DESIGN PATTERNS

Young B. Park (ybpark@dankook.ac.kr)

ICE BREAKING

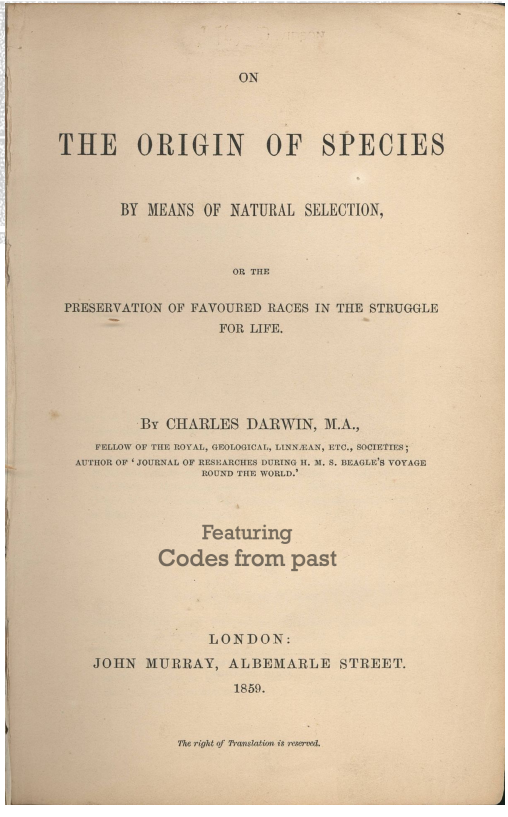
- **Who am I:** 박 용 범
  - 단국대학교 소프트웨어 학과 교수
  - 연락처 **email: ybpark@dankook.ac.kr, cell: 8210-3438-8840**

- **Who are you?**
  - 이름
  - 여기에 있는 이유
  - 이 과정에서 얻고자 하는 지식
  - 좋아하는 숫자

CHANGES + OBLIVION

- It is not the most intellectual of the species that survives; it is not the strongest that survives; but the species that survives is the one that is able best to adapt and adjust to the changing environment in which it finds itself.
  - Leon C. Megginson -
  (as his interpretation of Charles Darwin's ideas)

- Absent of programming, soon forgotten.



ON

THE ORIGIN OF SPECIES

BY MEANS OF NATURAL SELECTION,

OR THE

PRESERVATION OF FAVOURED RACES IN THE STRUGGLE FOR LIFE.

By CHARLES DARWIN, M.A.,

FELLOW OF THE ROYAL, GEOLOGICAL, LINNÆAN, ETC., SOCIETIES;
AUTHOR OF 'JOURNAL OF RESEARCHES DURING H. M. S. BEAGLE'S VOYAGE ROUND THE WORLD.'

Featuring
Codes from past

LONDON:
JOHN MURRAY, ALBEMARLE STREET.
1859.

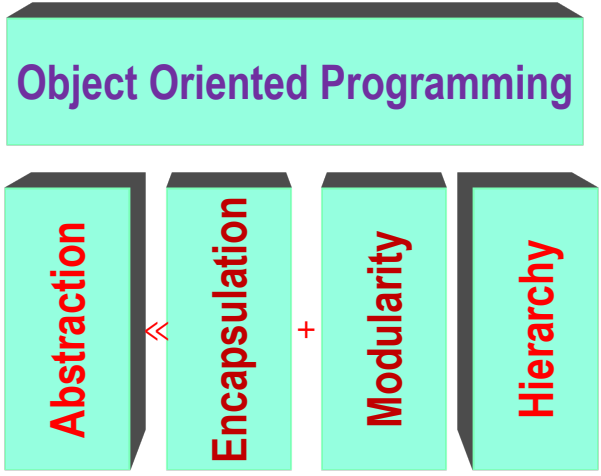
OBJECT ORIENTED DESIGN

1. Goal: make code more understandable and/or more flexible.
2. Design Critiques 실습

OBJECT ORIENTED TECHNOLOGY (REVIEW)

- A set of principles guiding software construction together with languages, databases, and other tools that support those principles.
  (Object Technology - A Manager's Guide, Taylor, 1997)
- A single paradigm
- Facilitates architectural and code reuse
- Models more closely reflect the real world
- Stability
  - A small change in requirements does not mean massive changes in the system under development
- Adaptive to change

BASIC PRINCIPLES OF OOP



- Abstraction: hiding the unnecessary details from the users(who are you?)
- Encapsulation: hide the data in a single entity or unit along with a method to protect information from outside
- Modularity: the property of a system that has been decomposed into a set of cohesive and loosely coupled modules. (Booch)
- Hierarchy: systematic code reuse

ABSTRACTION (REVISE)

- For the purpose of simplifying and managing the design of complex software.
- Allows to separate categories and concepts related to problems from specific instances of implementation.
- Code can be written so that
  - it does not depend on the specific details (e.g. supporting applications, operating system software or hardware)
  - but it depends on an abstract concept of the solution to the problem that can be integrated with the system with minimal additional work.

data는 변수에 code

ENCAPSULATION OF DATA(STATE)&CODE (REVISE)

- Information Hiding: hiding internal design decisions related to the selected data structures and algorithm of an object from the users

- The internal design decisions are most likely to change.
- Thus reduce the side effects of any future maintenance or modification of the design.
- And hence minimizing the effect on the other object in the design.

접근제어자를 class별로 설정할 수 있다!

MODULARITY (REVISE)

- Partition the system into modules and assign responsibility among the components
- Modularity reduces the total complexity a programmer has to deal with at any one time
- OOP helps to archive,
  - Functions are assigned to object in a way that groups similar functions together (Separation of Concerns)
  - There are small, simple, well-defined interfaces between objects (Information Hiding)

HIERARCHY (REVISE)

- Systematic software reuse (hierarchy) is a strategy for increasing productivity and improving the quality of the software industry.
- Although it is simple in concept, successful software reuse implementation is difficult in practice.
- Since the constructing software reuse hierarchy (the dependence of software reuse) is based on the context in which it is implemented.

>INTERFACE (REVISE)

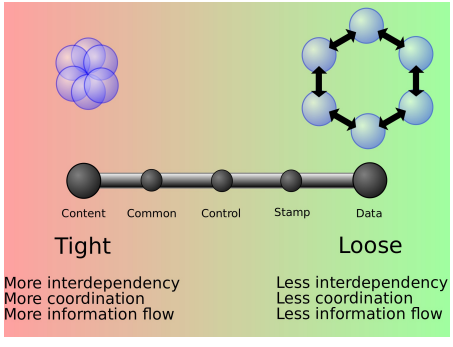
- Interfaces are the points of accesses through which modules or systems communicate.
- Each abstraction should possess a well-defined interface clearly describing the expected inputs to and outputs from the abstraction.
- If the objects are fully encapsulated, then the interface will describe the only way in which objects may be accessed by other objects.
- Some programming languages explicitly support interfaces

>SEPARATION OF CONCERNS (REVISE)

- Separating a computer program into distinct sections, such that each section addresses a separate concern. (“divide and conquer”)
- A concern is a set of information that affects the code of a computer program.
- A program that embodies SoC well is called modular.
- Modularity, and hence separation of concerns, is achieved by encapsulating information inside a section of code that has a well-defined interface.
  - ex) Layered Design
  - ex) Aspect-oriented Programming (AOP)

MEASURING MODULARITY (REVISE)

- Coupling and Cohesion
- Coupling
  - The degree of interdependence between software modules
  - A measure of how closely connected two routines or modules are
  - The strength of the relationships between modules
- Desirable Coupling → Loosely Couple
  - a relationship in which one module interacts with another module through a stable interface and does not need to be concerned with the other module's internal implementation

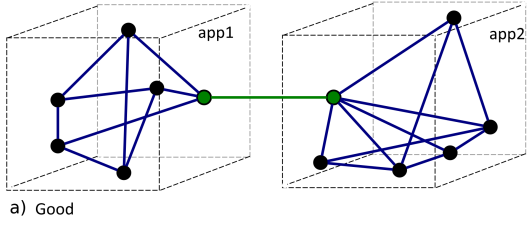


By wmasterj and Fabrice TIERCELIN
https://commons.wikimedia.org/w/index.php?curid=9853618

MEASURING MODULARITY CONTINUE… (REVISE)

- Cohesion
  - The degree to which all elements of a module are directed towards a single task.
  - The degree to which all elements directed towards a task are contained in a single module
  - The degree to which all responsibilities of a single class are related.

- Goal: Loosely Coupled, High Cohesion



HIGH COHESION & LOOSELY COUPLED

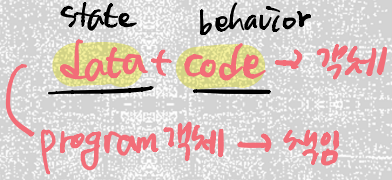
coincidental	• parts of a module are grouped arbitrarily
logical	• grouped because they are logically categorized to do the same thing
Temporal	• processed at a particular time in program execution
procedural	• they always follow a certain sequence of execution
communication	• they operate on the same data
sequential	• the output from one part is the input to another part
functional	• they all contribute to a single well-defined task of the module

Bad

Good

contents	• one module uses the code of another module
common	• access to the same global data
external	• share an externally imposed data format
control	• one module controlling the flow of another
stamp	• share a composite data structure and use only parts of it
data	• modules share data through, for example, parameters

WHAT IS OBJECT



- Informally, an object represents an entity, either physical, conceptual, or Software
- An object is an entity with a well-defined boundary and identity that encapsulates state(data) and behavior.
  - Each object has a unique identity, even if the state is identical to that of another object.
  - State is represented by attributes and relationships.
    The state of an object is one of the possible conditions in which an object may exist.
    The state of an object normally changes
  - Behavior is represented by operations, methods, and state machines.
    Behavior determines how an object acts and reacts.
    The visible behavior of an object is modeled by the set of messages it can respond to (operations the object can perform).

DESIGN PRINCIPLES OF OOP - SOLID

- The Single-Responsibility Principle(SRP)
- The Open Close Principle(OCP)
- The Liskov Substitution Principle(LSP)
- The Interface Segregation Principle(ISP)
- The Dependency Inversion Principle(DIP)

- cf. GRASP (General Responsibility Assignment Software Patterns/Principles) by Craig Larman

단일책임의 원칙

SINGLE-RESPONSIBILITY PRINCIPLE (SRP)

- *"A class should have only one reason to change."* 단 하나의 변할 이유.
- Rationale behind SRP
  - changes in requirements ➔ *changes in class responsibilities*
  - a 'cohesive' responsibility is a single axis of change ➔ *a class should have only one responsibility*
  - responsibility = A reason to change

코드 수정/개선할 때마다 하나의 class를 수정하면 X

책임이 1가지만 있어야함

- Violation of SRP causes spurious transitive dependencies between modules that are hard to anticipate. ➔ fragility

SINGLE-RESPONSIBILITY PRINCIPLE (SRP)

```
public class SuperDashboard extends Jframe implements MetaDataUser {
    public Component getLastFocusedComponent() { … }
    public void setLastFocused(Component lastFocused) { … }
    public int getMajorVersionNumber() { … }
    public int getMinorVersionNumber() { … }
    public int getBuildNumber() { … }
}
```

메소드의 이름 ✿✿✿

Focus

versionNum

두개씩 고쳐야할때 모두 이 class를 건드려야함

→ 단일책임의 원칙 위배

SINGLE-RESPONSIBILITY PRINCIPLE (SRP)

```
public class SuperDashboard extends Jframe implements MetaDataUser {
  public Component getLastFocusedComponent() { … }
  public void setLastFocused(Component lastFocused) { … }
}
```

```
public class Version {
  public int getMajorVersionNumber() { … }
  public int getMinorVersionNumber() { … }
  public int getBuildNumber() { … }
}
```

OPEN/CLOSE PRINCIPLE(OCP)

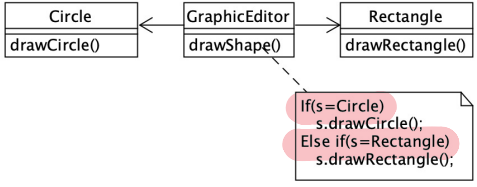
수정할 일 없게 짜라.

살짝만 더하면 추가 가능하도록

- *"Software entities should be open for extension, but closed for modification."*

ex) 상속기능

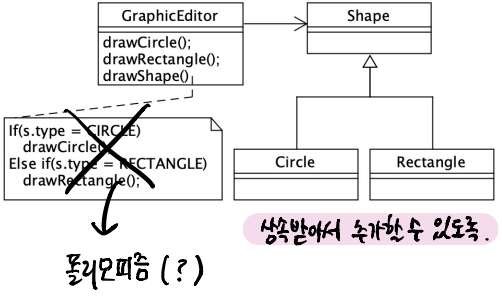
- New functionality should be added with minimum changes in the existing code.
- Reduce Rigidity : a change does not cause a cascade of related changes in dependent module.



<Bad Design?>

다음으로 일일히 대응하는 코드 불필요함

OPEN/CLOSE PRINCIPLE(OCP)



- *"Software entities should be open for extension, but closed for modification."*
- New functionality should be added with minimum changes in the existing code.
- Reduce Rigidity : a change does not cause a cascade of related changes in dependent module.

LISKOV SUBSTITUTION PRINCIPLE(LSP)

- *"Subtypes must be substitutable for their base types."*
  대체가능
- *Let Φ(x) be a property provable about objects x of type T. Then Φ(y) should be true for objects y of type S where S is a subtype of T.*
- A user of a base class should continue to function property if a derivative of that base is passed to it.

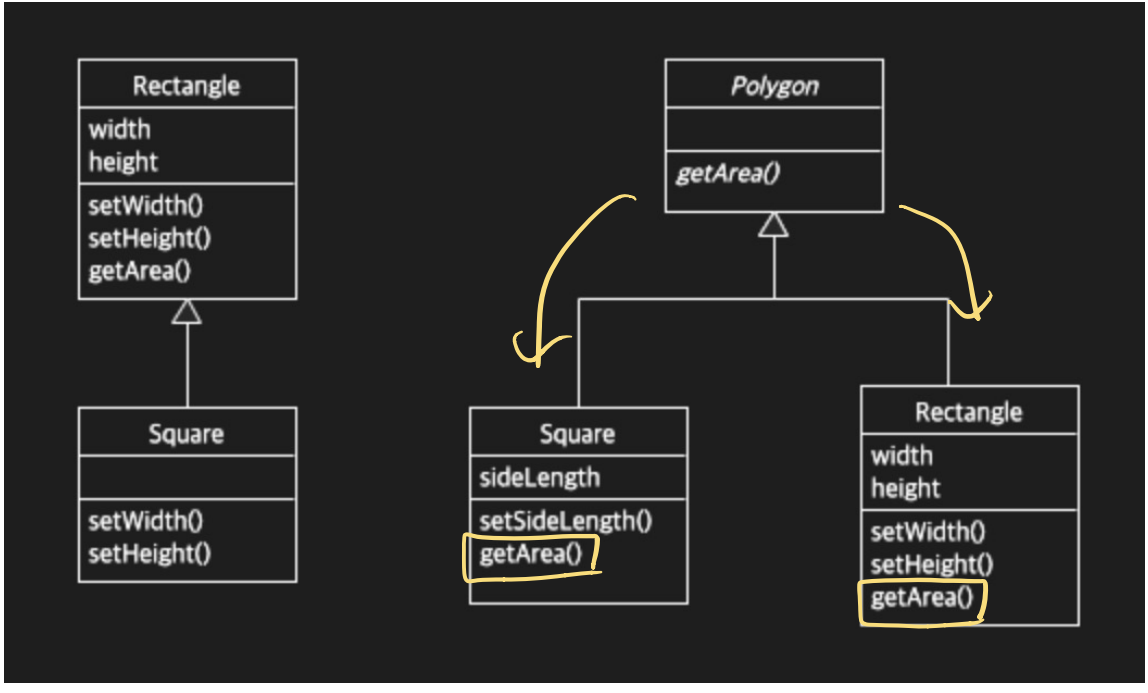
LISKOV SUBSTITUTION PRINCIPLE(LSP)

```
public class Rectangle {

    double width;
    double height;

    public void setWidth(double width) {
        this.width = width;
    }

    public void setHeight(double height) {
        this.height = height;
    }

    public double getArea() {
        return width * height;
    }
}
```

```
public class Square extends Rectangle {
    @Override
    public void setWidth(double width) {
        this.height = width;
        this.width = width;
    }
    @Override
    public void setHeight(double height) {
        this.height = height;
        this.width = height;
    }
}
```

```
Rectangle rectangle = new Rectangle();
rectangle.setHeight(3);
rectangle.setWidth(4);
System.out.println(rectangle.getArea())

Rectangle square = new Square();
square.setHeight(4);
square.setWidth(5);
System.out.println(square.getArea())
```

객체지향에서
상속 ⇔ 코드 재사용성


Rectangle
width
height
setWidth()
setHeight()
getArea()
Square
setWidth()
setHeight()
Polygon
getArea()
Square
sideLength
setSideLength()
getArea()
Rectangle
width
height
setWidth()
setHeight()
getArea()

LISKOV SUBSTITUTION PRINCIPLE(LSP)

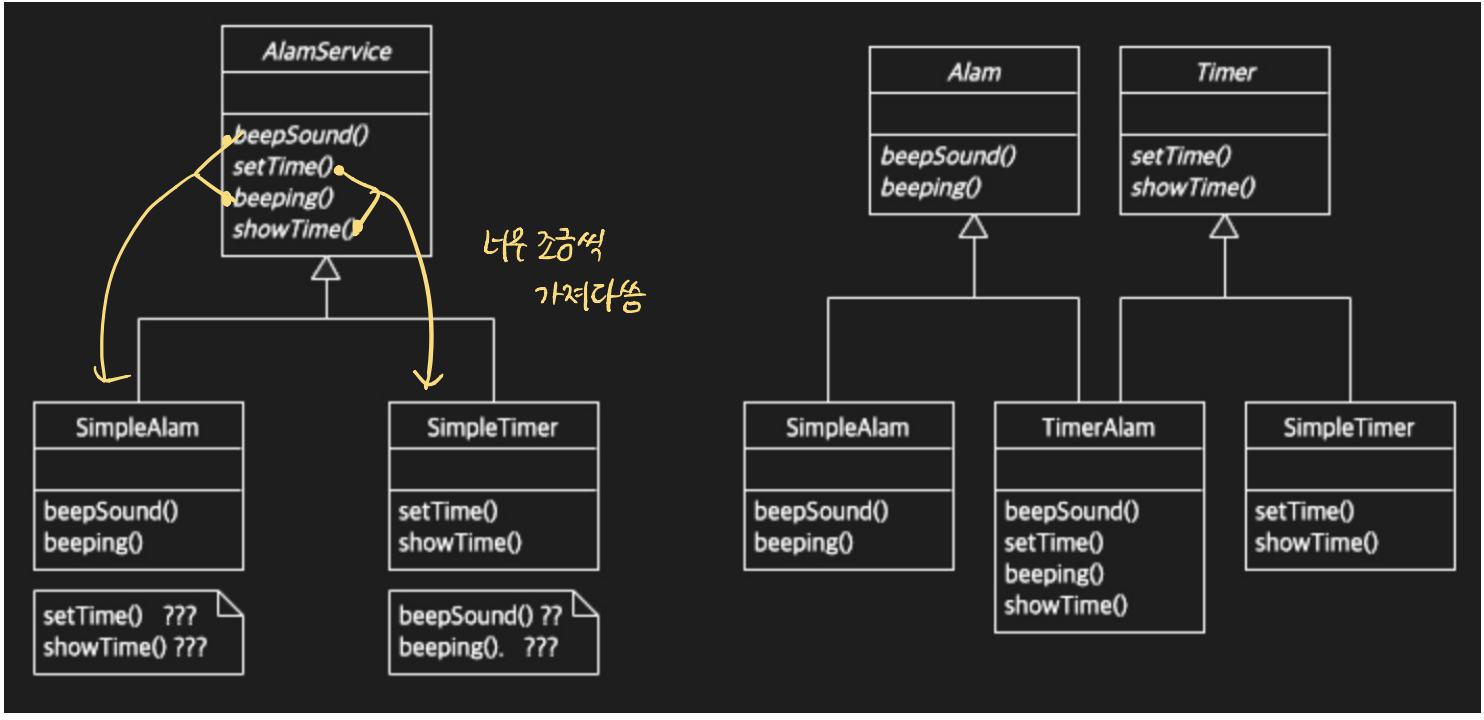
```
public class  Polygon {
    public double getArea(double width) {
    }
}

public class Rectangle extends Polygon {
    double width;
    double height;

    public void setWidth(double width) {
        this.width = width;
    }
    public void setHeight(double height) {
        this.height = height;
    }
    @Override
    public double getArea() {
        return width * height;
    }
}
```

```
public class Square extends Rectangle {
    double sideLength

    public void setSideLength(double sideLength) {
        this.sideLength = sideLength;
    }
    @Override
    public double getArea() {
        return sideLength * sideLength;
    }
}
```

```
Polygon rectangle = new Rectangle();
rectangle.setHeight(3);
rectangle.setWidth(4);
System.out.println(rectangle.getArea())

Polygon square = new Square();
square.setSideLength(4);
System.out.println(square.getArea())
```

INTERFACE-SEGREGATION PRINCIPLE(ISP)

- *"Clients should not be forced to depend upon methods that they do not use."*
- Many client specific interfaces are better than one general purpose interface
  - no 'fat' interfaces
  - no non-cohesive interfaces
- Related to SRP
- e.g., Input and Output interfaces


AlamService
beepSound()
setTime()
beeping()
showTime()
나쁜 결합
가져다씀
SimpleAlam
beepSound()
beeping()
setTime() ???
showTime() ???
SimpleTimer
setTime()
showTime()
beepSound() ??
beeping(). ???
Alam
beepSound()
beeping()
Timer
setTime()
showTime()
SimpleAlam
beepSound()
beeping()
TimerAlam
beepSound()
setTime()
beeping()
showTime()
SimpleTimer
setTime()
showTime()

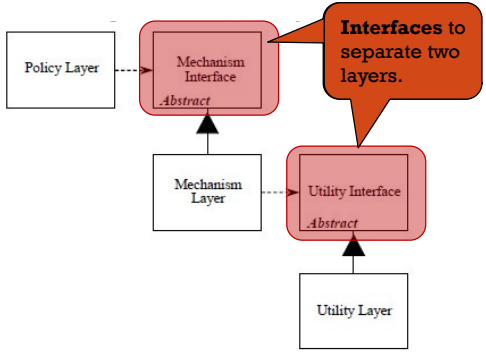
어려움

DEPENDENCY INVERSION PRINCIPLE(DIP)

- *"High-level modules should not depend on low-level modules. Both should depend on abstractions.*
- *Abstractions should not depend on details. Details should depend on abstractions."*
- High-level modules, which provide complex logic, should be easily reusable and unaffected by changes in low-level modules, which provide utility features. To achieve that, you need to introduce an abstraction that decouples the high-level and low-level modules from each other.

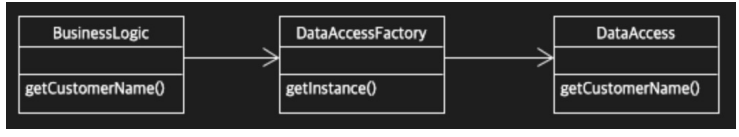
DEPENDENCY INVERSION PRINCIPLE(DIP)

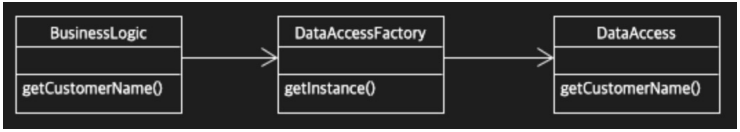
- High-level and low-level modules depend on the abstraction.
- Splits the dependency between the high-level and low-level modules by introducing an abstraction between them. As results
  - the high-level module depends on the abstraction
  - the low-level depends on the same abstraction.

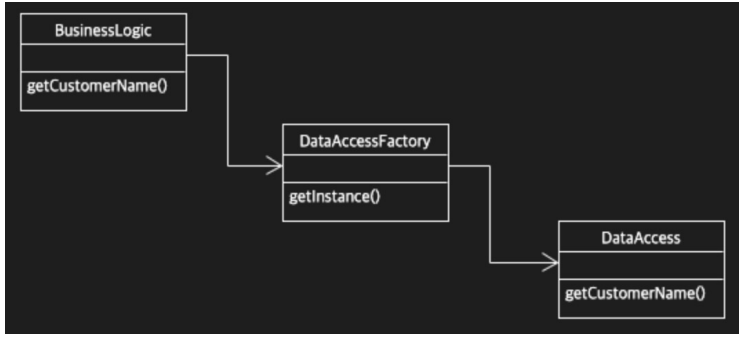


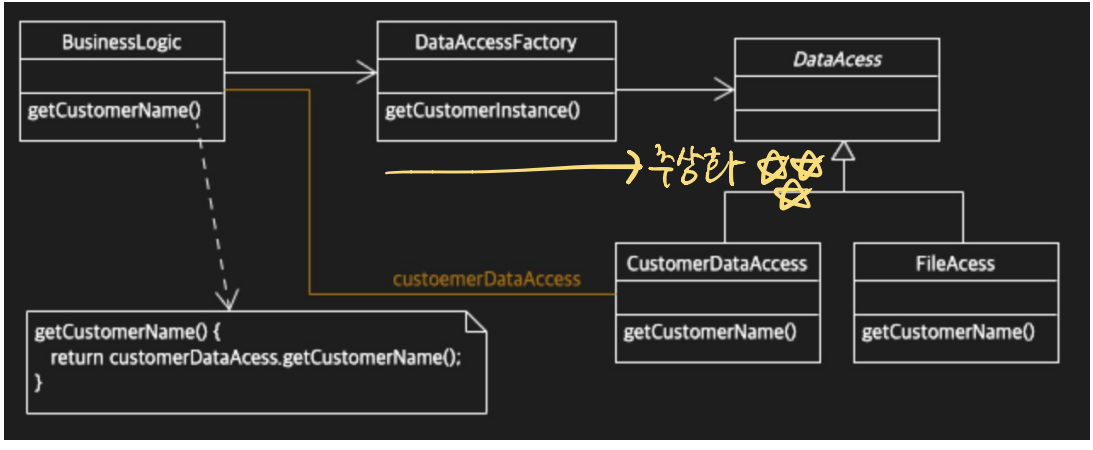
DEPENDENCY INVERSION PRINCIPLE(DIP)

```
public class BusinessLogic {
        public string getCustomerName(int id) {
                DataAccess dataAccess = DataAccessFactory.GetInstance();
                return dataAccess.getCustomerName(id);
}       }

public class DataAccessFactory {
        public static DataAccess getInstance() {
                return new DataAccess();
}       }

public class DataAccess {
        public string getCustomerName(int id) {
                return "Customer Name From DB";
}       }
```

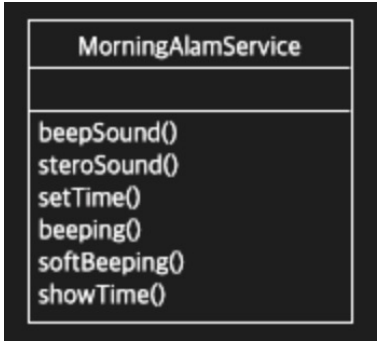


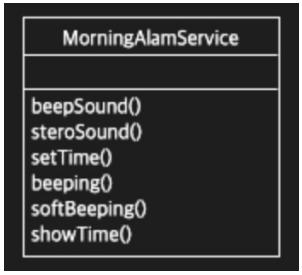

BusinessLogic
getCustomerName()
DataAccessFactory
getInstance()
DataAccess
getCustomerName()

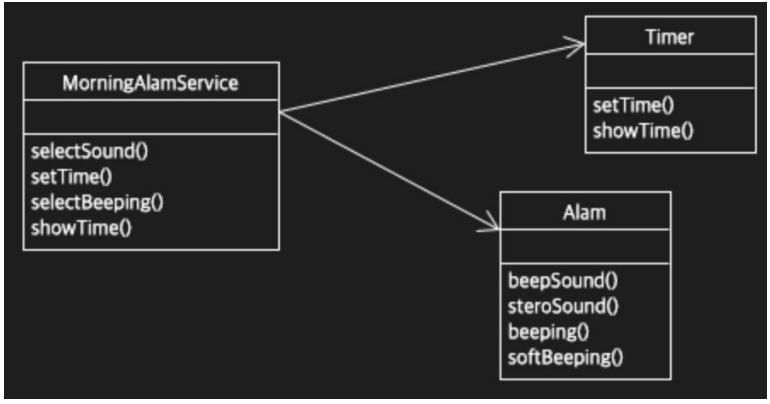

BusinessLogic
getCustomerName()
DataAccessFactory
getInstance()
DataAccess
getCustomerName()

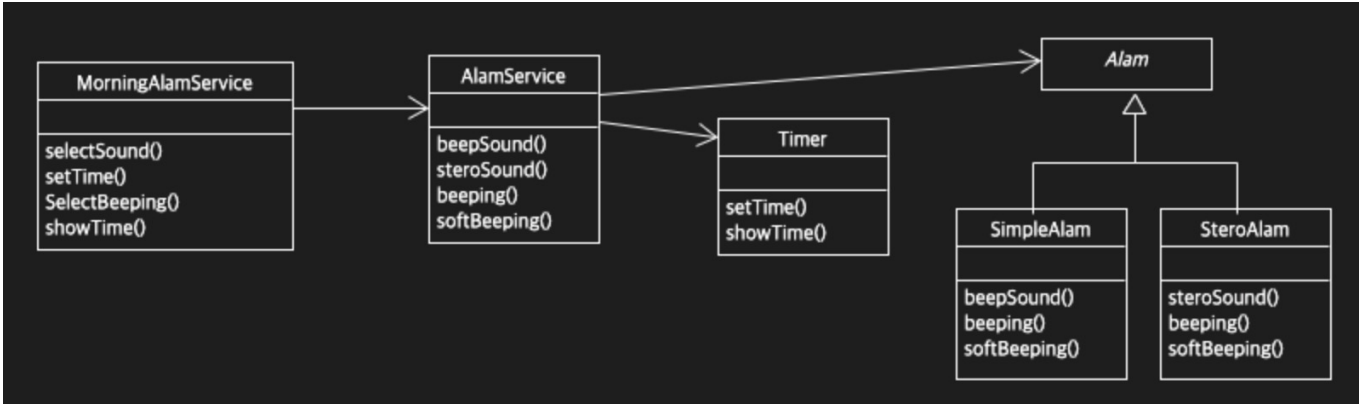

BusinessLogic
getCustomerName()
DataAccessFactory
getCustomerInstance()
DataAcess
추상화
custoemerDataAccess
CustomerDataAccess
getCustomerName()
FileAcess
getCustomerName()
getCustomerName() {
return customerDataAcess.getCustomerName();
}

DESIGN CRITIQUES 실습




MorningAlamService
beepSound()
steroSound()
setTime()
beeping()
softBeeping()
showTime()


MorningAlamService
selectSound()
setTime()
selectBeeping()
showTime()
Timer
setTime()
showTime()
Alam
beepSound()
steroSound()
beeping()
softBeeping()


MorningAlamService
selectSound()
setTime()
SelectBeeping()
showTime()
AlamService
beepSound()
steroSound()
beeping()
softBeeping()
Timer
setTime()
showTime()
Alam
SimpleAlam
beepSound()
beeping()
softBeeping()
SteroAlam
steroSound()
beeping()
softBeeping()