Wrap-up

Review

- ❖ AA 교육과정에서 가장 기억나는 용어는?

면접 Review

- SOLID
- SRP와 응집도
- Component vs module

- 품질 모델
- ADD
- Tactics

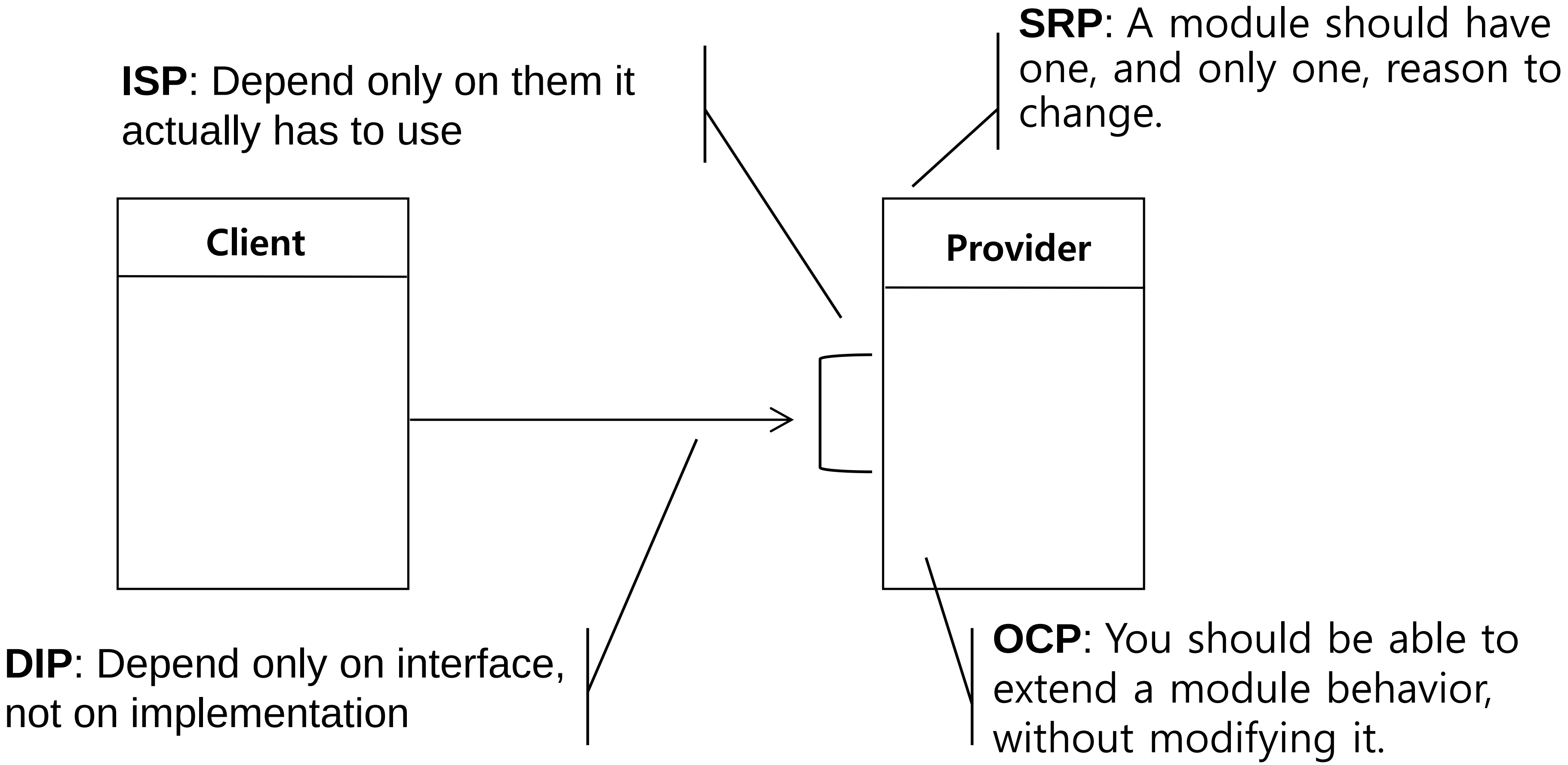
- Architecture style vs Design pattern
- N-Tier vs layer
- Dispatcher vs Broker
- Data-flow patterns
- MSA

- Strategy pattern vs Template method pattern
- Strategy pattern vs Factory method pattern
- Strategy pattern vs State pattern

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	Subclasses must be substitutable for their superclass.	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



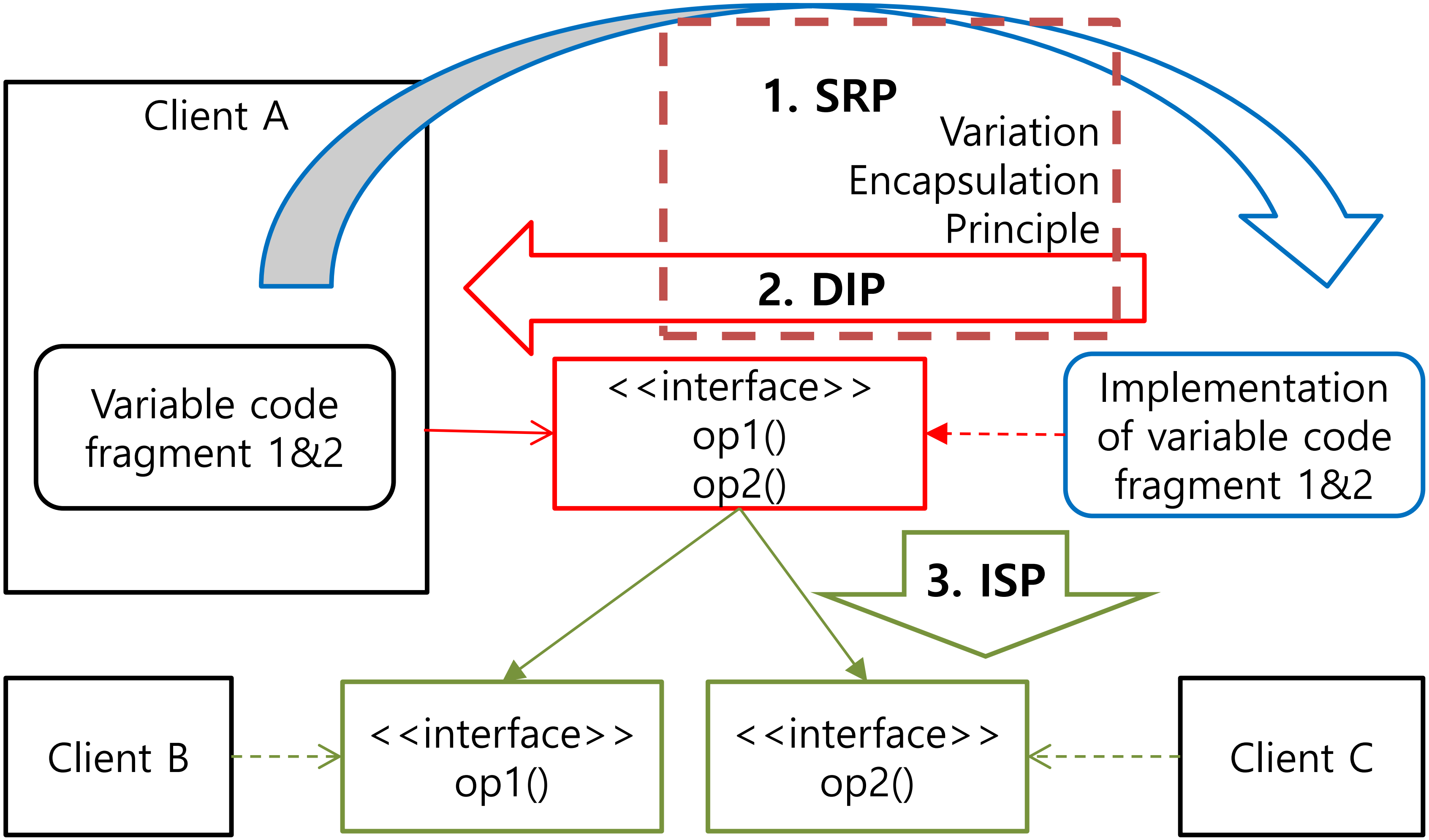
ISP: Depend only on them it actually has to use

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

응집도

❖ Strength of **functional relatedness** of elements within a module

```
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;
}
```

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

SRP

- ❖ A class should have only one reason to change

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

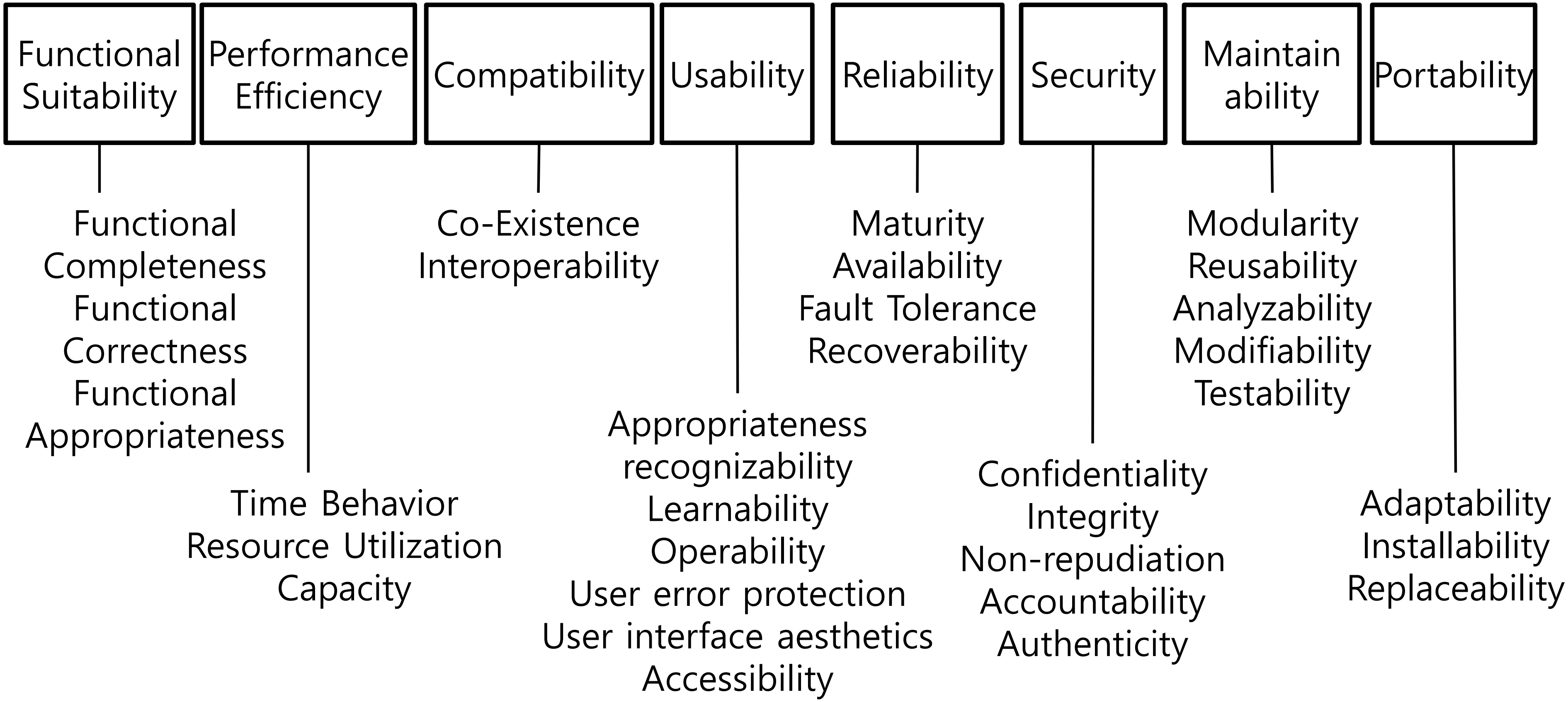
```
T getSum(const list<T>& values, int s, int e, bool(*select)(const T) ) {
    T sum = 0;
    for (unsigned int i = s; i <= e; i++)
        if ( select(values[i]) )
            sum += values[i];
    return sum;
}
```

Package vs. Module vs. Component

	Package	Module	Component
Role	namespace	Functional Implementation	
Runtime instance	No	No	Yes
Multiple instances	No	No	Yes
Encapsulation	No	Yes	Yes
Communicates via	X	interface	Port/connector

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

ISO/IEC 25010:2011 Quality Model



- Functional Suitability
  - Functional Completeness
  - Functional Correctness
  - Functional Appropriateness
- Performance Efficiency
  - Time Behavior
  - Resource Utilization
  - Capacity
- Compatibility
  - Co-Existence
  - Interoperability
- Usability
  - Appropriateness recognizability
  - Learnability
  - Operability
  - User error protection
  - User interface aesthetics
  - Accessibility
- Reliability
  - Maturity
  - Availability
  - Fault Tolerance
  - Recoverability
- Security
  - Confidentiality
  - Integrity
  - Non-repudiation
  - Accountability
  - Authenticity
- Maintain ability
  - Modularity
  - Reusability
  - Analyzability
  - Modifiability
  - Testability
- Portability
  - Adaptability
  - Installability
  - Replaceability

ISO/IEC 25010:2011 Systems and software engineering -- Systems and software Quality Requirements and Evaluation (SQuaRE) -- System and software quality models

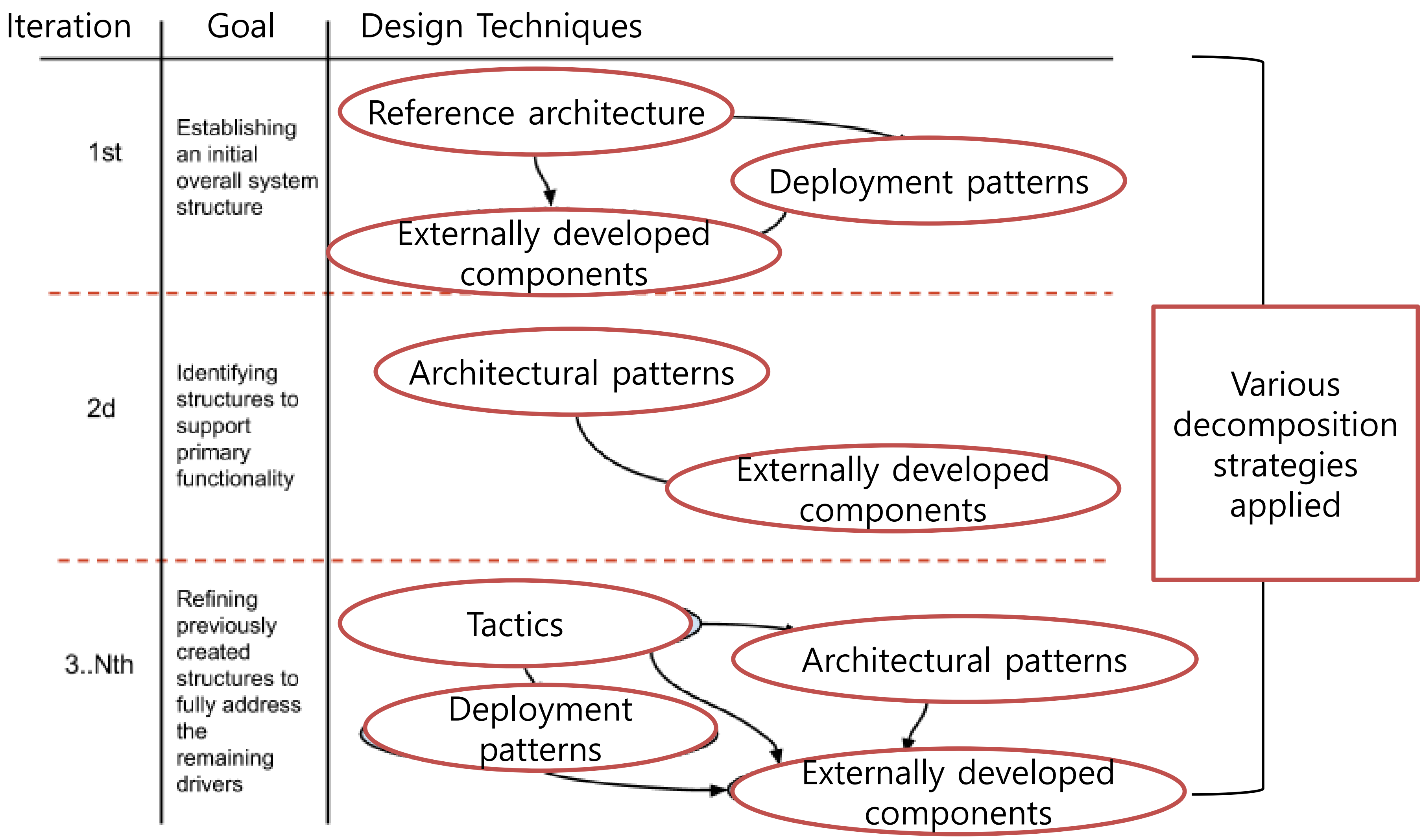
ADD

❖ ADD(Attribute-Driven Design)

- Small systems usually focus their attention on functionality, while larger systems must pay more attention to achieving quality attributes.
- The larger the system is, the more likely it is that components will have stringent quality attribute requirements
- The Attribute Driven Design (ADD) process describes how quality attributes can be used to drive recursive design of components

1. Choose the system/component to decompose
2. Refine the system/component
    a) Choose the architecture drivers (Use case, QA)
    b) Choose or invent a suitable architecture pattern
    c) Create components and allocate **responsibilities**
    d) Define component interfaces
    e) Verify functionality scenarios and QA scenarios (by Sequence diagrams)
3. Repeat for every component (Component-level Design)

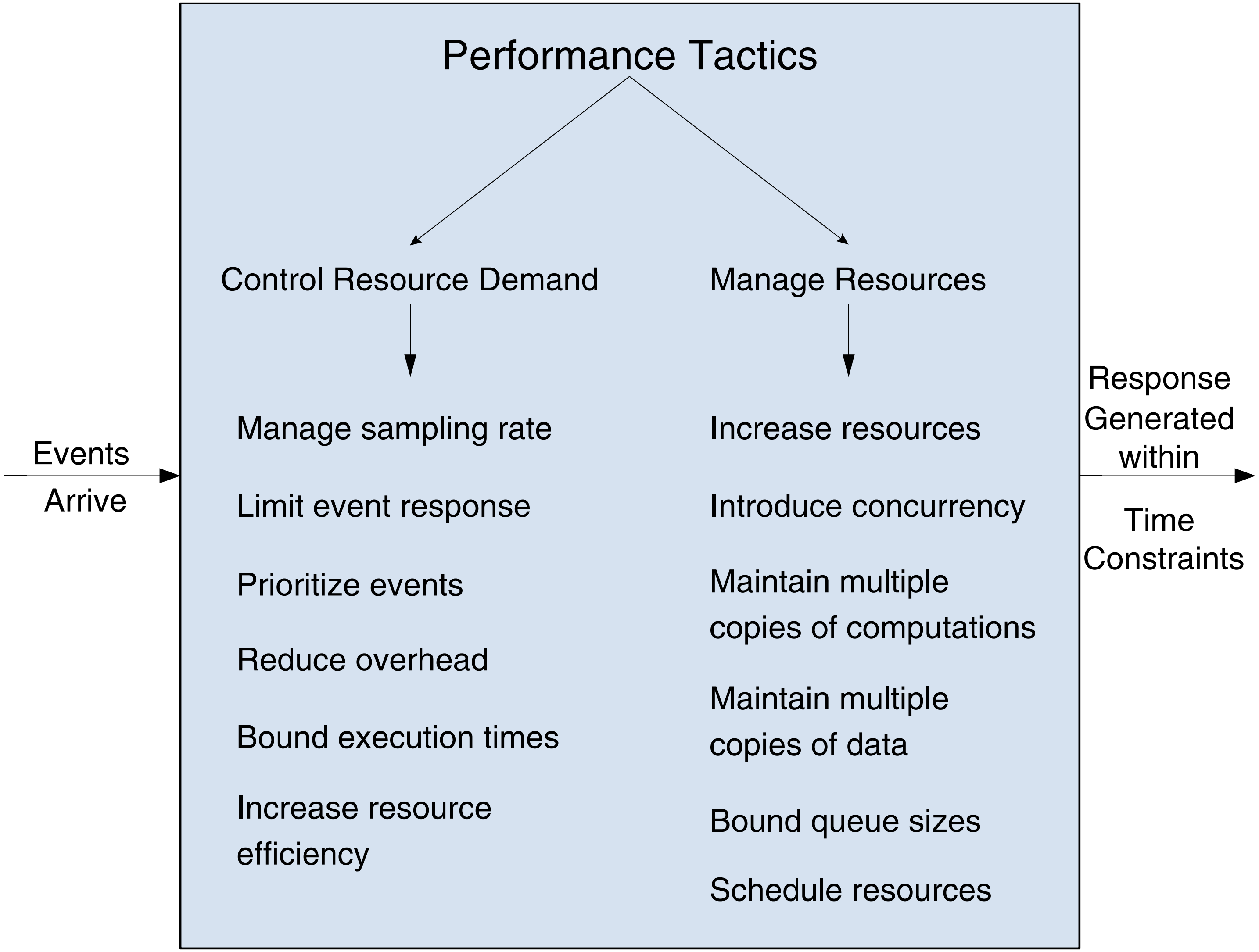
ADD: Design of greenfield systems for mature domains



Iteration	Goal	Design Techniques
1st	Establishing an initial overall system structure	Reference architecture Deployment patterns Externally developed components
2d	Identifying structures to support primary functionality	Architectural patterns Externally developed components
3..Nth	Refining previously created structures to fully address the remaining drivers	Tactics Architectural patterns Deployment patterns Externally developed components

Various decomposition strategies applied

Performance Tactics



Performance Tactics

Control Resource Demand

- Manage sampling rate
- Limit event response
- Prioritize events
- Reduce overhead
- Bound execution times
- Increase resource efficiency

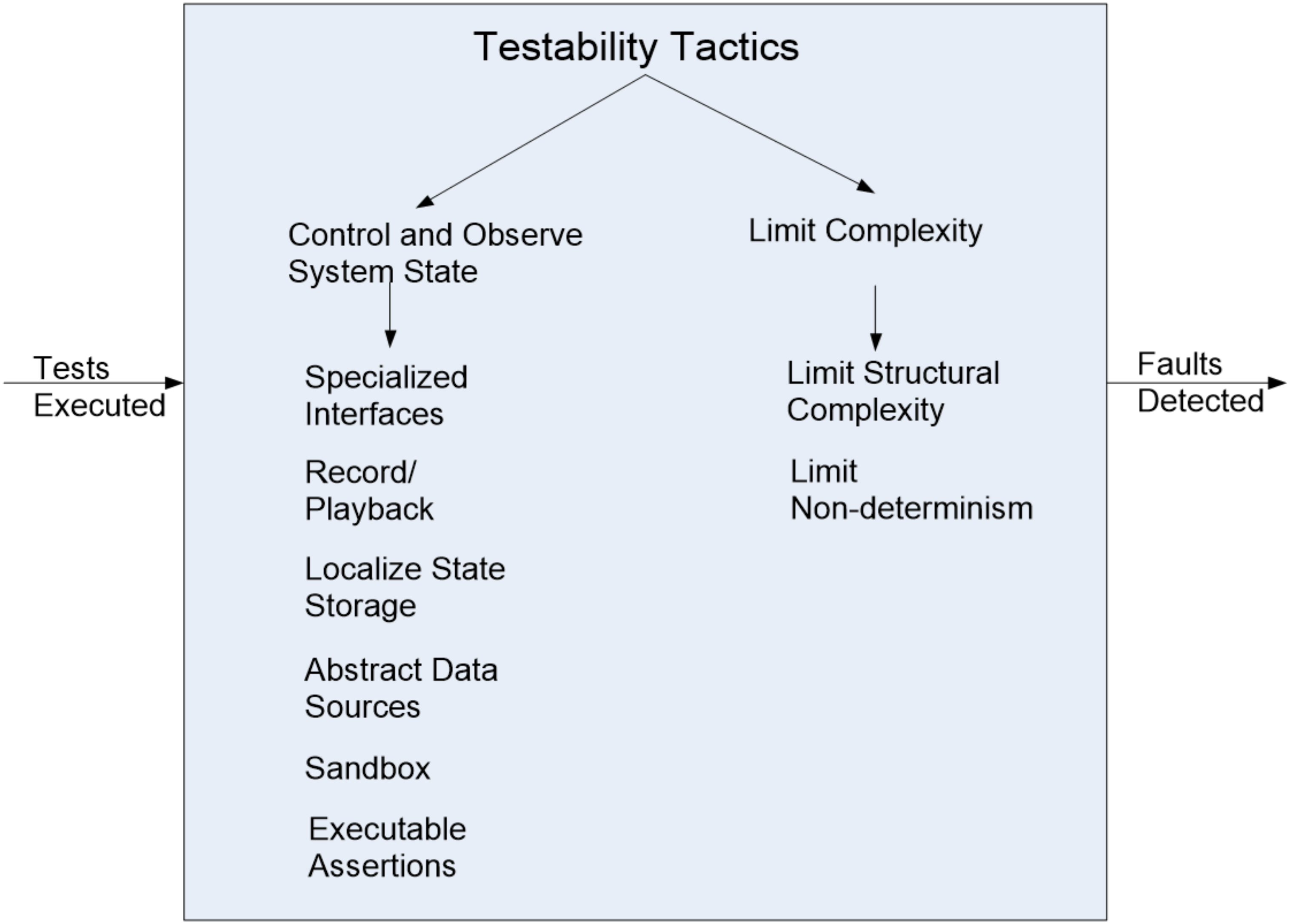
Manage Resources

- Increase resources
- Introduce concurrency
- Maintain multiple copies of computations
- Maintain multiple copies of data
- Bound queue sizes
- Schedule resources

Events Arrive

Response Generated within Time Constraints

Testability Tactics



Testability Tactics

Tests Executed

Control and Observe System State

- Specialized Interfaces
- Record/ Playback
- Localize State Storage
- Abstract Data Sources
- Sandbox
- Executable Assertions

Limit Complexity

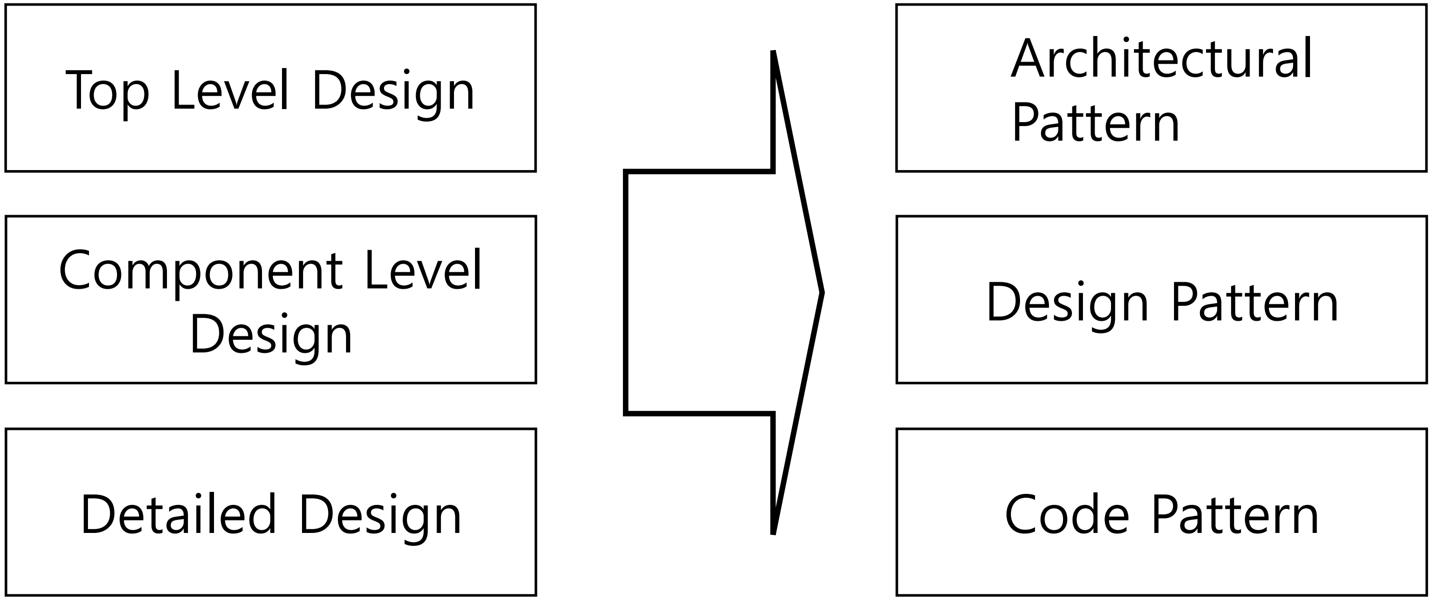
- Limit Structural Complexity
- Limit Non-determinism

Faults Detected

Patterns

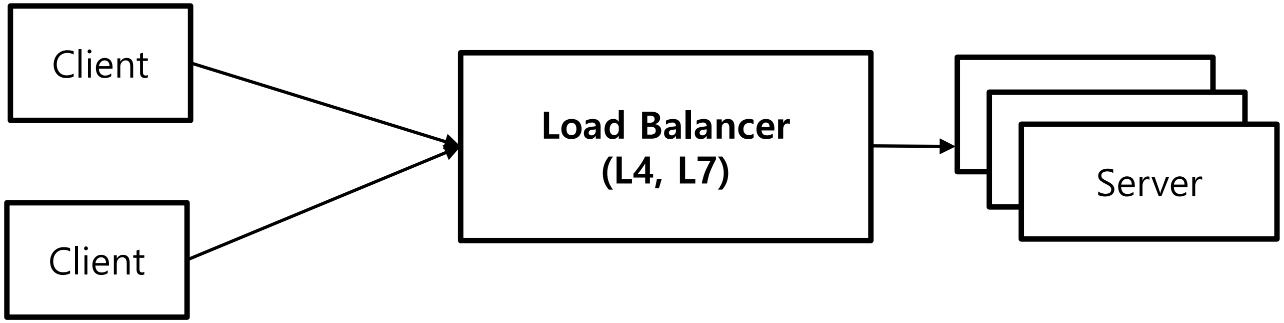
- Patterns are an well established solution to a recurring problem.

> Patterns help you learn from other's successes,
> instead of your own failures
> **Mark Johnson** (cited by B. Eckel)



Load Balancer

- **Load balancing** refers to efficiently distributing incoming network traffic across a group of backend servers, also known as a server farm or server pool.
  - Provides the scalability by distributing client requests efficiently across multiple servers
  - Ensures high availability by sending requests only to servers that are online
  - Provides the flexibility to add or subtract servers as demand dictates



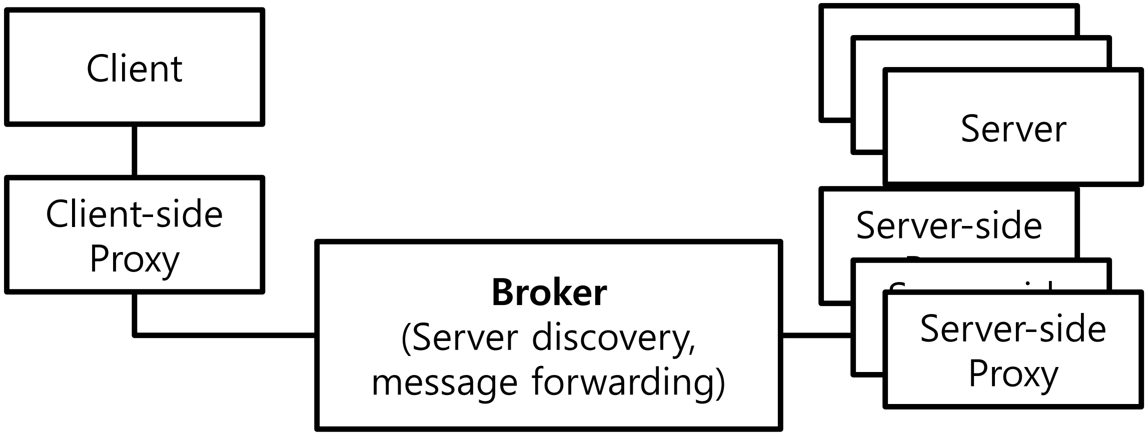
- **Layer 4 load balancer** bases the routing decision on the source and destination IP addresses and ports
- **Layer 7 load balancers** base the routing decision on various characteristics of the HTTP header and on the actual contents of the message, such as the URL, the type of data (text, video, graphics), or information in a cookie.

https://www.nginx.com/resources/glossary/load-balancing/

Broker Pattern

❖ Concept

- need a communication infrastructure that shields applications from the complexities of component location and IPC



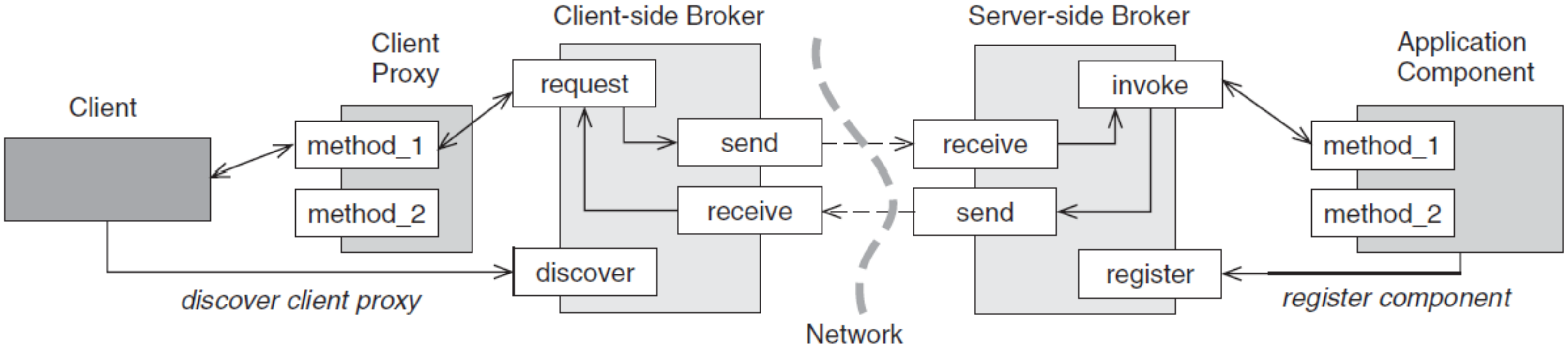
✓ so that service users <u>do not need to know the nature and location of service providers</u>, ➔ location transparency(modifiability: defer binding tactic)

✓ making it easy to <u>dynamically change the bindings between users and providers</u>? ➔ load balancing(scalability) and fail-over(availability)

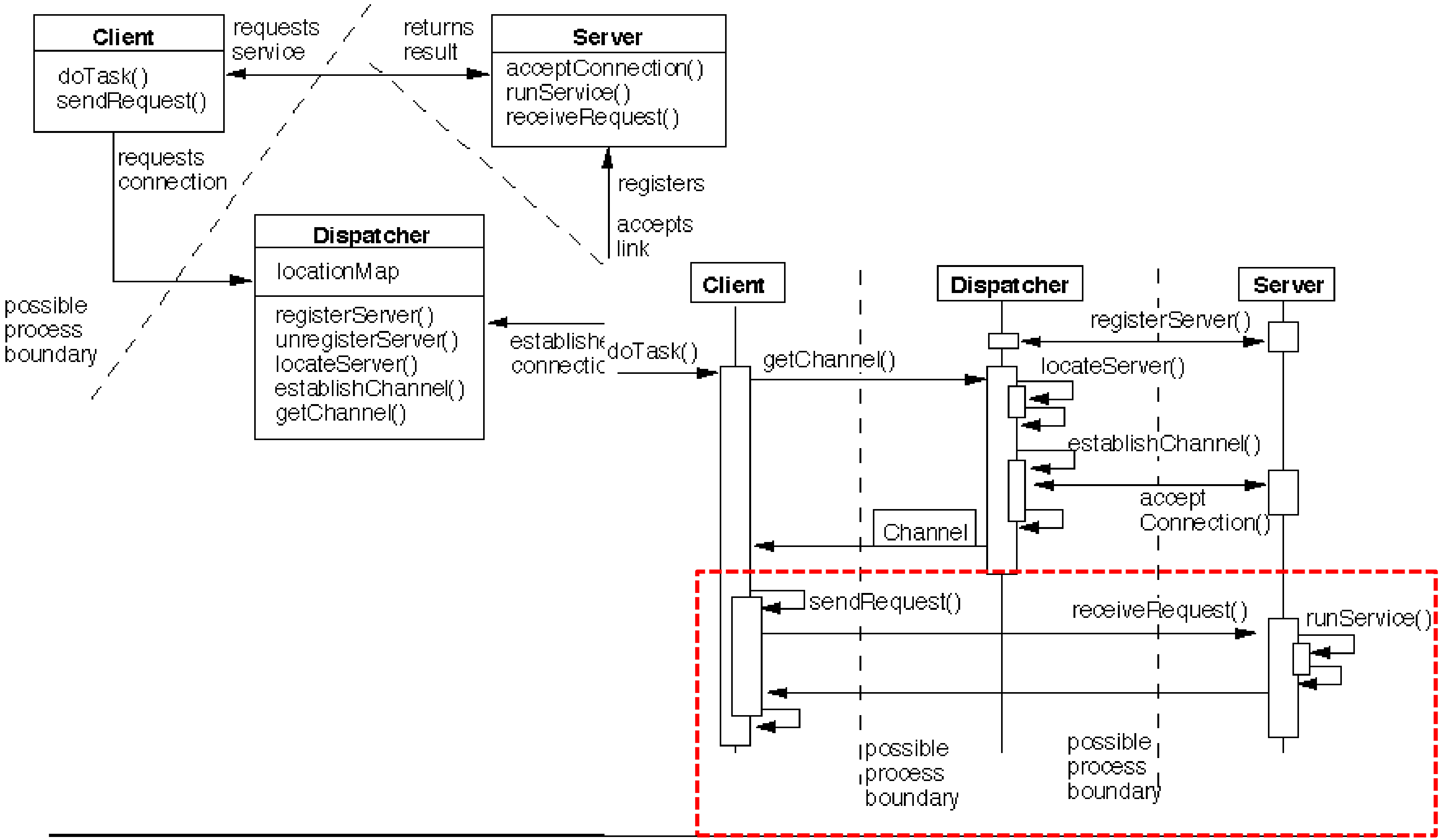
Broker Pattern

❖ Behavior

- When a client needs a service, it queries a broker via a service interface.
- The broker then forwards the client's service request to a server, which processes the request.
- The service result is communicated from the server back to the broker, which then returns the result (and any exceptions) back to the requesting client



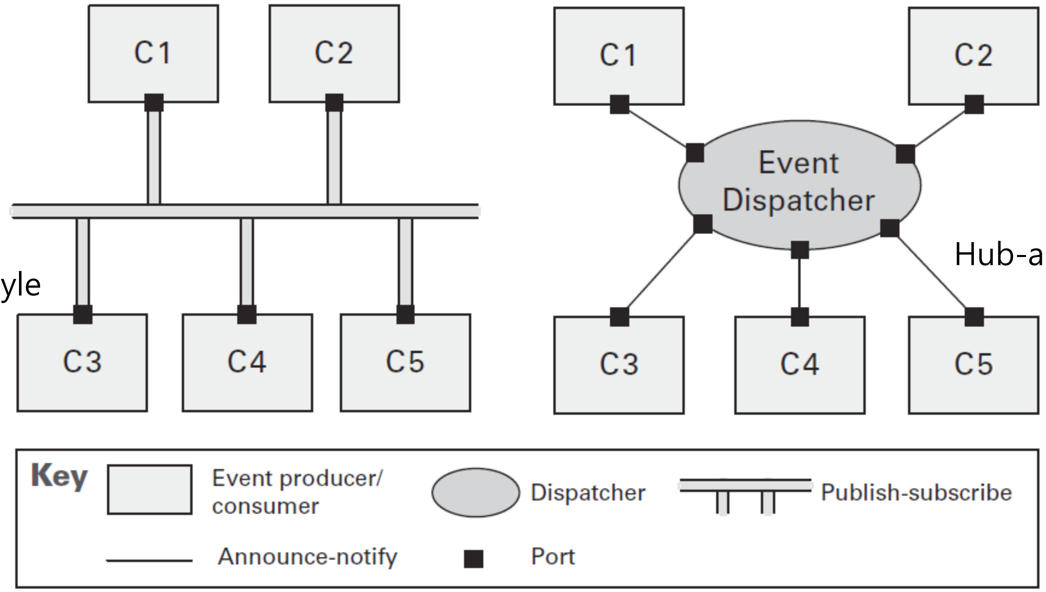
Client-Dispatcher-Server Pattern



Publish-Subscribe Pattern

- Components in some distributed applications are loosely coupled and operate largely independently.

- **Notification mechanism** is needed to inform the components about **state changes or other interesting events** that affect or coordinate their own computation.



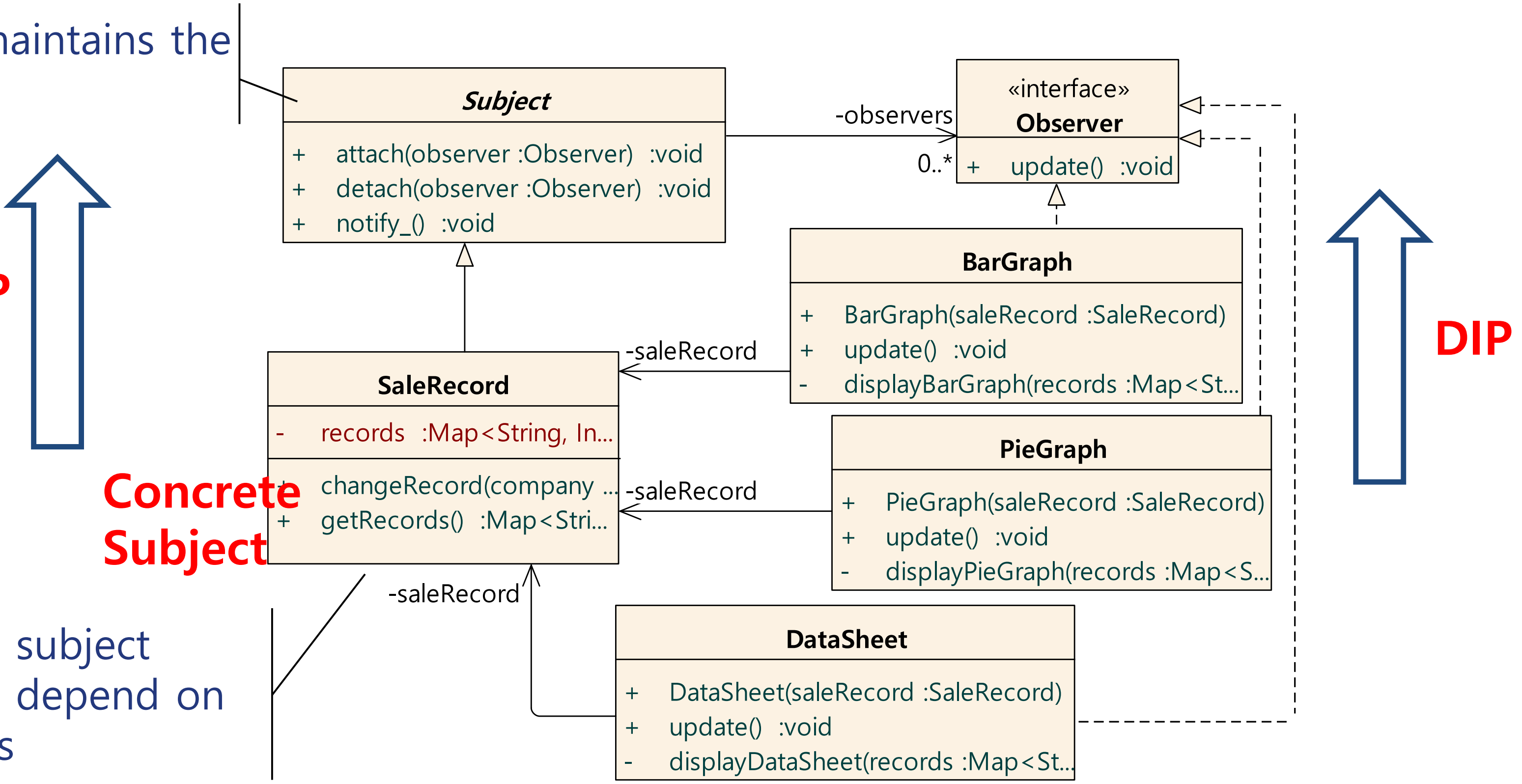
Message bus style

Hub-and-spoke style

Observer Pattern

❖ Define a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically.

Subject maintains the observers


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

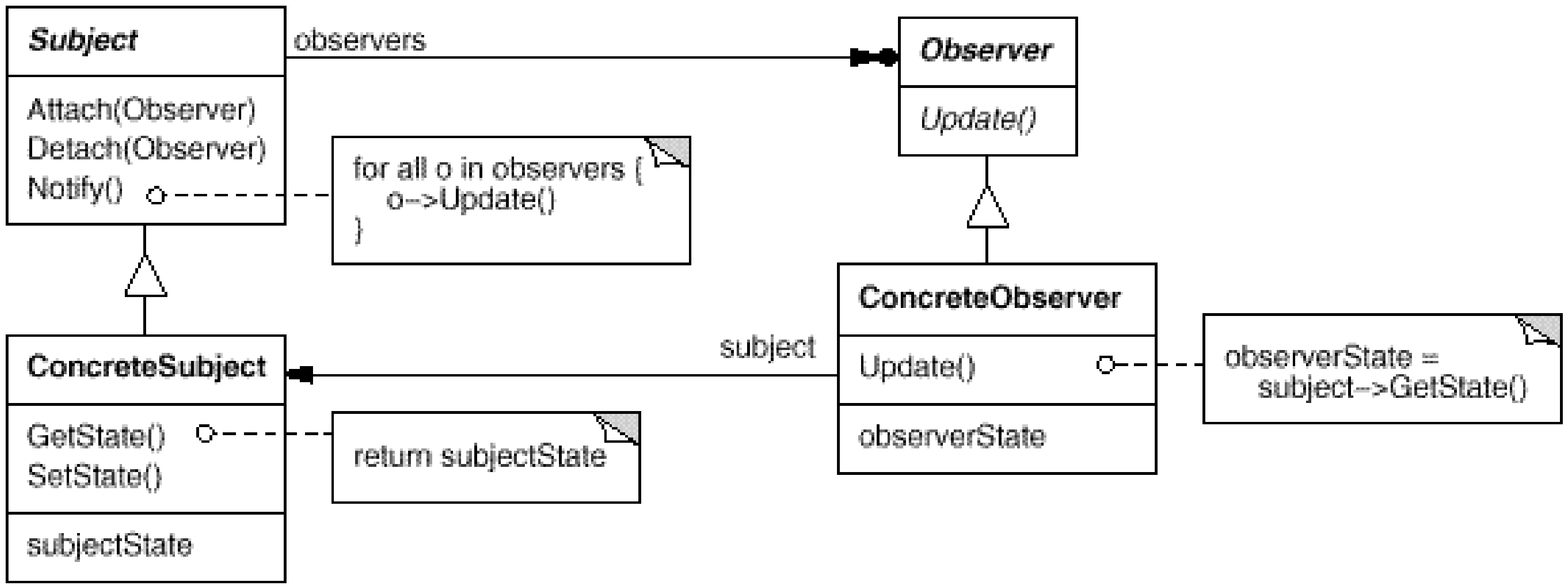
SRP

DIP

Concrete Subject

Concrete subject does not depend on observers

Observer Pattern



Publish-Subscribe Pattern vs Observer Pattern

- In **Publisher/Subscriber** pattern, components are loosely coupled as opposed to **Observer** pattern.
  - In the Observer pattern, Observers are aware of the Subject, but Publish-Subscribe does not require Subjects and Observers to know about each other because they communicate through a central broker.

- **Publish-Subscribe** is asynchronous, while **Observer** is synchronous
  - The use of message queues/passing as the primary implementation mechanism for Publish-Subscribe vs. a direct call with the Observer pattern.

- **Publish-Subscribe** is primarily used for communications in distributed systems, but **Observer** is used for communicating objects within a system.

N-Tier(Multi-tier) vs Layer

- N-tier architecture is a client-server architecture concept where the presentation, processing and data management functions are both logically and **physically separated**

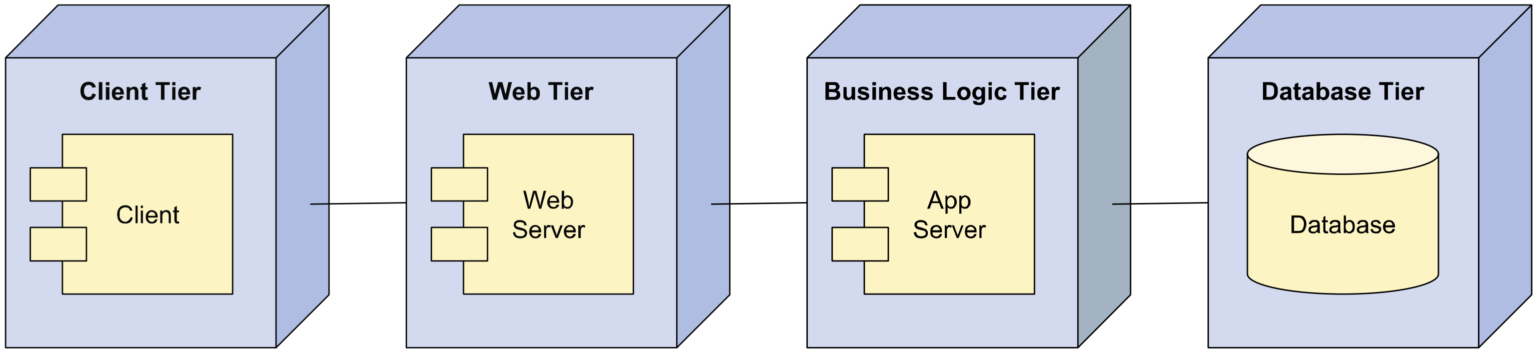
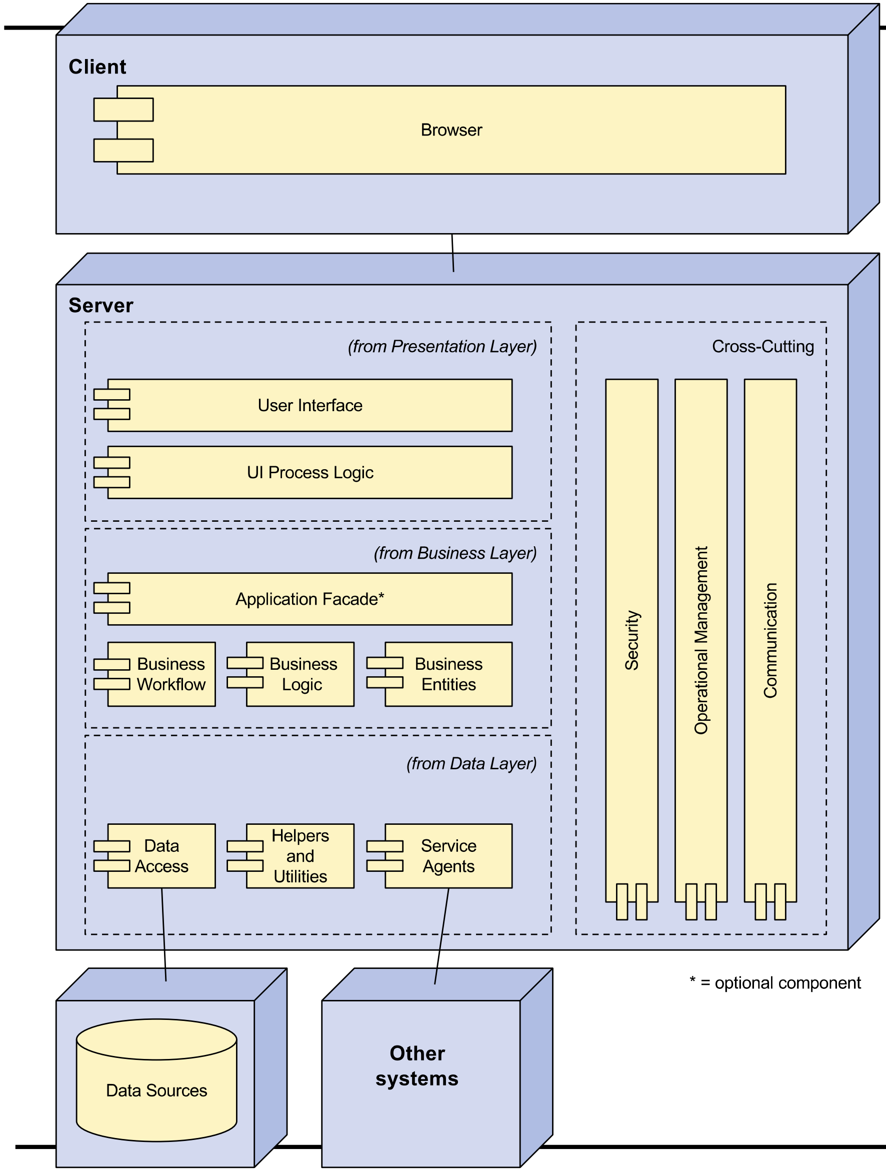


FIGURE 2.6 Four-tier deployment pattern from the *Microsoft Application Architecture Guide* (Key: UML)

N-Tier vs Layer



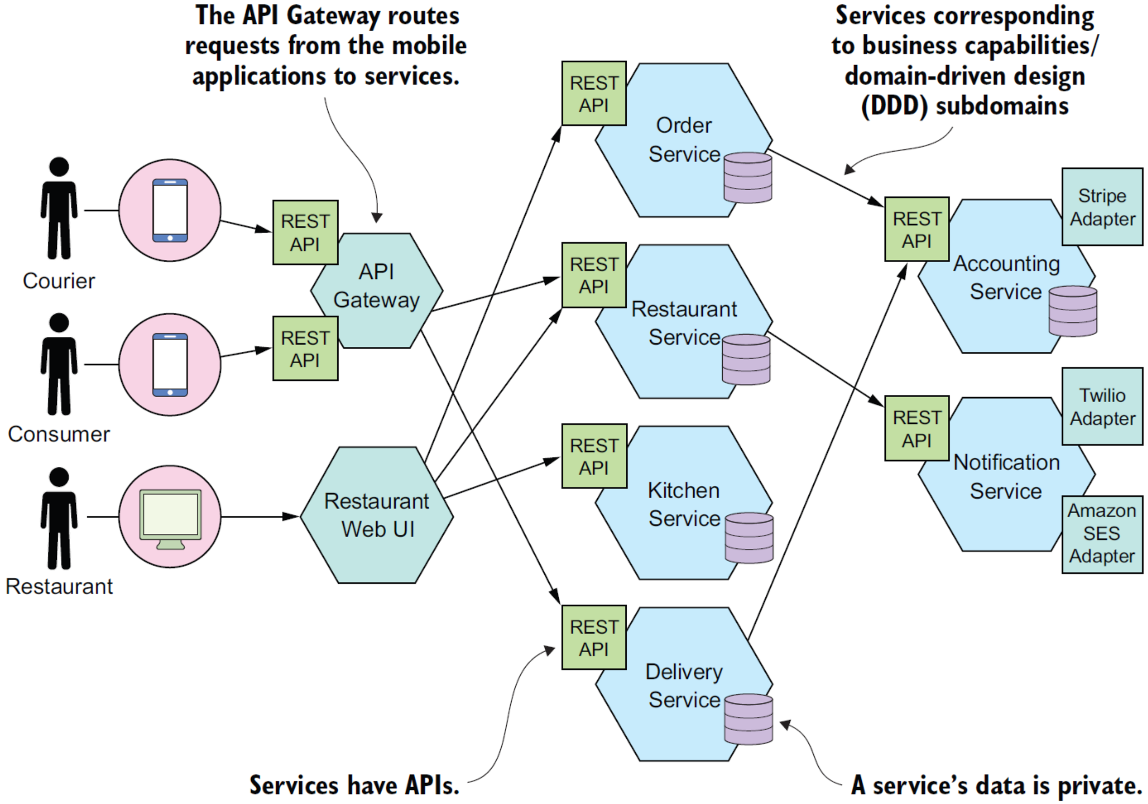
Architectural Pattern Classification

Style	Description	Examples
Data flow	computation is driven by the flow of data through the system.	• Batch sequential • Pipe-and-filter • Process control
Call-return	components interact through synchronous invocation of capabilities provided by other components	• Client-server • Peer-to-peer • SOA
Event-based	components interact through asynchronous events or messages	• Publish-subscribe • Point-to-point messaging • Blackboard
Repository	components interact through large collections of persistent, shared data	• Shared-data • Blackboard

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

MSA

- ❖ The microservice architecture uses services as the unit of modularity.
- ❖ A key characteristic of the microservice architecture is that the services are loosely coupled and communicate only via APIs.
- ❖ One way to achieve loose coupling is by each service having its own datastore.



The API Gateway routes requests from the mobile applications to services.

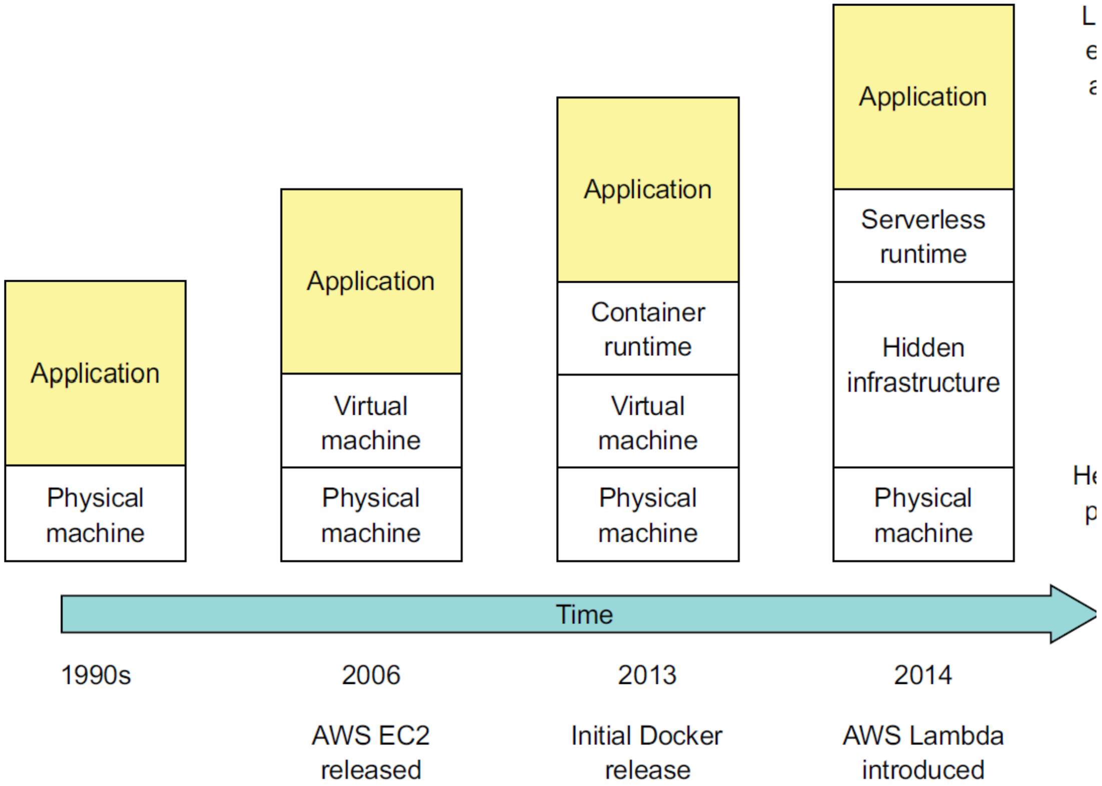
Services corresponding to business capabilities/ domain-driven design (DDD) subdomains

Services have APIs.

A service's data is private.

MSA

❖ Deployment



Application

Physical machine

Application

Virtual machine

Physical machine

Application

Container runtime

Virtual machine

Physical machine

Application

Serverless runtime

Hidden infrastructure

Physical machine

L
e
a

He
p

Time

1990s

2006

AWS EC2 released

2013

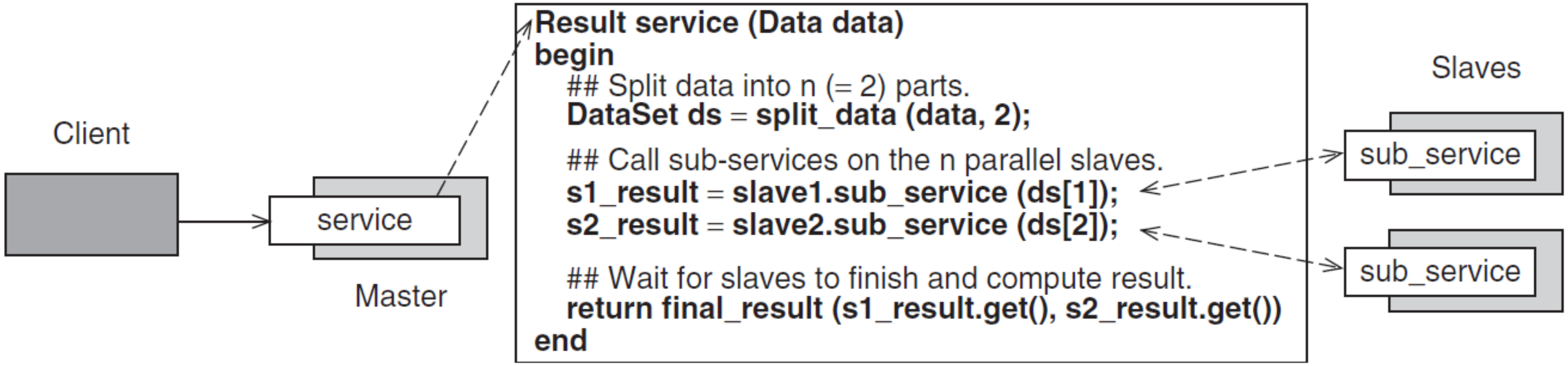
Initial Docker release

2014

AWS Lambda introduced

Master-Slave

- ❖ Meet the performance, fault-tolerance(availability), or accuracy (reliability) of the component via a ‘divide and conquer’ strategy.
- ❖ Split its services into independent subtasks that can be executed in parallel, and combine the partial results returned by these subtasks to provide the service’s final result.
- ❖ The divide and conquer strategy is determined by the intent(goal) of the pattern: performance, availability and reliability

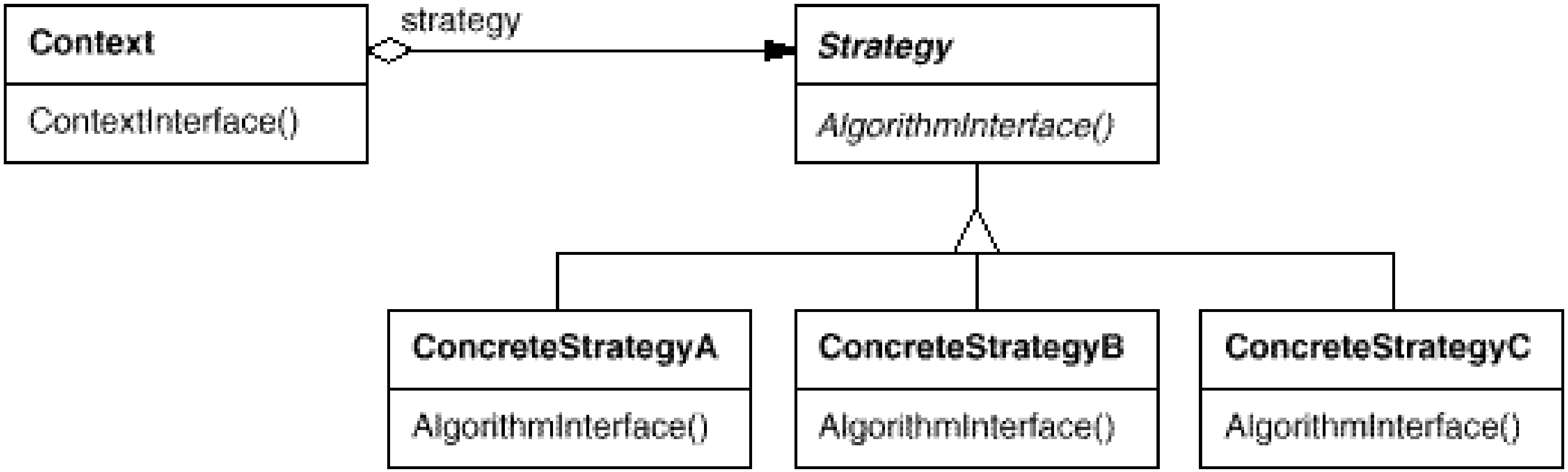


```
Result service (Data data)
begin
    ## Split data into n (= 2) parts.
    DataSet ds = split_data (data, 2);
    ## Call sub-services on the n parallel slaves.
    s1_result = slave1.sub_service (ds[1]);
    s2_result = slave2.sub_service (ds[2]);
    ## Wait for slaves to finish and compute result.
    return final_result (s1_result.get(), s2_result.get())
end
```

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



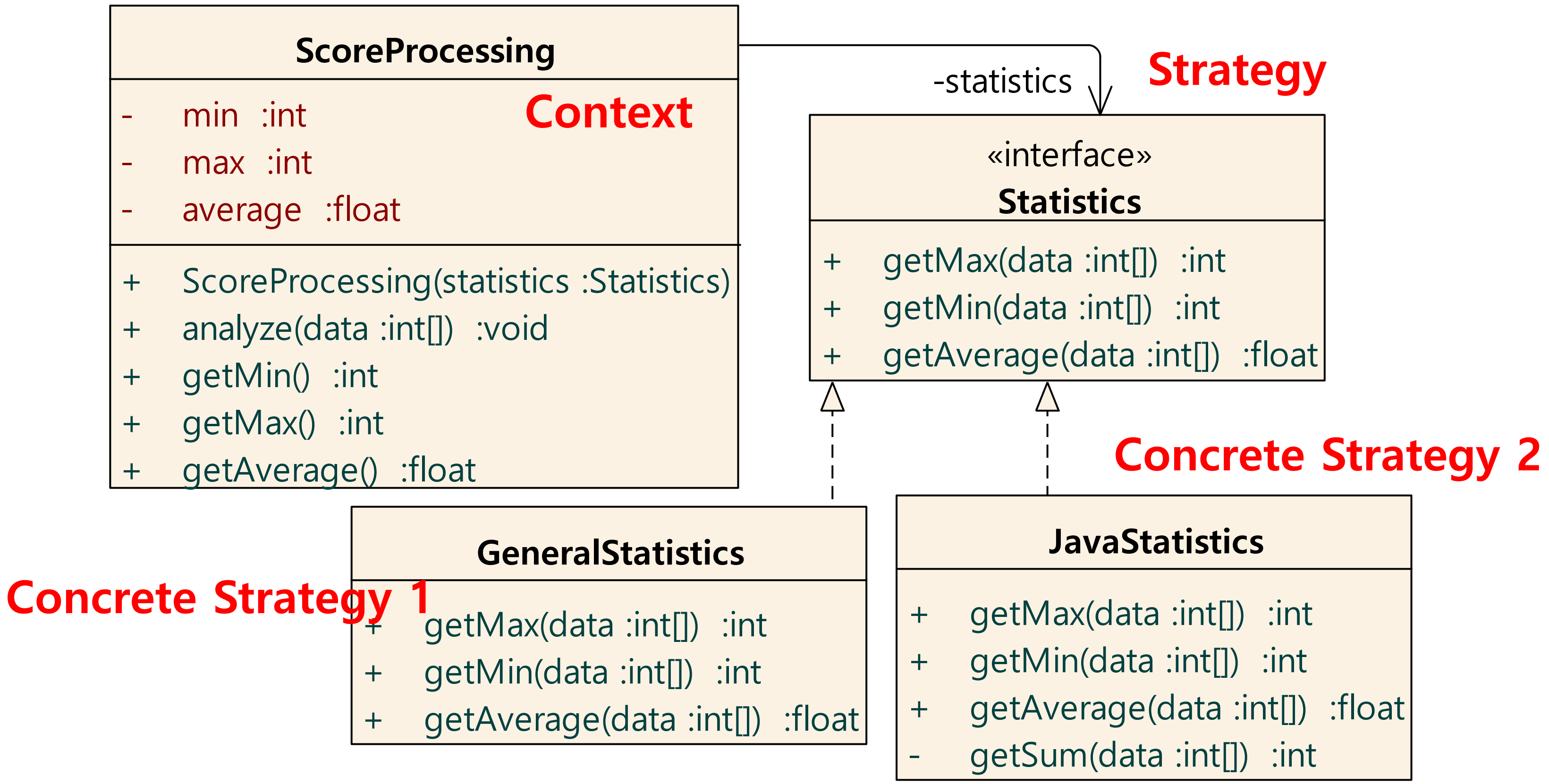
Strategy Pattern: Motivating Example

```
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

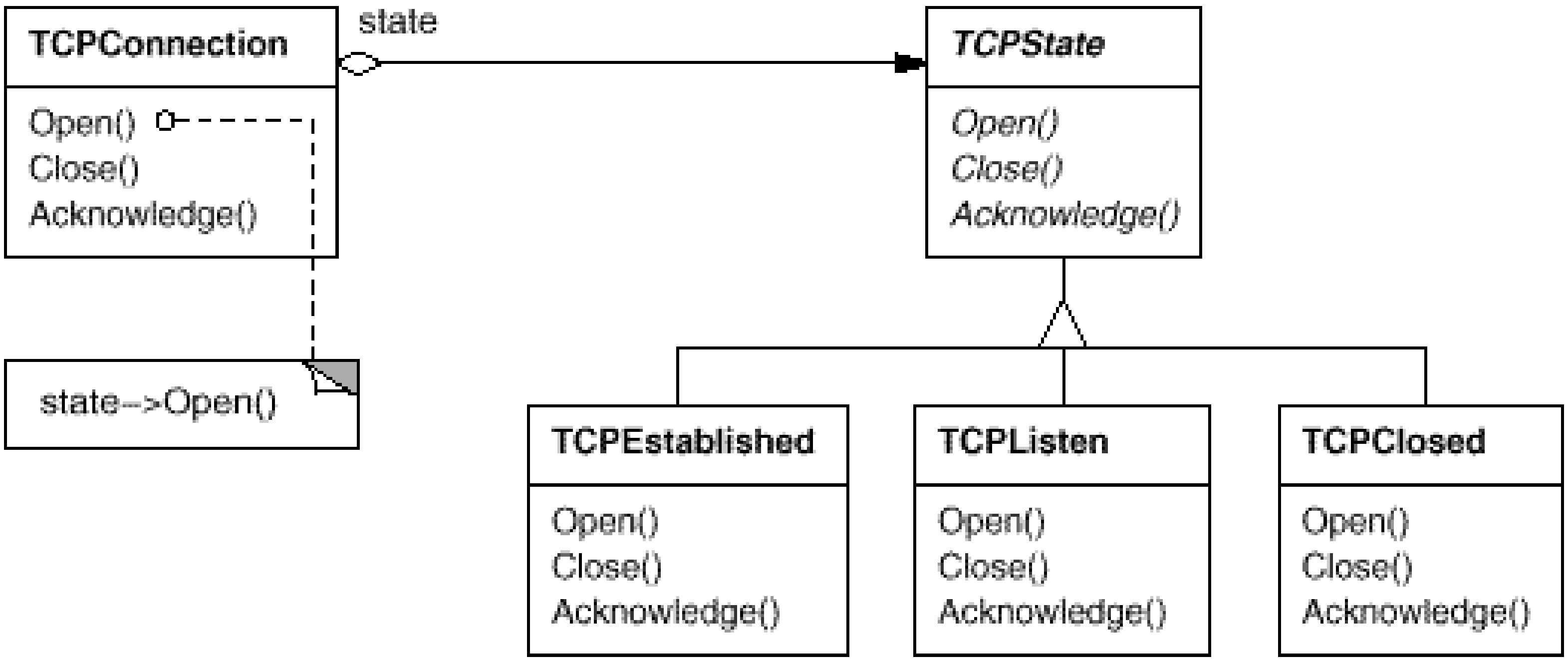
Strategy Pattern



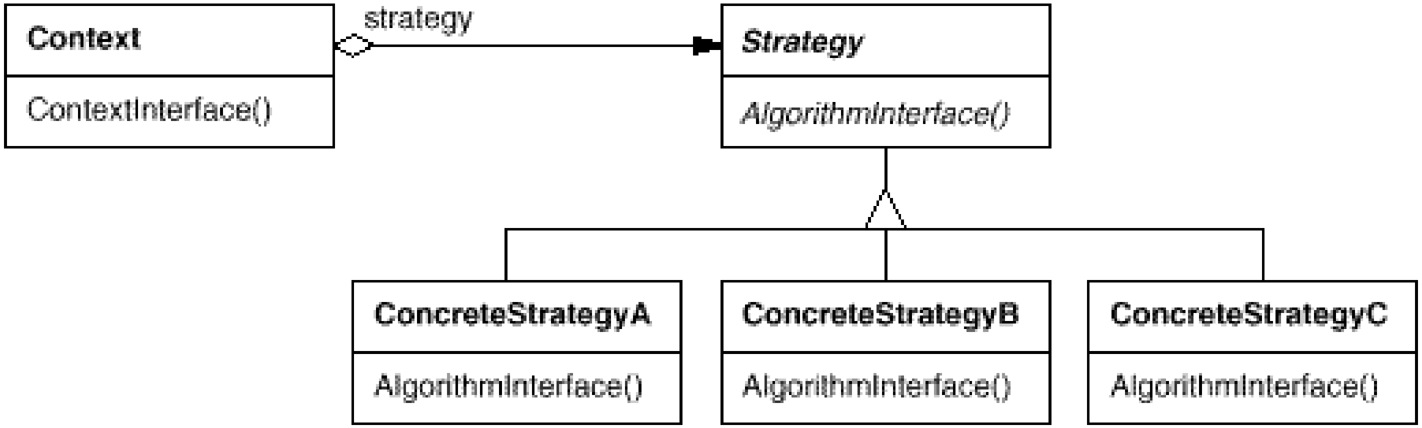
ScoreProcessing
- min :int
- max :int
- average :float
+ ScoreProcessing(statistics :Statistics)
+ analyze(data :int[]) :void
+ getMin() :int
+ getMax() :int
+ getAverage() :float
Context
-statistics
Strategy
«interface»
Statistics
+ getMax(data :int[]) :int
+ getMin(data :int[]) :int
+ getAverage(data :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

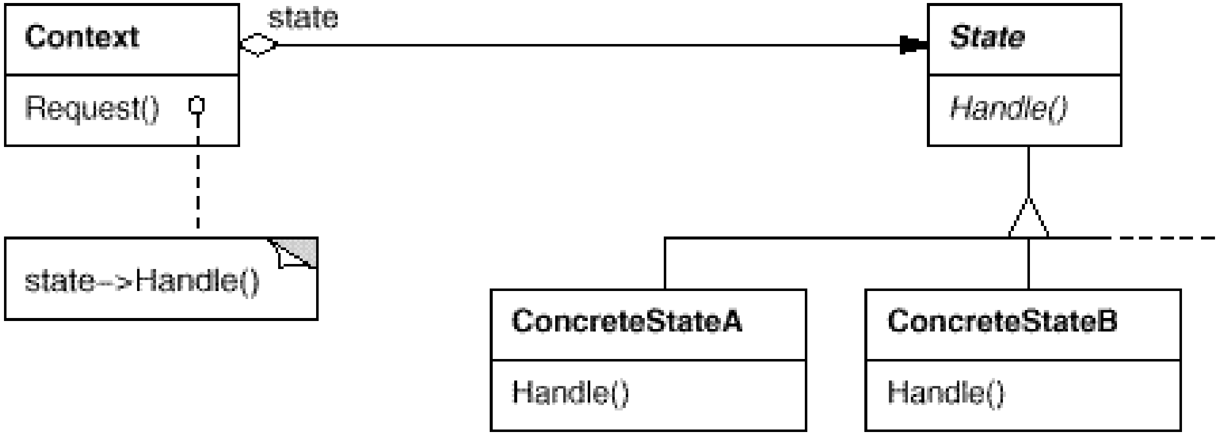
State Pattern

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



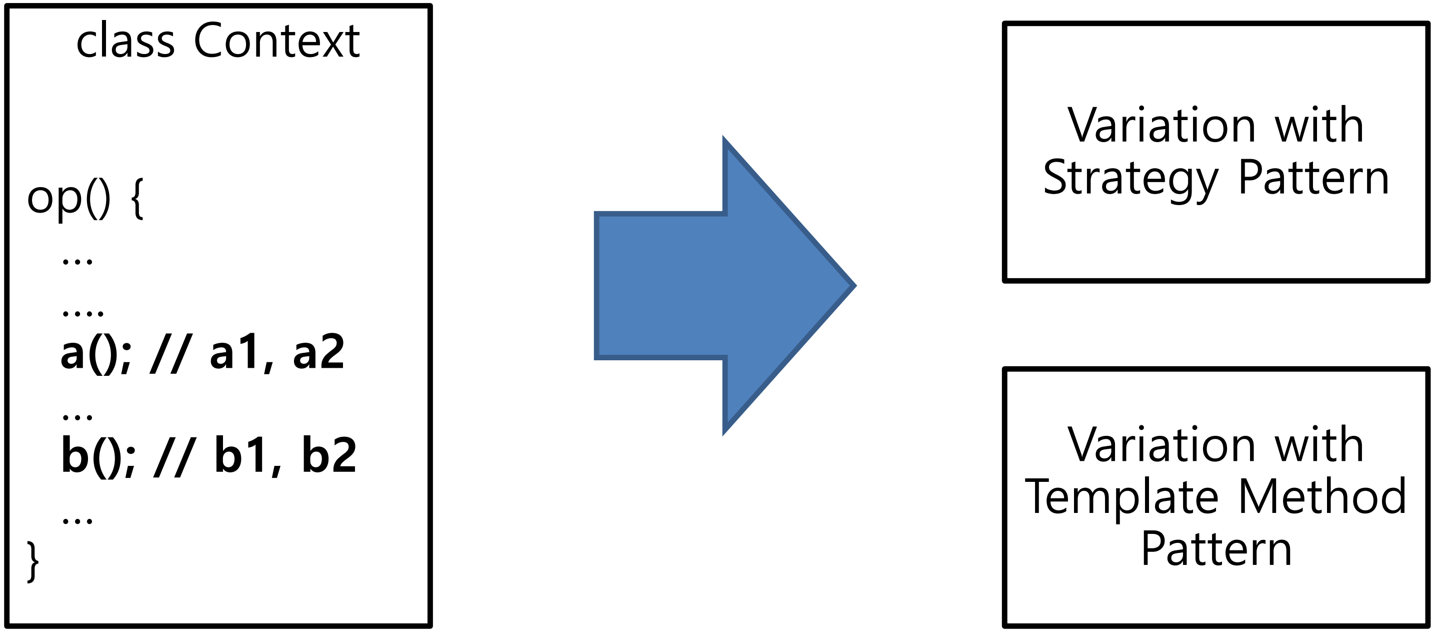
Strategy Pattern vs State Pattern





STRATEGY VS TEMPLATE METHOD

Strategy Pattern vs Template Method Pattern



class Context

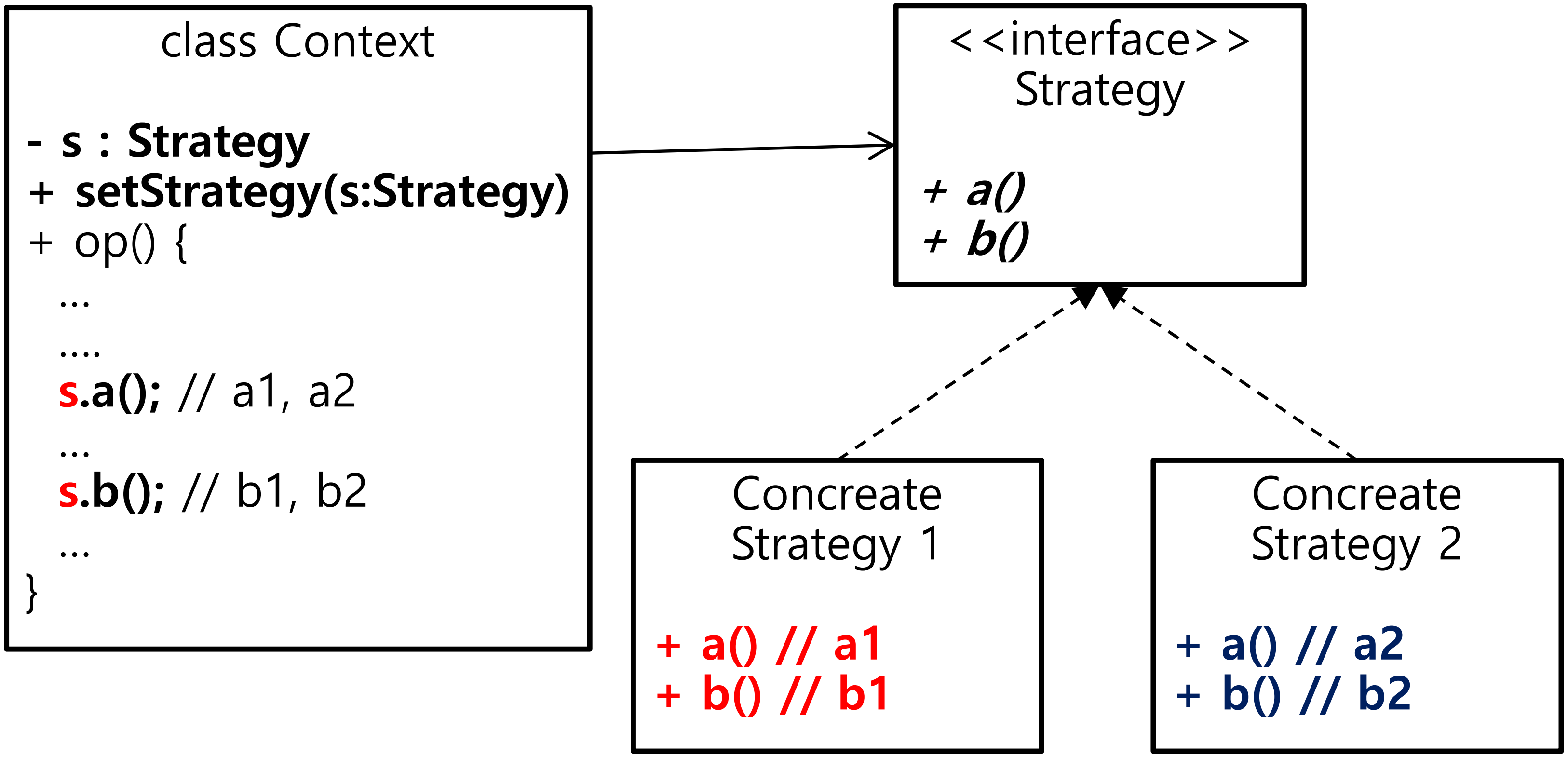
```
op() {
  …
  ….
  a(); // a1, a2
  …
  b(); // b1, b2
  …
}
```

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

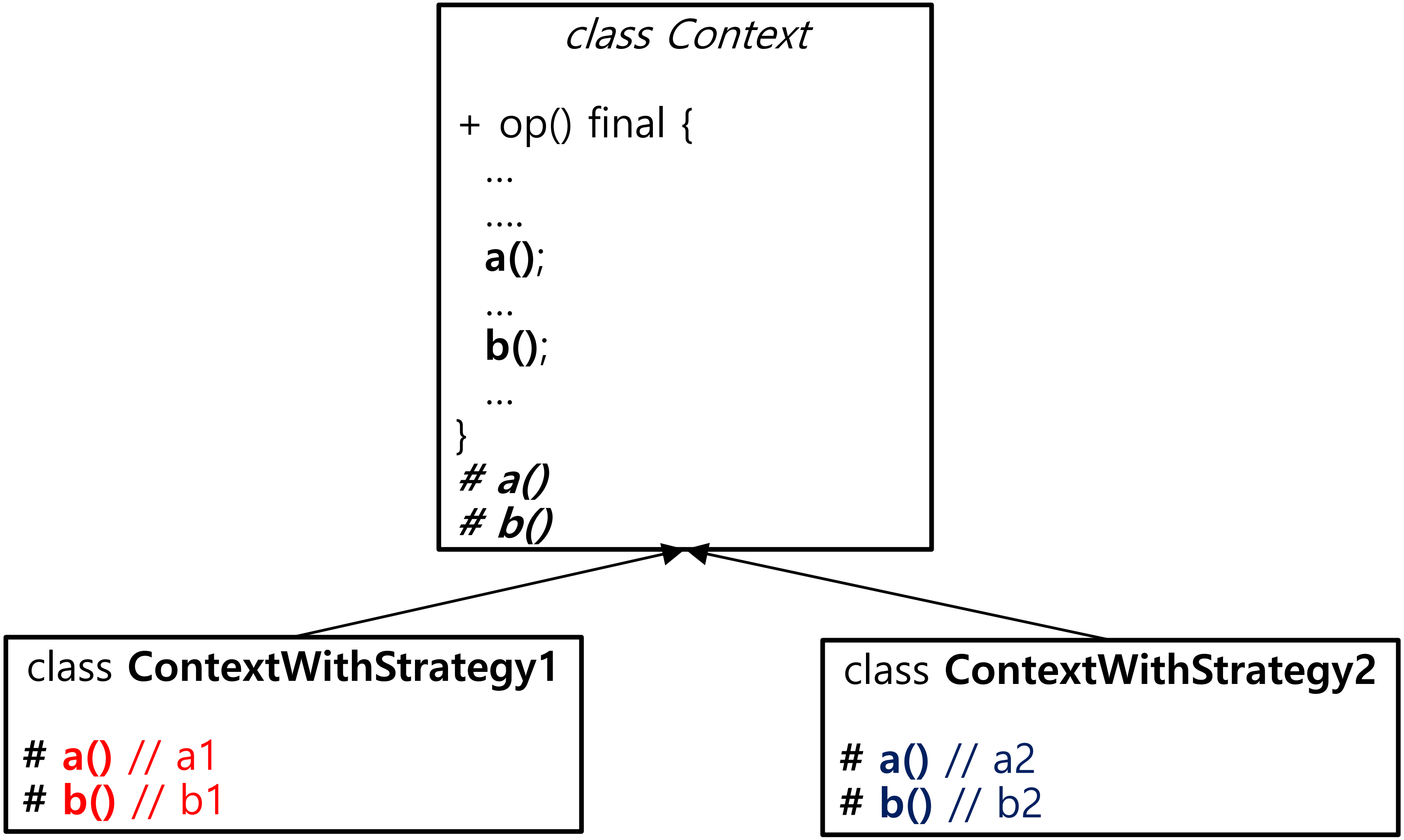
+ a() // a1
+ b() // b1

Concreate
Strategy 2

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

Variation with Template Method Pattern

- Implement the variation with subclasses



class Context

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

class **ContextWithStrategy1**

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

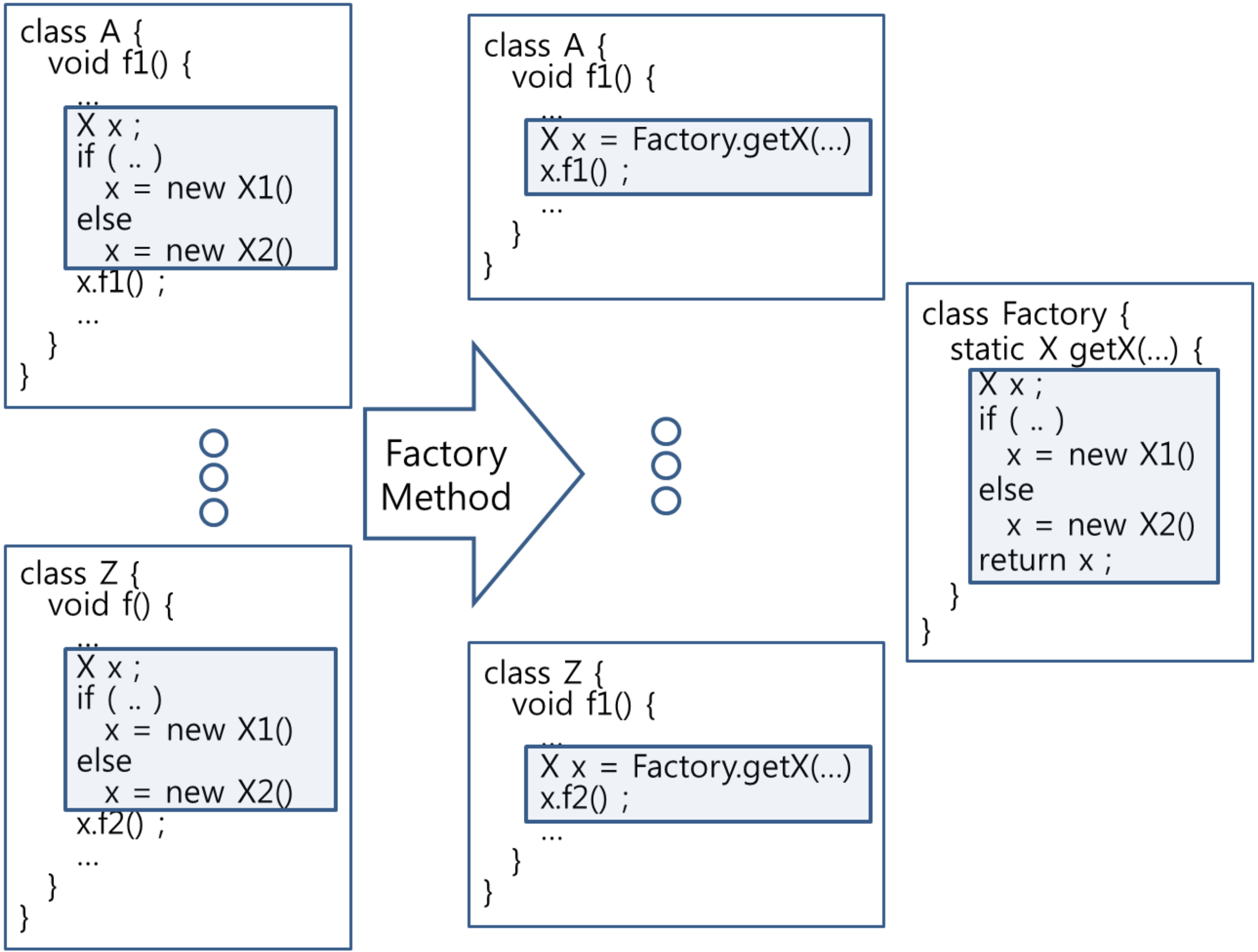
class **ContextWithStrategy2**

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

FACTORY METHOD VS ABSTRACT FACTORY

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() ;
    …
  }
}
```

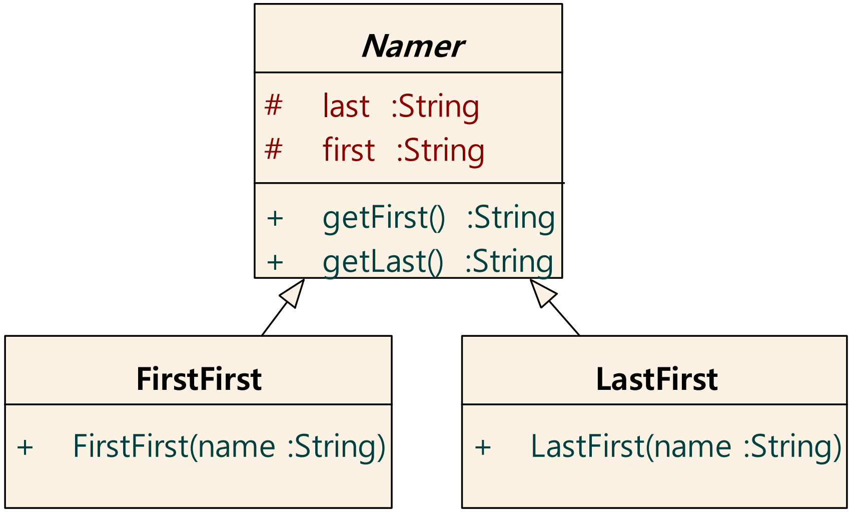
```
class Z {
  void f() {
    …
    X x ;
    if ( .. )
      x = new X1()
    else
      x = new X2()
    x.f2() ;
    …
  }
}
```

```
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 ;
  }
}
```

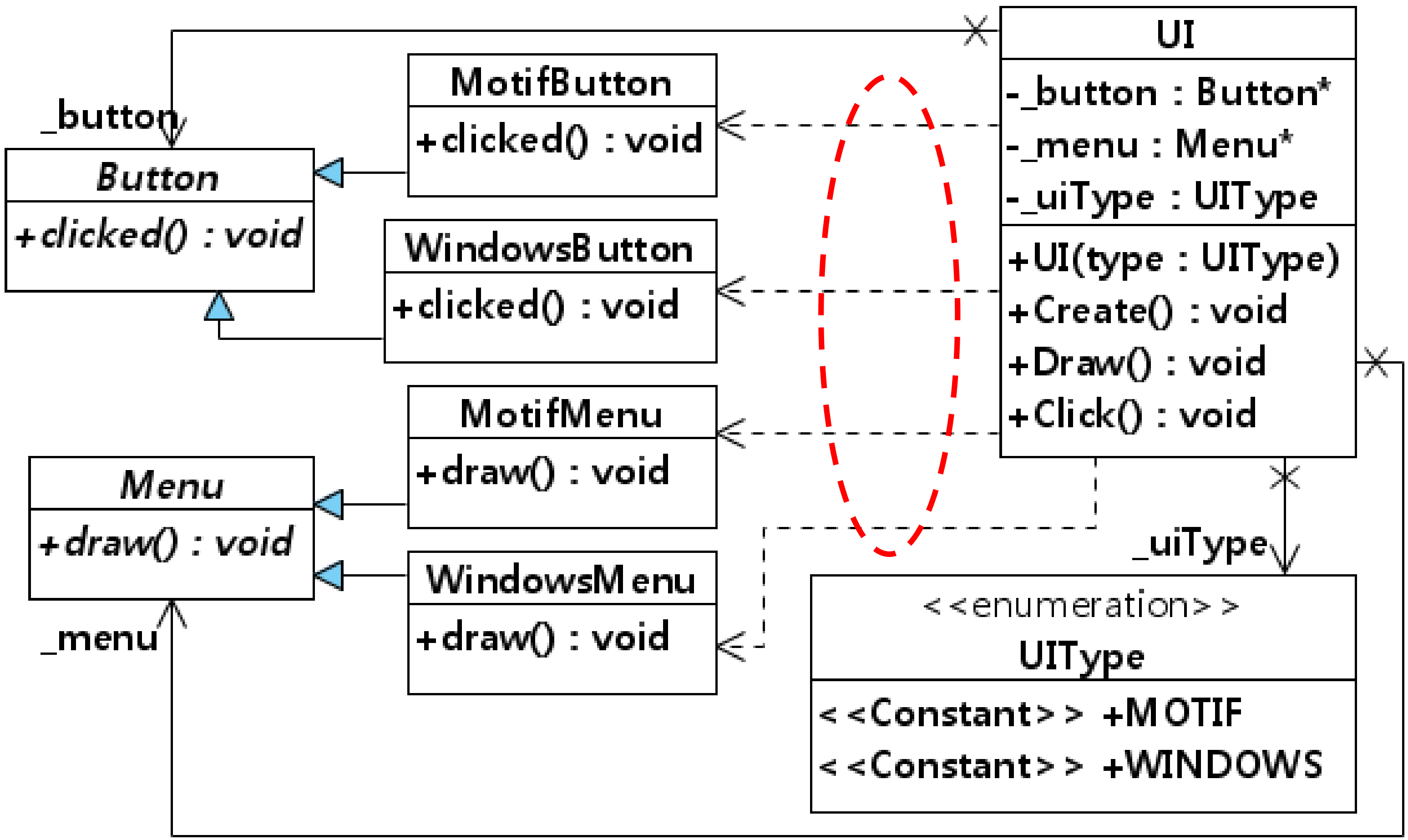
Replace Object Creation Behavior with Factory



```
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() ; }
}
```

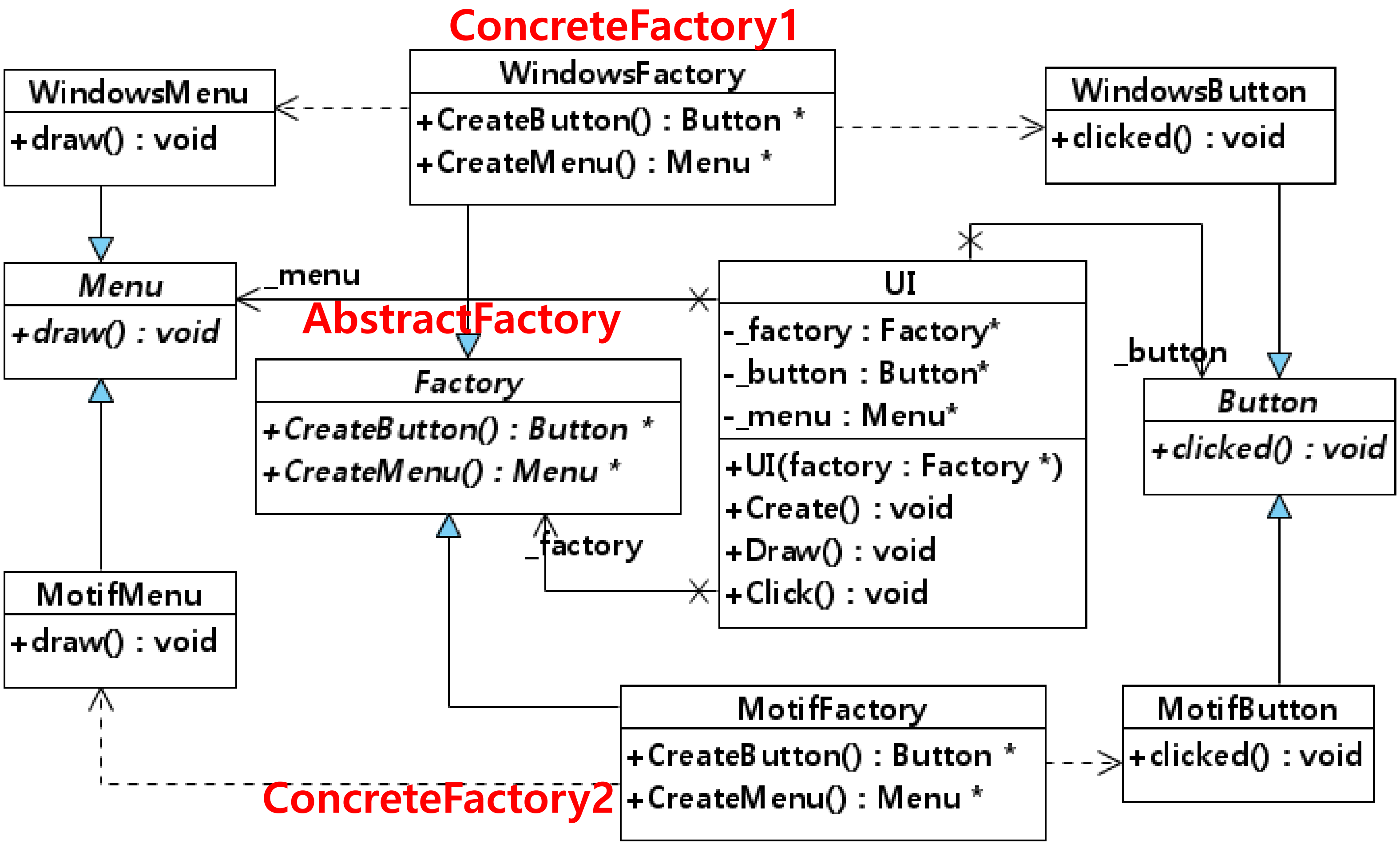
The Client (UI) depends on platform-specific Products

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

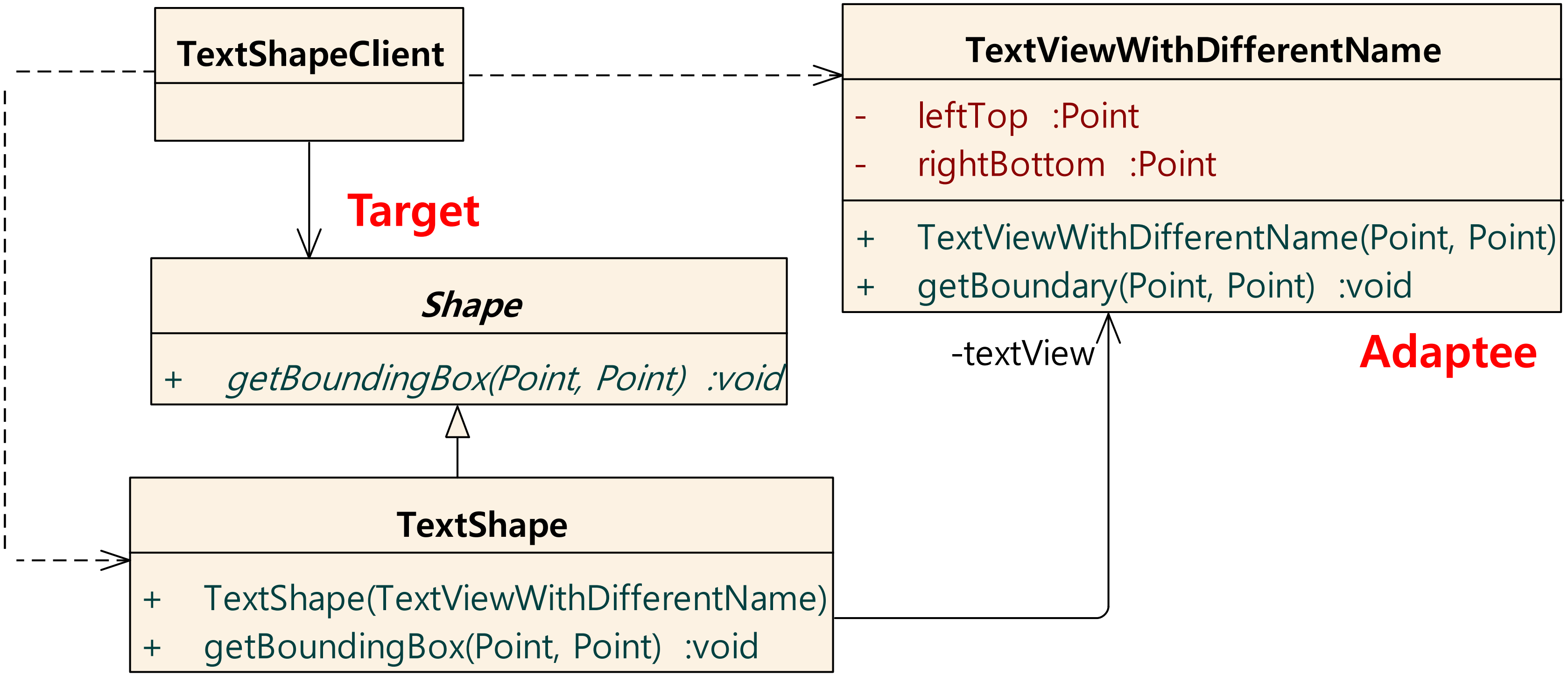
```
public class UI {
  private Button _button ;
  private Menu _menu ;
  private Factory _factory ;

  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() ; }
}
```

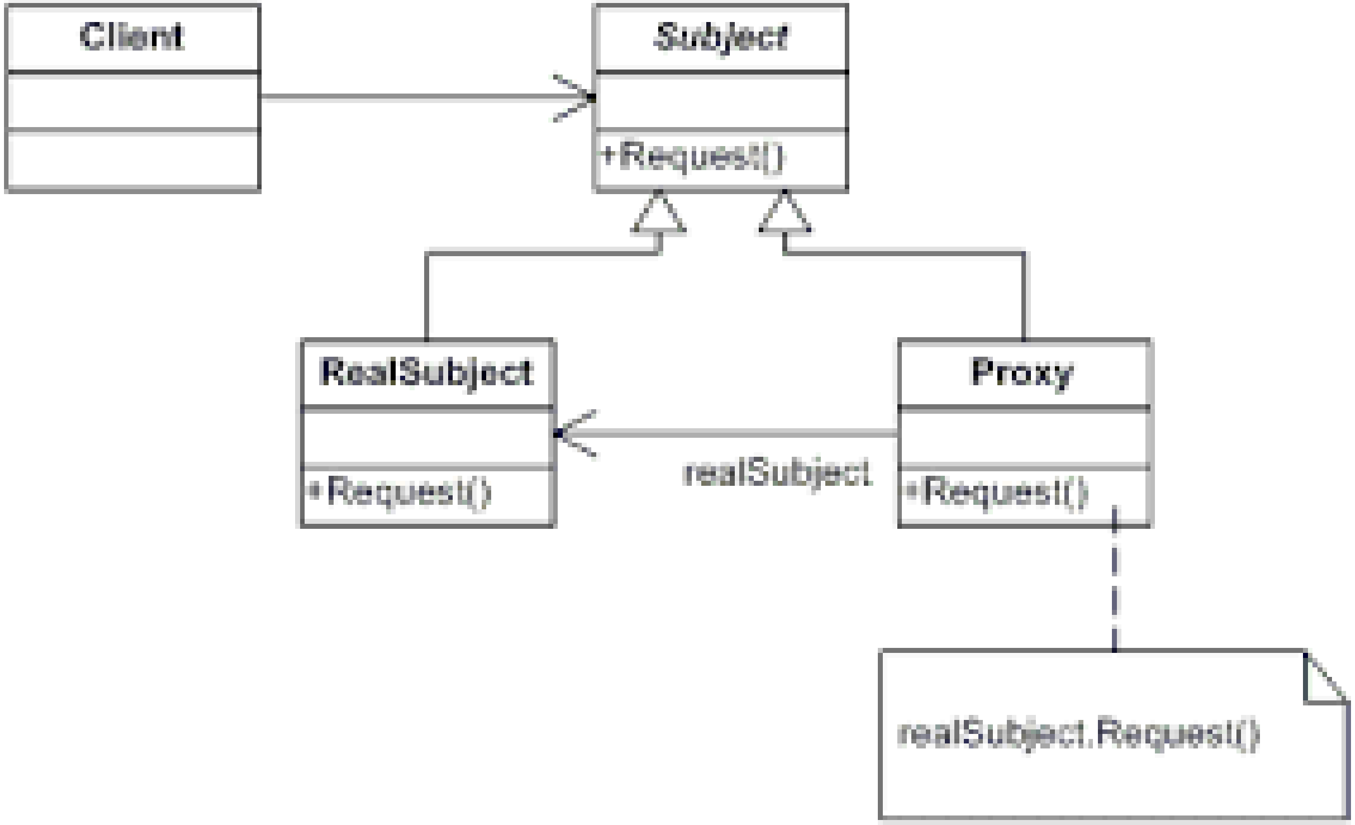
The Client (UI) does not depend on platform-specific products

ADAPTER VS PROXY

Adapter Pattern


TextShapeClient
Target
Shape
+ getBoundingBox(Point, Point) :void
TextShape
+ TextShape(TextViewWithDifferentName)
+ getBoundingBox(Point, Point) :void
TextViewWithDifferentName
- leftTop :Point
- rightBottom :Point
+ TextViewWithDifferentName(Point, Point)
+ getBoundary(Point, Point) :void
-textView
Adaptee

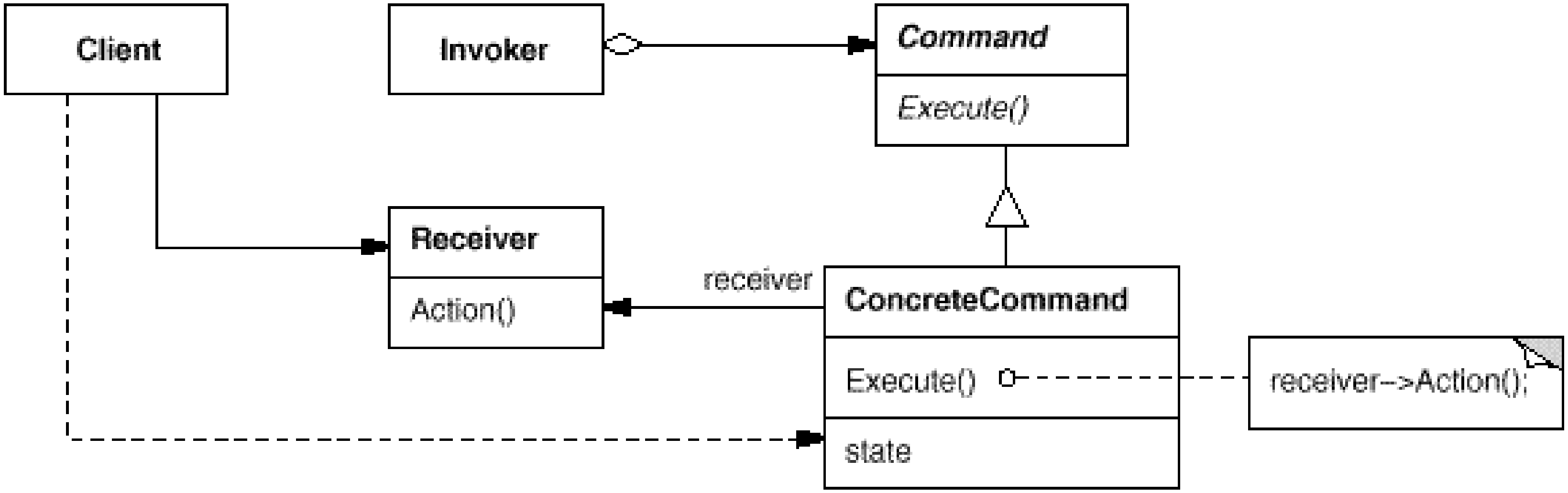
Proxy Pattern

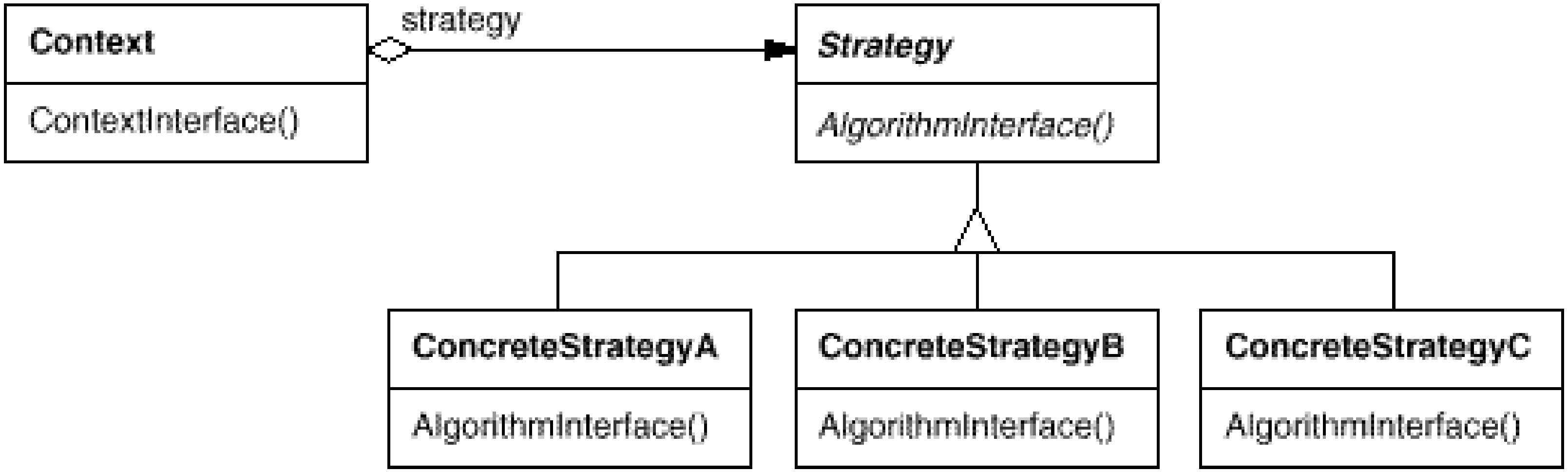


Remote proxy, virtual proxy, protection proxy, smart pointer, …

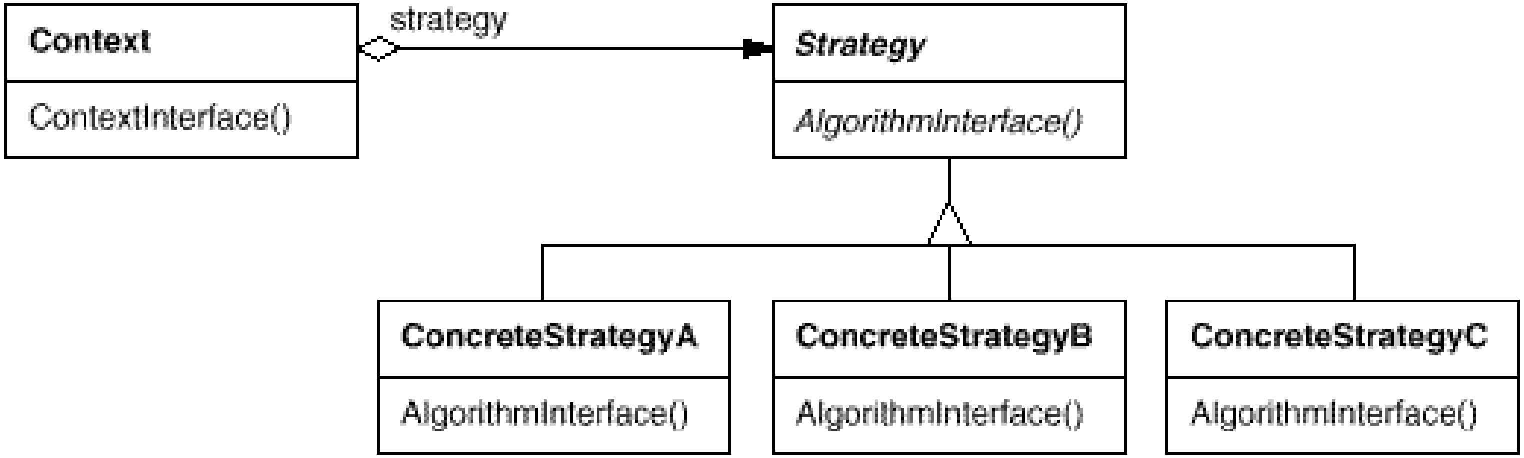
STRATEGY VS COMMAND VS OBSERVER

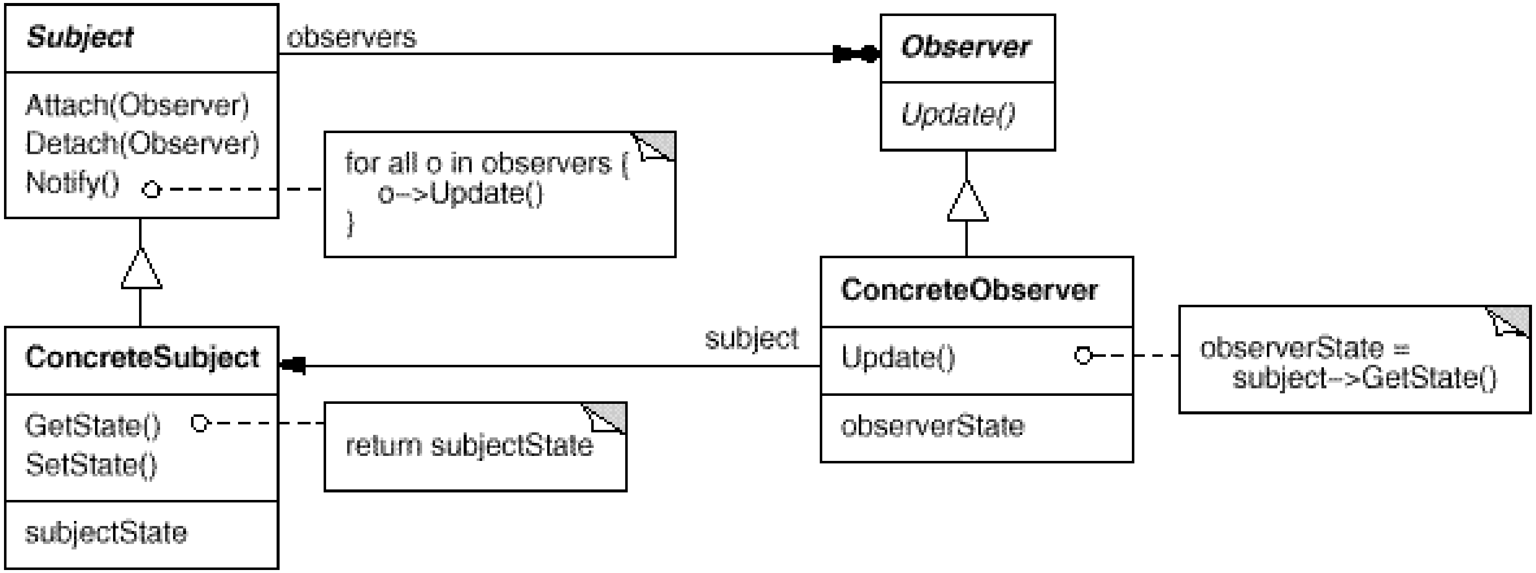
Command vs Strategy





Strategy vs Observer





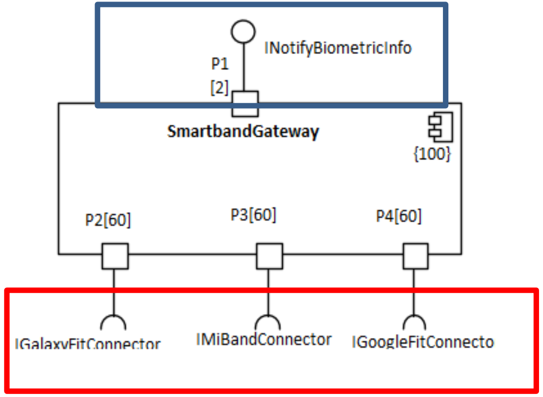
아키텍처 설계: Component Level

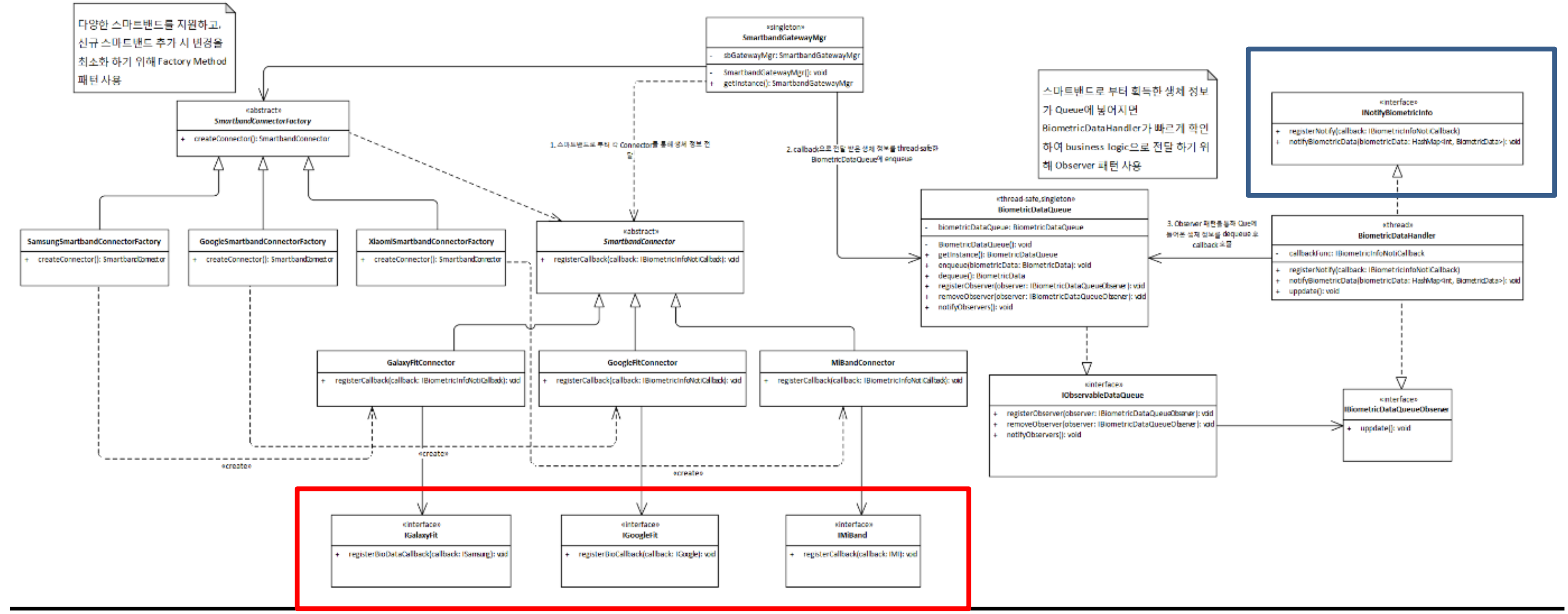
- 컴포넌트 요구사항 반영
  - 기능
  - 성능: multi-thread, thread pool, thread-safe

- 컴포넌트 상세 설계
  - 하나의 컴포넌트가 하나의 클래스로?
  - 설계 원칙(응집도, SOLID)을 고려한 세분화 필요

- 디자인 패턴의 혼동
  - State vs Strategy
  - Strategy vs Template method
  - Factory method vs Abstract factory
  - Adaptor vs Proxy

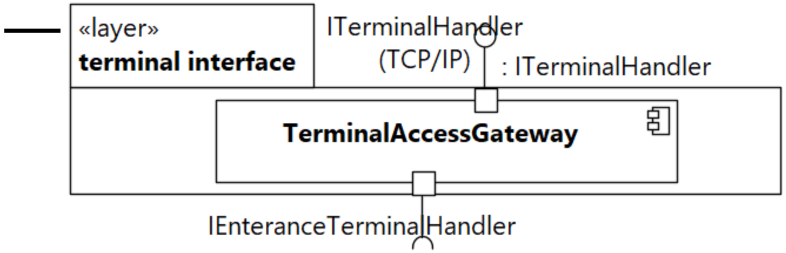
Interface 일관성 필요



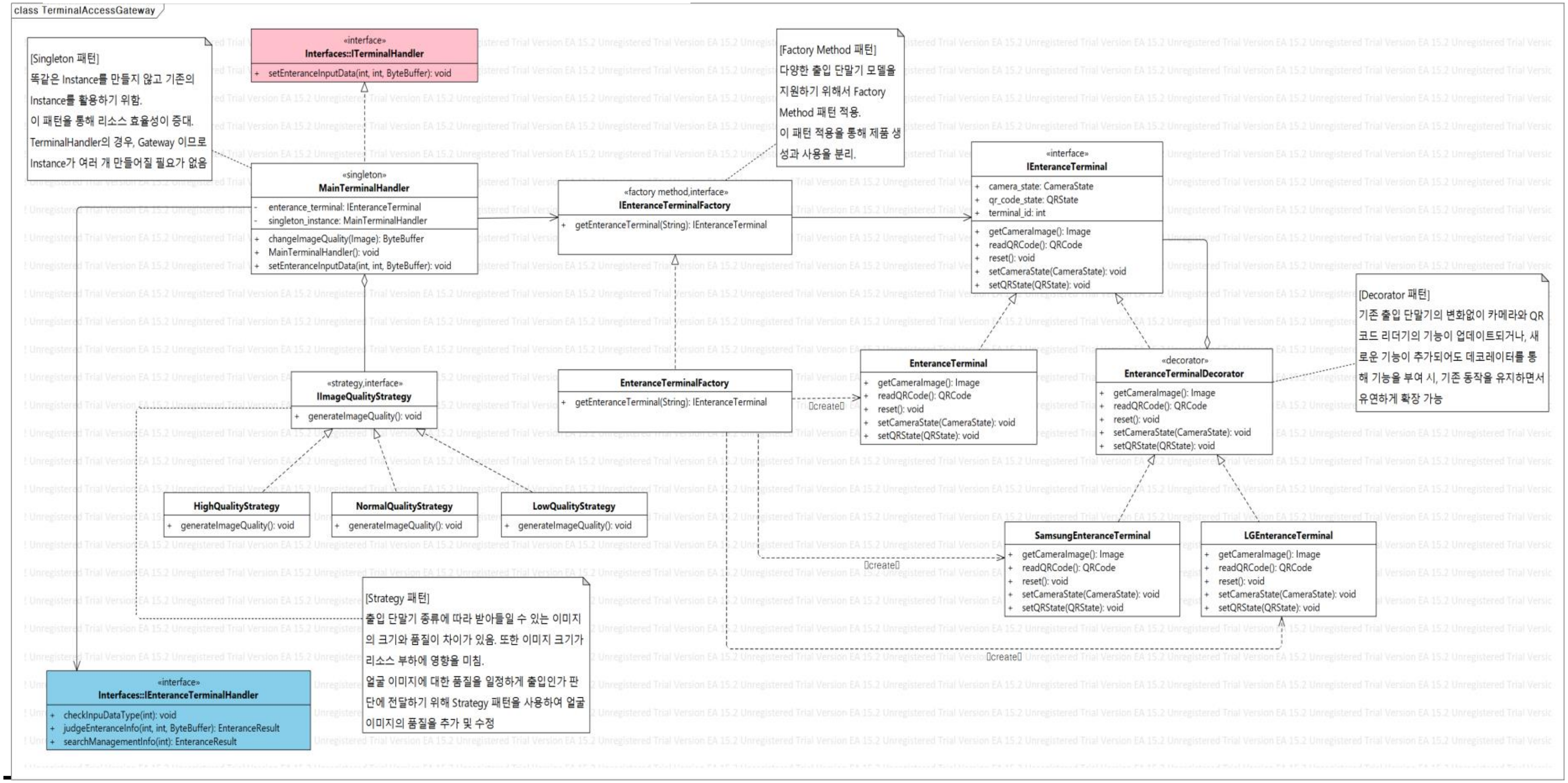

다양한 스마트밴드를 지원하고, 신규 스마트밴드 추가 시 변경을 최소화 하기 위해 Factory Method 패턴 사용

스마트밴드로 부터 획득한 생체 정보가 Queue에 넣어지면 BiometricDataHandler가 빠르게 확인하여 business logic으로 전달 하기 위해 Observer 패턴 사용

«singleton»
SmartbandGatewayMgr
- sbGatewayMgr: SmartbandGatewayMgr
- SmartbandGatewayMgr(): void
+ getInstance(): SmartbandGatewayMgr
«abstract»
SmartbandConnectorFactory
+ createConnector(): SmartbandConnector
SamsungSmartbandConnectorFactory
+ createConnector(): SmartbandConnector
GoogleSmartbandConnectorFactory
+ createConnector(): SmartbandConnector
XiaomiSmartbandConnectorFactory
+ createConnector(): SmartbandConnector
1. 스마트밴드로 부터 각 Connector를 통해 생체 정보 전달
2. callback으로 전달 받은 생체 정보를 thread-safe한 BiometricDataQueue에 enqueue
«abstract»
SmartbandConnector
+ registerCallback(callback: IBiometricInfoNotiCallback): void
GalaxyFitConnector
+ registerCallback(callback: IBiometricInfoNotiCallback): void
GoogleFitConnector
+ registerCallback(callback: IBiometricInfoNotiCallback): void
MiBandConnector
+ registerCallback(callback: IBiometricInfoNotiCallback): void
«create»
«create»
«create»
«interface»
IGalaxyFit
+ registerBioDataCallback(callback: ISamsung): void
«interface»
IGoogleFit
+ registerBioCallback(callback: IGoogle): void
«interface»
IMiBand
+ registerCallback(callback: IMI): void
«thread-safe,singleton»
BiometricDataQueue
- biometricDataQueue: BiometricDataQueue
- BiometricDataQueue(): void
+ getInstance(): BiometricDataQueue
+ enqueue(biometricData: BiometricData): void
+ dequeue(): BiometricData
+ registerObserver(observer: IBiometricDataQueueObserver): void
+ removeObserver(observer: IBiometricDataQueueObserver): void
+ notifyObservers(): void
3. Observer 패턴을 통해 Queue에 들어온 생체 정보를 dequeue 후 callback 호출
«interface»
INotifyBiometricInfo
+ registerNotify(callback: IBiometricInfoNotiCallback)
+ notifyBiometricData(biometricData: HashMap<int, BiometricData>): void
«thread»
BiometricDataHandler
- callbackFunc: IBiometricInfoNotiCallback
+ registerNotify(callback: IBiometricInfoNotiCallback)
+ notifyBiometricData(biometricData: HashMap<int, BiometricData>): void
+ uppdate(): void
«interface»
IObservableDataQueue
+ registerObserver(observer: IBiometricDataQueueObserver): void
+ removeObserver(observer: IBiometricDataQueueObserver): void
+ notifyObservers(): void
«interface»
IBiometricDataQueueObserver
+ uppdate(): void



Stategy, Factory method, Decorator 등의 적용 시도는 좋음. 다만, generateImageQuality() 에 적절한 인자 및 return type 필요함
Handler에서 IEntranceTerminal로의 dependency 필요
Decorator의 적용이 적합한지 Terminal의 각 동작에 대한 세밀한 분석 필요

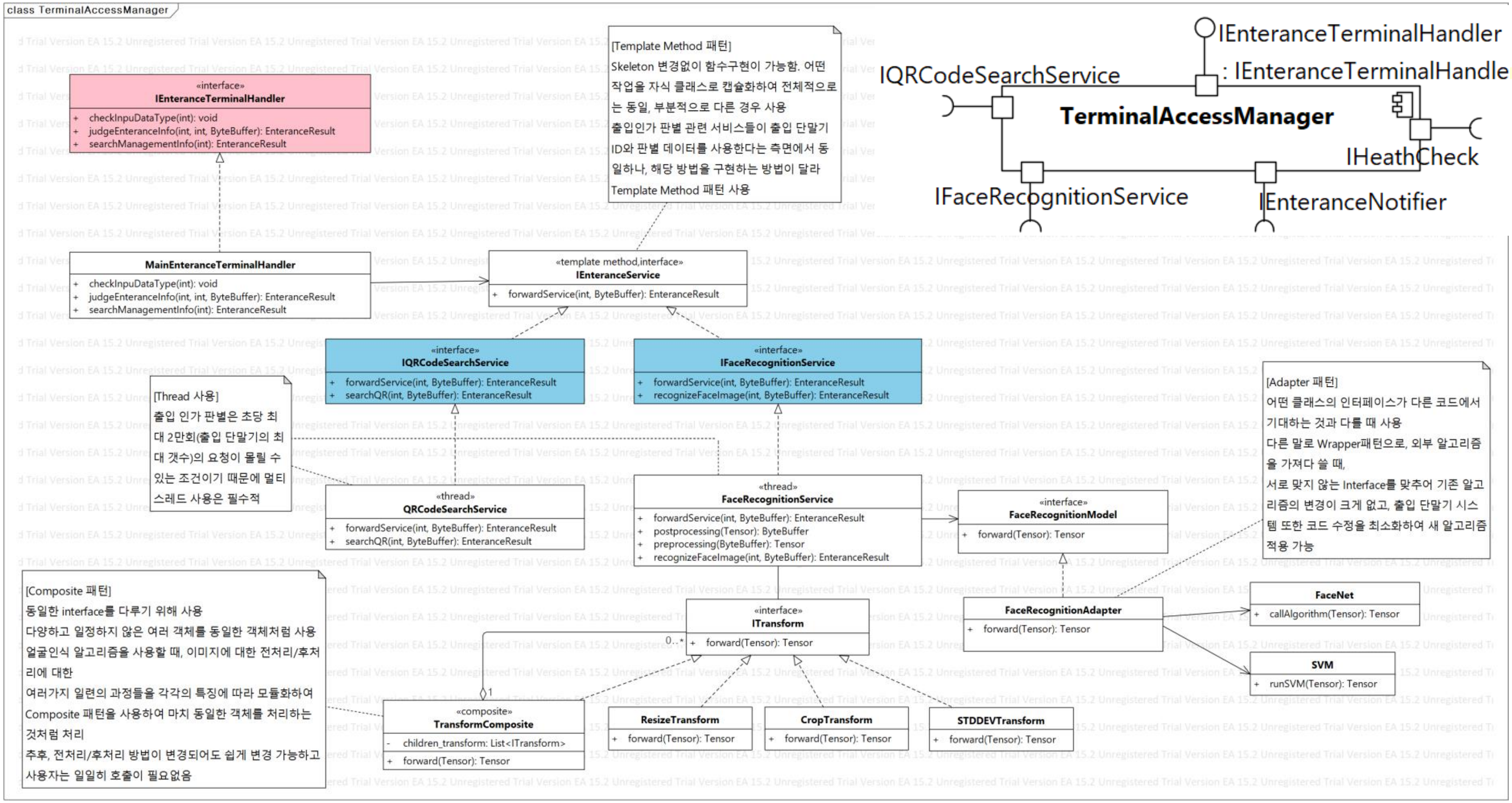

class TerminalAccessGateway
«interface»
Interfaces::ITerminalHandler
+ setEnteranceInputData(int, int, ByteBuffer): void
«singleton»
MainTerminalHandler
- enterance_terminal: IEnteranceTerminal
- singleton_instance: MainTerminalHandler
+ changeImageQuality(Image): ByteBuffer
+ MainTerminalHandler(): void
+ setEnteranceInputData(int, int, ByteBuffer): void
«factory method,interface»
IEnteranceTerminalFactory
+ getEnteranceTerminal(String): IEnteranceTerminal
«interface»
IEnteranceTerminal
+ camera_state: CameraState
+ qr_code_state: QRState
+ terminal_id: int
+ getCameraImage(): Image
+ readQRCode(): QRCode
+ reset(): void
+ setCameraState(CameraState): void
+ setQRState(QRState): void
«strategy,interface»
IImageQualityStrategy
+ generateImageQuality(): void
EnteranceTerminalFactory
+ getEnteranceTerminal(String): IEnteranceTerminal
create
EnteranceTerminal
+ getCameraImage(): Image
+ readQRCode(): QRCode
+ reset(): void
+ setCameraState(CameraState): void
+ setQRState(QRState): void
«decorator»
EnteranceTerminalDecorator
+ getCameraImage(): Image
+ readQRCode(): QRCode
+ reset(): void
+ setCameraState(CameraState): void
+ setQRState(QRState): void
HighQualityStrategy
+ generateImageQuality(): void
NormalQualityStrategy
+ generateImageQuality(): void
LowQualityStrategy
+ generateImageQuality(): void
SamsungEnteranceTerminal
+ getCameraImage(): Image
+ readQRCode(): QRCode
+ reset(): void
+ setCameraState(CameraState): void
+ setQRState(QRState): void
LGEnteranceTerminal
+ getCameraImage(): Image
+ readQRCode(): QRCode
+ reset(): void
+ setCameraState(CameraState): void
+ setQRState(QRState): void
create
create
«interface»
Interfaces::IEnteranceTerminalHandler
+ checkInpuDataType(int): void
+ judgeEnteranceInfo(int, int, ByteBuffer): EnteranceResult
+ searchManagementInfo(int): EnteranceResult

[Singleton 패턴]
똑같은 Instance를 만들지 않고 기존의 Instance를 활용하기 위함.
이 패턴을 통해 리소스 효율성이 증대.
TerminalHandler의 경우, Gateway 이므로 Instance가 여러 개 만들어질 필요가 없음

[Factory Method 패턴]
다양한 출입 단말기 모델을 지원하기 위해서 Factory Method 패턴 적용.
이 패턴 적용을 통해 제품 생성과 사용을 분리.

[Decorator 패턴]
기존 출입 단말기의 변화없이 카메라와 QR 코드 리더기의 기능이 업데이트되거나, 새로운 기능이 추가되어도 데코레이터를 통해 기능을 부여 시, 기존 동작을 유지하면서 유연하게 확장 가능

[Strategy 패턴]
출입 단말기 종류에 따라 받아들일 수 있는 이미지의 크기와 품질이 차이가 있음. 또한 이미지 크기가 리소스 부하에 영향을 미침.
얼굴 이미지에 대한 품질을 일정하게 출입인가 판단에 전달하기 위해 Strategy 패턴을 사용하여 얼굴 이미지의 품질을 추가 및 수정


class TerminalAccessManager
«interface»
IEnteranceTerminalHandler
+ checkInpuDataType(int): void
+ judgeEnteranceInfo(int, int, ByteBuffer): EnteranceResult
+ searchManagementInfo(int): EnteranceResult
MainEnteranceTerminalHandler
+ checkInpuDataType(int): void
+ judgeEnteranceInfo(int, int, ByteBuffer): EnteranceResult
+ searchManagementInfo(int): EnteranceResult
«template method,interface»
IEnteranceService
+ forwardService(int, ByteBuffer): EnteranceResult
«interface»
IQRCodeSearchService
+ forwardService(int, ByteBuffer): EnteranceResult
+ searchQR(int, ByteBuffer): EnteranceResult
«interface»
IFaceRecognitionService
+ forwardService(int, ByteBuffer): EnteranceResult
+ recognizeFaceImage(int, ByteBuffer): EnteranceResult
«thread»
QRCodeSearchService
+ forwardService(int, ByteBuffer): EnteranceResult
+ searchQR(int, ByteBuffer): EnteranceResult
«thread»
FaceRecognitionService
+ forwardService(int, ByteBuffer): EnteranceResult
+ postprocessing(Tensor): ByteBuffer
+ preprocessing(ByteBuffer): Tensor
+ recognizeFaceImage(int, ByteBuffer): EnteranceResult
«interface»
FaceRecognitionModel
+ forward(Tensor): Tensor
FaceRecognitionAdapter
+ forward(Tensor): Tensor
FaceNet
+ callAlgorithm(Tensor): Tensor
SVM
+ runSVM(Tensor): Tensor
«interface»
ITransform
+ forward(Tensor): Tensor
0..*
1
«composite»
TransformComposite
- children_transform: List<ITransform>
+ forward(Tensor): Tensor
ResizeTransform
+ forward(Tensor): Tensor
CropTransform
+ forward(Tensor): Tensor
STDDEVTransform
+ forward(Tensor): Tensor
IEnteranceTerminalHandler
: IEnteranceTerminalHandle
IQRCodeSearchService
TerminalAccessManager
IHeathCheck
IFaceRecognitionService
IEnteranceNotifier

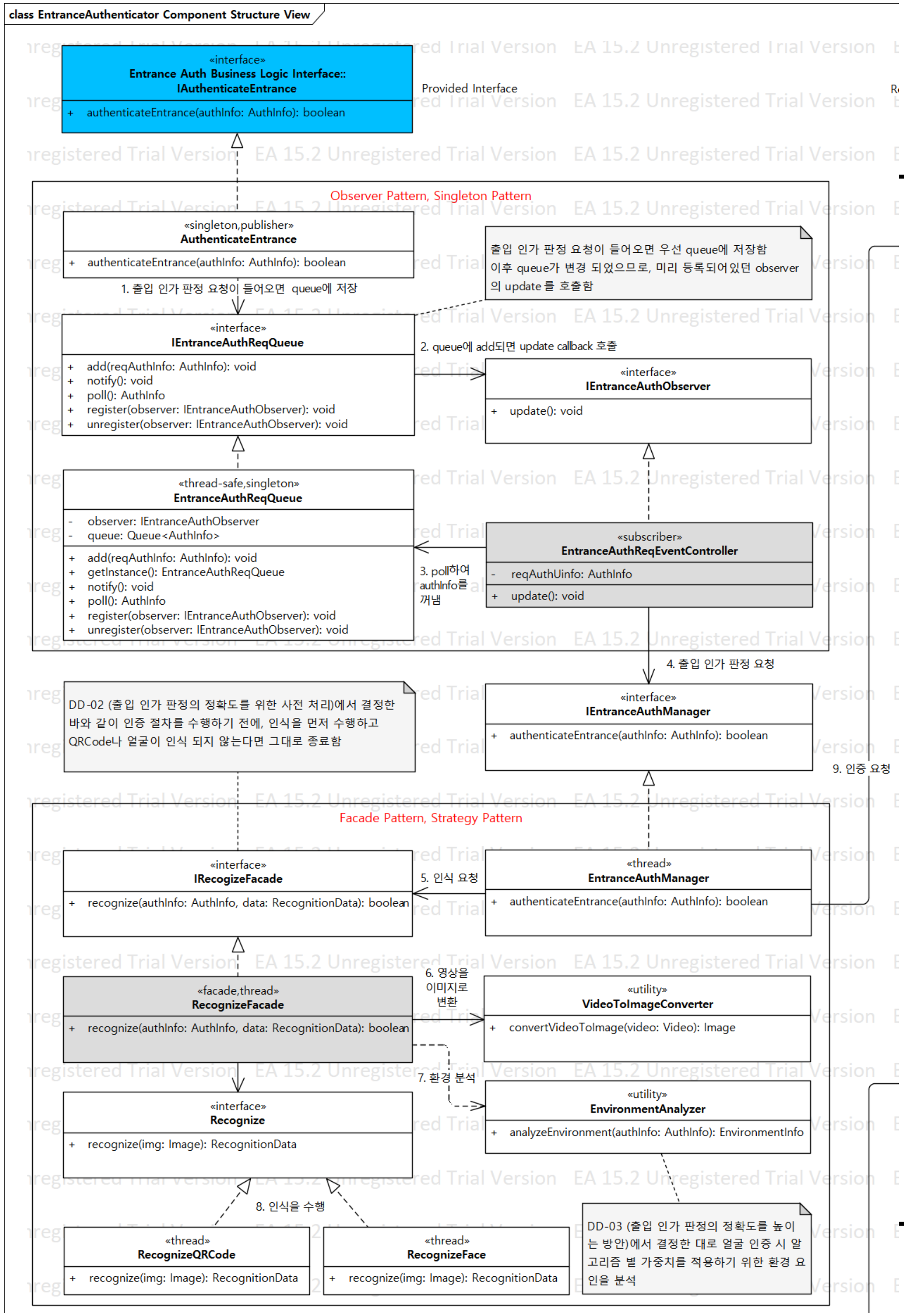
[Template Method 패턴]
Skeleton 변경없이 함수구현이 가능함. 어떤 작업을 자식 클래스로 캡슐화하여 전체적으로는 동일, 부분적으로 다른 경우 사용
출입인가 판별 관련 서비스들이 출입 단말기 ID와 판별 데이터를 사용한다는 측면에서 동일하나, 해당 방법을 구현하는 방법이 달라 Template Method 패턴 사용

[Thread 사용]
출입 인가 판별은 초당 최대 2만회(출입 단말기의 최대 갯수)의 요청이 몰릴 수 있는 조건이기 때문에 멀티스레드 사용은 필수적

[Adapter 패턴]
어떤 클래스의 인터페이스가 다른 코드에서 기대하는 것과 다를 때 사용
다른 말로 Wrapper패턴으로, 외부 알고리즘을 가져다 쓸 때,
서로 맞지 않는 Interface를 맞추어 기존 알고리즘의 변경이 크게 없고, 출입 단말기 시스템 또한 코드 수정을 최소화하여 새 알고리즘 적용 가능

[Composite 패턴]
동일한 interface를 다루기 위해 사용
다양하고 일정하지 않은 여러 객체를 동일한 객체처럼 사용
얼굴인식 알고리즘을 사용할 때, 이미지에 대한 전처리/후처리에 대한
여러가지 일련의 과정들을 각각의 특징에 따라 모듈화하여 Composite 패턴을 사용하여 마치 동일한 객체를 처리하는 것처럼 처리
추후, 전처리/후처리 방법이 변경되어도 쉽게 변경 가능하고 사용자는 일일히 호출이 필요없음

구체적인 EntranceService 생성은 누가 어떤 기준으로 하는지?
interface가 interface를 realize?
template method의 구조에 일치하지 않음
required inteface 사용 필요

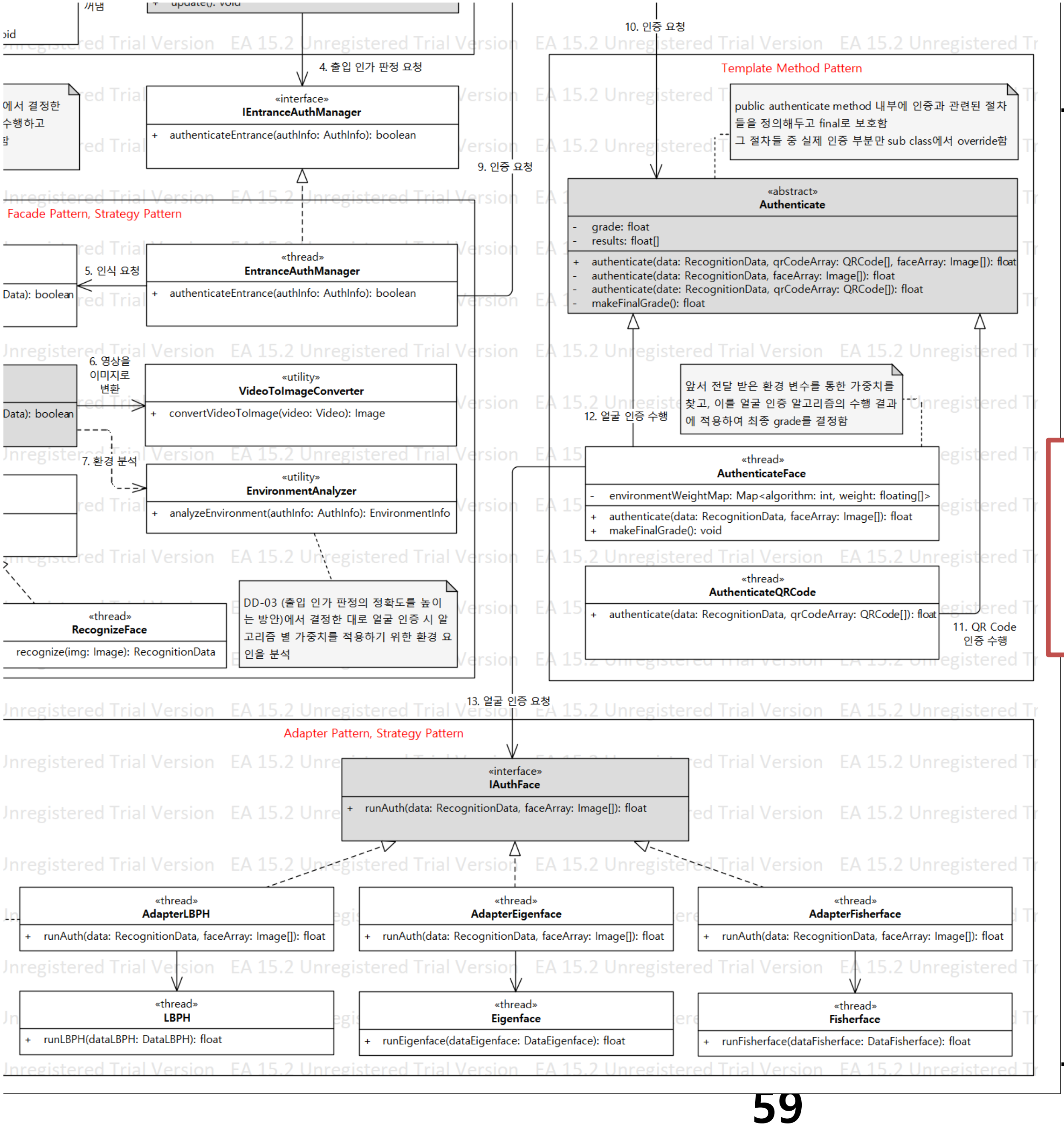

class EntranceAuthenticator Component Structure View
«interface»
Entrance Auth Business Logic Interface::
IAuthenticateEntrance
+ authenticateEntrance(authInfo: AuthInfo): boolean
Provided Interface
Observer Pattern, Singleton Pattern
«singleton,publisher»
AuthenticateEntrance
+ authenticateEntrance(authInfo: AuthInfo): boolean
1. 출입 인가 판정 요청이 들어오면 queue에 저장
«interface»
IEntranceAuthReqQueue
+ add(reqAuthInfo: AuthInfo): void
+ notify(): void
+ poll(): AuthInfo
+ register(observer: IEntranceAuthObserver): void
+ unregister(observer: IEntranceAuthObserver): void
2. queue에 add되면 update callback 호출
«interface»
IEntranceAuthObserver
+ update(): void
«thread-safe,singleton»
EntranceAuthReqQueue
- observer: IEntranceAuthObserver
- queue: Queue<AuthInfo>
+ add(reqAuthInfo: AuthInfo): void
+ getInstance(): EntranceAuthReqQueue
+ notify(): void
+ poll(): AuthInfo
+ register(observer: IEntranceAuthObserver): void
+ unregister(observer: IEntranceAuthObserver): void
3. poll하여 authInfo를 꺼냄
«subscriber»
EntranceAuthReqEventController
- reqAuthUinfo: AuthInfo
+ update(): void
4. 출입 인가 판정 요청
«interface»
IEntranceAuthManager
+ authenticateEntrance(authInfo: AuthInfo): boolean
9. 인증 요청
Facade Pattern, Strategy Pattern
«interface»
IRecogizeFacade
+ recognize(authInfo: AuthInfo, data: RecognitionData): boolean
5. 인식 요청
«thread»
EntranceAuthManager
+ authenticateEntrance(authInfo: AuthInfo): boolean
«facade,thread»
RecognizeFacade
+ recognize(authInfo: AuthInfo, data: RecognitionData): boolean
6. 영상을 이미지로 변환
«utility»
VideoToImageConverter
+ convertVideoToImage(video: Video): Image
7. 환경 분석
«utility»
EnvironmentAnalyzer
+ analyzeEnvironment(authInfo: AuthInfo): EnvironmentInfo
«interface»
Recognize
+ recognize(img: Image): RecognitionData
8. 인식을 수행
«thread»
RecognizeQRCode
+ recognize(img: Image): RecognitionData
«thread»
RecognizeFace
+ recognize(img: Image): RecognitionData

출입 인가 판정 요청이 들어오면 우선 queue에 저장함
이후 queue가 변경 되었으므로, 미리 등록되어있던 observer의 update 를 호출함

DD-02 (출입 인가 판정의 정확도를 위한 사전 처리)에서 결정한 바와 같이 인증 절차를 수행하기 전에, 인식을 먼저 수행하고 QRCode나 얼굴이 인식 되지 않는다면 그대로 종료함

DD-03 (출입 인가 판정의 정확도를 높이는 방안)에서 결정한 대로 얼굴 인증 시 알고리즘 별 가중치를 적용하기 위한 환경 요인을 분석

Eventcontroller가 class에 직접 접근하는 것이 적절한지?
DIP 위반?
Façade는 적절함.

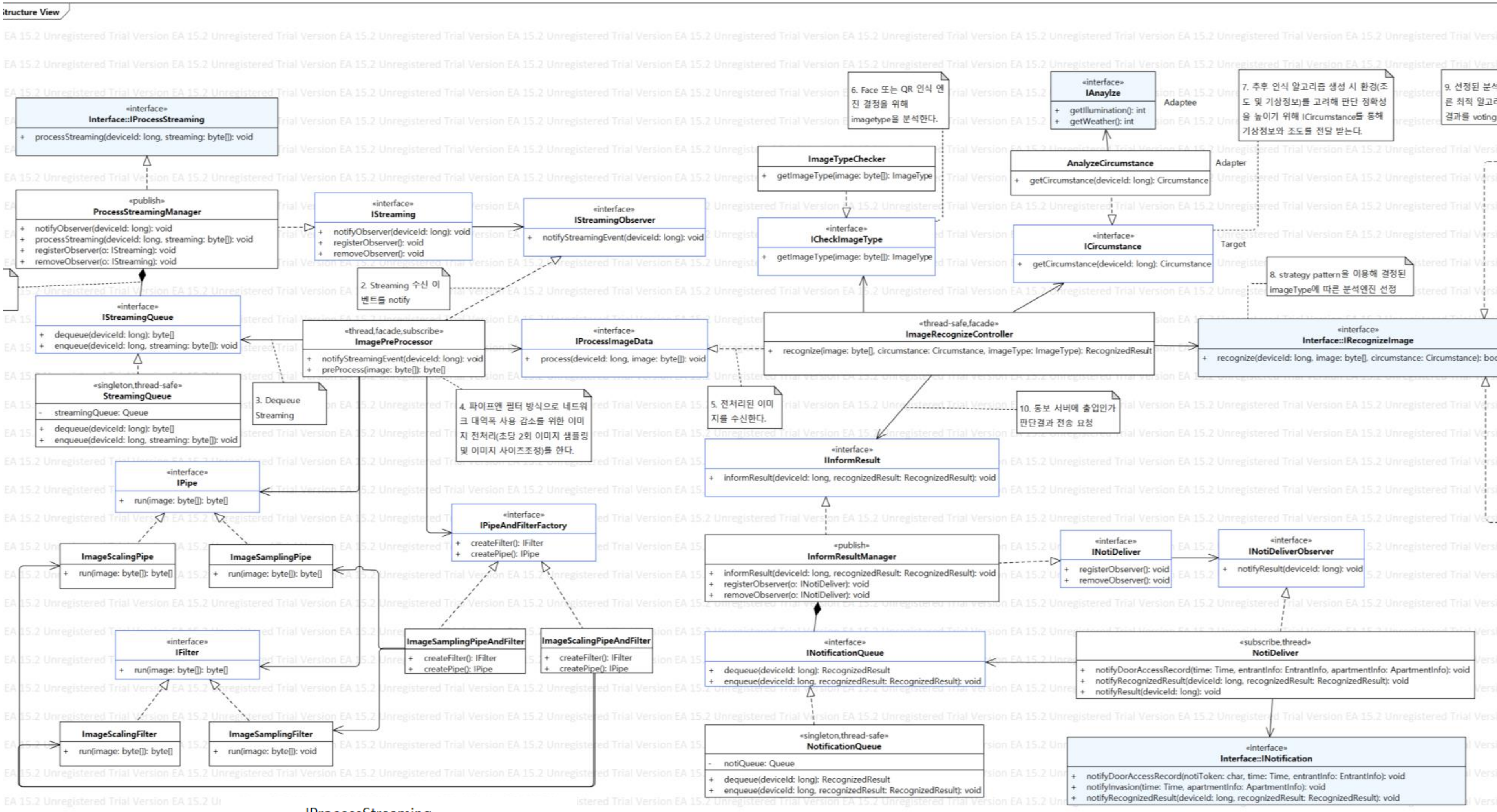

10. 인증 요청
Template Method Pattern
4. 출입 인가 판정 요청
«interface»
IEntranceAuthManager
+ authenticateEntrance(authInfo: AuthInfo): boolean
9. 인증 요청
«abstract»
Authenticate
- grade: float
- results: float[]
+ authenticate(data: RecognitionData, qrCodeArray: QRCode[], faceArray: Image[]): float
- authenticate(data: RecognitionData, faceArray: Image[]): float
- authenticate(date: RecognitionData, qrCodeArray: QRCode[]): float
- makeFinalGrade(): float
Facade Pattern, Strategy Pattern
5. 인식 요청
Data): boolean
«thread»
EntranceAuthManager
+ authenticateEntrance(authInfo: AuthInfo): boolean
6. 영상을 이미지로 변환
Data): boolean
«utility»
VideoToImageConverter
+ convertVideoToImage(video: Video): Image
12. 얼굴 인증 수행
7. 환경 분석
«utility»
EnvironmentAnalyzer
+ analyzeEnvironment(authInfo: AuthInfo): EnvironmentInfo
«thread»
AuthenticateFace
- environmentWeightMap: Map<algorithm: int, weight: floating[]>
+ authenticate(data: RecognitionData, faceArray: Image[]): float
+ makeFinalGrade(): void
«thread»
AuthenticateQRCode
+ authenticate(data: RecognitionData, qrCodeArray: QRCode[]): float
11. QR Code 인증 수행
«thread»
RecognizeFace
recognize(img: Image): RecognitionData
13. 얼굴 인증 요청
Adapter Pattern, Strategy Pattern
«interface»
IAuthFace
+ runAuth(data: RecognitionData, faceArray: Image[]): float
«thread»
AdapterLBPH
+ runAuth(data: RecognitionData, faceArray: Image[]): float
«thread»
AdapterEigenface
+ runAuth(data: RecognitionData, faceArray: Image[]): float
«thread»
AdapterFisherface
+ runAuth(data: RecognitionData, faceArray: Image[]): float
«thread»
LBPH
+ runLBPH(dataLBPH: DataLBPH): float
«thread»
Eigenface
+ runEigenface(dataEigenface: DataEigenface): float
«thread»
Fisherface
+ runFisherface(dataFisherface: DataFisherface): float

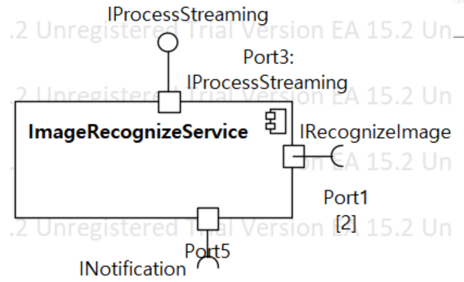
public authenticate method 내부에 인증과 관련된 절차들을 정의해두고 final로 보호함 그 절차들 중 실제 인증 부분만 sub class에서 override함

앞서 전달 받은 환경 변수를 통한 가중치를 찾고, 이를 얼굴 인증 알고리즘의 수행 결과에 적용하여 최종 grade를 결정함

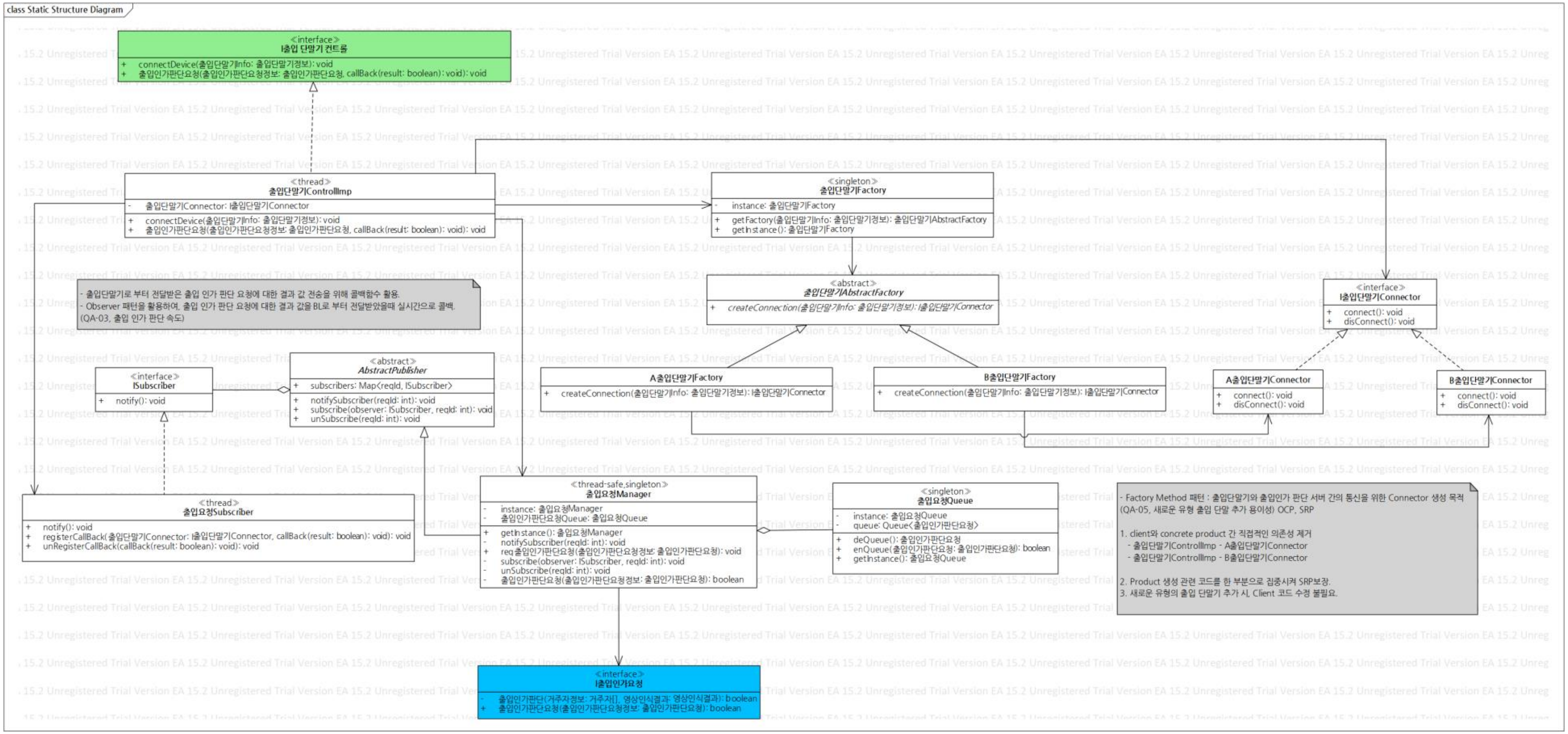
DD-03 (출입 인가 판정의 정확도를 높이는 방안)에서 결정한 대로 얼굴 인증 시 알고리즘 별 가중치를 적용하기 위한 환경 요인을 분석

Template method pattern은 부적절함
Adaptee는 외부 코드이어야 함
환경을 감안하여 알고리즘 별 가중치 부여 ➔ 11 번 multiplicity 필요. 3개 인식 알고리즘에게 데이터 전달의 overhead는?


Structure View
«interface» Interface::IProcessStreaming
+ processStreaming(deviceId: long, streaming: byte[]): void
«publish» ProcessStreamingManager
+ notifyObserver(deviceId: long): void
+ processStreaming(deviceId: long, streaming: byte[]): void
+ registerObserver(o: IStreaming): void
+ removeObserver(o: IStreaming): void
«interface» IStreaming
+ notifyObserver(deviceId: long): void
+ registerObserver(): void
+ removeObserver(): void
«interface» IStreamingObserver
+ notifyStreamingEvent(deviceId: long): void
2. Streaming 수신 이벤트를 notify
«interface» IStreamingQueue
+ dequeue(deviceId: long): byte[]
+ enqueue(deviceId: long, streaming: byte[]): void
«thread,facade,subscribe» ImagePreProcessor
+ notifyStreamingEvent(deviceId: long): void
+ preProcess(image: byte[]): byte[]
«interface» IProcessImageData
+ process(deviceId: long, image: byte[]): void
«singleton,thread-safe» StreamingQueue
- streamingQueue: Queue
+ dequeue(deviceId: long): byte[]
+ enqueue(deviceId: long, streaming: byte[]): void
3. Dequeue Streaming
4. 파이프엔 필터 방식으로 네트워크 대역폭 사용 감소를 위한 이미지 전처리(초당 2회 이미지 샘플링 및 이미지 사이즈조정)를 한다.
5. 전처리된 이미지를 수신한다.
«interface» IPipe
+ run(image: byte[]): byte[]
ImageScalingPipe
+ run(image: byte[]): byte[]
ImageSamplingPipe
+ run(image: byte[]): byte[]
«interface» IPipeAndFilterFactory
+ createFilter(): IFilter
+ createPipe(): IPipe
ImageSamplingPipeAndFilter
+ createFilter(): IFilter
+ createPipe(): IPipe
ImageScalingPipeAndFilter
+ createFilter(): IFilter
+ createPipe(): IPipe
«interface» IFilter
+ run(image: byte[]): byte[]
ImageScalingFilter
+ run(image: byte[]): byte[]
ImageSamplingFilter
+ run(image: byte[]): void
6. Face 또는 QR 인식 엔진 결정을 위해 imagetype을 분석한다.
ImageTypeChecker
+ getImageType(image: byte[]): ImageType
«interface» ICheckImageType
+ getImageType(image: byte[]): ImageType
«interface» IAnaylze
+ getIllumination(): int
+ getWeather(): int
Adaptee
AnalyzeCircumstance
+ getCircumstance(deviceId: long): Circumstance
Adapter
«interface» ICircumstance
+ getCircumstance(deviceId: long): Circumstance
Target
7. 추후 인식 알고리즘 생성 시 환경(조도 및 기상정보)를 고려해 판단 정확성을 높이기 위해 ICircumstance를 통해 기상정보와 조도를 전달 받는다.
8. strategy pattern을 이용해 결정된 imageType에 따른 분석엔진 선정
9. 선정된 분석 른 최적 알고리 결과를 voting
«thread-safe,facade» ImageRecognizeController
+ recognize(image: byte[], circumstance: Circumstance, imageType: ImageType): RecognizedResult
«interface» Interface::IRecognizeImage
+ recognize(deviceId: long, image: byte[], circumstance: Circumstance): boo
10. 통보 서버에 출입인가 판단결과 전송 요청
«interface» IInformResult
+ informResult(deviceId: long, recognizedResult: RecognizedResult): void
«publish» InformResultManager
+ informResult(deviceId: long, recognizedResult: RecognizedResult): void
+ registerObserver(o: INotiDeliver): void
+ removeObserver(o: INotiDeliver): void
«interface» INotiDeliver
+ registerObserver(): void
+ removeObserver(): void
«interface» INotiDeliverObserver
+ notifyResult(deviceId: long): void
«interface» INotificationQueue
+ dequeue(deviceId: long): RecognizedResult
+ enqueue(deviceId: long, recognizedResult: RecognizedResult): void
«subscribe,thread» NotiDeliver
+ notifyDoorAccessRecord(time: Time, entrantInfo: EntrantInfo, apartmentInfo: ApartmentInfo): void
+ notifyRecognizedResult(deviceId: long, recognizedResult: RecognizedResult): void
+ notifyResult(deviceId: long): void
«singleton,thread-safe» NotificationQueue
- notiQueue: Queue
+ dequeue(deviceId: long): RecognizedResult
+ enqueue(deviceId: long, recognizedResult: RecognizedResult): void
«interface» Interface::INotification
+ notifyDoorAccessRecord(notiToken: char, time: Time, entrantInfo: EntrantInfo): void
+ notifyInvasion(time: Time, apartmentInfo: ApartmentInfo): void
+ notifyRecognizedResult(deviceId: long, recognizedResult: RecognizedResult): void



Observer pattern의 구조에 부합되는 abstract class, interface 사용 필요
특정 filter에 특정 pipe가 사용되어야 하는지? Filter와 pipe간의 독립성이 권장됨
이를 고려하지 않는 다면 abstract factory pattern의 사용이 적절함
pipe가 filter의 역할을 하는 것 아닌지?
이미지 타입의 결정은 누가 하는지?


class Static Structure Diagram
«interface»
I출입 단말기 컨트롤
+ connectDevice(출입단말기Info: 출입단말기정보): void
+ 출입인가판단요청(출입인가판단요청정보: 출입인가판단요청, callBack(result: boolean): void): void
«thread»
출입단말기ControllImp
- 출입단말기Connector: I출입단말기Connector
+ connectDevice(출입단말기Info: 출입단말기정보): void
+ 출입인가판단요청(출입인가판단요청정보: 출입인가판단요청, callBack(result: boolean): void): void
«singleton»
출입단말기Factory
- instance: 출입단말기Factory
+ getFactory(출입단말기Info: 출입단말기정보): 출입단말기AbstractFactory
+ getInstance(): 출입단말기Factory
«abstract»
출입단말기AbstractFactory
+ createConnection(출입단말기Info: 출입단말기정보): I출입단말기Connector
A출입단말기Factory
+ createConnection(출입단말기Info: 출입단말기정보): I출입단말기Connector
B출입단말기Factory
+ createConnection(출입단말기Info: 출입단말기정보): I출입단말기Connector
«interface»
I출입단말기Connector
+ connect(): void
+ disConnect(): void
A출입단말기Connector
+ connect(): void
+ disConnect(): void
B출입단말기Connector
+ connect(): void
+ disConnect(): void
«interface»
ISubscriber
+ notify(): void
«abstract»
AbstractPublisher
+ subscribers: Map<reqId, ISubscriber>
+ notifySubscriber(reqId: int): void
+ subscribe(observer: ISubscriber, reqId: int): void
+ unSubscribe(reqId: int): void
«thread»
출입요청Subscriber
+ notify(): void
+ registerCallBack(출입단말기Connector: I출입단말기Connector, callBack(result: boolean): void): void
+ unRegisterCallBack(callBack(result: boolean): void): void
«thread-safe,singleton»
출입요청Manager
- instance: 출입요청Manager
- 출입인가판단요청Queue: 출입요청Queue
+ getInstance(): 출입요청Manager
- notifySubscriber(reqId: int): void
+ req출입인가판단요청(출입인가판단요청정보: 출입인가판단요청): void
- subscribe(observer: ISubscriber, reqId: int): void
- unSubscribe(reqId: int): void
- 출입인가판단요청(출입인가판단요청정보: 출입인가판단요청): boolean
«singleton»
출입요청Queue
- instance: 출입요청Queue
- queue: Queue<출입인가판단요청>
+ deQueue(): 출입인가판단요청
+ enQueue(출입인가판단요청: 출입인가판단요청): boolean
+ getInstance(): 출입요청Queue
«interface»
I출입인가요청
- 출입인가판단(거주자정보: 거주자[], 영상인식결과: 영상인식결과): boolean
+ 출입인가판단요청(출입인가판단요청정보: 출입인가판단요청): boolean

- 출입단말기로 부터 전달받은 출입 인가 판단 요청에 대한 결과 값 전송을 위해 콜백함수 활용.
- Observer 패턴을 활용하여, 출입 인가 판단 요청에 대한 결과 값을 BL로 부터 전달받았을때 실시간으로 콜백.
(QA-03, 출입 인가 판단 속도)

- Factory Method 패턴 : 출입단말기와 출입인가 판단 서버 간의 통신을 위한 Connector 생성 목적
(QA-05, 새로운 유형 출입 단말 추가 용이성) OCP, SRP

1. client와 concrete product 간 직접적인 의존성 제거
 - 출입단말기ControllImp - A출입단말기Connector
 - 출입단말기ControllImp - B출입단말기Connector

2. Product 생성 관련 코드를 한 부분으로 집중시켜 SRP보장.
3. 새로운 유형의 출입 단말기 추가 시, Client 코드 수정 불필요.

Abstract factory vs factory method
Concrete Subject: subscribe 등의 구현?

Q&A