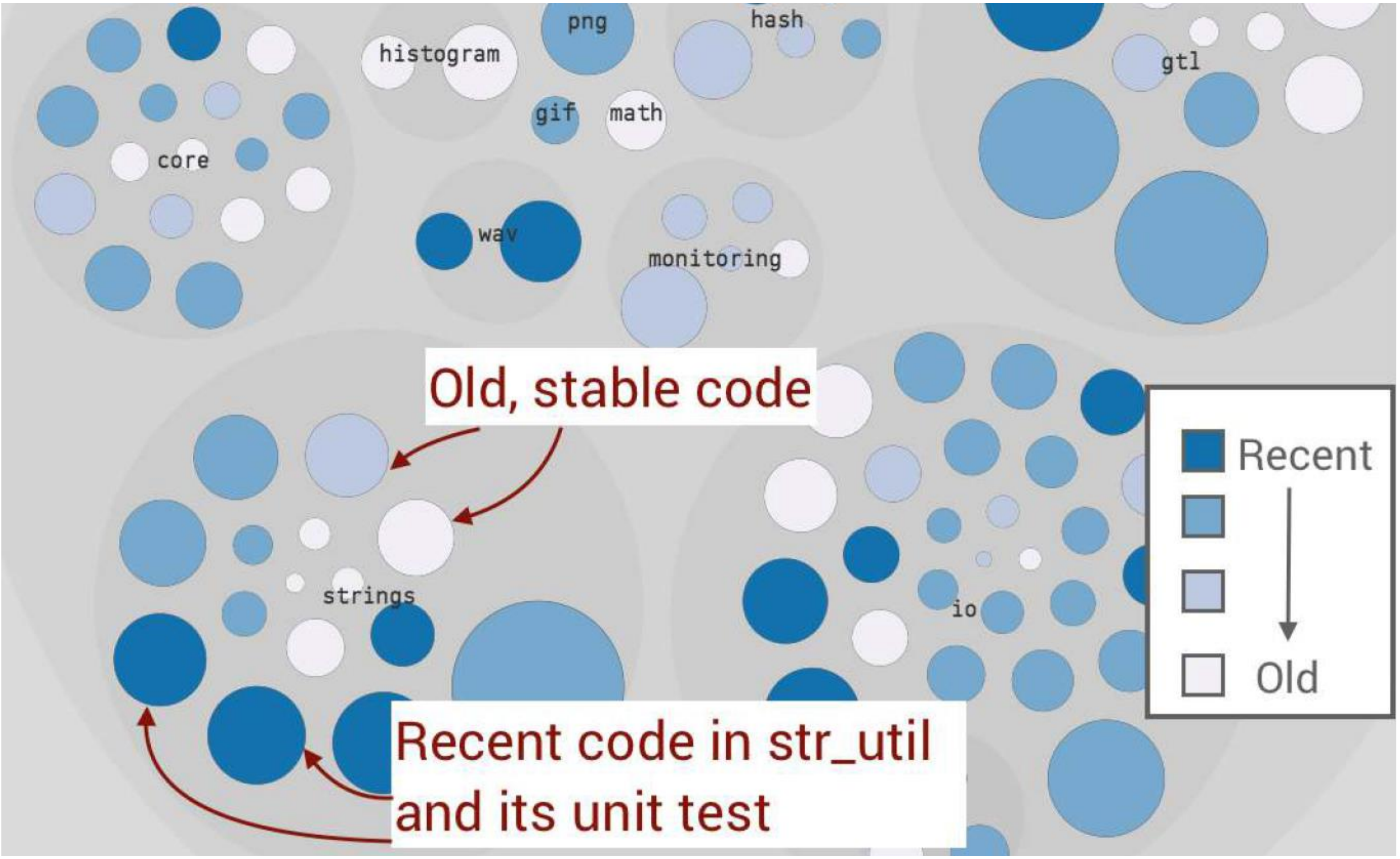
- Separate into different packages those classes that change at different times and for different reasons. ➔ package version of Single Responsibility Principle(SRP)

Package is a unit of CHANGE

- Age-oriented code organization
  - Software design X-Rays: Fix technical debt with behavioral code analysis, 2018

CCP: Common Closure Principle

- ❖ Age-oriented code organization



Old, stable code

Recent code in str_util and its unit test

CRP: Common Reuse Principle

- Classes are seldom reused in isolation.

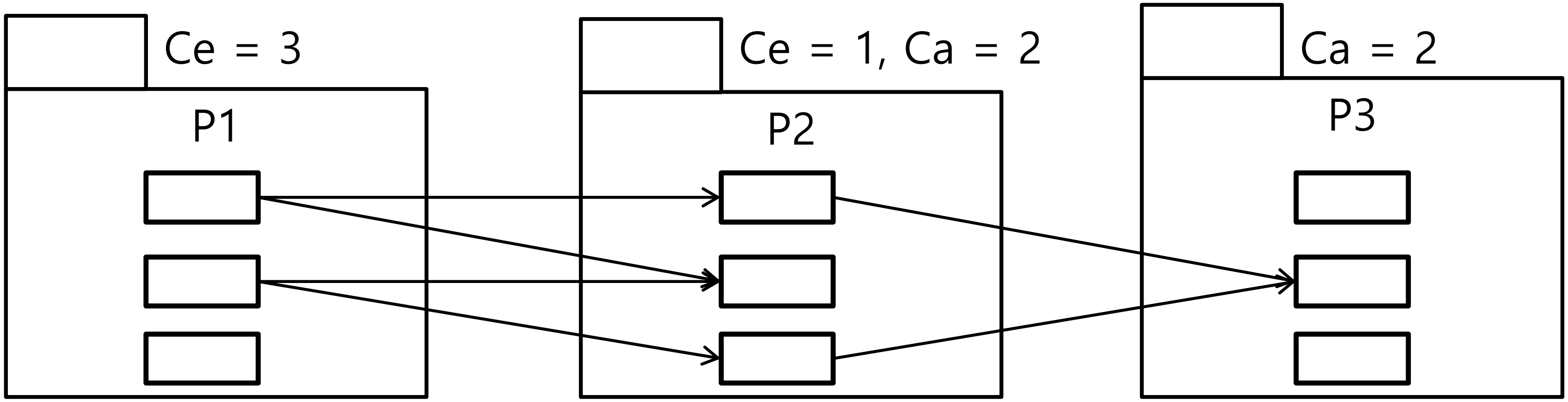
- Gather together in one package, classes and modules that tend to be reused together
- Thus when we depend on a package, we want to make sure we depend on every class in that package.

- Don’t depend on packages that have classes we don’t use ➔ package version of Interface Segregation Principle(ISP)

Package is a unit of REUSE

Package Coupling

- Efferent Couplings (Ce) – Fan out
  - The number of classes in other packages that the classes in the package depend upon
  - an indicator of the package's dependence on externalities
- Afferent Couplings (Ca) – Fan in
  - The number of classes in other packages that depend upon classes within the package
  - an indicator of the package's responsibility.



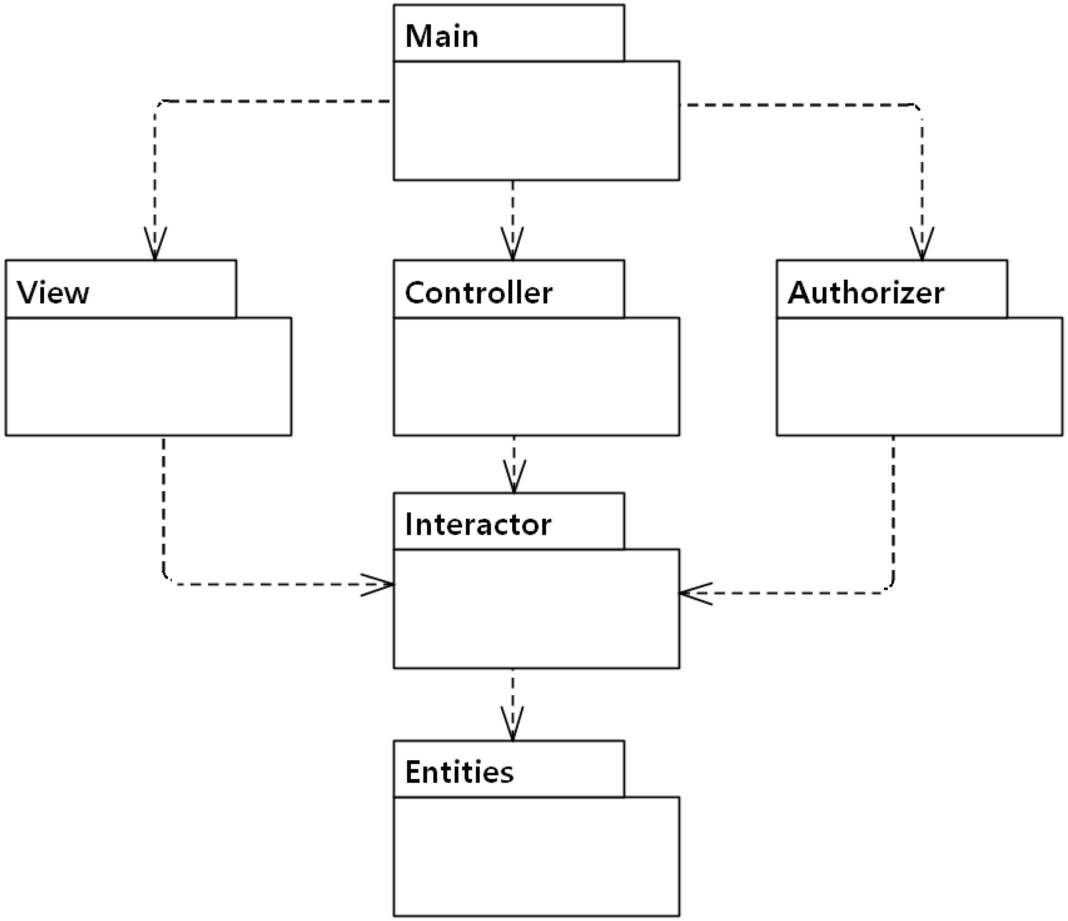
Package Coupling Principles

- R. C. Martin’s coupling principles on package

ADP	Acyclic Dependencies Principle	The dependency graph of packages must have no cycles.
SDP	Stable Dependencies Principle	Depend in the direction of stability.
SAP	Stable Abstractions Principle	Abstractness increases with stability.

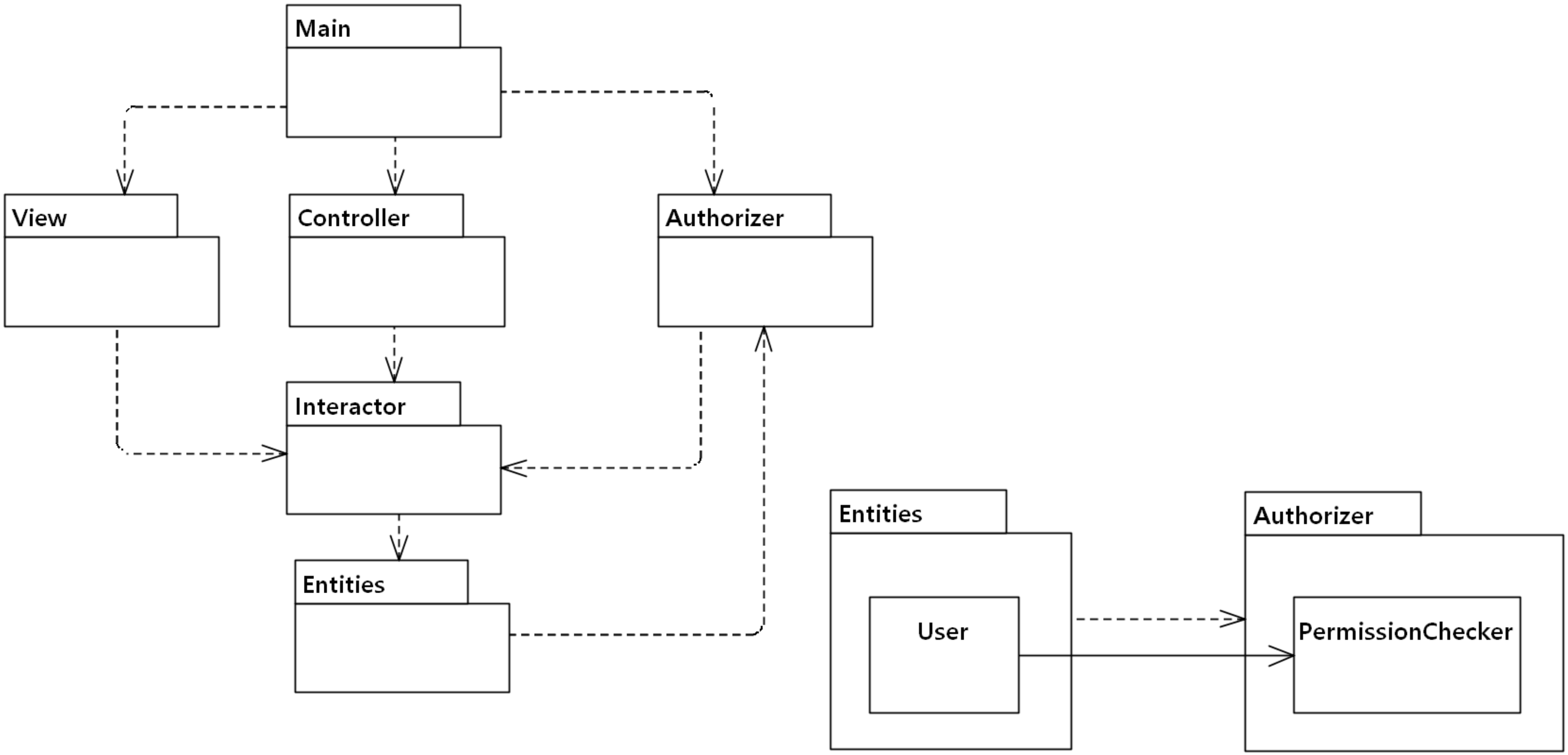
ADP: Acyclic Dependencies Principle

- ❖ Typical package structure



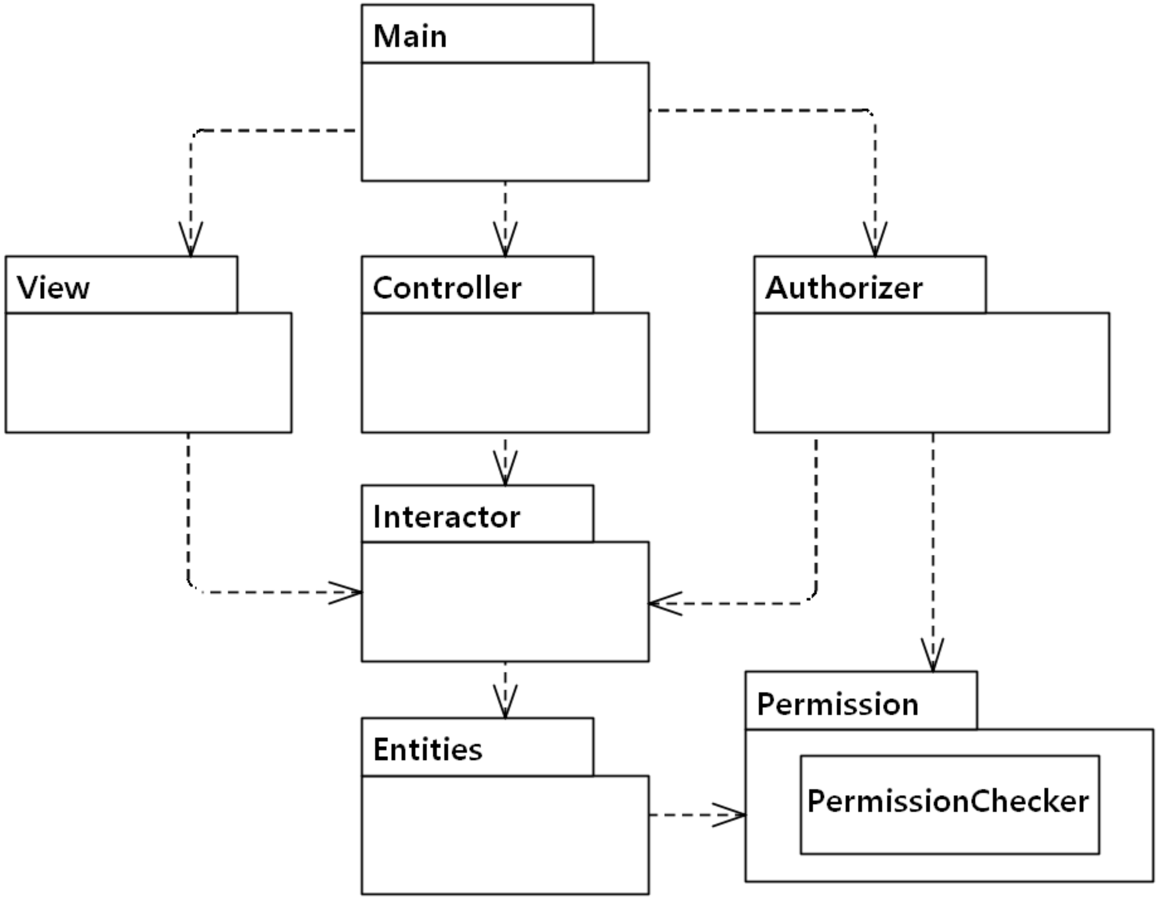
ADP: Acyclic Dependencies Principle

❖ There is a cycle: Interactor, Entities and Authorizer



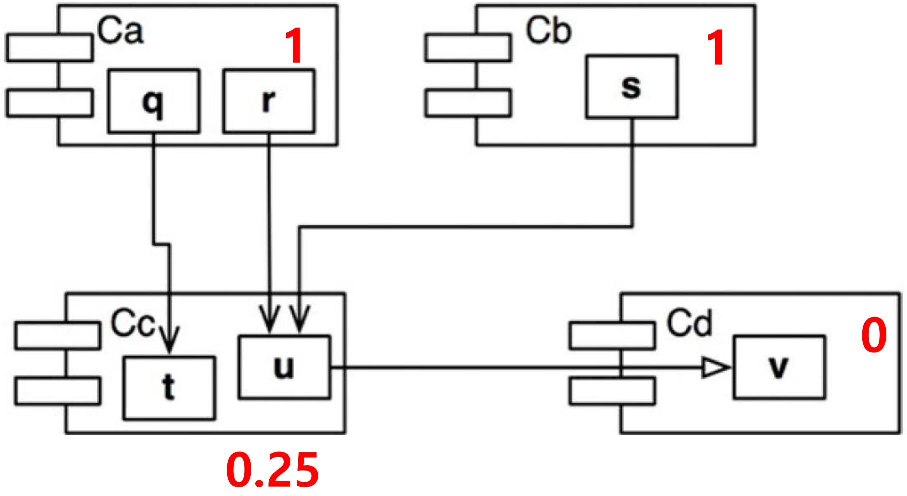
ADP: Acyclic Dependencies Principle

❖ Break the cycle by introducing Permission package



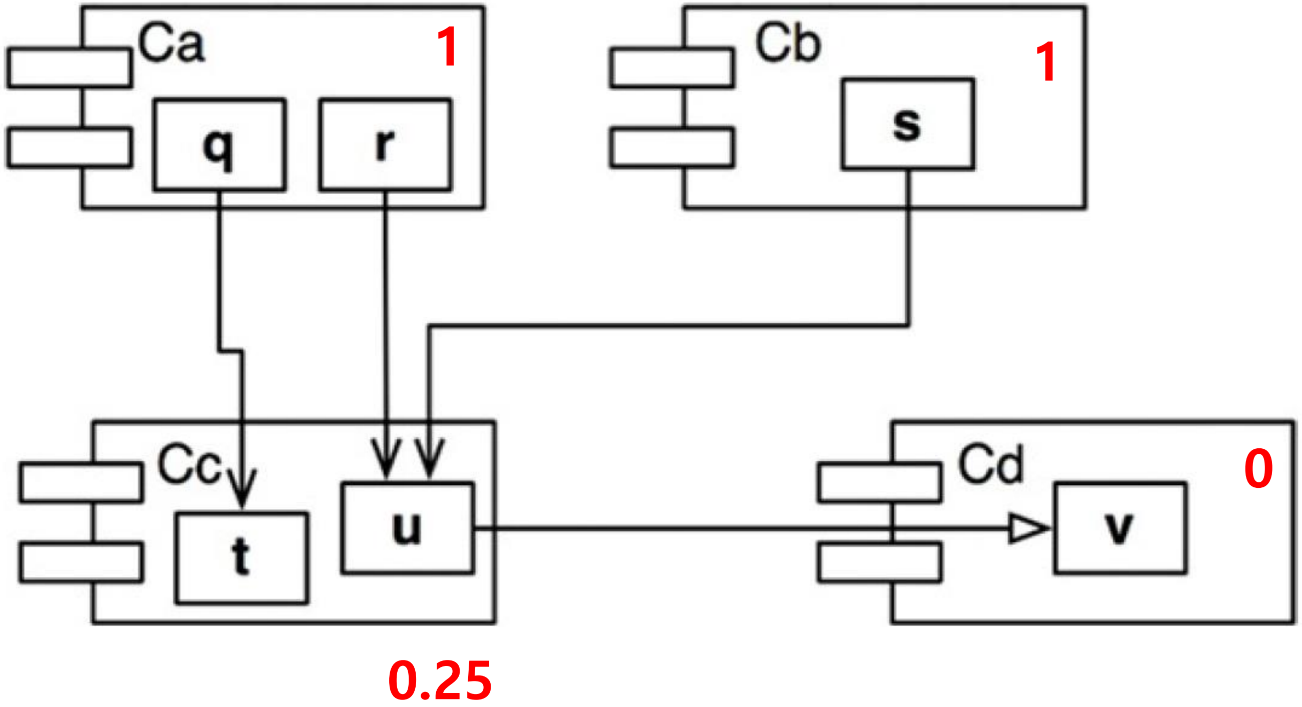
SDP: Stable Dependencies Principle

- Instability Metric = Fan-out / (Fan-In + Fan-Out)
  - Fan-in: the number of classes outside this package that depend on classes within the package
  - Fan-out: the number of classes inside this package that depend on classes outside the package
- Examples
  - Ca = 2 / (0 + 2) = 1
  - Cb = 1 / (0 + 1) = 1
  - Cc = 1 / (3 + 1) = 0.25
  - Cd = 0 / (1 + 0) = 0



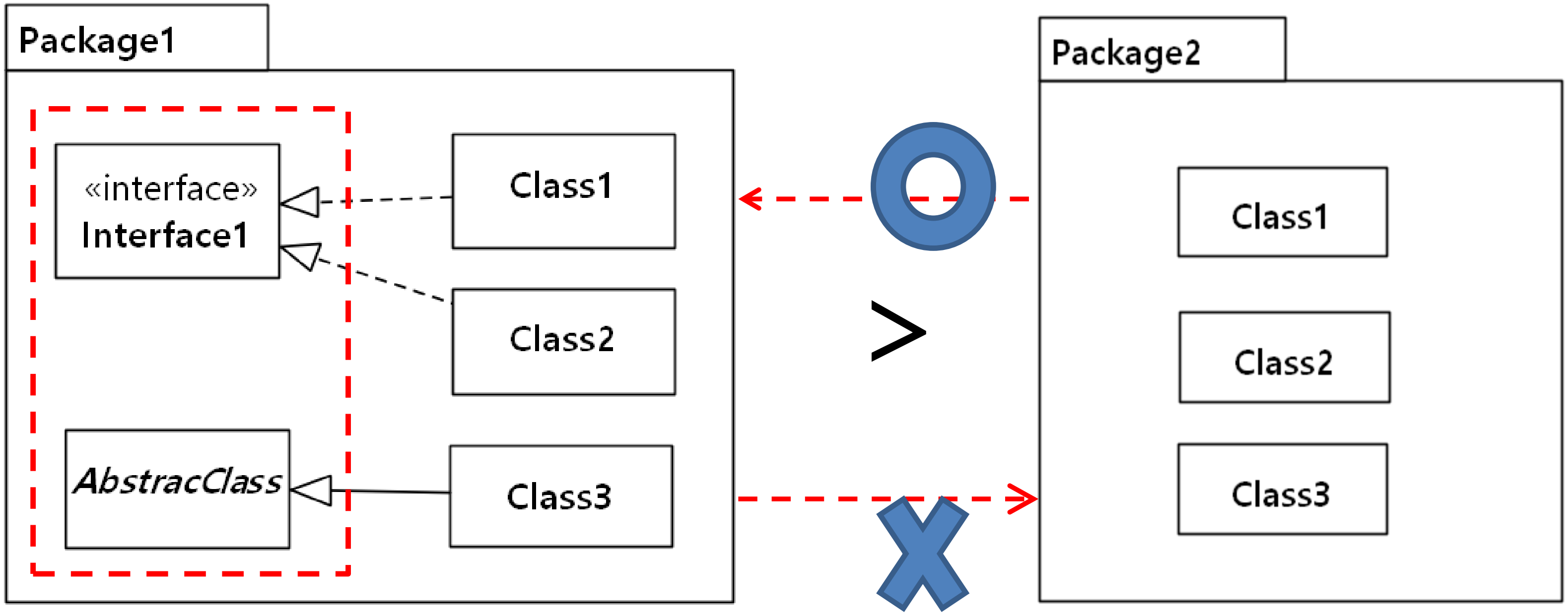
SDP: Stable Dependencies Principle

- Don't depend on the package of less stability
- Instability metrics should decrease in the direction of dependency



SAP: Stable Abstractions Principle

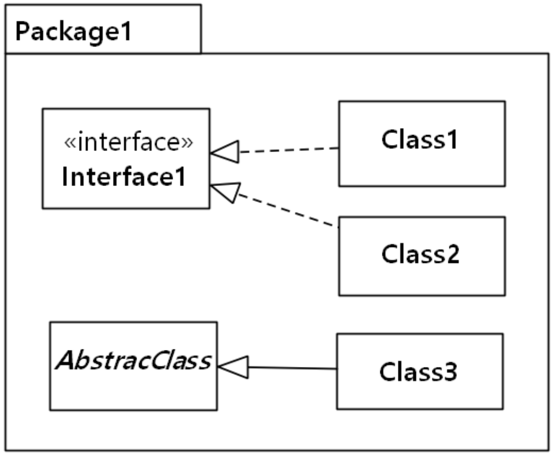
- A stable package should also be abstract.
- If a package is to be stable, it should consist of interfaces and abstract classes so that it can be extended



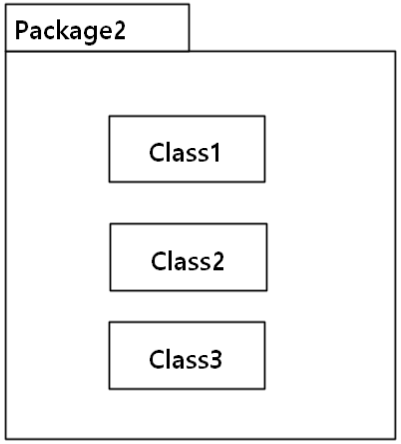
More abstract

Package Abstractness

- Abstractness Metric = Na / Nc
  - Na: number of abstract classes and interfaces in the package
  - Nc: number of all classes and interfaces in the package
- The *A* metric ranges from 0 to 1.
  - 0: concrete classes only
  - 1: abstract classes(or interfaces) only

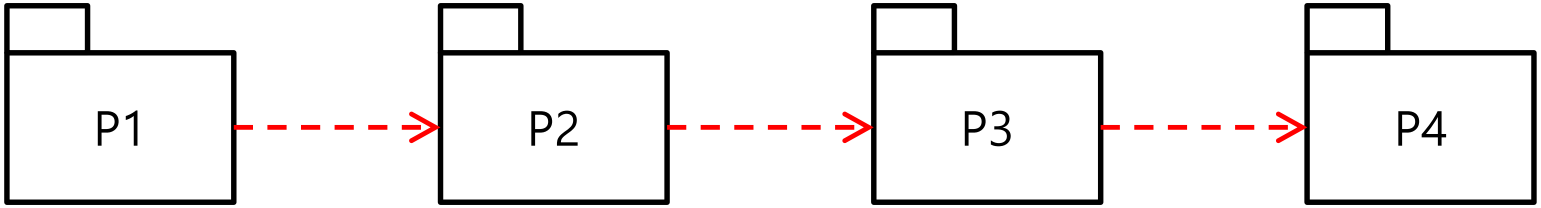


A : 0.4 (=2/5)



A : 0 (=0/3)

SAP and SDP



❖ SAP(Stable Abstraction Principle)

- Abstractness should increase in the direction of dependency

A1 < A2 < A3 < A4

❖ SDP(Stable Dependency Principle)

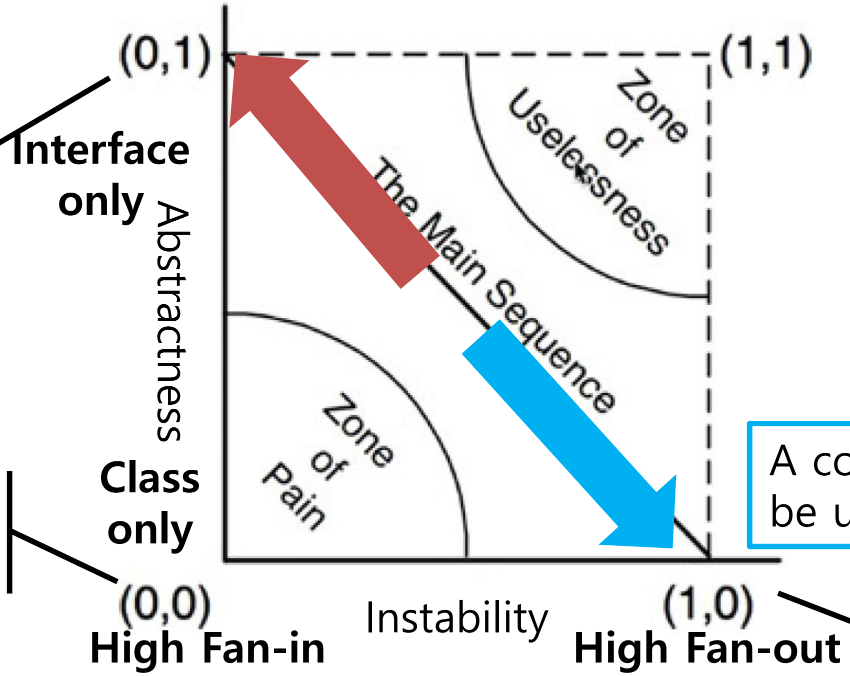
- Instability should decrease in the direction of dependency

I1 > I2 > I3 > I4

Stability and Abstractness

A stable package(High Fan-in) should be abstract, so that it can be extended

It is maximally abstract, yet has no clients (0 Fan-in). Such packages are useless.



Interface only package

Type only packages OK;
Type => Interface

A concrete package can be unstable(High Fan-out)

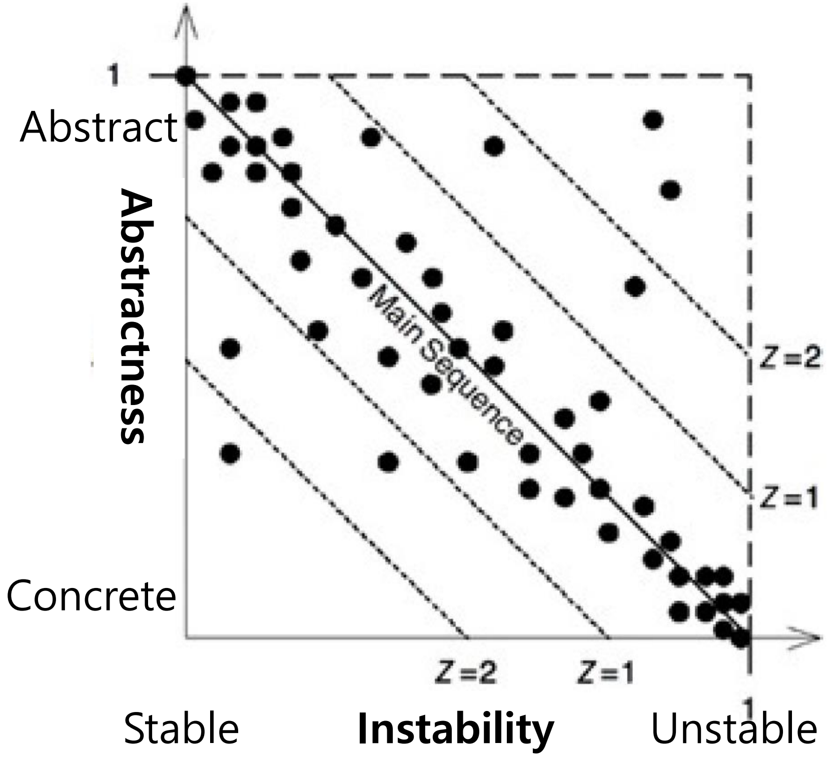
Concrete class only package

A package is not desirable because it is rigid.
It cannot be extended because it is not abstract, and it is very difficult to change because of its stability (High Fan-in).

Stability and Abstractness

❖ Distance from the Main Sequence

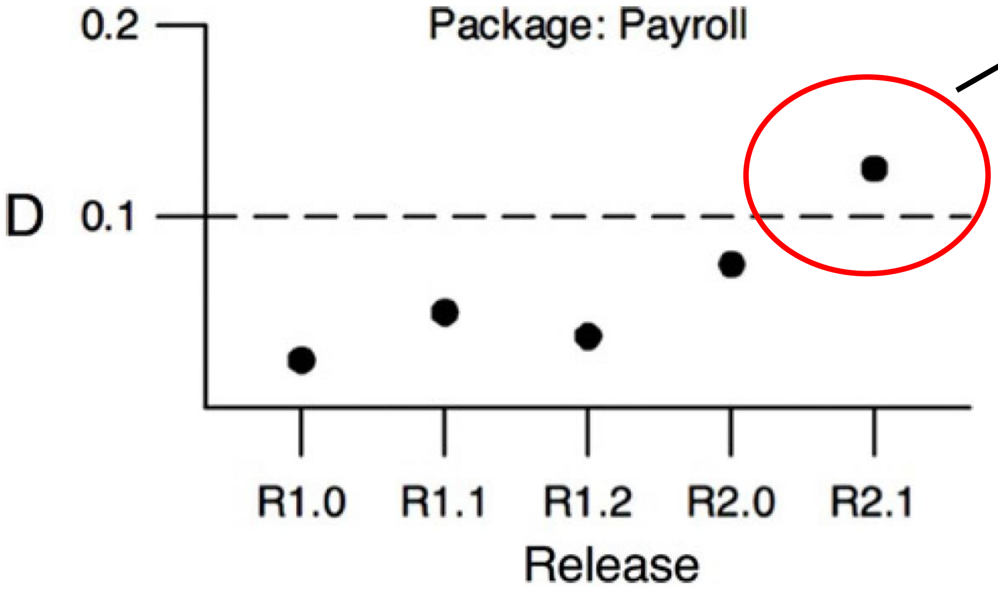
- Distance = | A + I – 1 |
  - ✓ 0: the package is directly on the Main Sequence.
  - ✓ 1: the package is as far away as possible from the Main Sequence.



The bulk of the components lie along the Main Sequence, but some of them are more than one standard deviation (Z = 1) away from the mean.

Stability and Abstractness

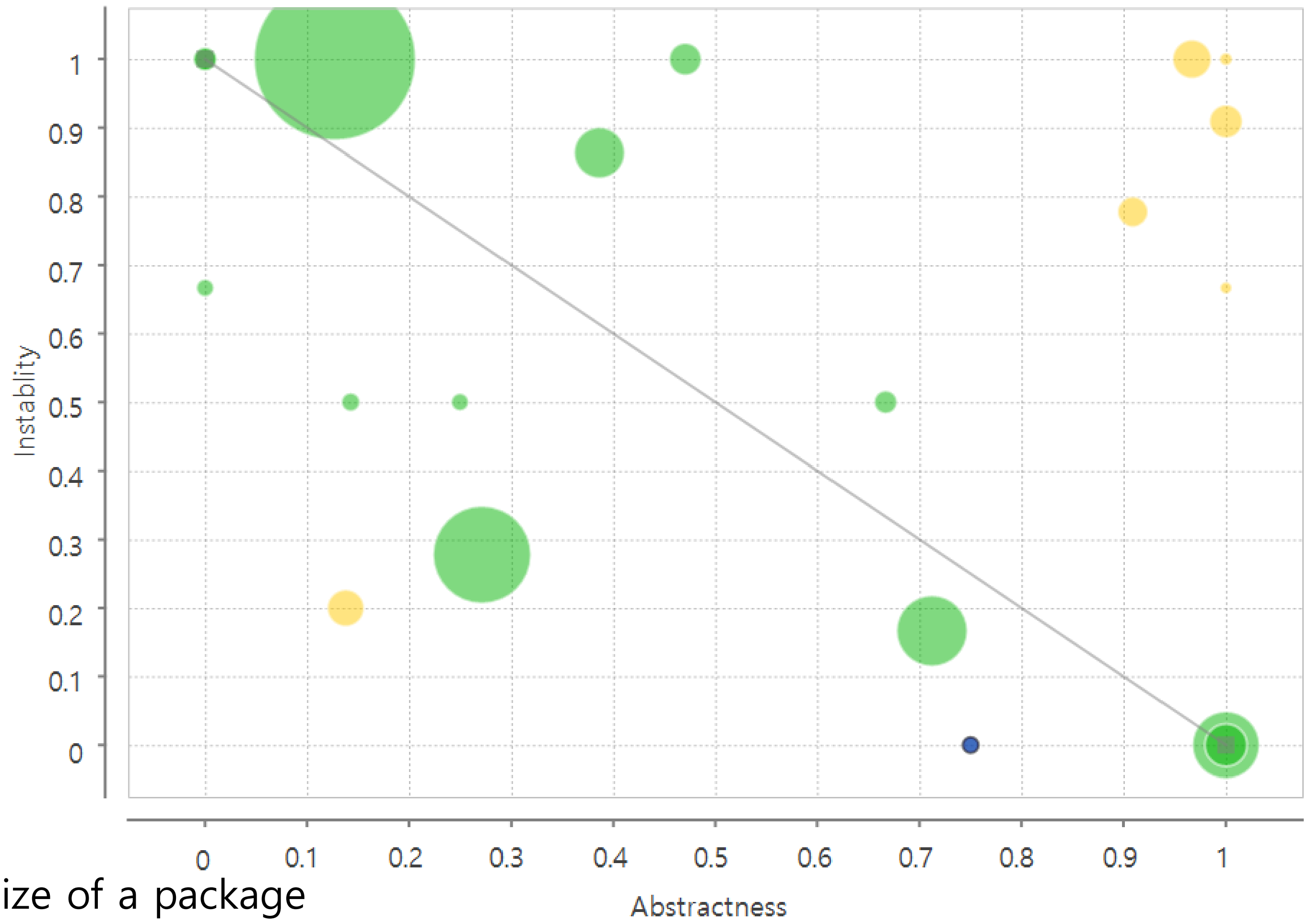
- Another way to use the metrics is to plot the D metric of each component over time.
- The plot shows a control threshold at *D* = 0.1.



The R2.1 point has exceeded this control limit, so we need to find out why this component is so far from the main sequence

STAN

- Structure Analysis for Java: stan4j.com



The radius: the size of a package

DESIGN TECHNIQUES

Design Techniques

- Design Patterns
  - Common use of patterns
  - Strategy vs state
  - Strategy vs template method
  - Strategy vs command vs observer
  - Combined uses of patterns

DESIGN PATTERNS

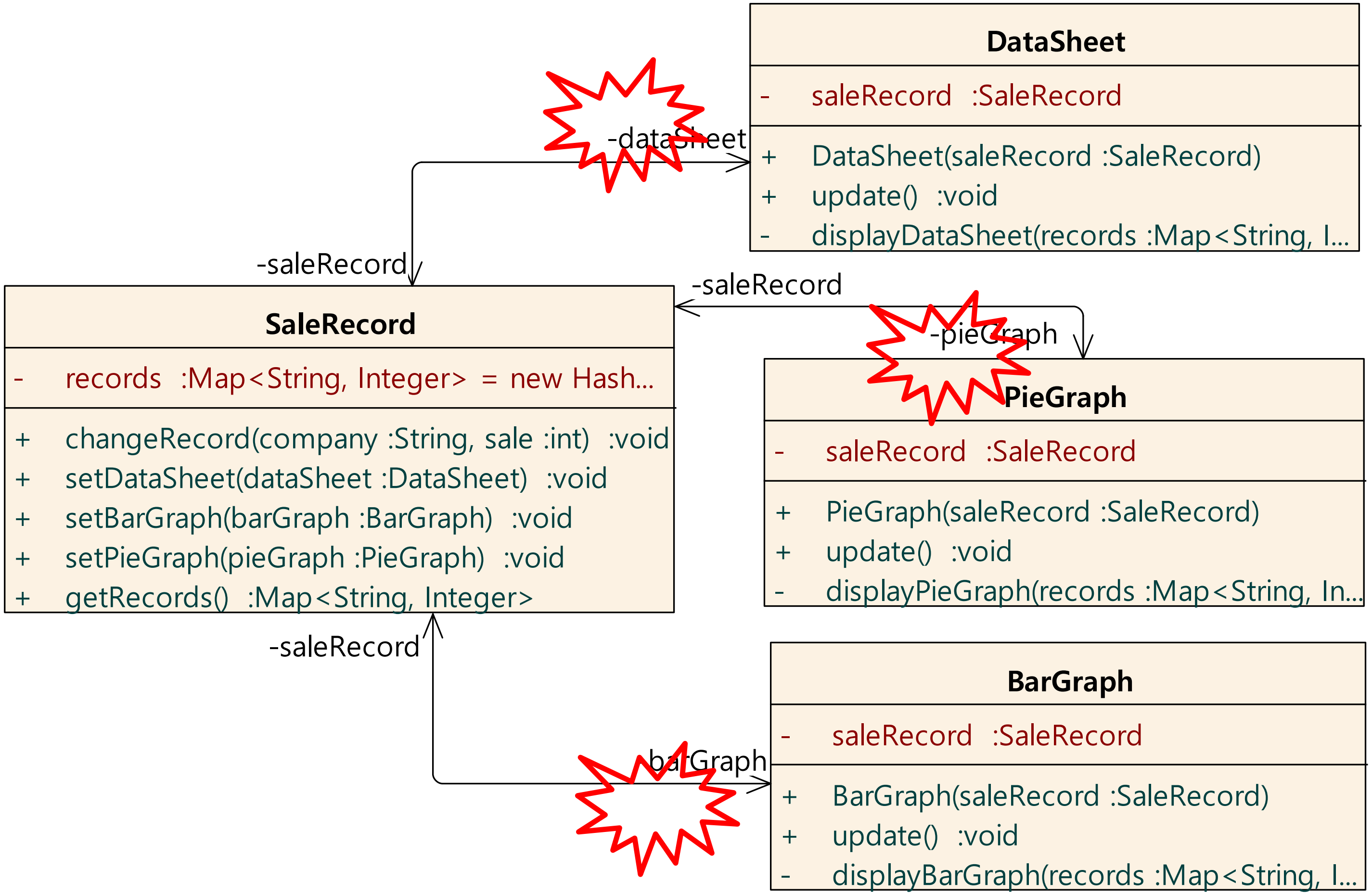
GoF Pattern Catalogue

	Creational	Structural	Behavioral
Class-level	Factory Method	Adapter (class)	Interpreter Template Method
Object-level	Abstract Factory Builder Prototype Singleton	Adapter (object) Bridge Composite Decorator Facade Flyweight Proxy	Chain of Responsibility Command Iterator Mediator Memento Observer State Strategy Visitor

COMMON USE OF PATTERNS

Replace State Change Notification with Observer

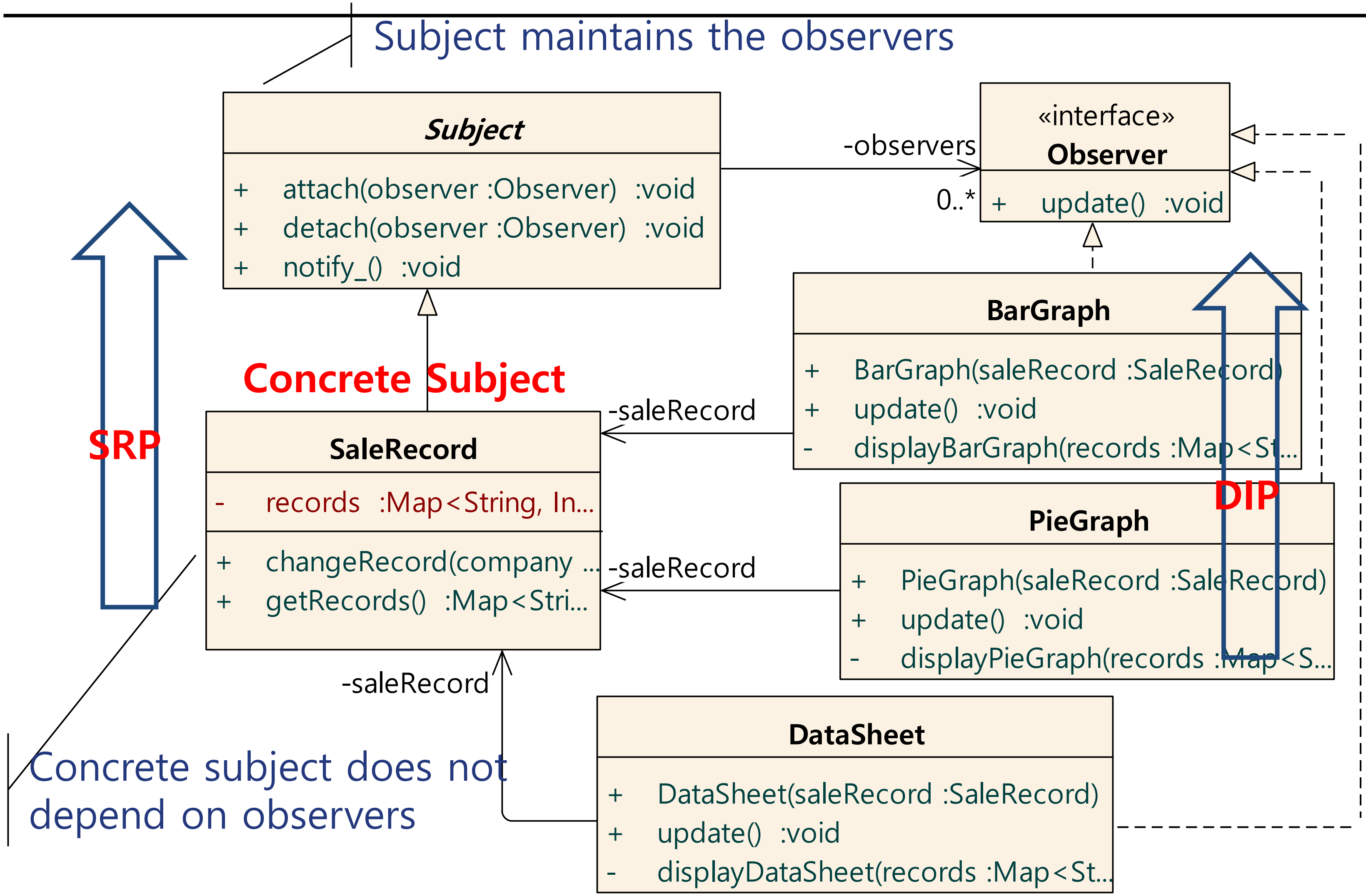
- Whenever new kinds of observers or new instances are considered, class SaleRecord should be modified



DataSheet
- saleRecord :SaleRecord
+ DataSheet(saleRecord :SaleRecord)
+ update() :void
- displayDataSheet(records :Map<String, I...
-dataSheet
-saleRecord
SaleRecord
- records :Map<String, Integer> = new Hash...
+ changeRecord(company :String, sale :int) :void
+ setDataSheet(dataSheet :DataSheet) :void
+ setBarGraph(barGraph :BarGraph) :void
+ setPieGraph(pieGraph :PieGraph) :void
+ getRecords() :Map<String, Integer>
-saleRecord
-pieGraph
PieGraph
- saleRecord :SaleRecord
+ PieGraph(saleRecord :SaleRecord)
+ update() :void
- displayPieGraph(records :Map<String, In...
-saleRecord
-barGraph
BarGraph
- saleRecord :SaleRecord
+ BarGraph(saleRecord :SaleRecord)
+ update() :void
- displayBarGraph(records :Map<String, I...

Replace State Change Notification with Observer

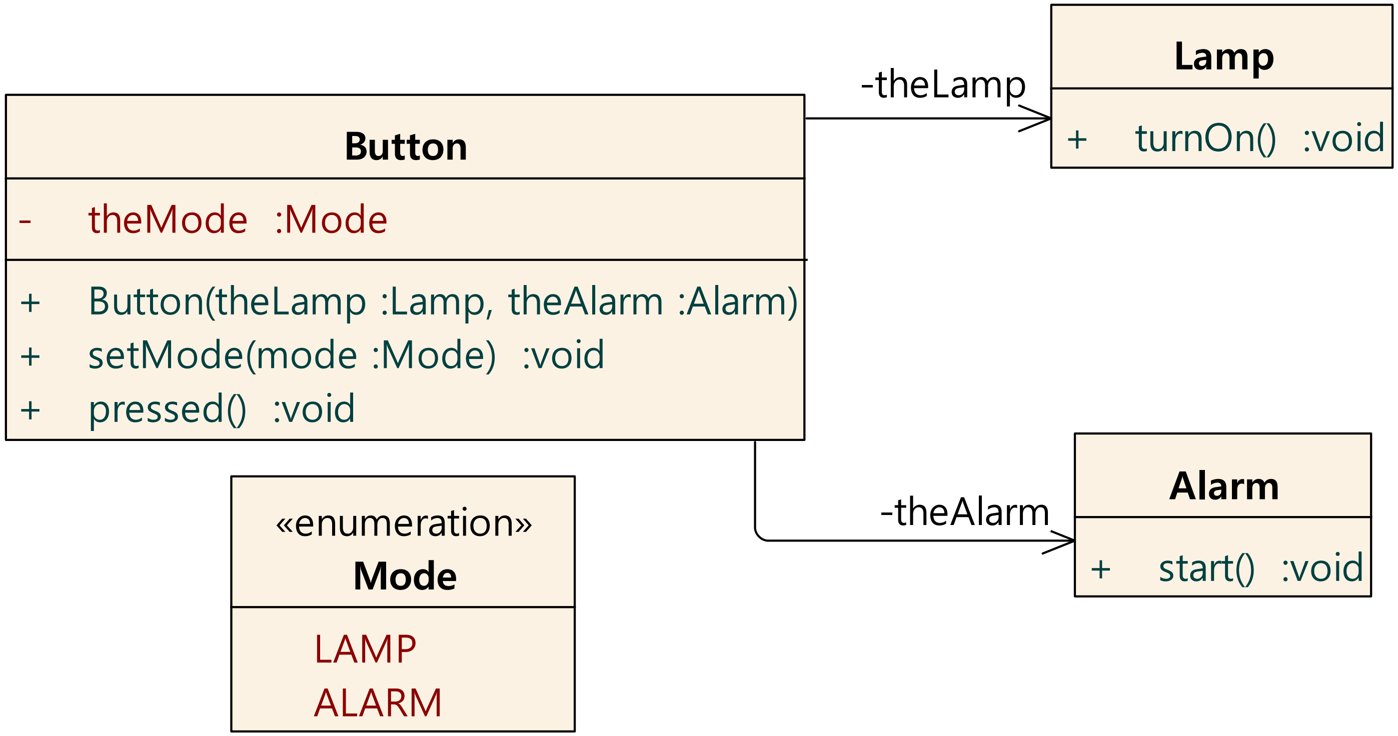
Subject maintains the observers


Subject
+ attach(observer :Observer) :void
+ detach(observer :Observer) :void
+ notify_() :void
-observers
0..*
«interface»
Observer
+ update() :void
Concrete Subject
SRP
SaleRecord
- records :Map<String, In...
+ changeRecord(company ...
+ getRecords() :Map<Stri...
-saleRecord
-saleRecord
-saleRecord
BarGraph
+ BarGraph(saleRecord :SaleRecord)
+ update() :void
- displayBarGraph(records :Map<St...
PieGraph
DIP
+ PieGraph(saleRecord :SaleRecord)
+ update() :void
- displayPieGraph(records :Map<S...
DataSheet
+ DataSheet(saleRecord :SaleRecord)
+ update() :void
- displayDataSheet(records :Map<St...

Concrete subject does not depend on observers

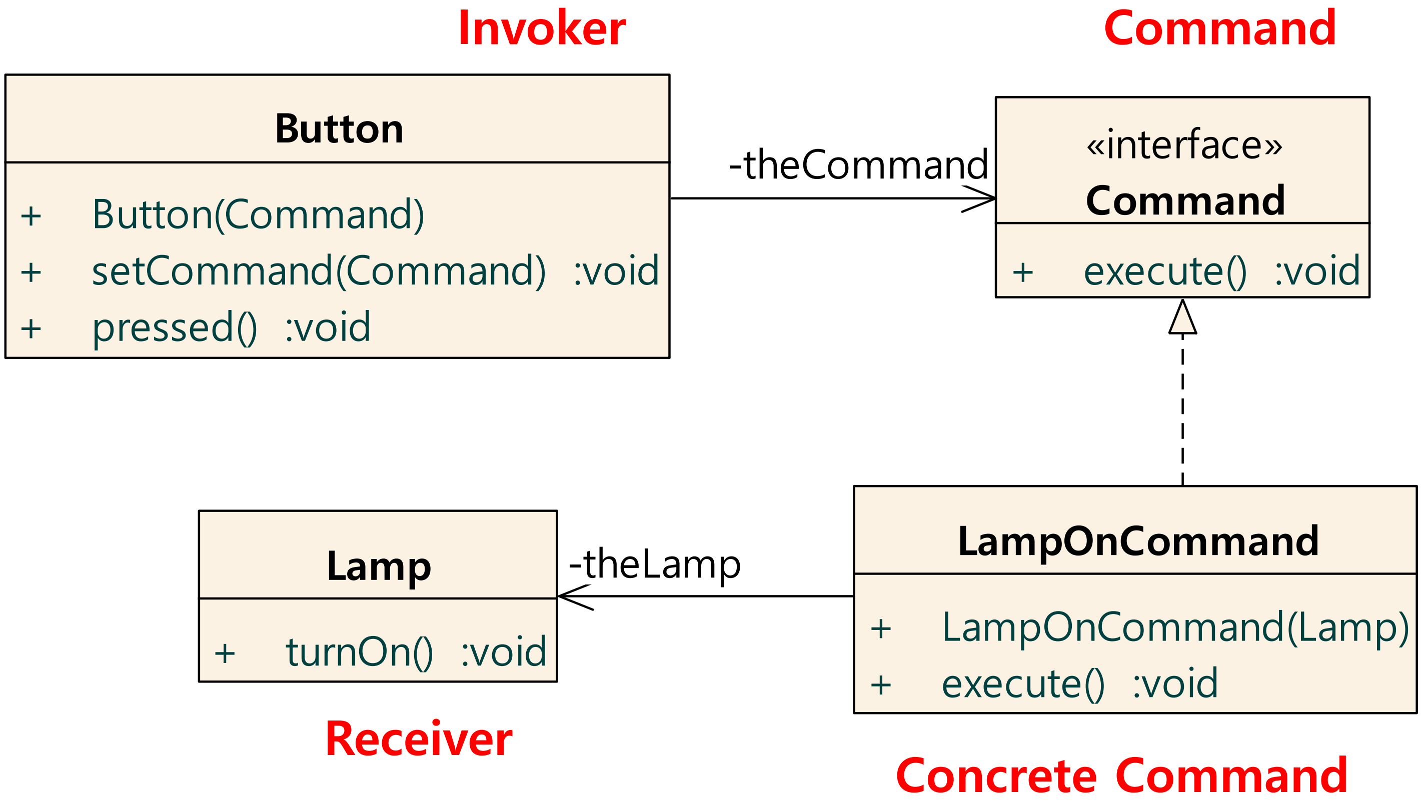
Replace Event Notification Behavior with Command

- Button itself invokes a specific operation of a specific target object



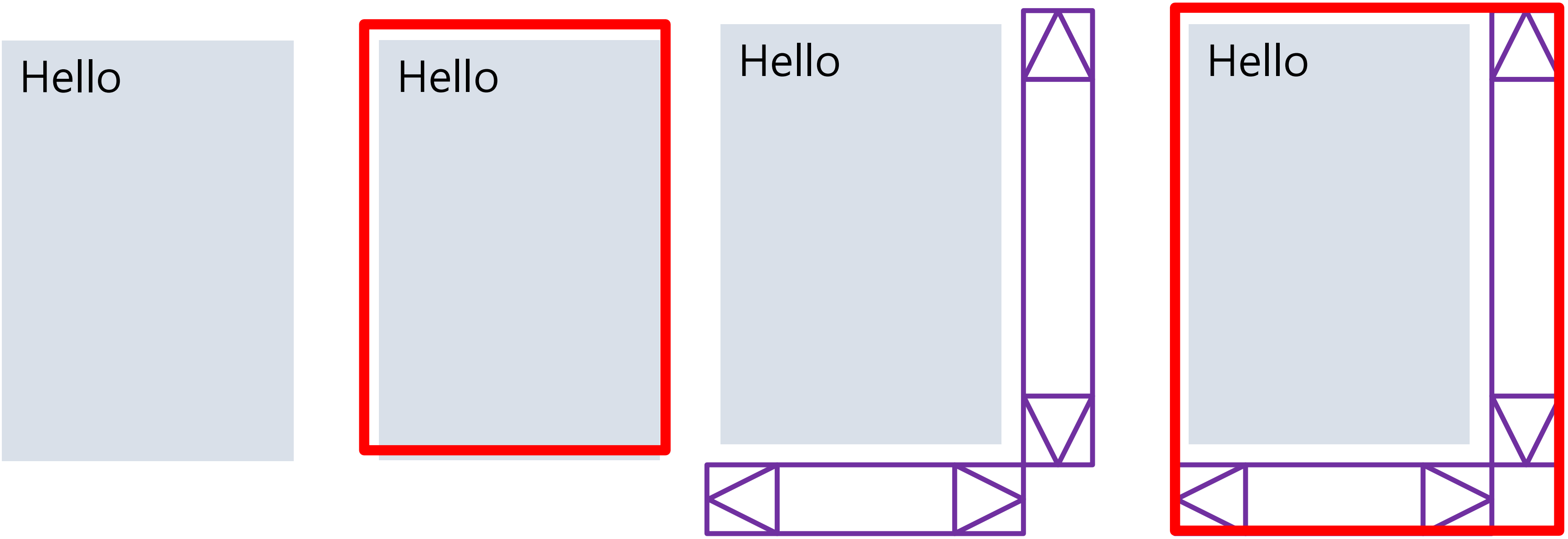
Replace Event Notification Behavior with Command

- Encapsulate a request as an object.
- It enables the Button to be independent of any behavior



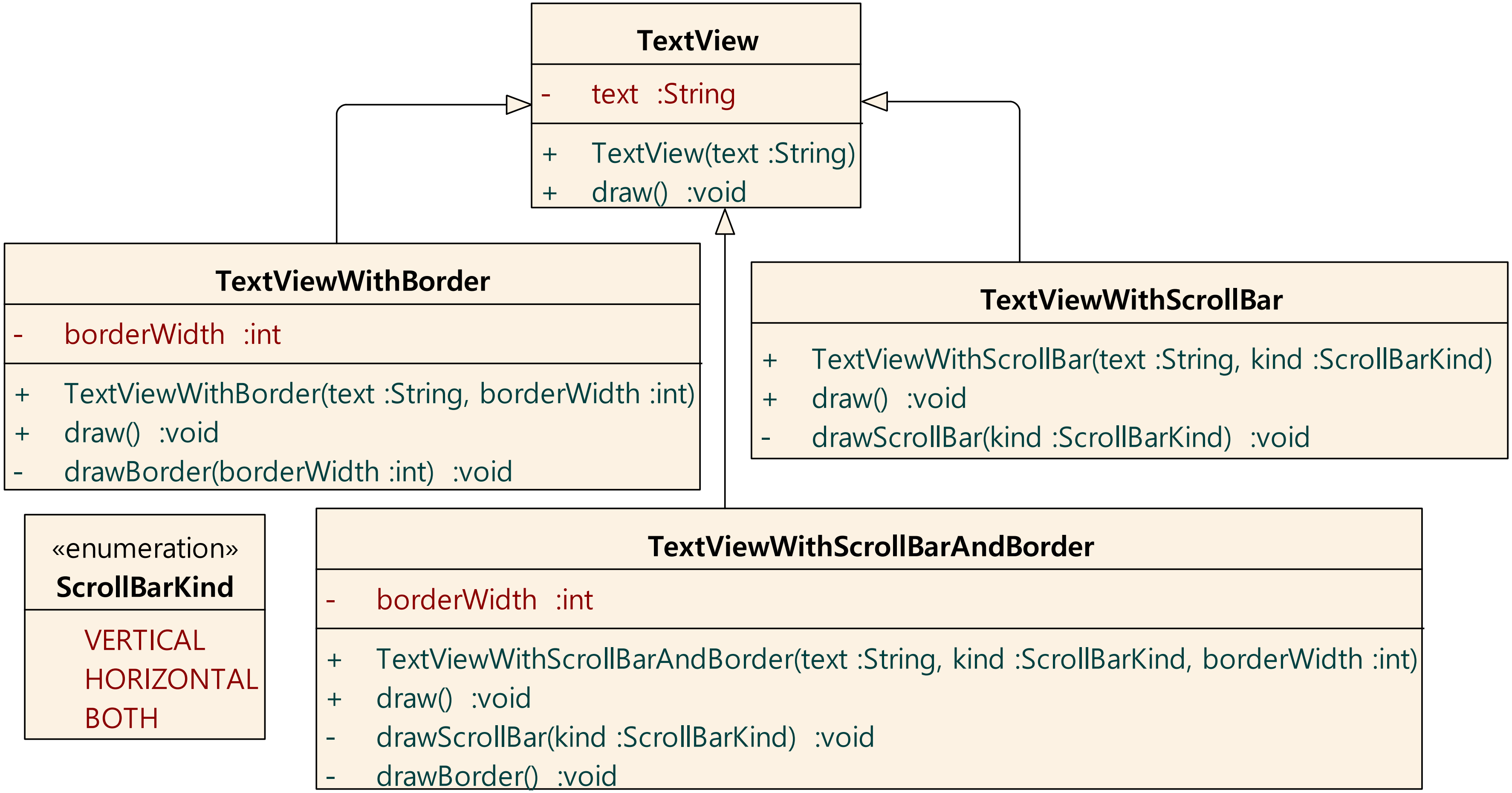
Replace Additional Behavior with Decorator

- We need various kinds of TextWindow
  - TextView with no ScrollBar and no Border
  - TextView with Border
  - TextView with ScrollBar
  - TextView with ScrollBar and Border



Replace Additional Behavior with Decorator

❖ Inheritance-based Approach



TextView

- text :String

+ TextView(text :String)
+ draw() :void

TextViewWithBorder

- borderWidth :int

+ TextViewWithBorder(text :String, borderWidth :int)
+ draw() :void
- drawBorder(borderWidth :int) :void

TextViewWithScrollBar

+ TextViewWithScrollBar(text :String, kind :ScrollBarKind)
+ draw() :void
- drawScrollBar(kind :ScrollBarKind) :void

«enumeration»
ScrollBarKind

VERTICAL
HORIZONTAL
BOTH

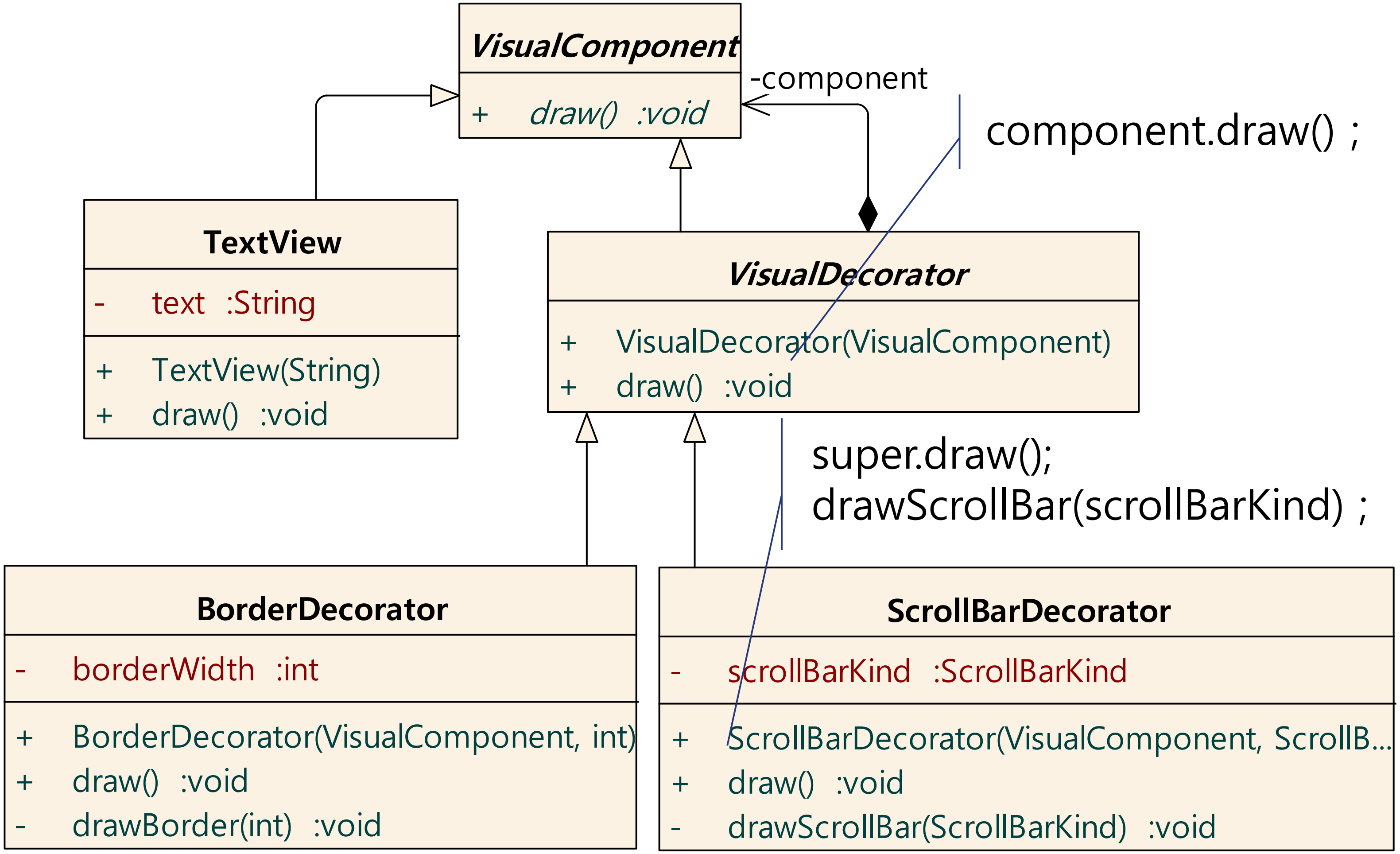
TextViewWithScrollBarAndBorder

- borderWidth :int

+ TextViewWithScrollBarAndBorder(text :String, kind :ScrollBarKind, borderWidth :int)
+ draw() :void
- drawScrollBar(kind :ScrollBarKind) :void
- drawBorder() :void

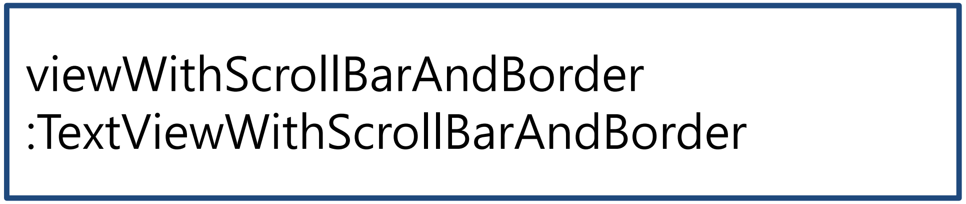
Replace Additional Behavior with Decorator

❖ Decorator class is defined for each additional behavior

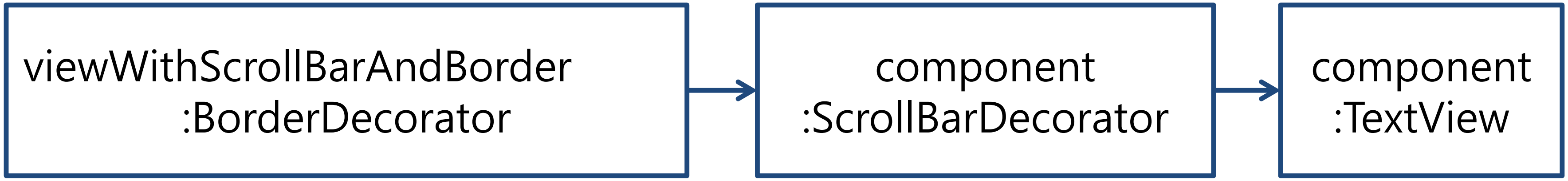


Replace Additional Behavior with Decorator

```
TextView viewWithScrollBarAndBorder = new
  TextViewWithScrollBarAndBorder("Hello", ScrollBarKind.BOTH, 5) ;
viewWithScrollBarAndBorder.draw() ;
```

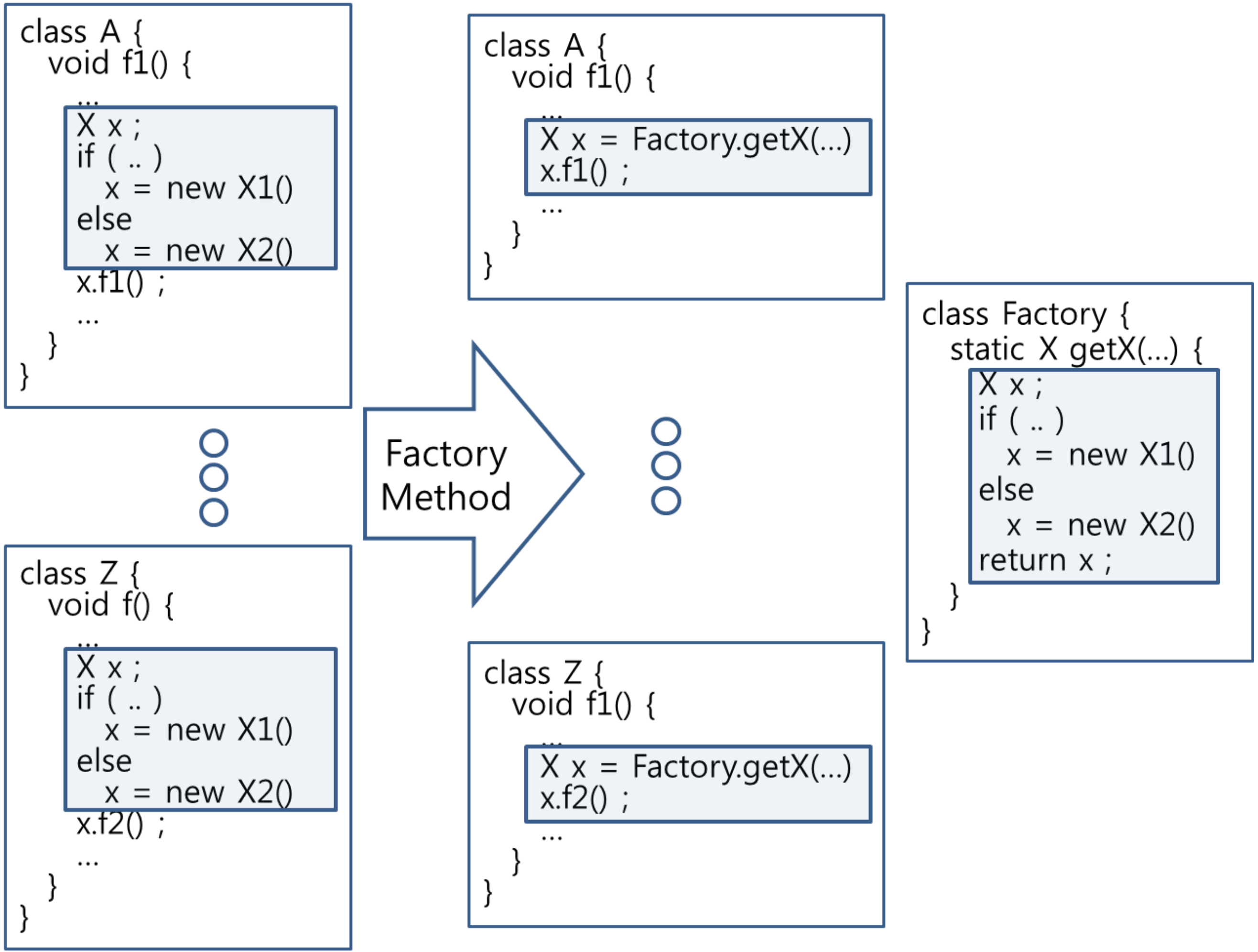


```
VisualComponent viewWithScrollBarAndBorder =
  new BorderDecorator(
      new ScrollBarDecorator(new TextView("Hello"), ScrollBarKind.BOTH), 5) ;
    viewWithScrollBarAndBorder.draw() ;
```



Replace Object Creation Behavior with Factory

- ❖ Localize and isolate object creation codes



```
class A {
   void f1() {
      …
      X x ;
      if ( .. )
         x = new X1()
      else
         x = new X2()
      x.f1() ;
      …
   }
}
```

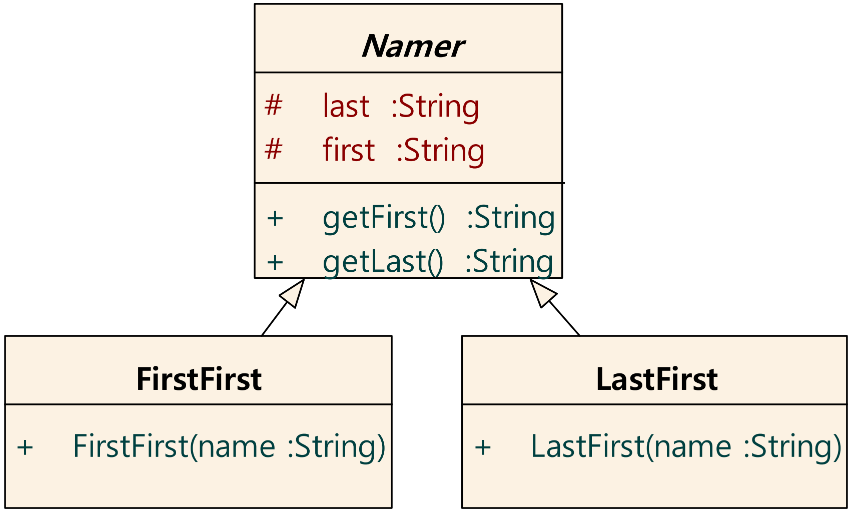
```
class Z {
   void f() {
      …
      X x ;
      if ( .. )
         x = new X1()
      else
         x = new X2()
      x.f2() ;
      …
   }
}
```

```
class A {
   void f1() {
      …
      X x = Factory.getX(…)
      x.f1() ;
      …
   }
}
```

```
class Z {
   void f1() {
      …
      X x = Factory.getX(…)
      x.f2() ;
      …
   }
}
```

```
class Factory {
   static X getX(…) {
      X x ;
      if ( .. )
         x = new X1()
      else
         x = new X2()
      return x ;
   }
}
```

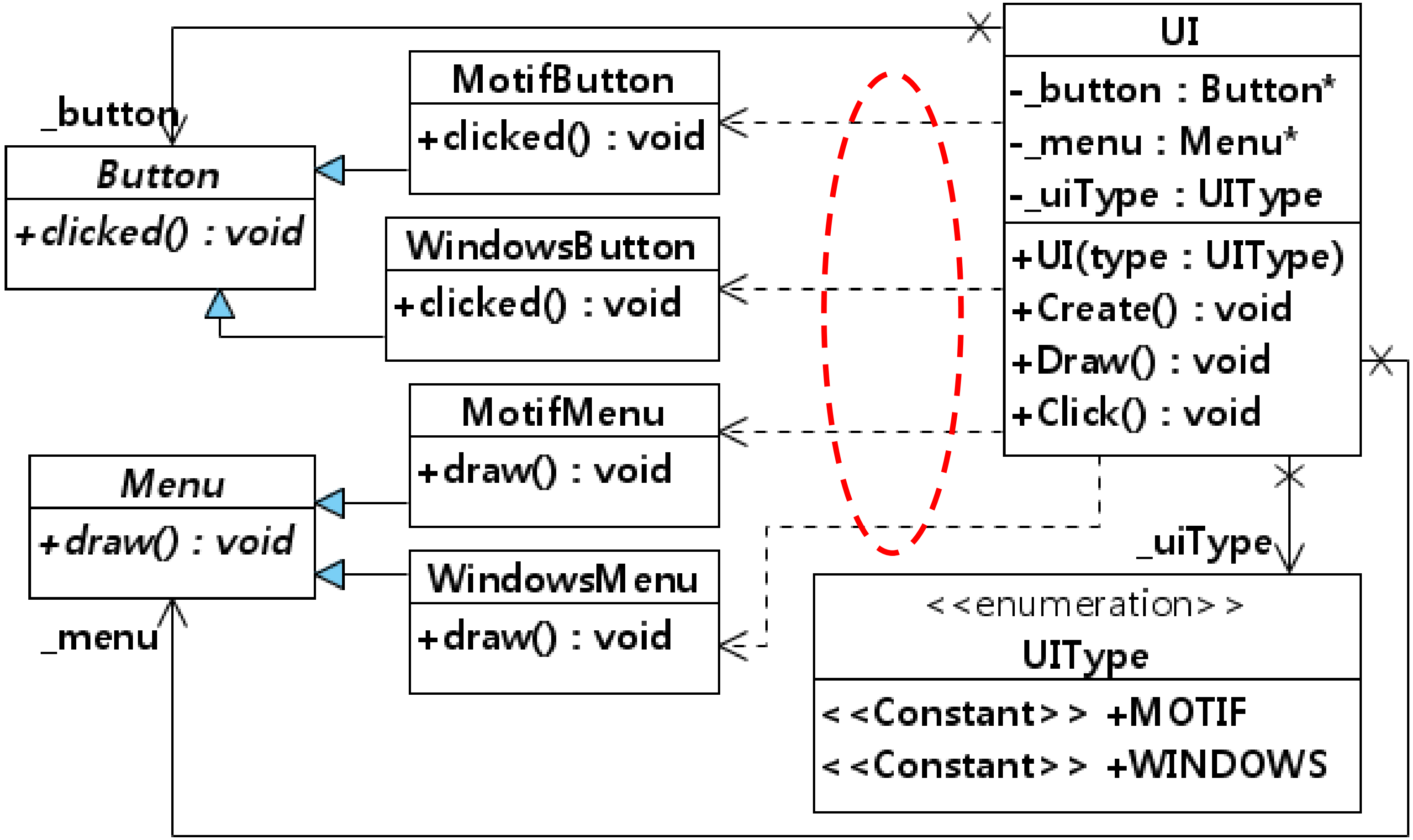
Replace Object Creation Behavior with Factory



```
public class NameFactory {
  public static Namer getInstance(String name) {
    int i = name.indexOf(",");
    if (i>0)
      return new LastFirst(name); //return an object of one class
    else
      return new FirstFirst(name); //or an object of the other
  }
}
```

Replace Dependent Object Creation Behavior with Abstract Factory

- ❖ The Client (UI) depends on platform-specific Products



Version 0

```
public class UI {
  private Button _button ;
  private Menu _menu ;
  private UIType _uiType ;
  public UI(UIType type ) { _uiType = type ; }
  public void Create() {
    switch ( _uiType ) {
      case MOTIF: {
        _button = new MotifButton() ; _menu = new MotifMenu() ;
          break ; }
      case WINDOWS: {
        _button = new WindowsButton() ; _menu = new WindowsMenu() ;
          break ; }
    }
  }
  public void Draw() { _menu.draw() ; }
  public void Click() { _button.clicked() ; }
}
```

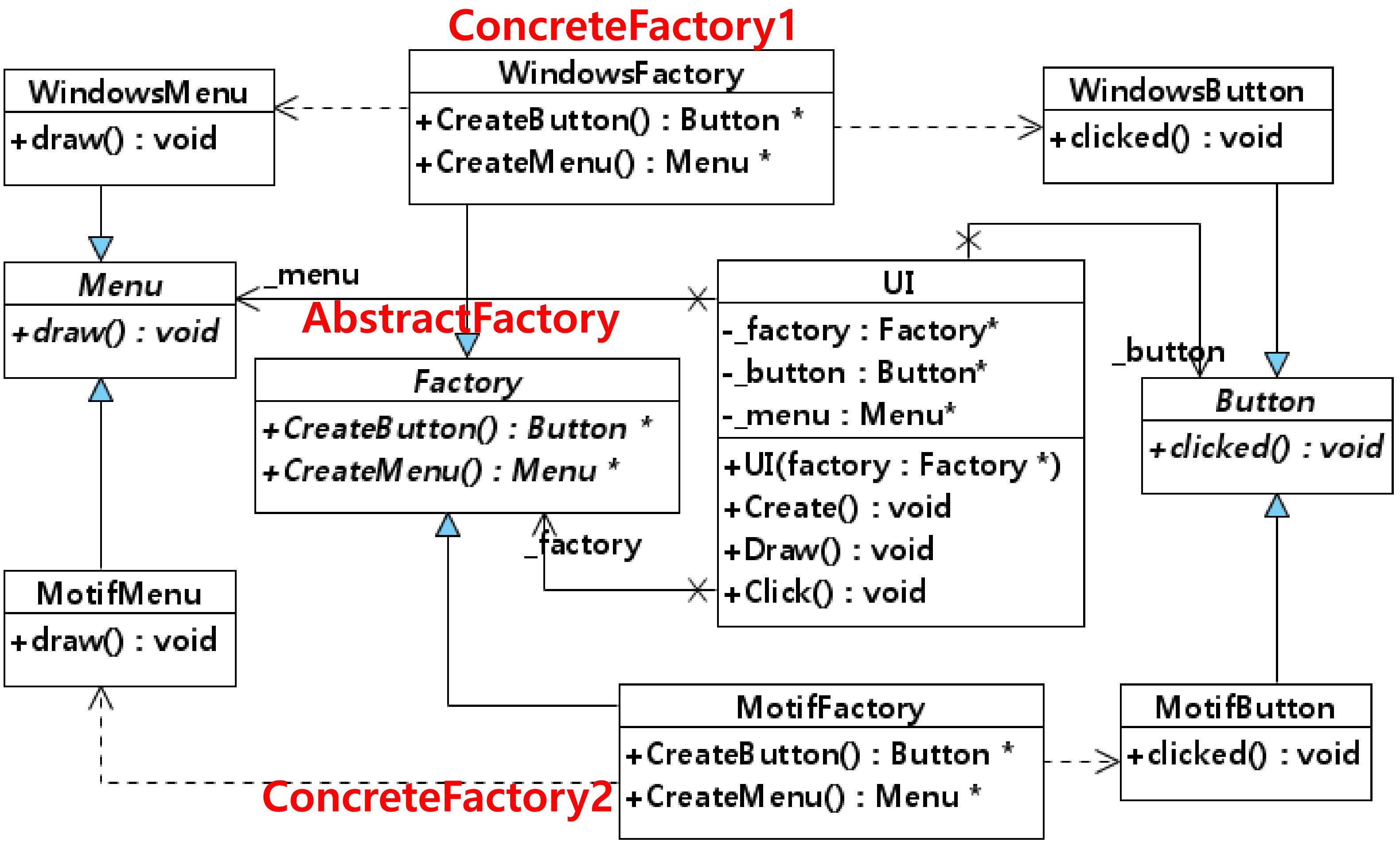
The Client (UI) depends on platform-specific Products

Version 1 – Factory Method Pattern

```
public class UI {
  private Button _button ;
  private Menu _menu ;
  private UIType _uiType ;
  public UI(UIType type ) { _uiType = type ; }
  public void Create() { improved by applying factory method pattern
    _button = ButtonFactory.getButton(_uiType);
    _menu = MenuFactory.getMenu(_uiType);
  }
  public void Draw() { _menu.draw() ; }
  public void Click() { _button.clicked() ; }
}
```

The Client (UI) **still depends on** platform-specific Products

Replace Dependent Object Creation Behavior with Abstract Factory



Version 2 – Abstract Factory Pattern

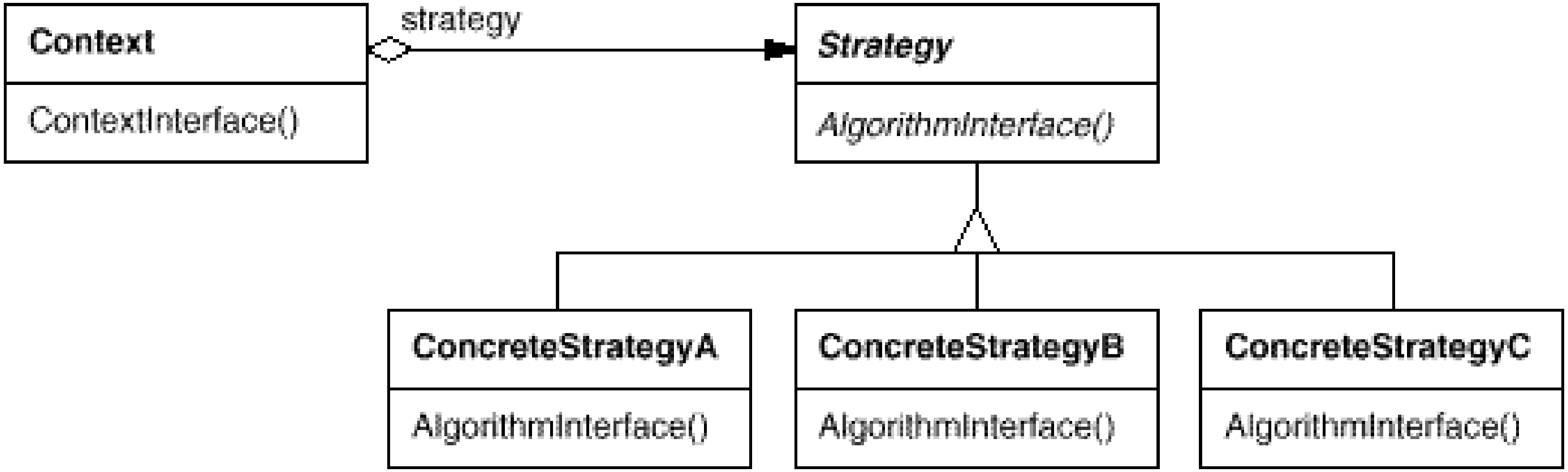
```
public class UI {
  private Button _button ;
  private Menu _menu ;
  private Factory _factory ;

  public UI(Factory factory ) { _factory = factory ; }
  public void Create() {
    _button = _factory.CreateButton() ;
    _menu = _factory.CreateMenu() ;
  }
  public void Draw() { _menu.draw() ; }
  public void Click() { _button.clicked() ; }
}
```

The Client (UI) does not depend on platform-specific products

STRATEGY VS STATE

Strategy Pattern

- Define a family of algorithms, encapsulate each one and make them interchangeable
- Strategy lets the algorithm vary independently from clients that use it



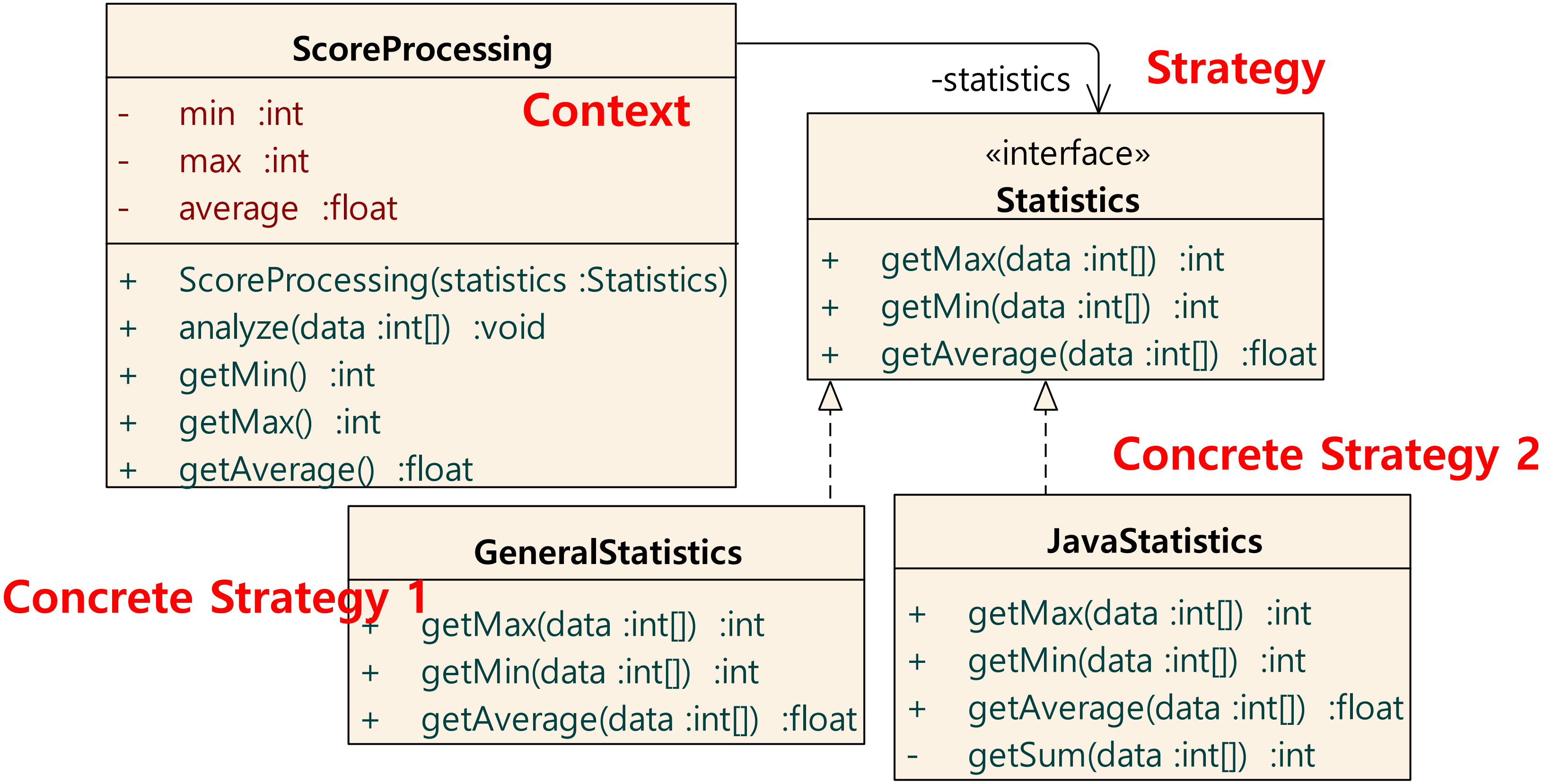
Strategy Pattern: Motivating Example

```
public class ScoreProcessing {
  private int min, max ;
  private float average ;
  public void analyze(int[] data) {
    min = max = data[0] ;
    int sum = data[0] ;
    for ( int i = 1 ; i < data.length ; i ++ ) {
      if ( min > data[i] ) min = data[i] ;
      if ( max < data[i] ) max = data[i] ;
      sum += data[i] ;
    }
    average = (float) sum / data.length ;
  }
  public int getMin() { return min; }
  public int getMax() { return max; }
  public float getAverage() { return average; }
}
```

analyze() has poor cohesion. It performs three different functions: min, max, and average

In addition, the source code should be modified to change algorithm

Strategy Pattern



ScoreProcessing

- min :int
- max :int
- average :float

+ ScoreProcessing(statistics :Statistics)
+ analyze(data :int[]) :void
+ getMin() :int
+ getMax() :int
+ getAverage() :float

Context

-statistics

Strategy

«interface»
Statistics

+ getMax(data :int[]) :int
+ getMin(data :int[]) :int
+ getAverage(data :int[]) :float

Concrete Strategy 2

GeneralStatistics

+ getMax(data :int[]) :int
+ getMin(data :int[]) :int
+ getAverage(data :int[]) :float

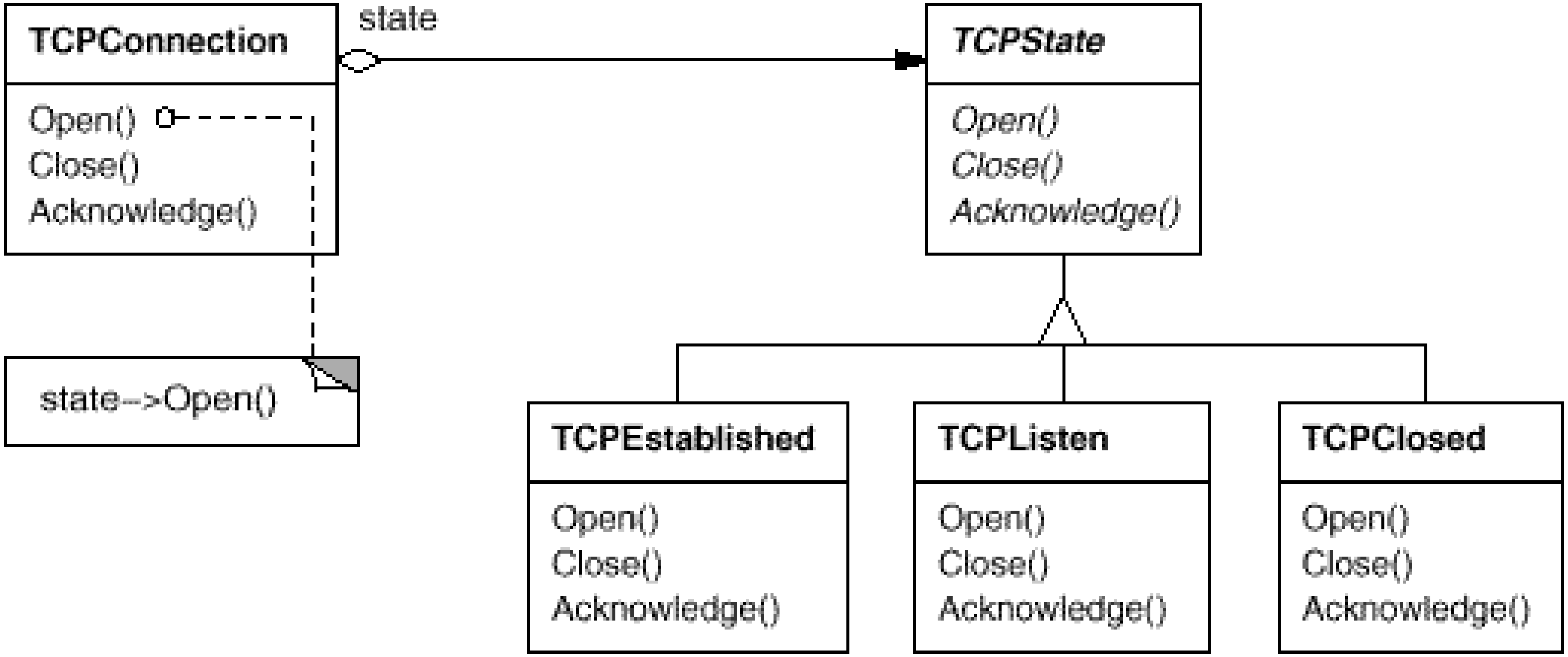
Concrete Strategy 1

JavaStatistics

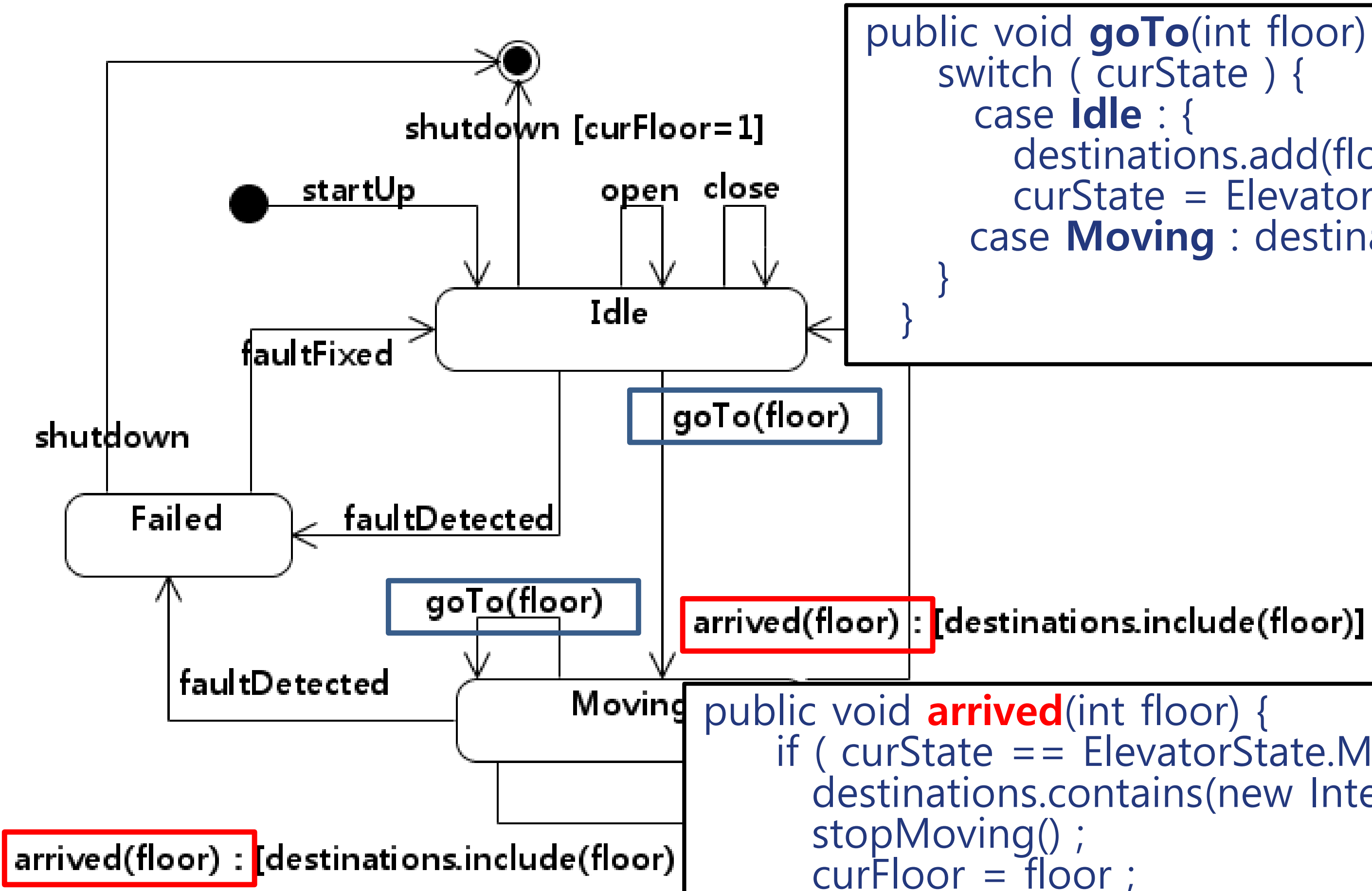
+ getMax(data :int[]) :int
+ getMin(data :int[]) :int
+ getAverage(data :int[]) :float
- getSum(data :int[]) :int

State Pattern

❖ Allow an object to alter its behavior when its internal state changes.



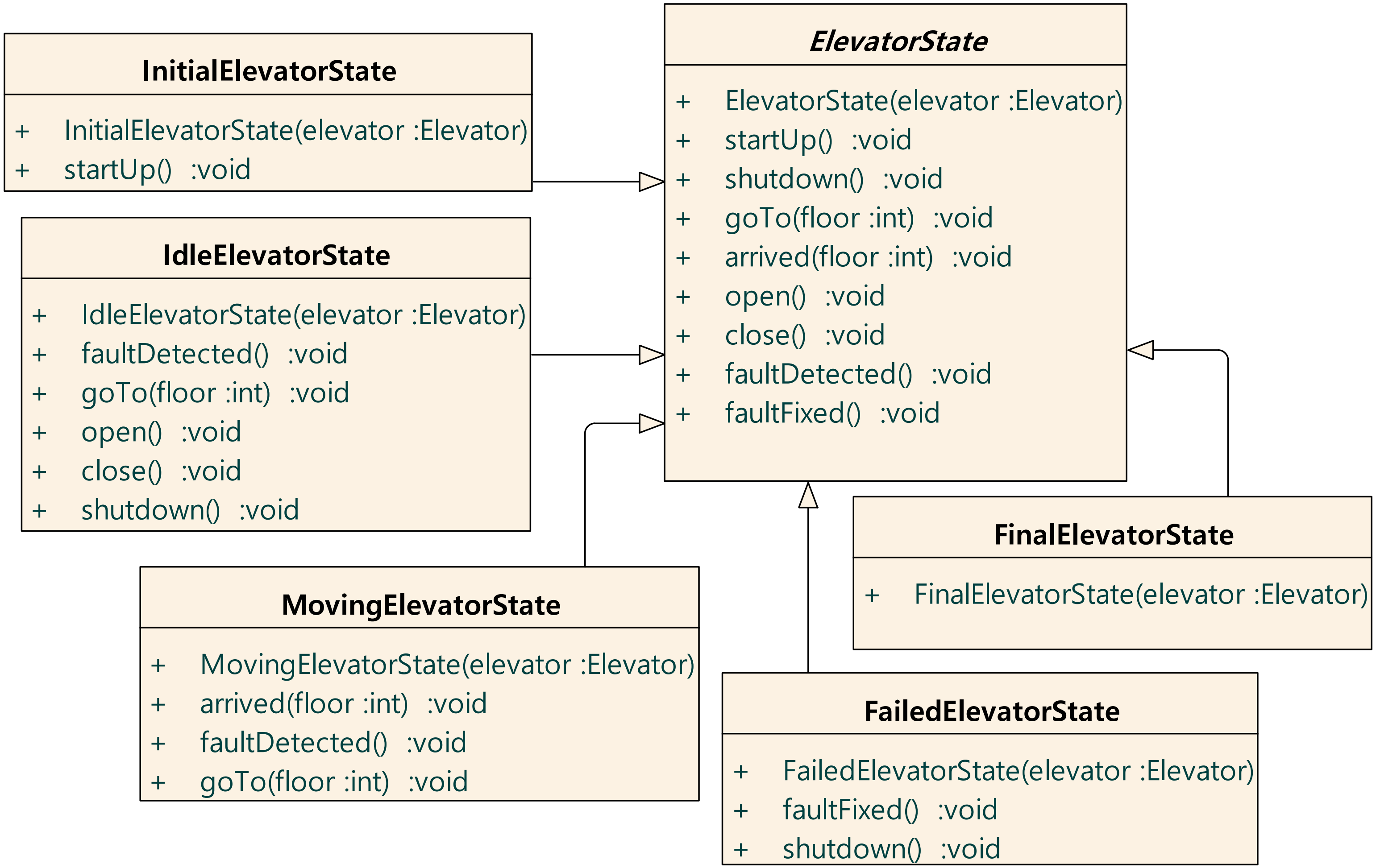
State Pattern: Motivating Example



```
public void goTo(int floor) {
      switch ( curState ) {
         case Idle : {
            destinations.add(floor) ; startMoving() ;
            curState = ElevatorState.Moving ; } break ;
         case Moving : destinations.add(floor) ; break ;
      }
   }
```

```
public void arrived(int floor) {
      if ( curState == ElevatorState.Moving &&
         destinations.contains(new Integer(floor))) {
         stopMoving() ;
         curFloor = floor ;
         destinations.remove(new Integer(floor));
         if ( ! hasDestination() ) curState = ElevatorState.Idle ;
      }
   }
```

State Pattern



InitialElevatorState

+ InitialElevatorState(elevator :Elevator)
+ startUp() :void

IdleElevatorState

+ IdleElevatorState(elevator :Elevator)
+ faultDetected() :void
+ goTo(floor :int) :void
+ open() :void
+ close() :void
+ shutdown() :void

ElevatorState

+ ElevatorState(elevator :Elevator)
+ startUp() :void
+ shutdown() :void
+ goTo(floor :int) :void
+ arrived(floor :int) :void
+ open() :void
+ close() :void
+ faultDetected() :void
+ faultFixed() :void

MovingElevatorState

+ MovingElevatorState(elevator :Elevator)
+ arrived(floor :int) :void
+ faultDetected() :void
+ goTo(floor :int) :void

FinalElevatorState

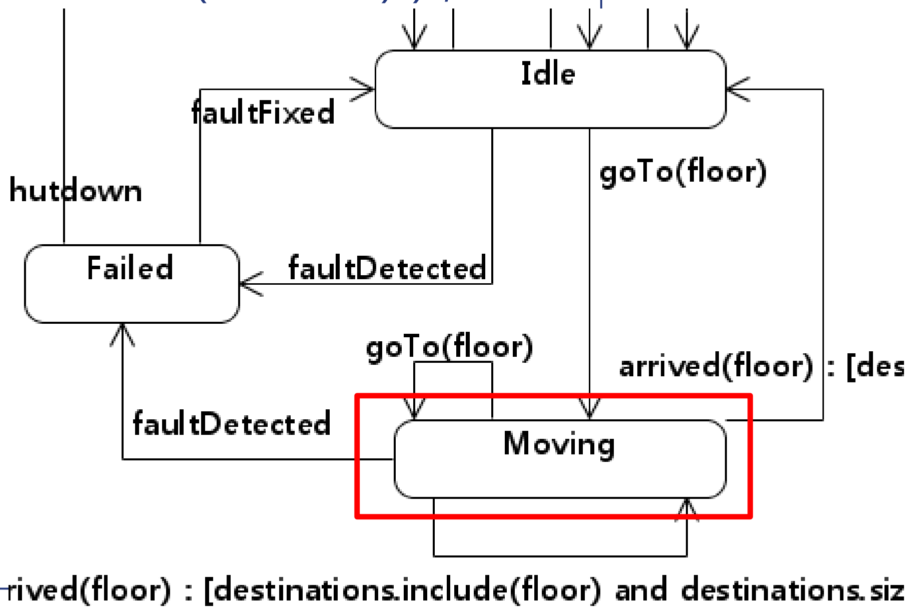
+ FinalElevatorState(elevator :Elevator)

FailedElevatorState

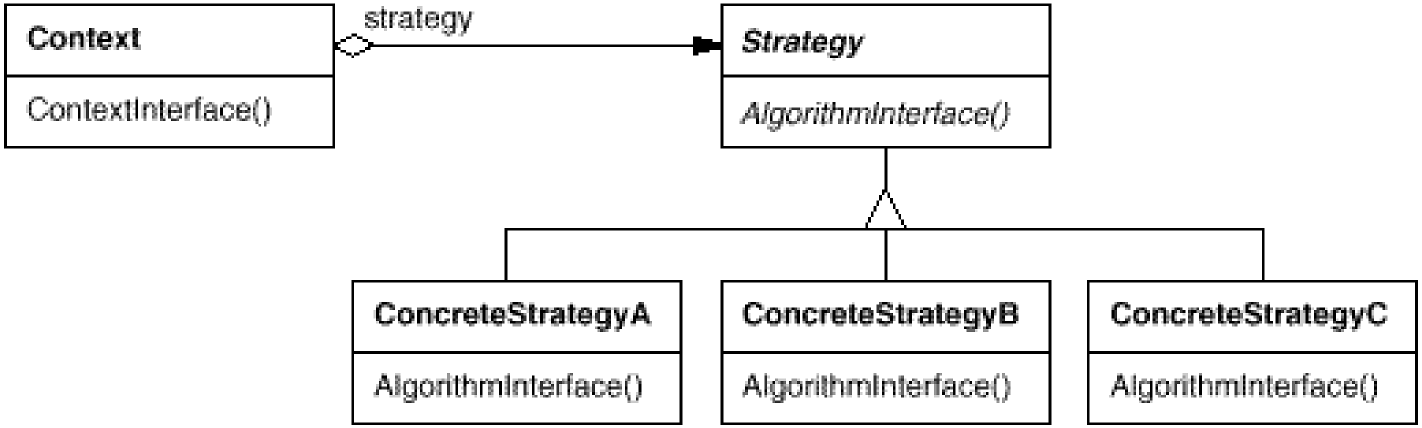
+ FailedElevatorState(elevator :Elevator)
+ faultFixed() :void
+ shutdown() :void

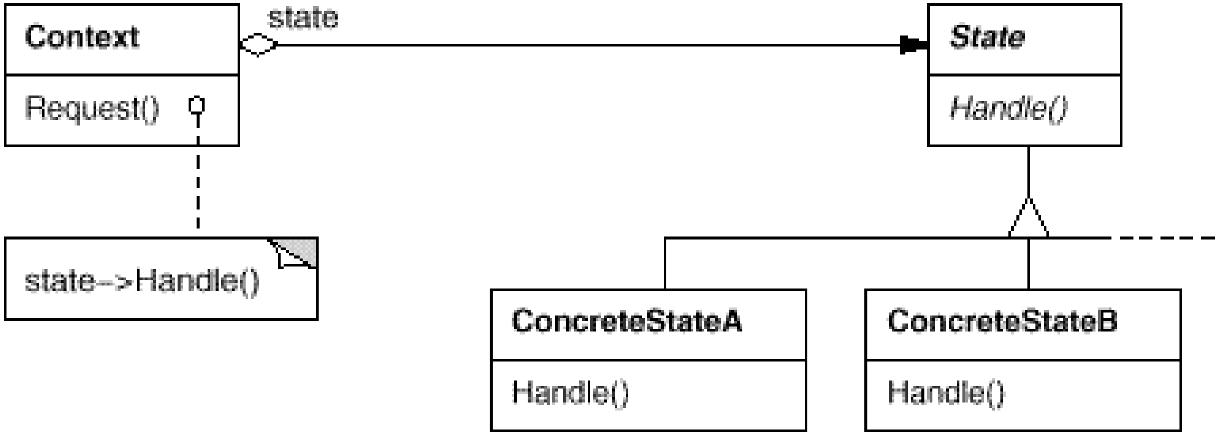
State Pattern

```
public class MovingElevatorState extends ElevatorState {
  public MovingElevatorState(Elevator elevator) { super(elevator); }
  public void arrived(int floor) {
    if ( elevator.isInDestination(floor) ) {
      elevator.stopMoving() ;
      elevator.setCurFloor(floor) ;
      elevator.removeDestination(floor) ;
      if ( ! elevator.hasDestination() )
        elevator.setCurState( new IdleElevatorState(elevator) ) ;
    }
  }
  public void faultDetected() {
    elevator.setCurState(
      new FailedElevatorState(elevator)) ;
  }
  public void goTo(int floor) {
    elevator.addDestination(floor) ;
  }
}
```



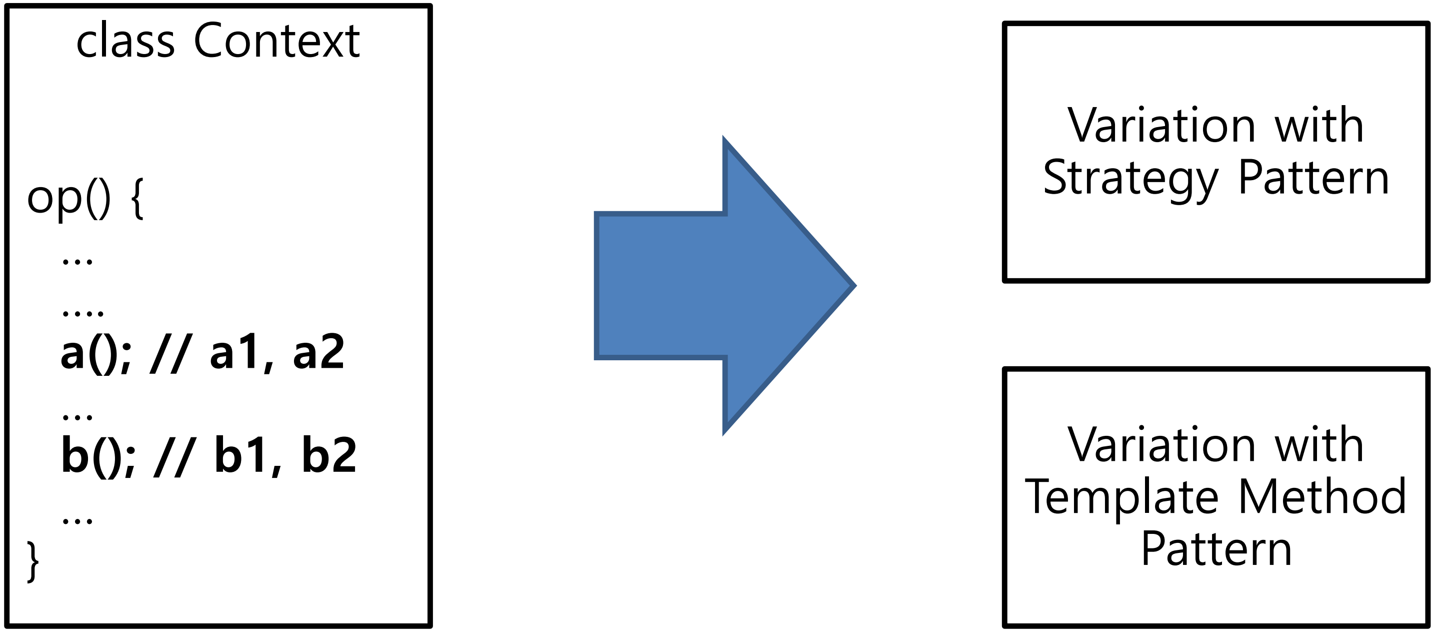
Strategy Pattern vs State Pattern





STRATEGY VS TEMPLATE METHOD

Strategy Pattern vs Template Method Pattern



class Context

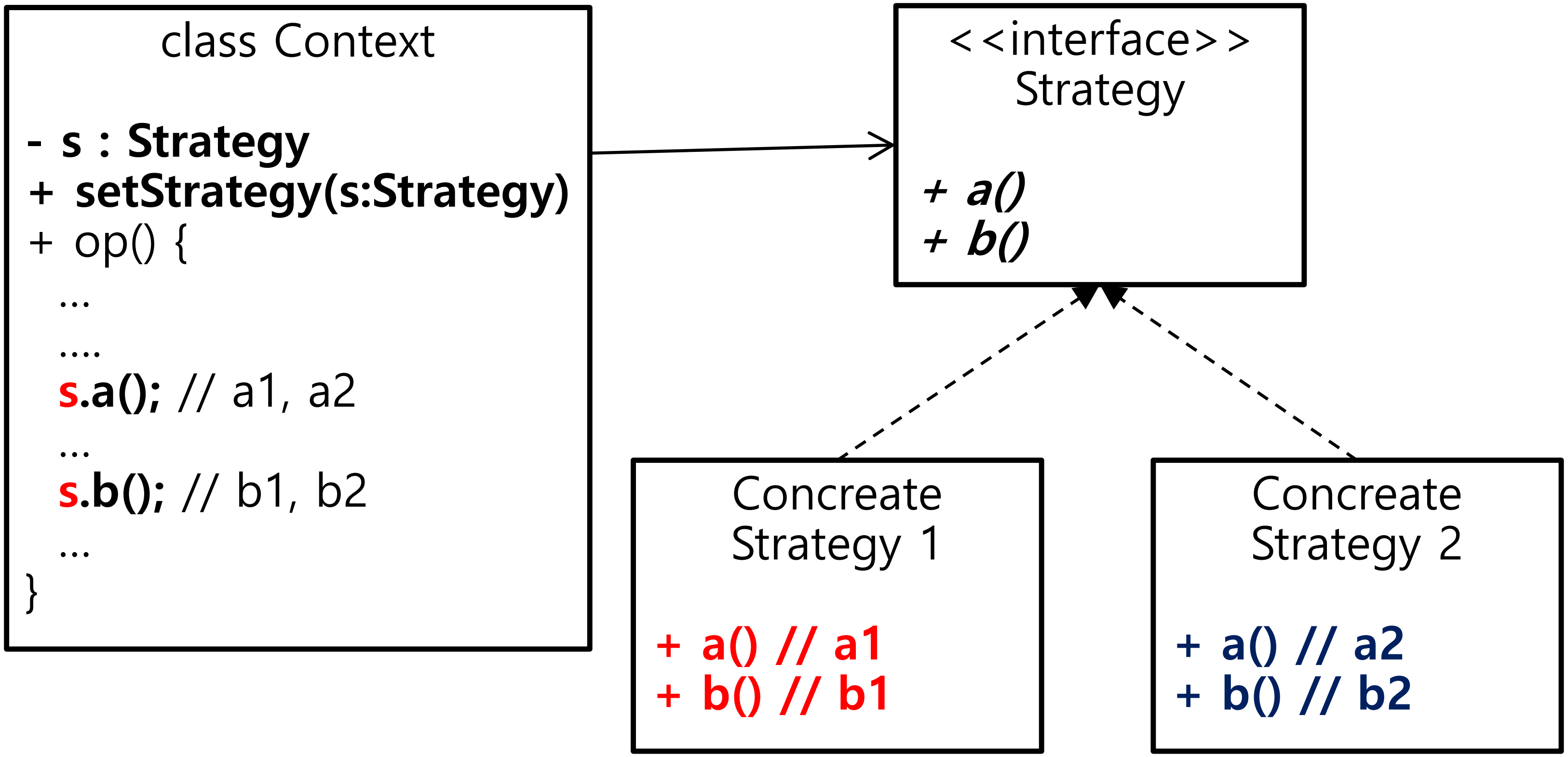
```
op() {
   …
   ….
   a(); // a1, a2
   …
   b(); // b1, b2
   …
}
```

Variation with Strategy Pattern

Variation with Template Method Pattern

Variation with Strategy Pattern

- Implement the variation with strategies



class Context

```
- s : Strategy
+ setStrategy(s:Strategy)
+ op() {
   …
   ….
   s.a(); // a1, a2
   …
   s.b(); // b1, b2
   …
}
```

<<interface>>
Strategy

+ a()
+ b()

Concreate
Strategy 1

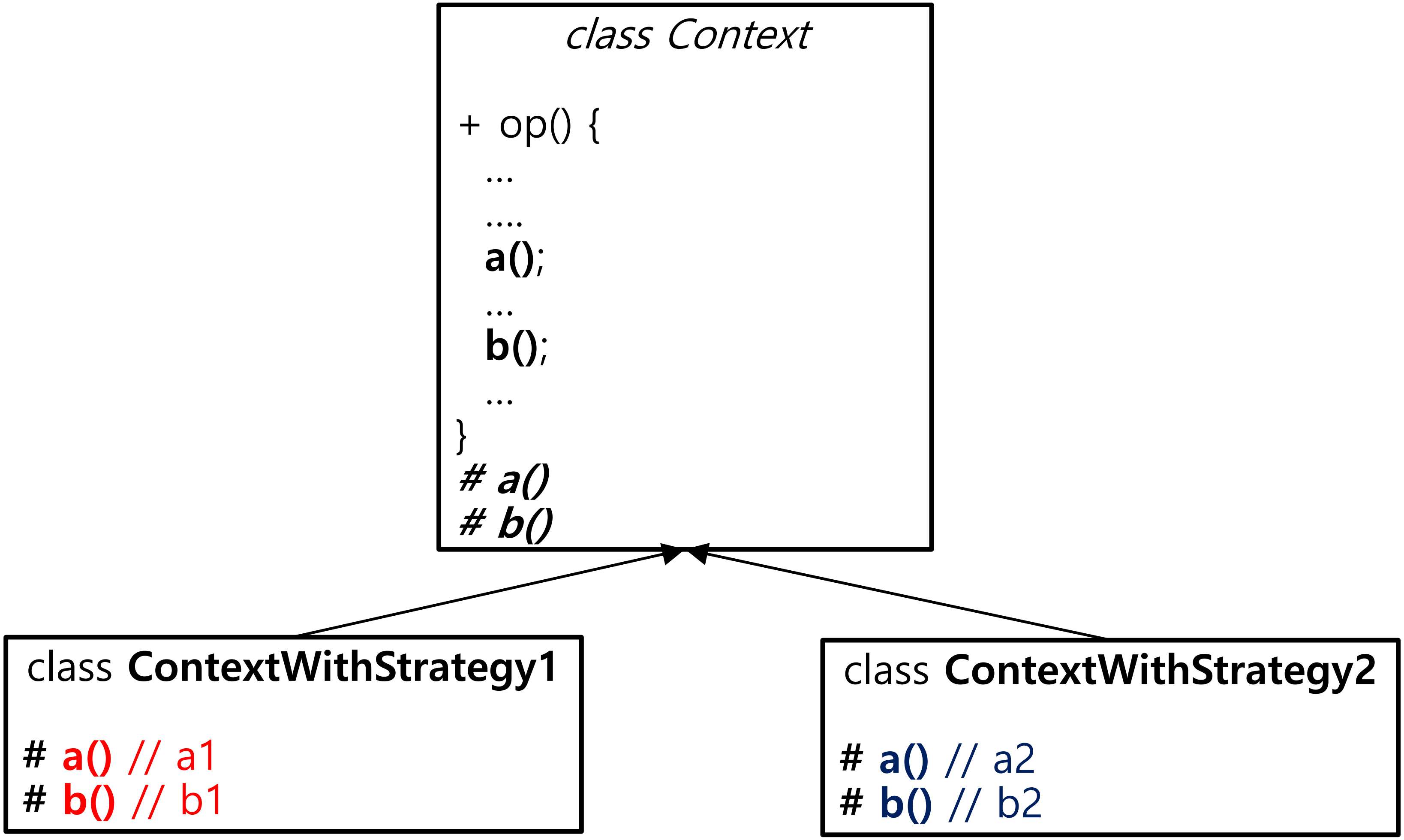
+ a() // a1
+ b() // b1

Concreate
Strategy 2

+ a() // a2
+ b() // b2

Variation with Template Method Pattern

- ❖ Implement the variation with subclasses



class Context

```
+ op() {
   …
   ….
   a();
   …
   b();
   …
}
```

# a()
# b()

class **ContextWithStrategy1**

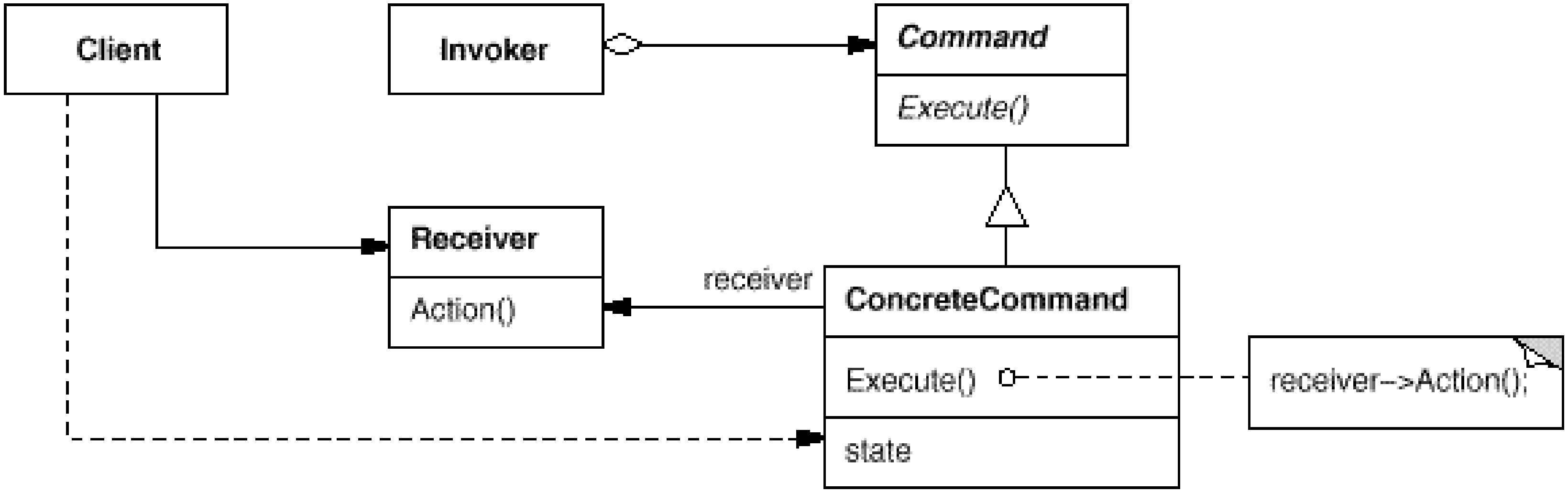
# **a()** // a1
# **b()** // b1

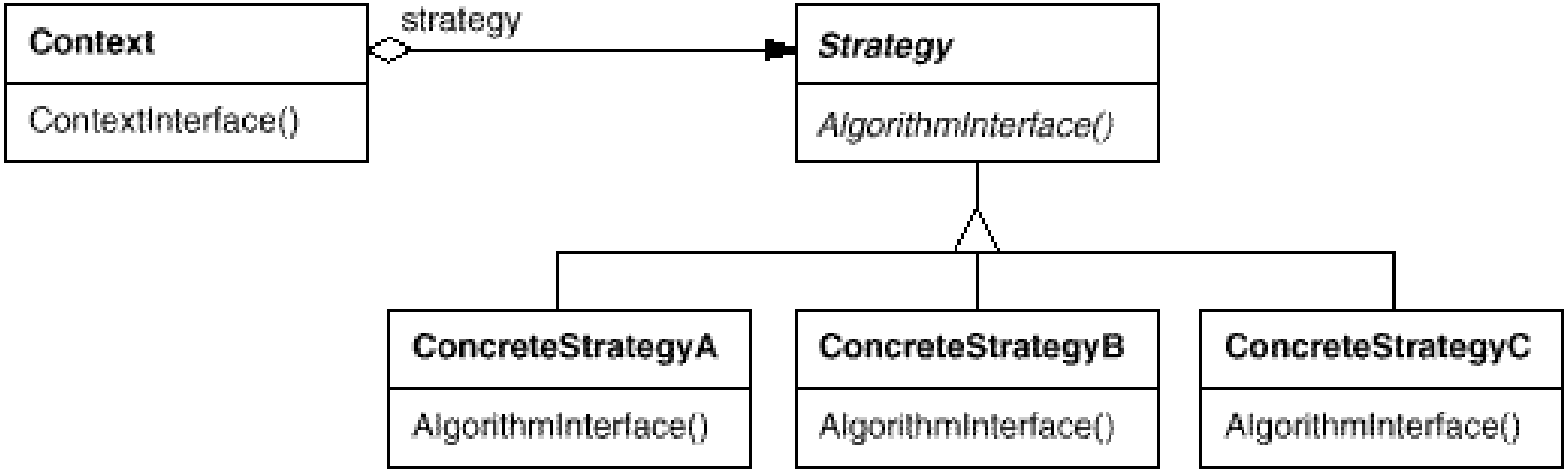
class **ContextWithStrategy2**

# **a()** // a2
# **b()** // b2

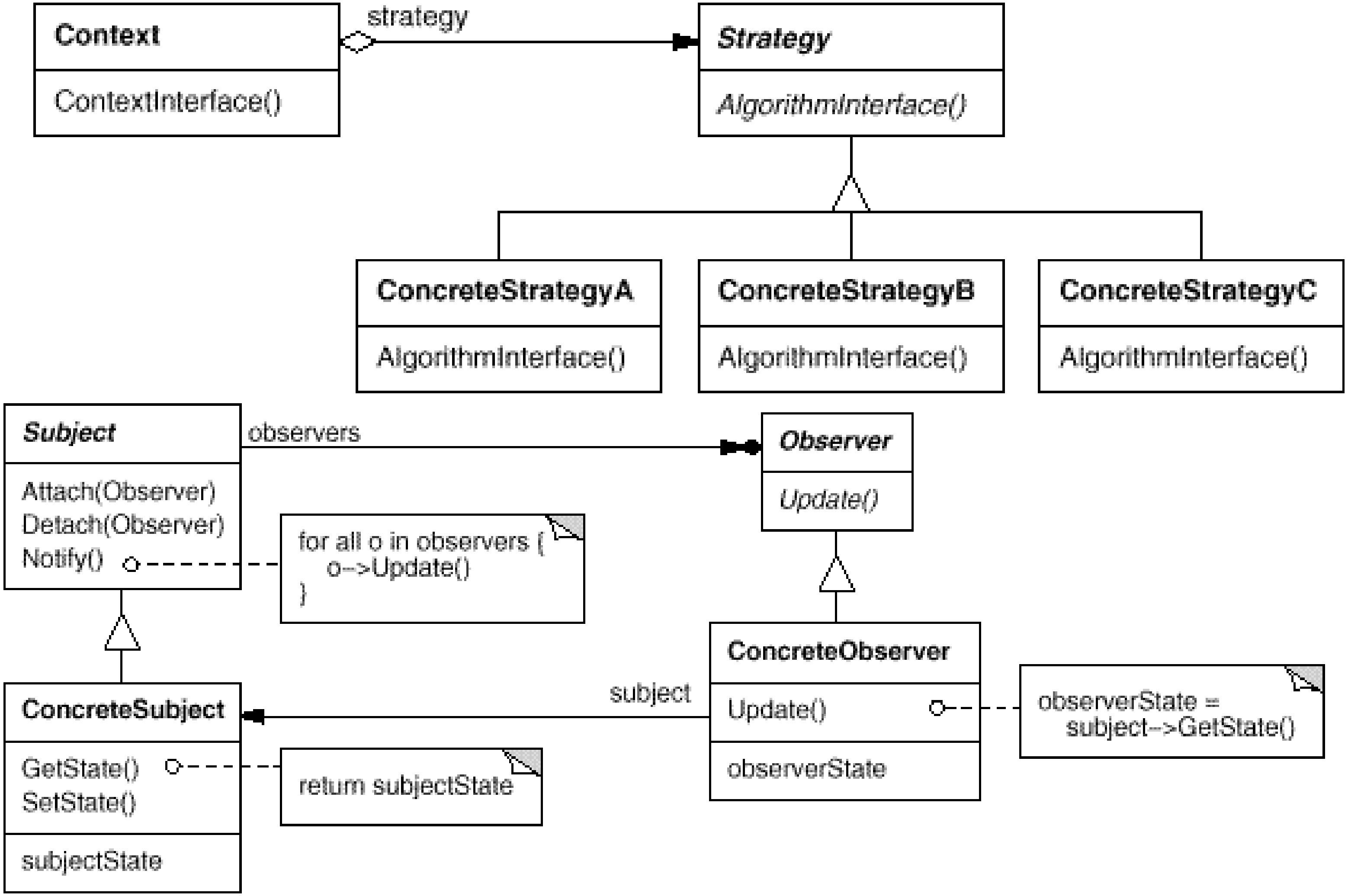
STRATEGY VS COMMAND VS OBSERVER

Command vs Strategy



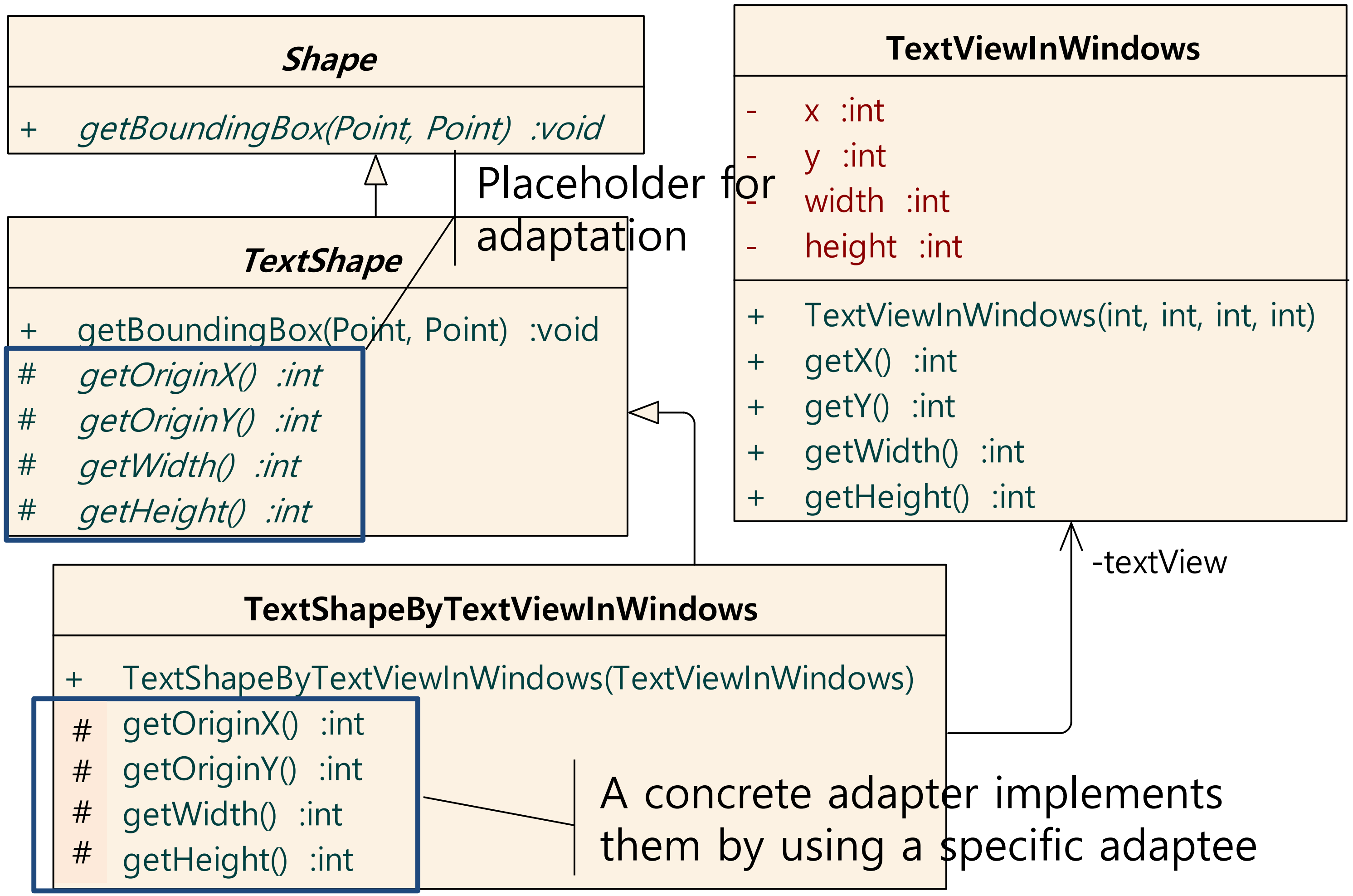


Strategy vs Observer

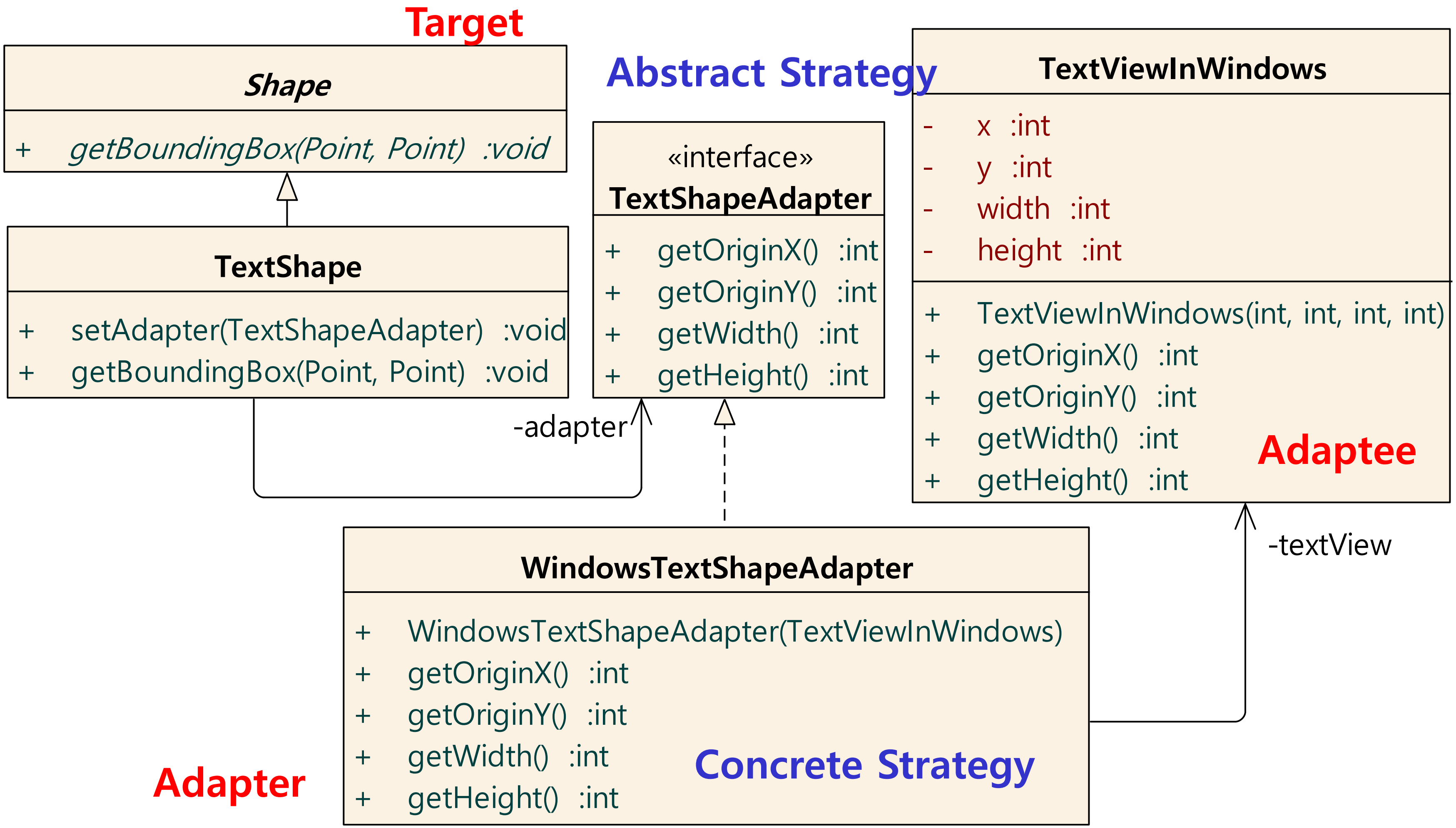


COMBINED USES OF PATTERNS

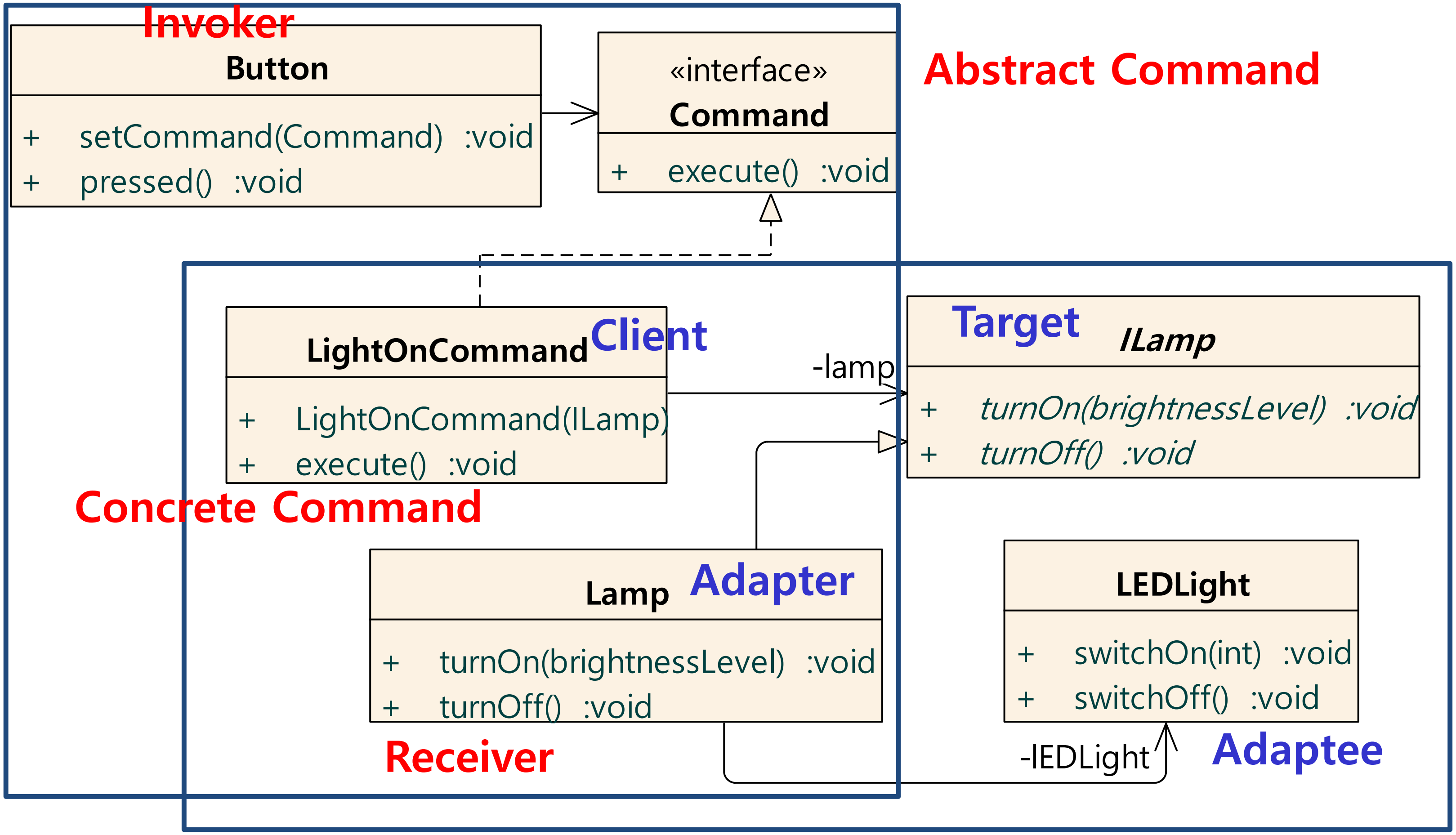
Adapter and Template Method



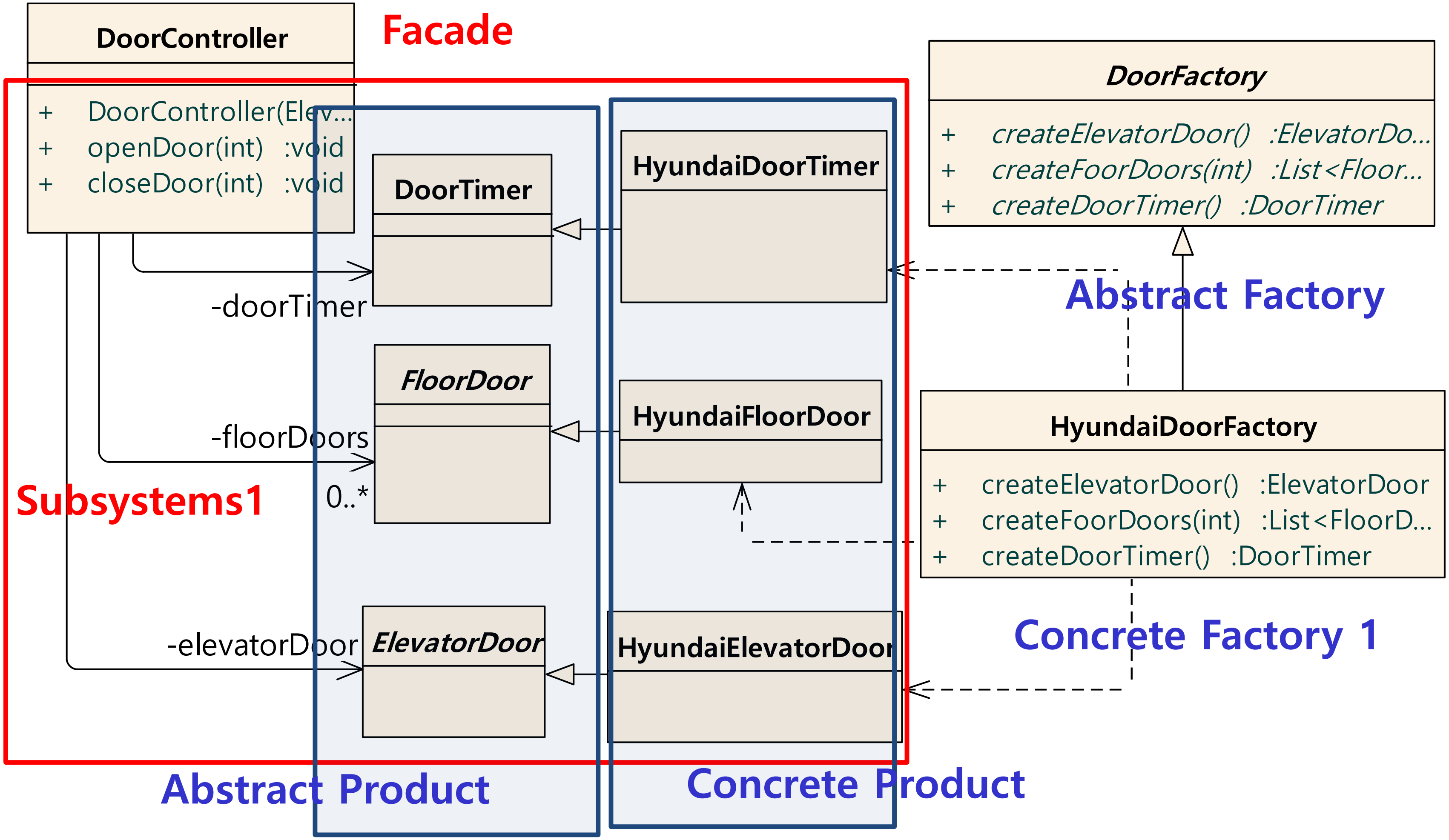
Adapter and Strategy



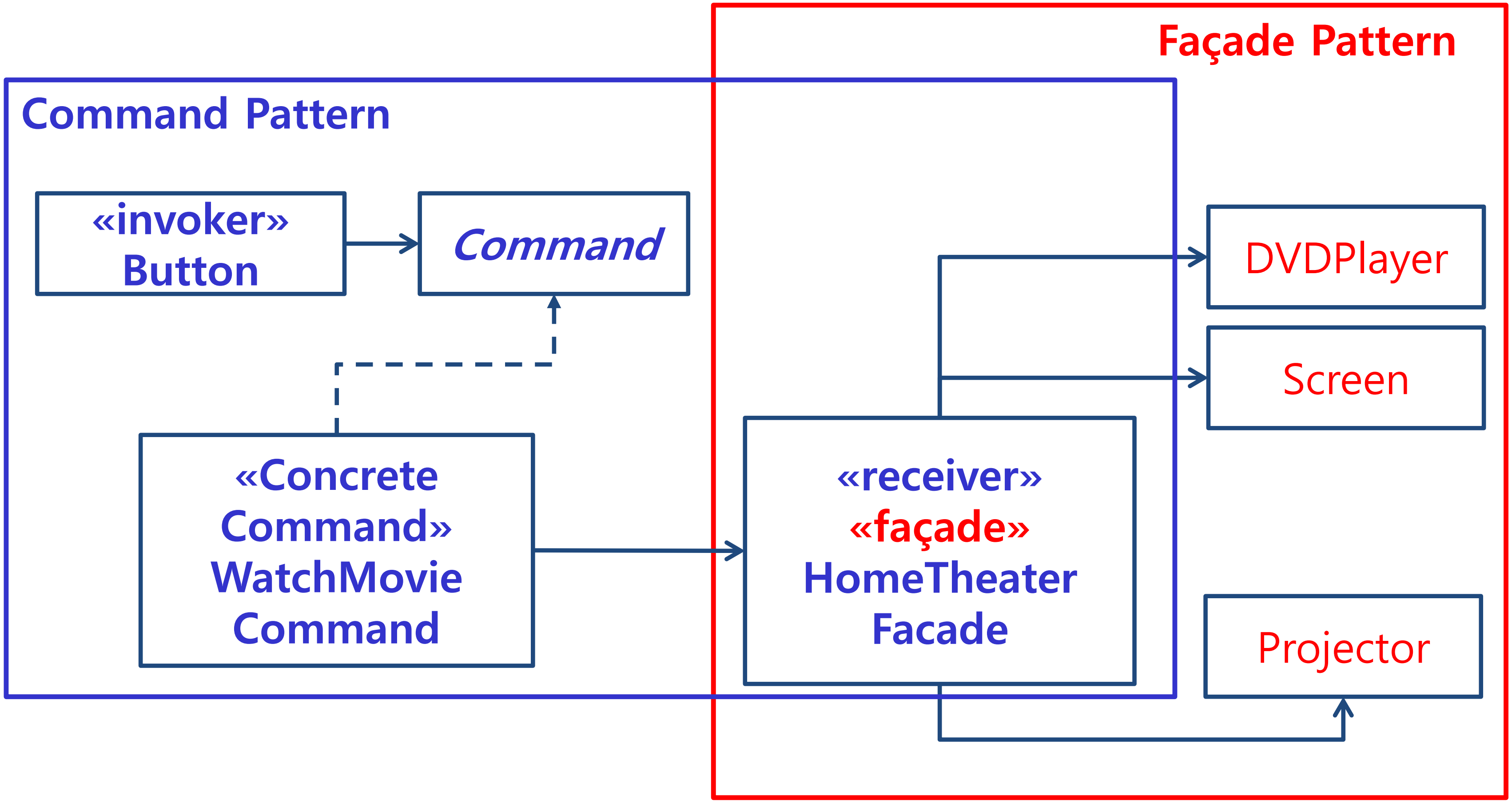
Command and Adapter



Façade and Abstract Factory



Command and Façade



Q&A