6. Design: Design Patterns

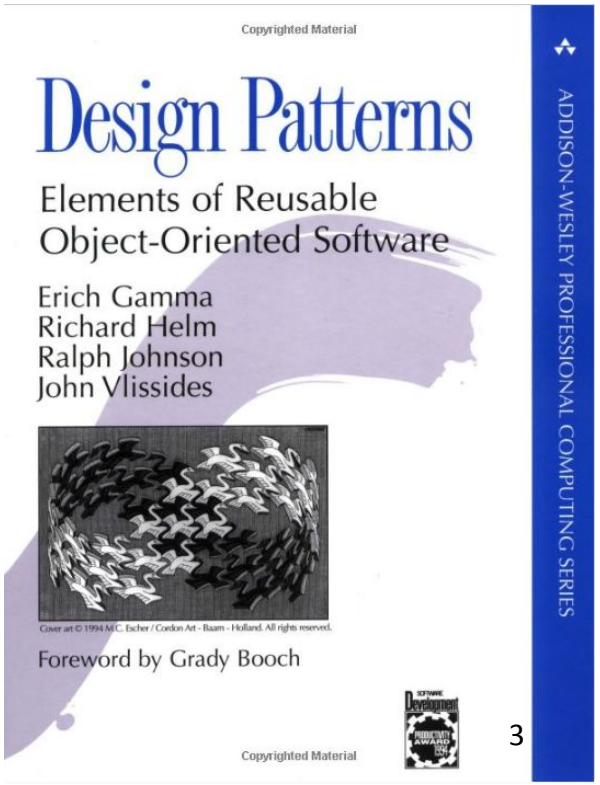
Choi, Kwanghoon

Chonnam National University

Table of Contents

6.5 Design Patterns

- A description of problem and the essence of its solution reusable in different settings
- Design Patterns: Elements of Reusable Object-Oriented Software by GoF (Gang of Four)
  - 23 Design Patterns in 3 types:
    - Creational Pattern types
    - Structural Pattern types
    - Behavioral Pattern types



http://www.uml.org.cn/c++/pdf/DesignPatterns.pdf

Design Pattern Types

- Creational patterns
  - Creation and composition of objects, mechanism to instantiate objects easier, and constraints on the type and number of objects
- Structural patterns
  - How classes and objects are organized and integrated to build a larger structure
- Behavioral patterns
  - The assignment of responsibility between objects and the manner in which communication is effected between objects

23 Design Patterns

Creational Patterns	Structural Patterns	Behavioral Patterns
▪ Abstract Factory ▪ Factory method ▪ Builder ▪ Prototype ▪ Singleton	▪ Adapter ▪ Bridge ▪ Composite ▪ Decorator ▪ Façade ▪ Flyweight ▪ Proxy	▪ Chain of Responsibility ▪ Command ▪ Interpreter ▪ Iterator ▪ Mediator ▪ Memento ▪ Observer ▪ State ▪ Strategy ▪ Template Method ▪ Visitor

Creational Patterns

- **Abstract factory pattern** groups object factories that have a common theme.
- **Factory method pattern** creates an object without specifying the exact class to create.
- **Builder pattern** constructs complex objects by separating construction and representation.
- **Prototype pattern** creates objects by cloning an existing object.
- **Singleton pattern** restricts object creation for a class to only one instance.

Structural Patterns

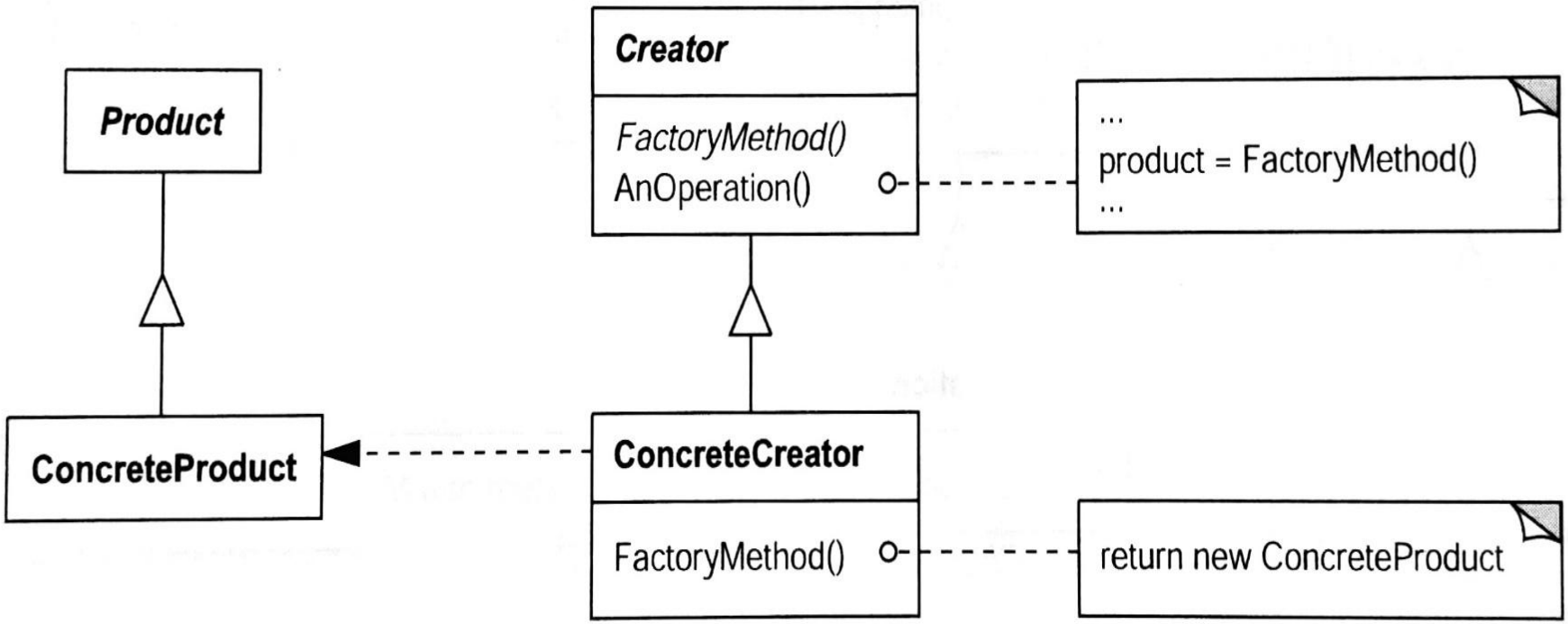
- **Adapter pattern** allows classes with incompatible interfaces to work together by wrapping its own interface around that of an already existing class.
- **Bridge pattern** decouples an abstraction from its implementation so that the two can vary independently.
- **Composite pattern** composes zero-or-more similar objects so that they can be manipulated as one object.
- **Decorator pattern** dynamically adds/overrides behavior in an existing method of an object.
- **Façade pattern** provides a simplified interface to a large bod of code.
- **Flyweight pattern** reduces the cost of creating and manipulating a large number of similar objects.
- **Proxy pattern** provides a placeholder for another object to control access, reduce cost, and reduce complexity.

Behavioral Patterns

- **Chain of Responsibility pattern** delegates commands to a chain of processing objects.
- **Command pattern** creates objects which encapsulate actions and parameters.
- **Interpreter pattern** implements a specialized language.
- **Iterator pattern** accesses the elements of an object sequentially without exposing its underlying representation.
- **Mediator pattern** allows loose coupling between classes by being the only class that has detailed knowledge of their methods.
- **Memento pattern** provides the ability to restore an object to its previous state (undo).
- **Observer pattern** is a publish/subscribe pattern which allows a number of observer objects to see an event.
- **State pattern** allows an object to alter its behavior when its internal state changes.
- **Strategy pattern** allows one of a family of algorithms to be selected on-the-fly at runtime.
- **Template Method** pattern defines the skeleton of an algorithm as an abstract class, allowing its subclasses to provide concrete behavior.
- **Visitor pattern** separates an algorithm from an object structure by moving the hierarchy of methods into one object.

Factory Method Pattern

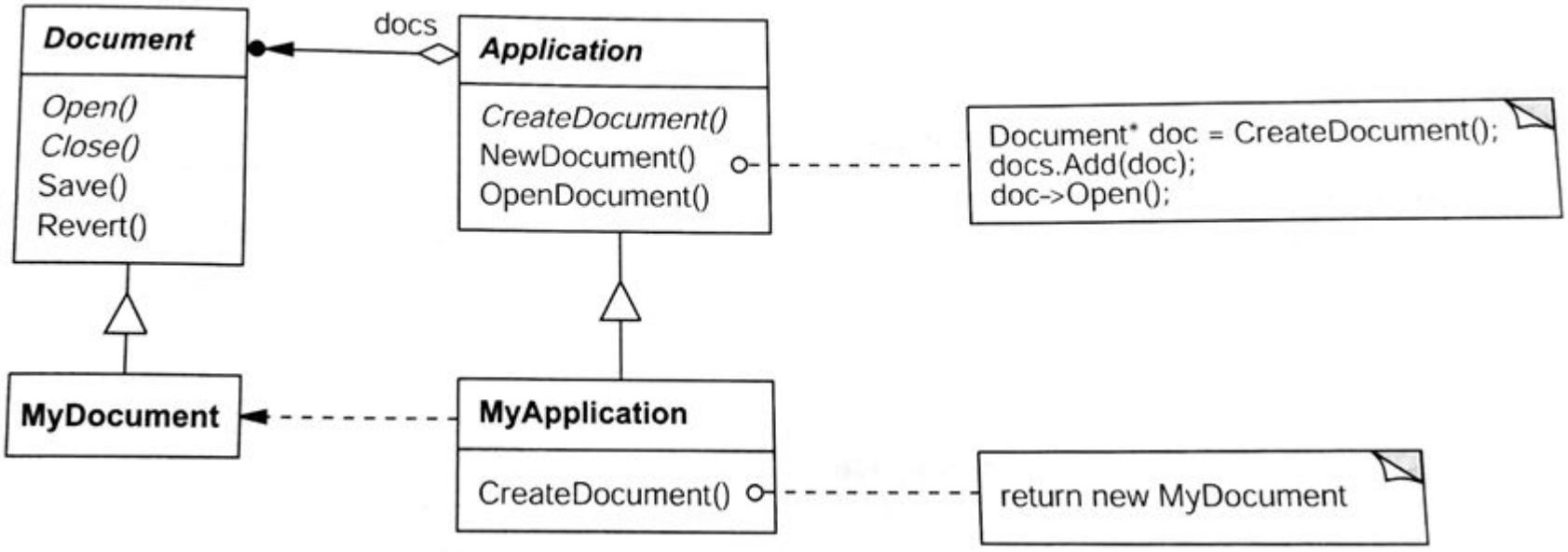
- Creates an object without specifying the exact class (name) to create

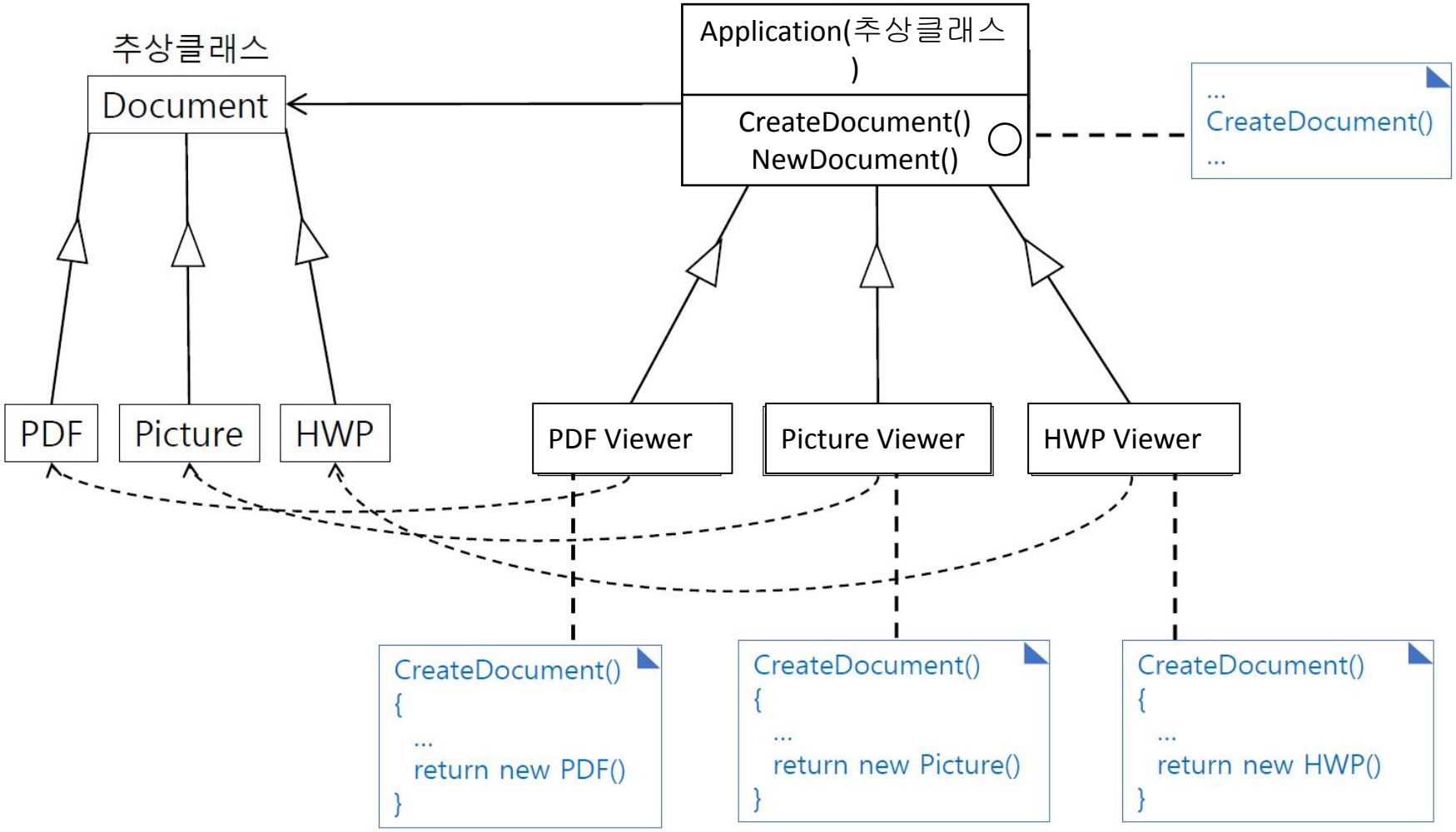


Factory Method Pattern

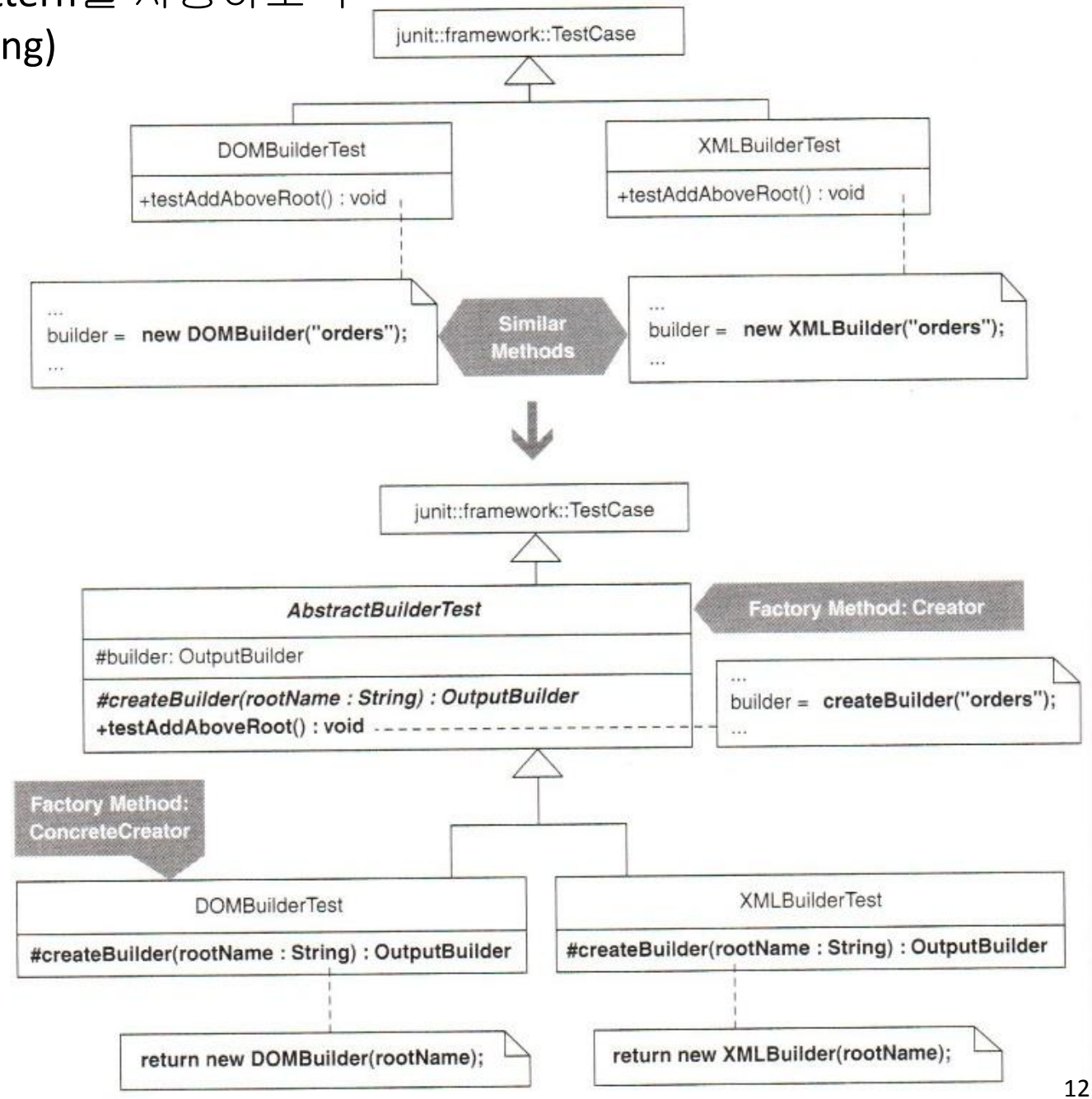
- Creates an object without specifying the exact class to create

Example) CreateDocument()




추상클래스
Document
Application(추상클래스)
CreateDocument()
NewDocument()
...
CreateDocument()
...
PDF
Picture
HWP
PDF Viewer
Picture Viewer
HWP Viewer
CreateDocument()
{
...
return new PDF()
}
CreateDocument()
{
...
return new Picture()
}
CreateDocument()
{
...
return new HWP()
}

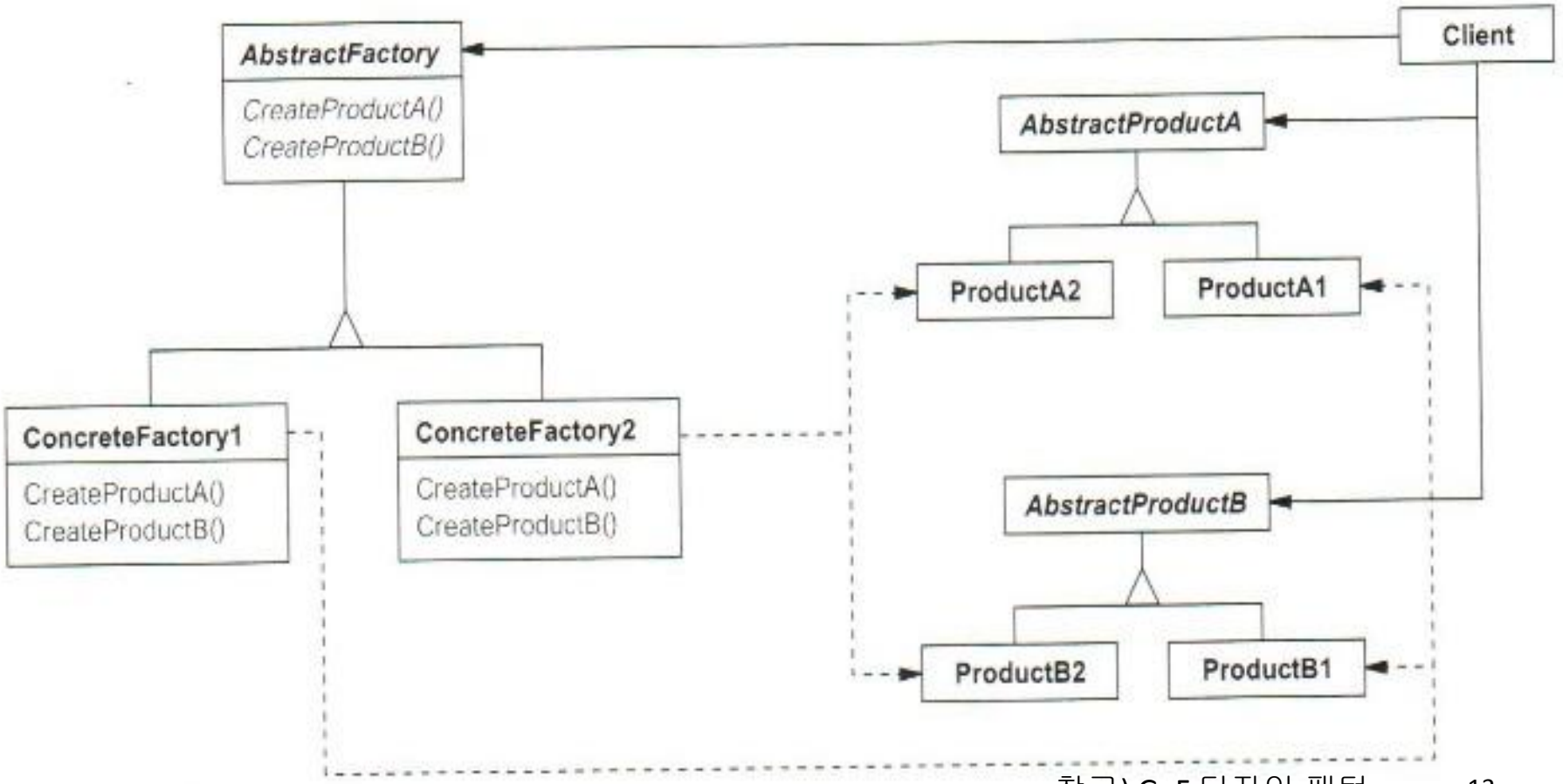
factory method pattern을 사용하도록 리팩토링(refactoring)

- 

참고) Joshua Kerievsky, Refactoring to design patterns

Abstract Factory Pattern

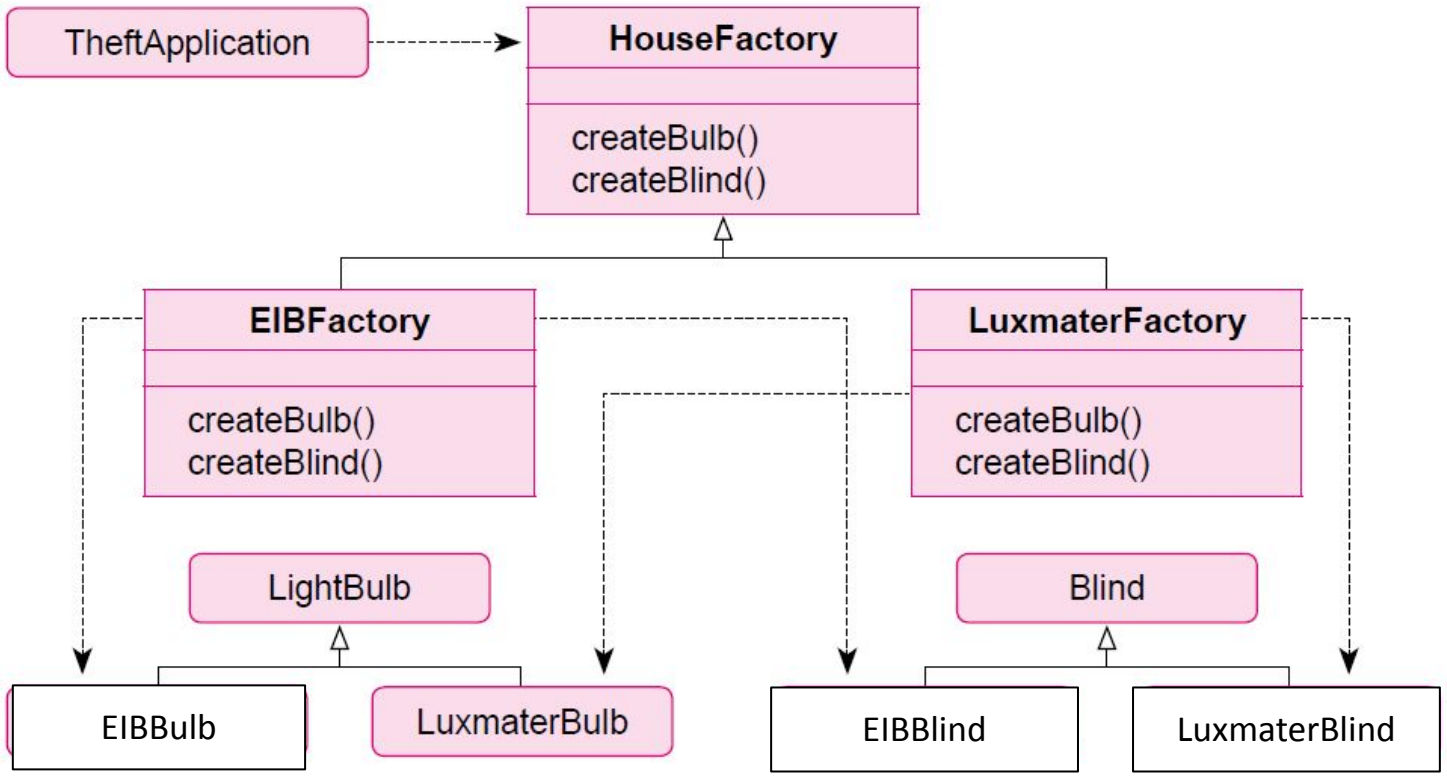
- groups object factories that have a common theme



참고) GoF 디자인 패턴

Abstract Factory Pattern

- groups object factories that have a common theme



참고) 최은만 교재

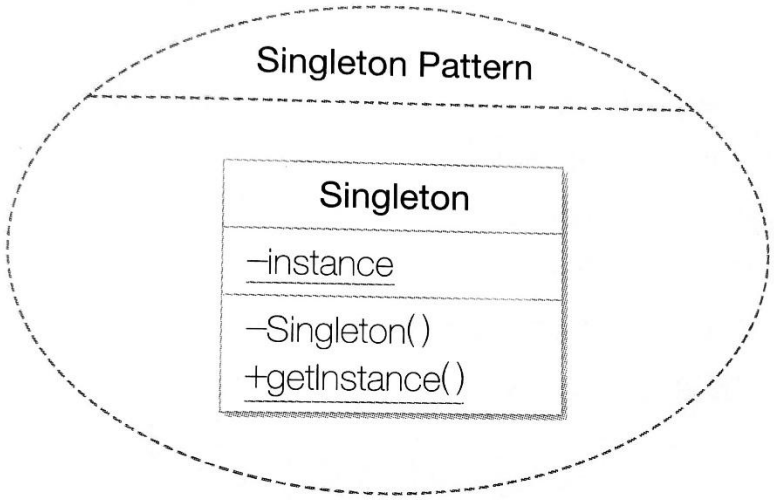
Abstract Factory Pattern

- Q. Draw a class diagram for the following Java programs
  - DesignPatternExamples/src/com/example/designpattern/abstractfactory
- Q. Explain the role of FactoryFactory
  - The ElevatorFactoryFactory class

Singleton Pattern

- restricts object creation for a class to only one instance.

```
public class Singleton {
  private static Singleton instance = null;
  private Singleton() { … }
  public static Singleton getInstance() {
    if (instance == null)
      instance = new Singleton();
    return instance;
  }
  public void doSomething() {
    …
  }
}
```

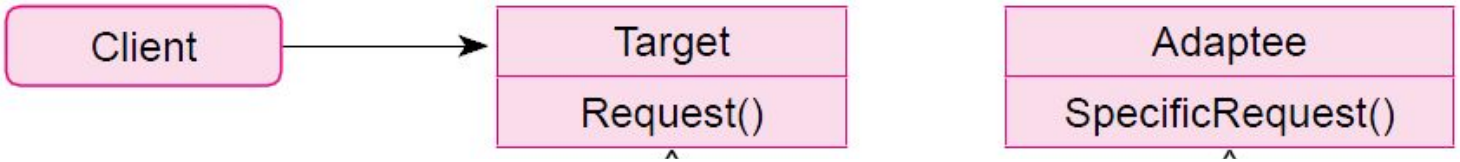


참고) 정인상, 채흥석, Java로 배우는 디자인패턴

Adapter Pattern

- allows classes with incompatible interfaces to work together by wrapping its own interface around a legacy system.

[Problem]

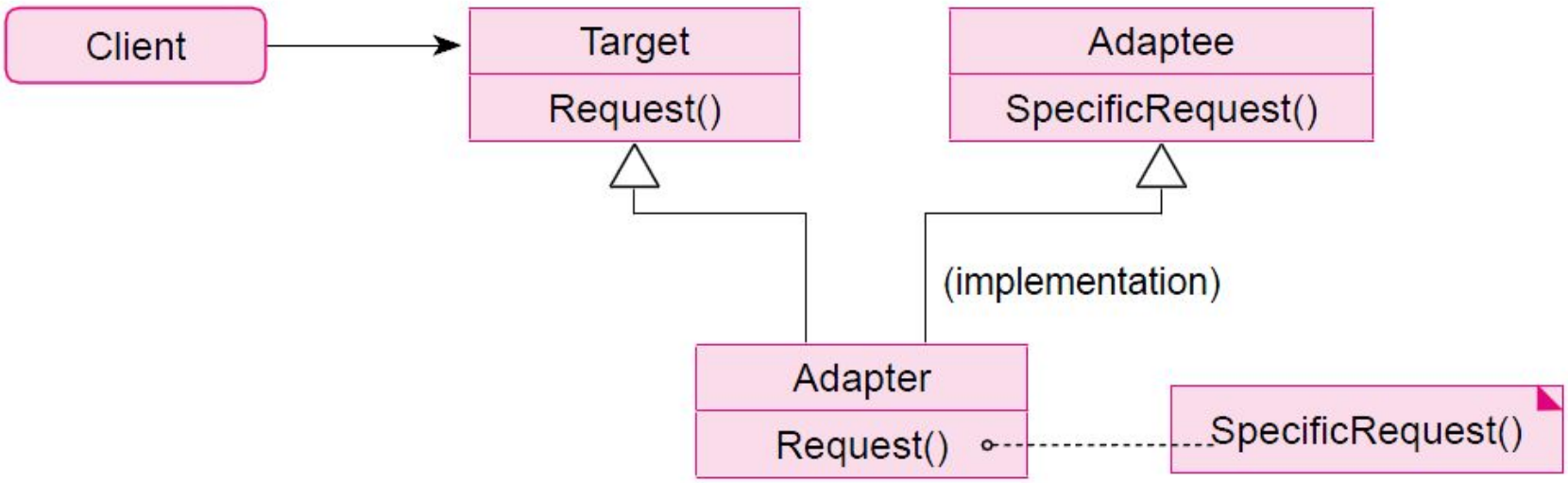


참고) 최은만 교재

Adapter Pattern

- allows classes with incompatible interfaces to work together by wrapping its own interface around a legacy system.

[Solution]



참고) 최은만 교재

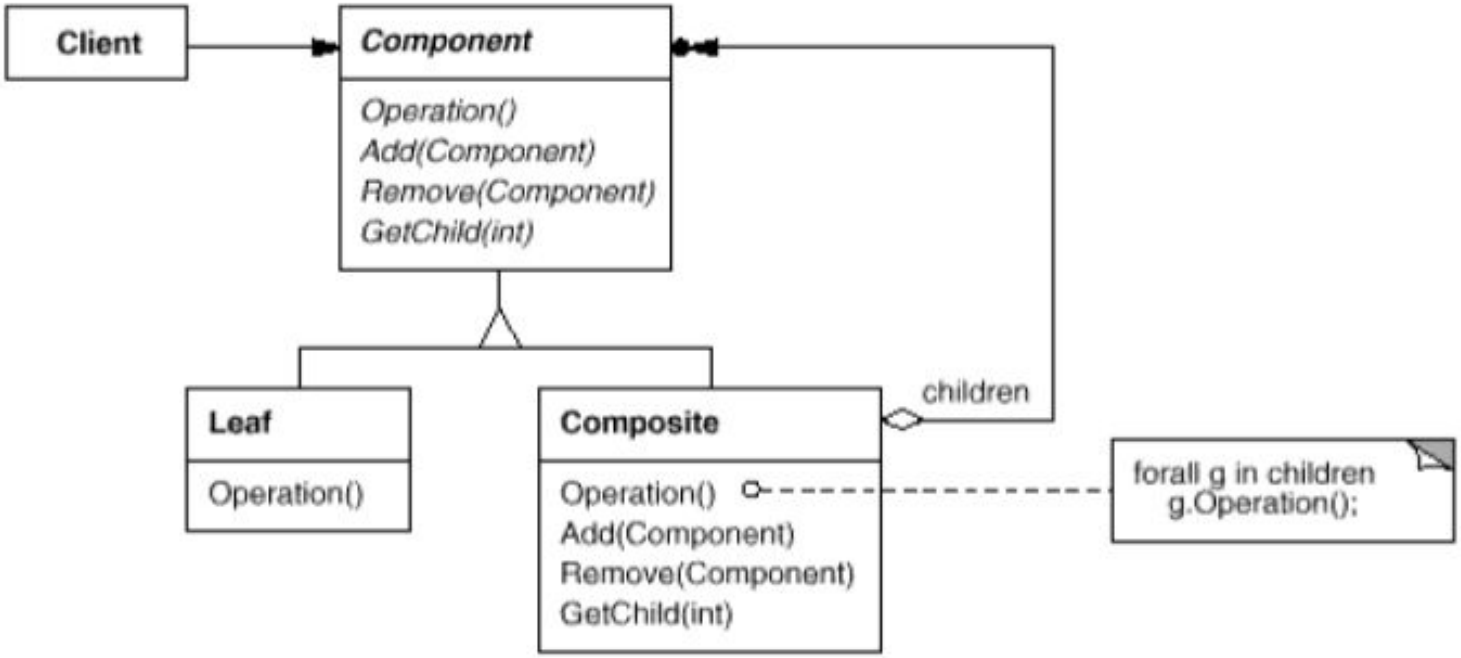
Adapter Pattern

- Q. Draw a class diagram for the following Java programs
  - DesignPatternExamples/src/com/example/designpattern/adapter
  - Explain how Client can call specificRequest() in ClassB.

- Q. How is it different from the class diagram in the previous slide?
  - Explain each role of the two inheritance relationships in the previous slide. (Target <- Adapter, Adaptee <- Adapter)
  - Draw a new class diagram for the adapter pattern based on the answer to the second question.

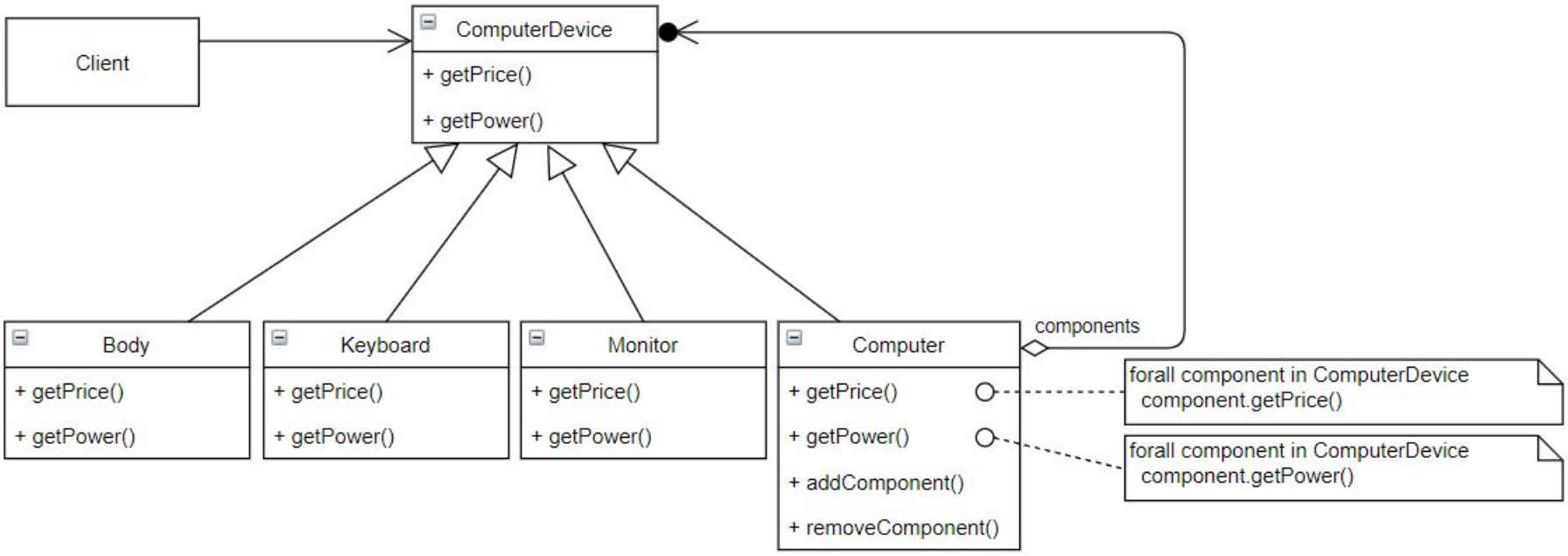
Composite Pattern

- composes zero-or-more similar objects so that they can be manipulated as one object.



Composite Pattern

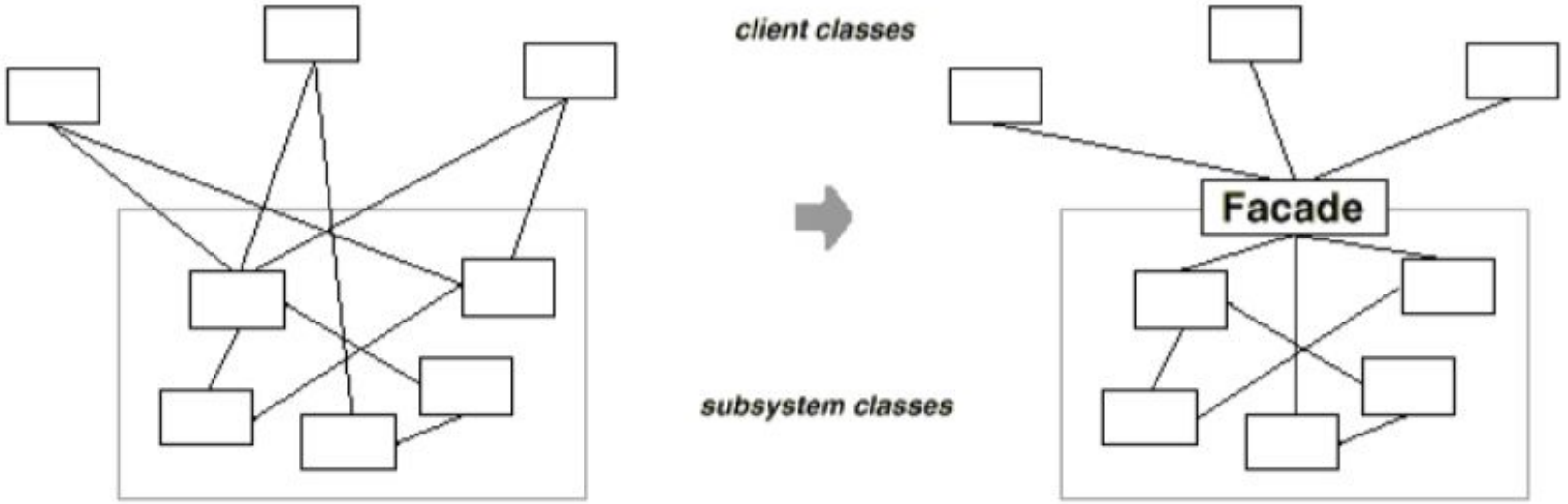
- Draw a class diagram for Computer, ComputerDevice, Client, etc. using this pattern



참고) 정인상, 채흥석, Java로 배우는 디자인패턴

Façade Pattern

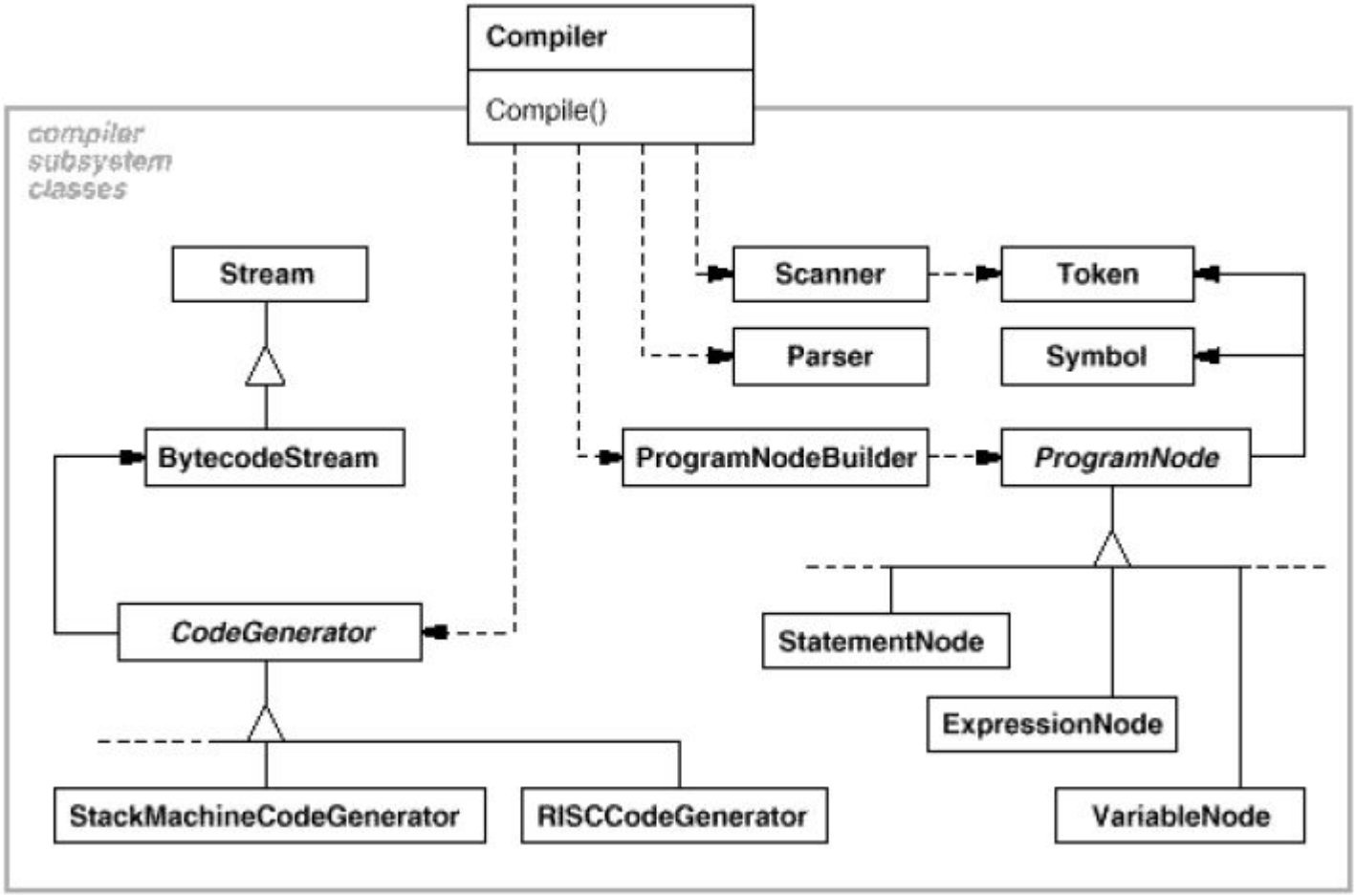
- provides a simplified interface to a large body of code.



참고) 최은만 교재

Façade Pattern

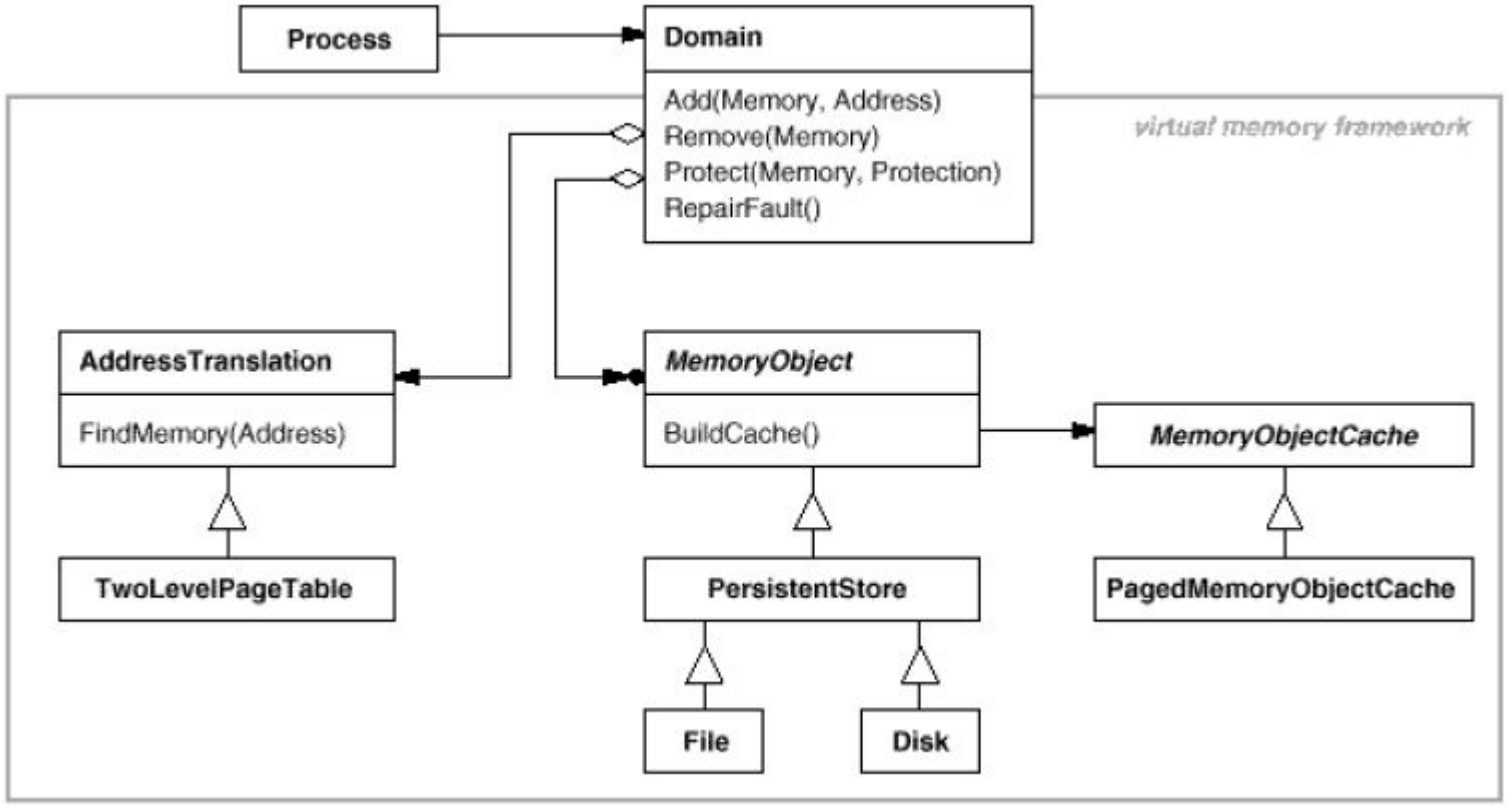
- Compiler subsystem classes



참고) GoF 디자인 패턴

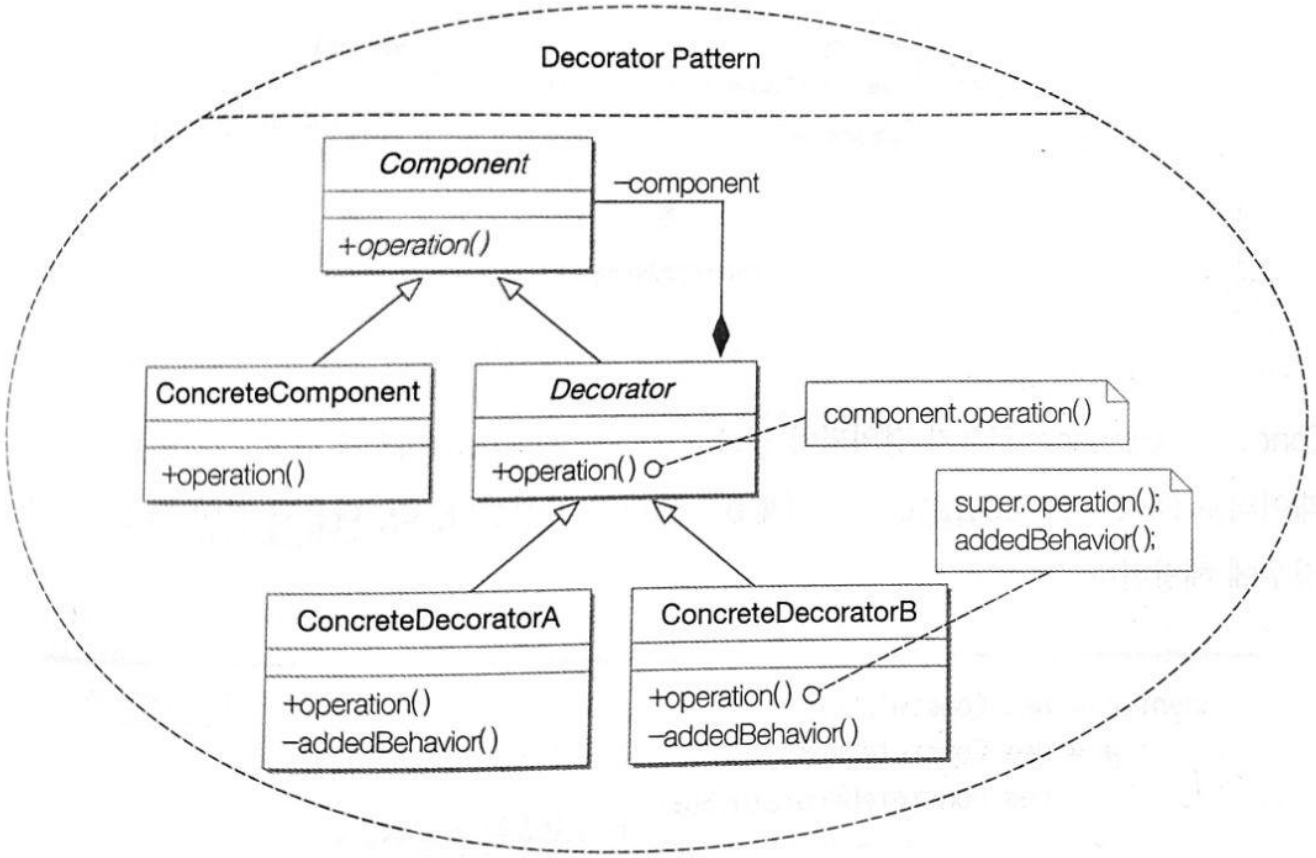
Façade Pattern

- Virtual memory framework



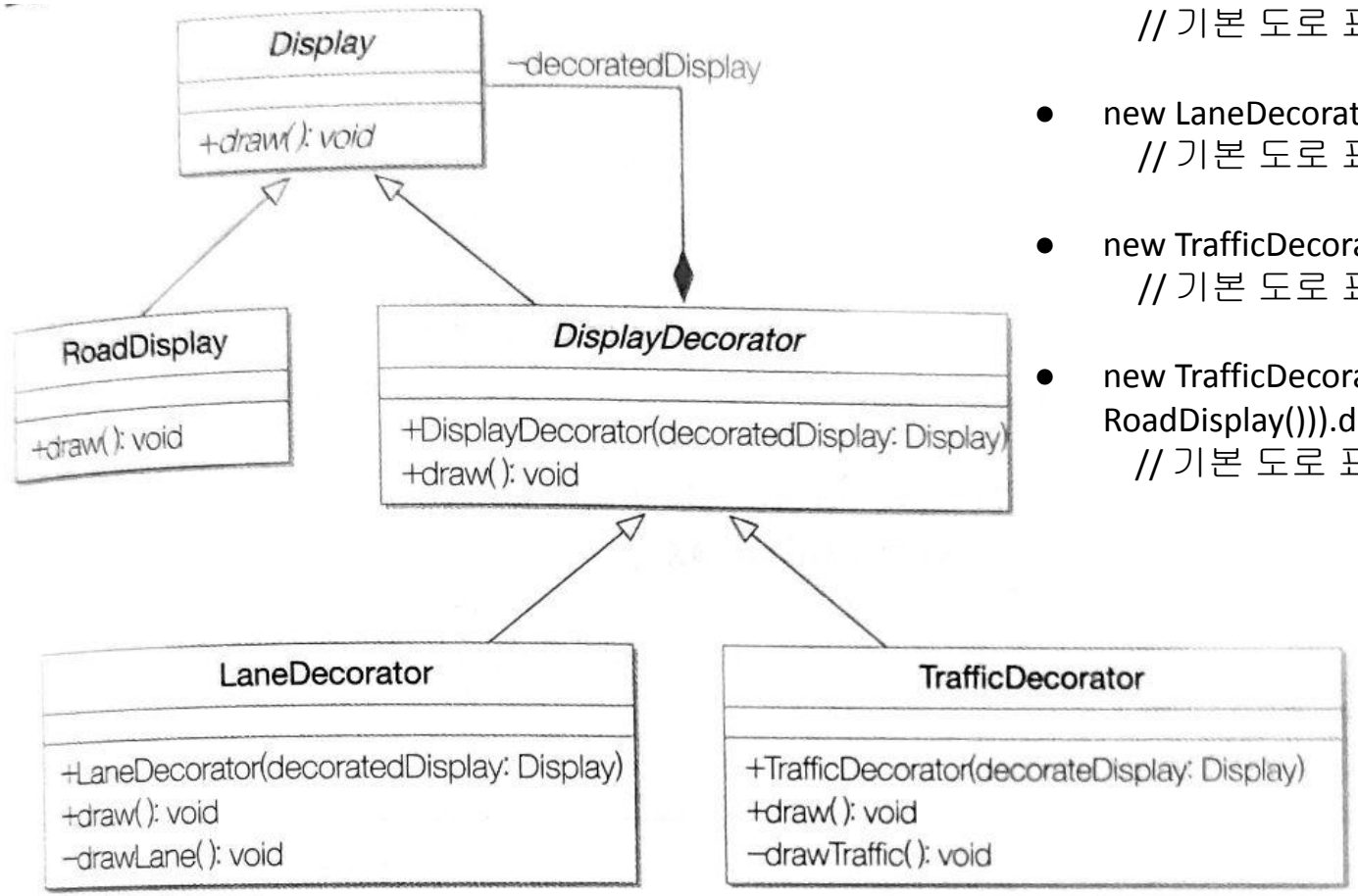
Decorator Pattern

- dynamically adds/overrides behavior in an existing method of an object.



Decorator Pattern

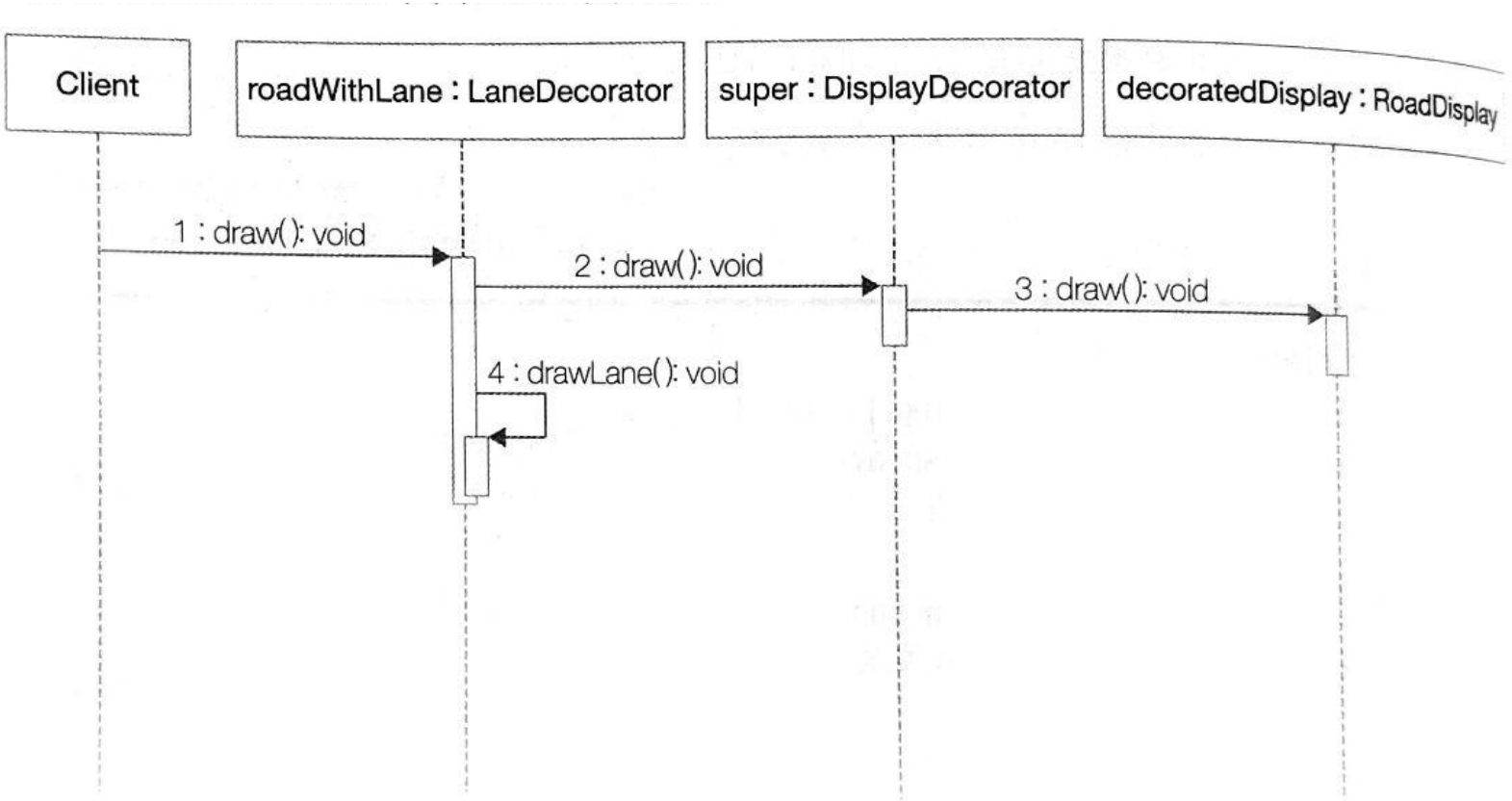
- Various combinations of road displays



- new RoadyDisplay().draw()
  // 기본 도로 표시
- new LaneDecorator(new RoadDisplay()).draw()
  // 기본 도로 표시 + 차선 표시
- new TrafficDecorator(new RoadDisplay()).draw()
  // 기본 도로 표시 + 교통량 표시
- new TrafficDecorator(new LaneDecorator(new RoadDisplay())).draw()
  // 기본 도로 표시 + 차선 표시 + 교통량 표시

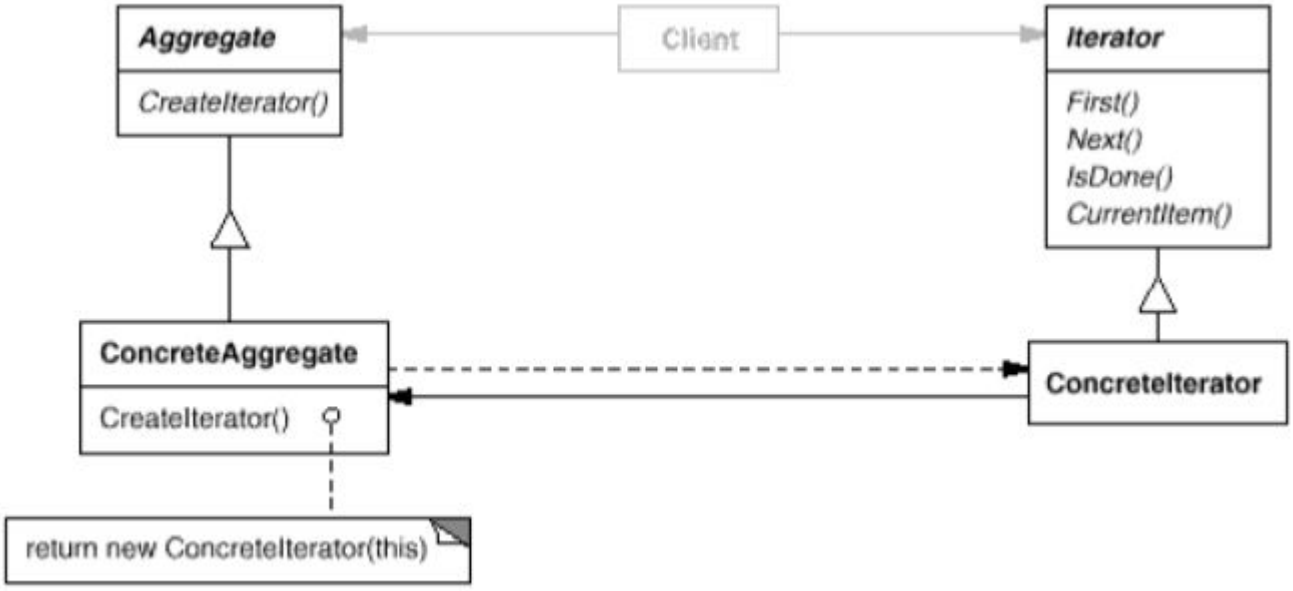
Decorator Pattern

- Sequence diagram (roadWithLane.draw)



Iterator Pattern

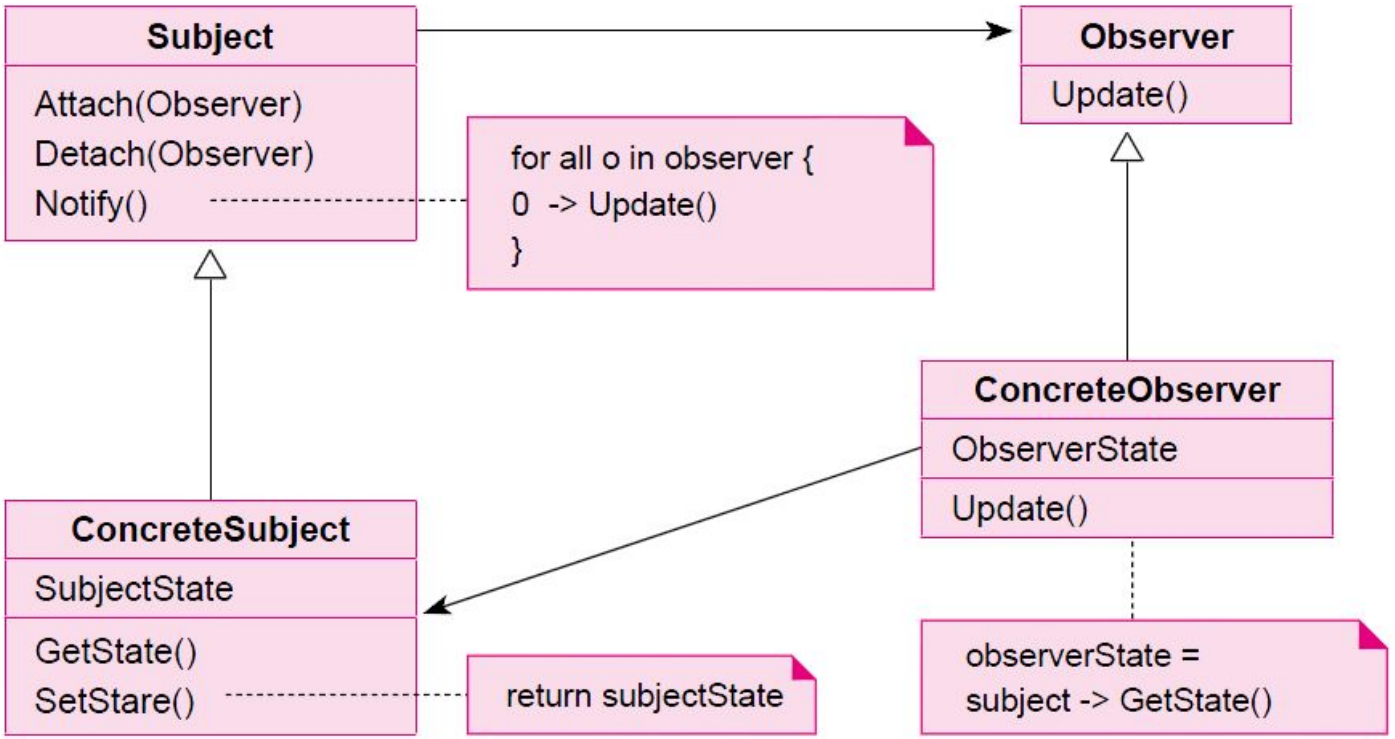
- accesses the elements of an object sequentially without exposing its underlying representation.



- cf. *Iterator, Iterable, for each* statement in Java
  - java.lang.Iterator<E>
  - java.util.Iterable<E>

Observer Pattern

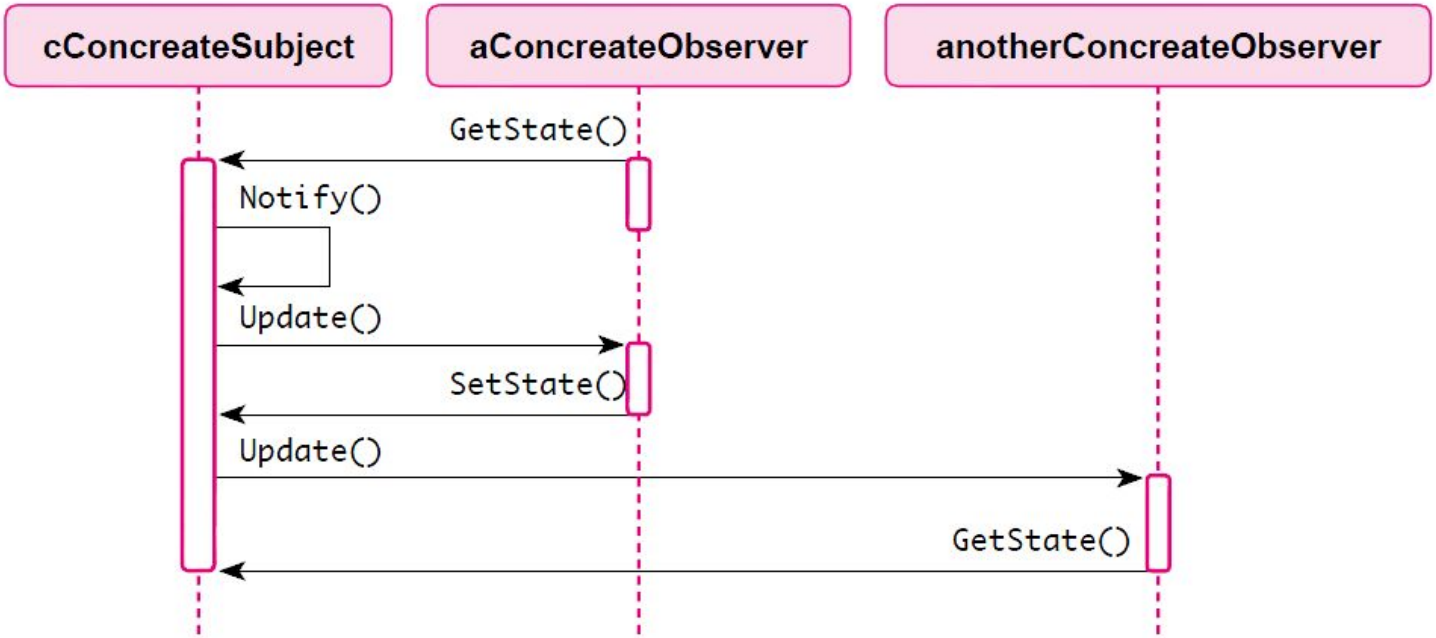
- a publish/subscribe pattern which allows a number of observer objects to see an event.



참고) 최은만 교재

Observer Pattern

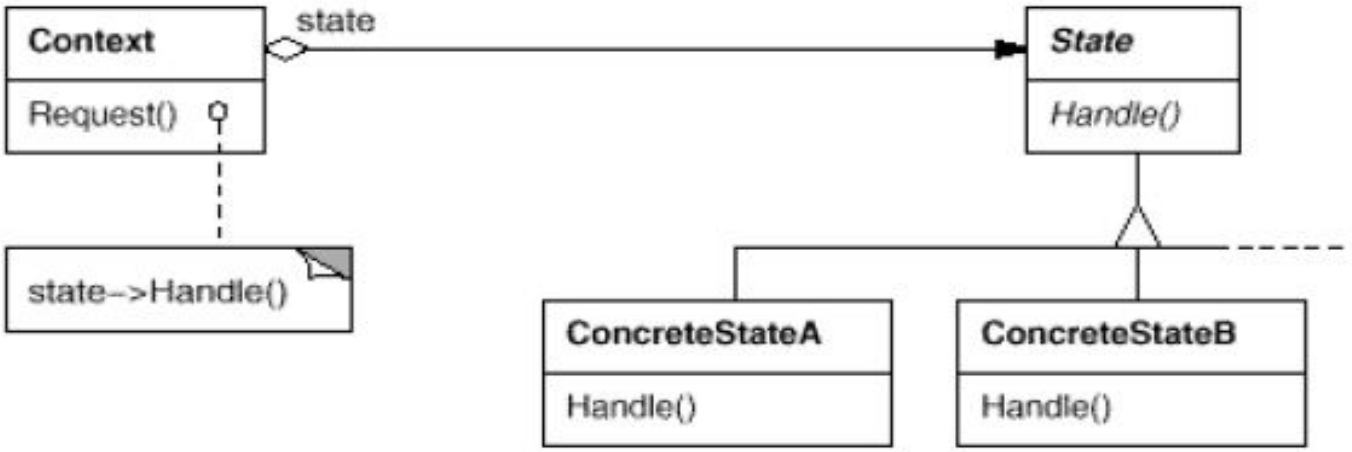
- a publish/subscribe pattern which allows a number of observer objects to see an event.



참고) 최은만 교재

State Pattern

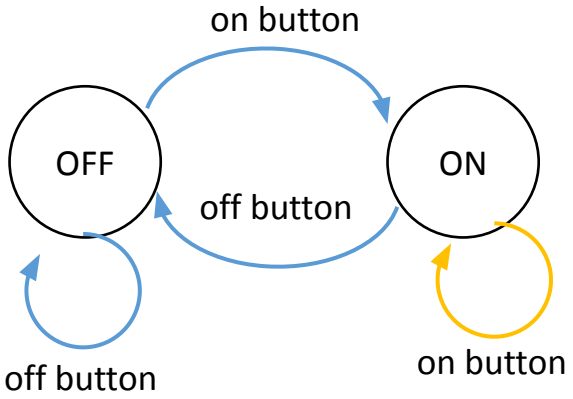
- allows an object to alter its behavior when its internal state changes (a.k.a. objects for states)



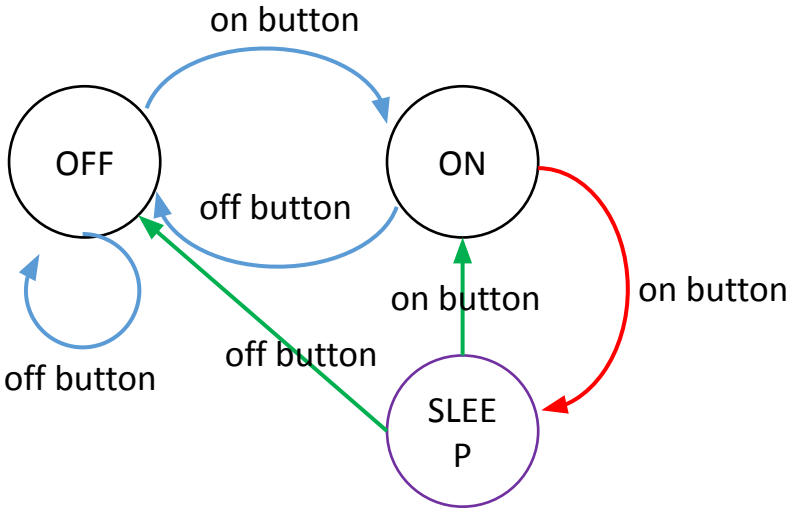
State Pattern

- Light

[The initial design]



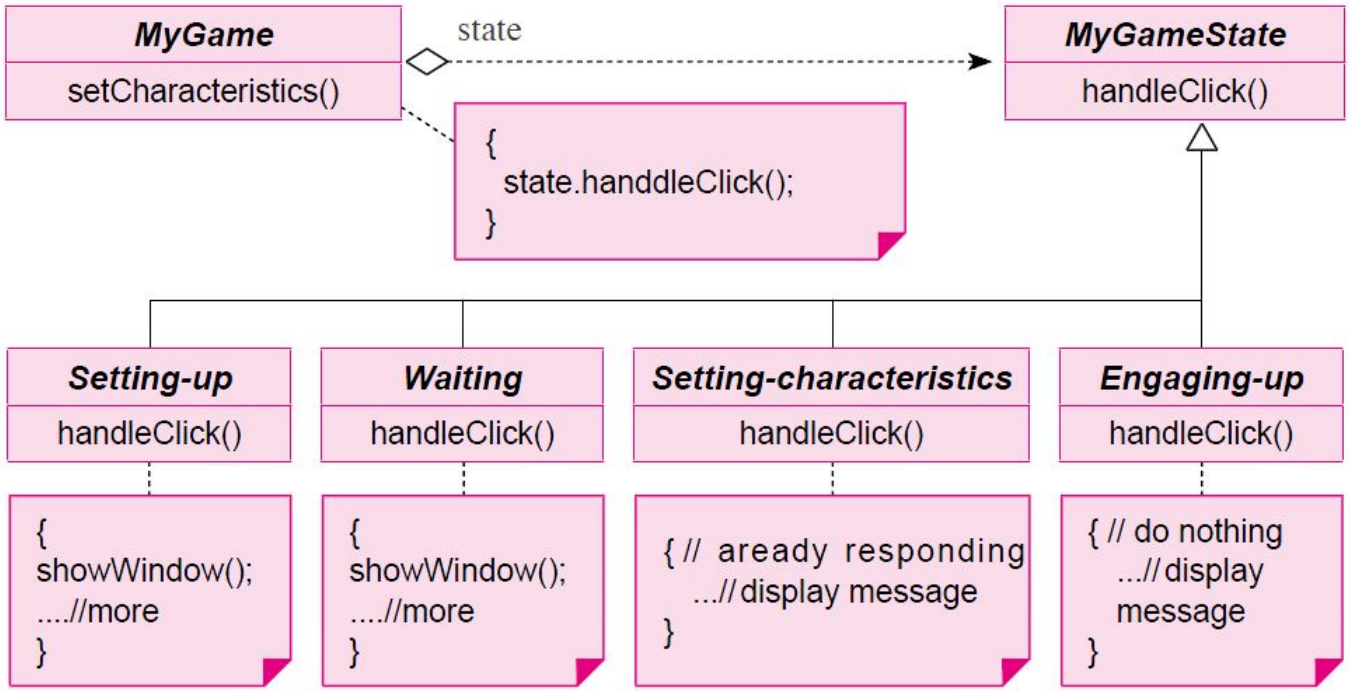
[A new design]



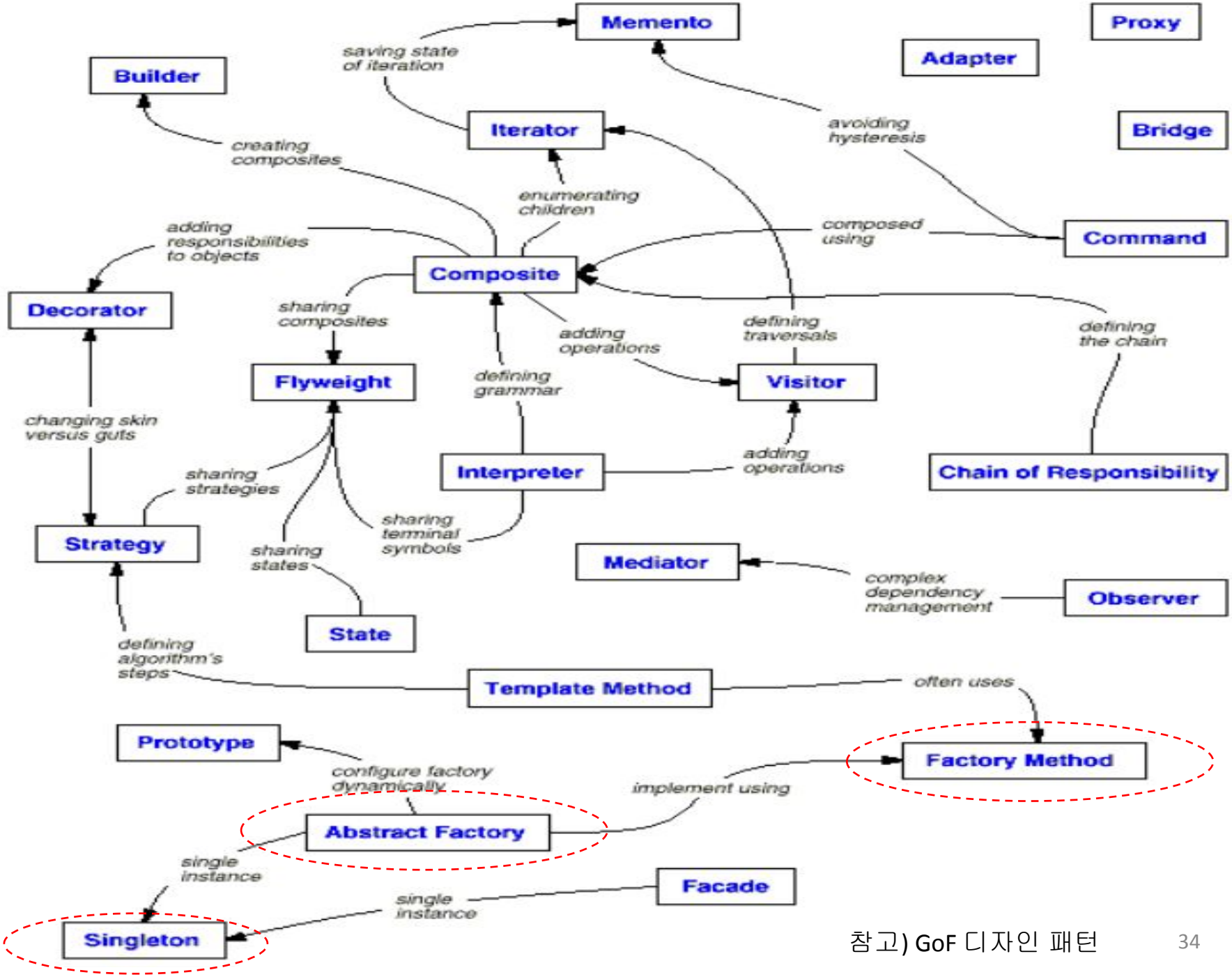
참고) 정인상, 채흥석, Java로 배우는 디자인패턴

State Pattern

- allows an object to alter its behavior when its internal state changes (a.k.a. objects for states)



참고) 최은만 교재


Memento
Proxy
saving state of iteration
Adapter
Builder
Iterator
avoiding hysteresis
Bridge
creating composites
enumerating children
adding responsibilities to objects
composed using
Command
Composite
Decorator
sharing composites
adding operations
defining traversals
defining the chain
Flyweight
defining grammar
Visitor
changing skin versus guts
sharing strategies
Interpreter
adding operations
Chain of Responsibility
Strategy
sharing terminal symbols
sharing states
Mediator
complex dependency management
Observer
State
defining algorithm's steps
Template Method
often uses
Prototype
Factory Method
configure factory dynamically
implement using
Abstract Factory
single instance
Facade
single instance
Singleton

참고) GoF 디자인 패턴