OOP 소개

제주대학교 고급객체지향자바개발론 수업 2012-09-07

최범균
트위터: @madvirus, 이메일: madvirus@madvirus.net

강사 소개

- 최범균
  - 트위터: @madvirus
  - 이메일: madvirus@madvirus.net

- 이력
  - 현) 에스씨지솔루션즈
  - 전) 위메이드 엔터테인먼트
  - 전) 다음 커뮤니케이션
  - 자바 7 프로그래밍, JSP 프로그래밍 등 저

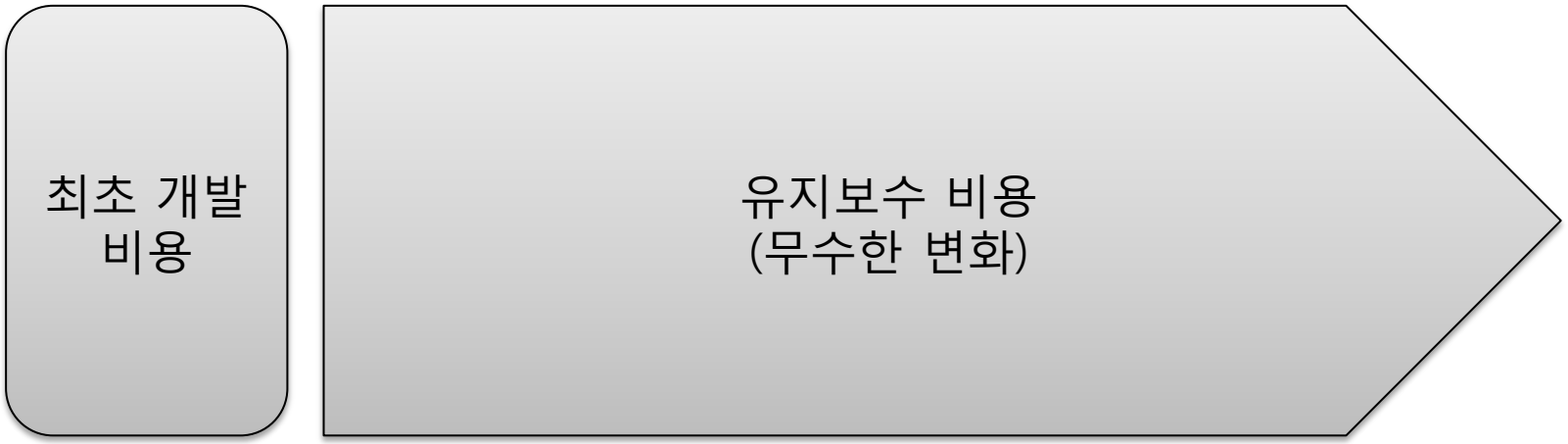
TOC

- 비용
- 절차 지향 vs 객체 지향
- 객체 지향
- 캡슐화
- 추상화
- 유연함
- 정리

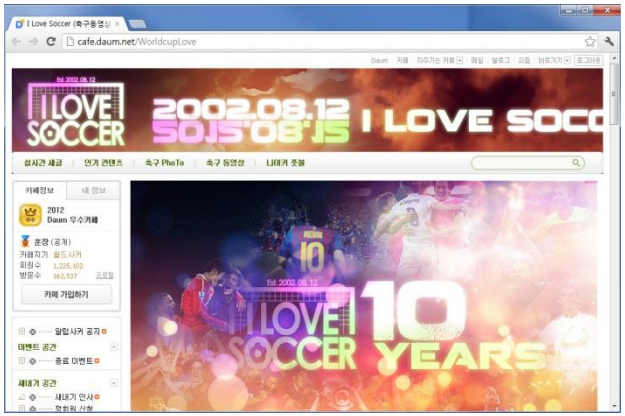
비용

개발 비용?

비용의 구성



유지보수 사례



다음 카페

아고라 게시판

비용을 줄이려면

- 유지보수 비용 줄이려면
  - 코드 변경이 용이
  - 코드 분석이 용이

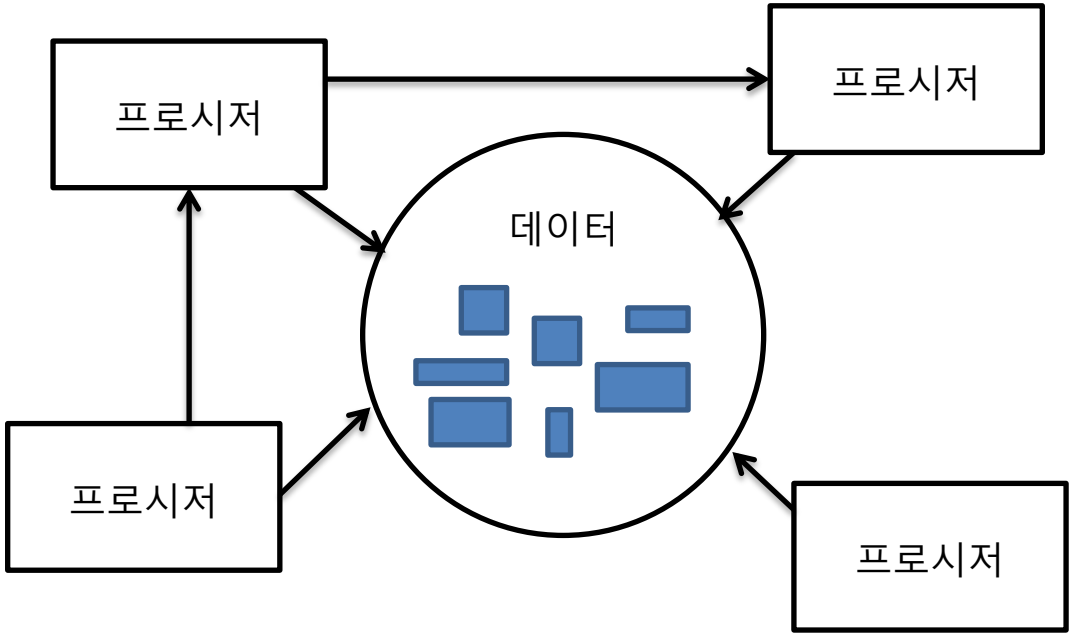
- 코드 변경/분석이 용이하려면,
  - 코드 자체의 가독성 향상
  - 변경에 유연하게 대처 가능한 구조
  - 코드가 비즈니스 모델을 표현

답 중의 하나

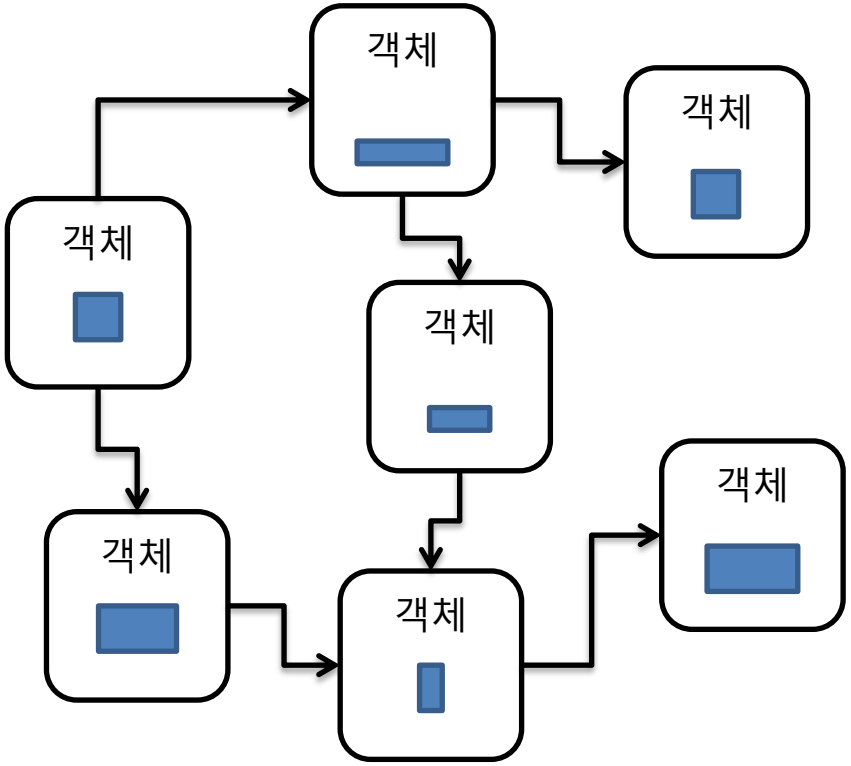
Object Oriented

객체 지향

절차 지향

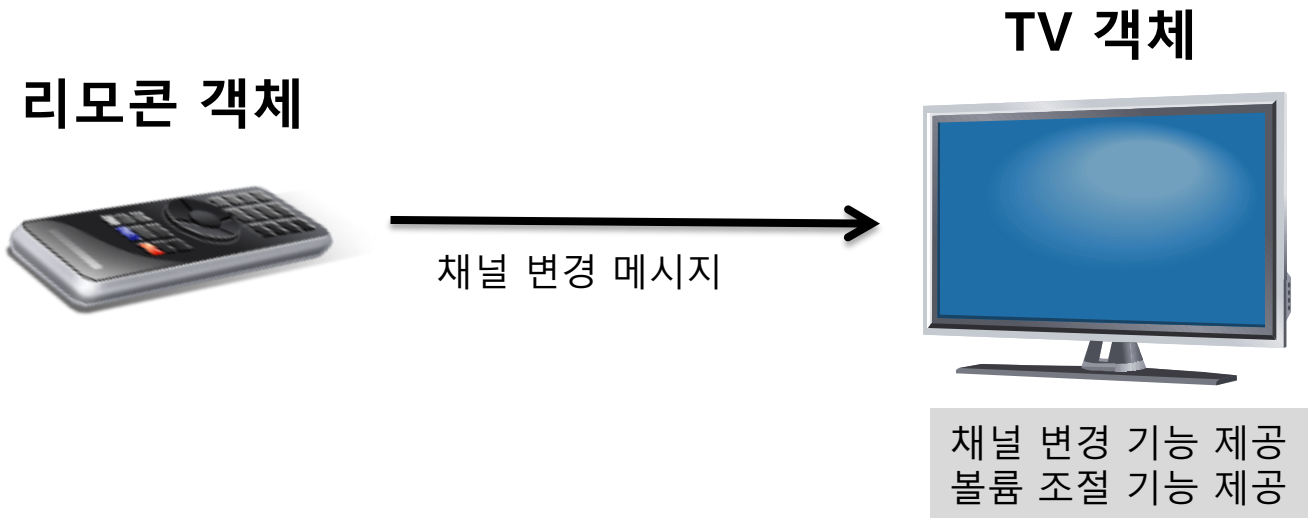


객체 지향



객체(Object)

- 객체는 제공하는 기능으로 정의
  - 객체는 역할을 가짐
  - 객체가 어떤 데이터를 보관하는지로 정의하지 않음
- 메시징
  - 객체와 객체는 기능 실행을 위해 메시지를 주고 받음
  - 일반적으로 메서드 호출을 통해 메시지 전달



클래스

- 객체를 표현하기 위한 수단
  - 자바, C#과 같은 언어에서 객체를 정의하기 위한 수단으로 클래스를 사용함

```
public class TV {

    public void increaseVolume() {
        …
    }
    public void decreaseVolume() {
        …
    }
}
```

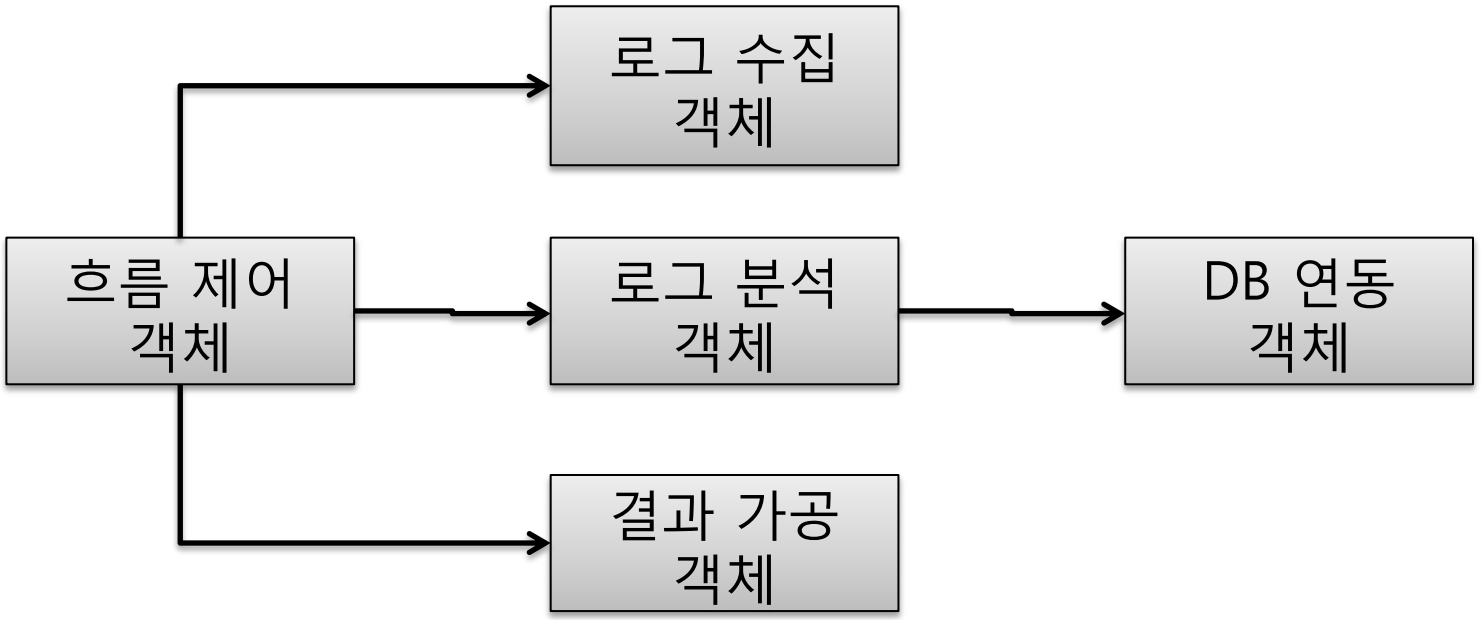
제공할 기능을 정의

```
TV tv = new TV();

tv.increaseVolume();
```

객체에 메시지를 전달

역할/책임(Responsibility) 그리고 관계

- 각 객체들은 자신만의 역할을 가짐
- 객체 간 관계 형
  - 서로 다른 역할을 수행하는 객체 간 메시지 전달을 통해 전체 기능 수행



객체 설계의 기본 과정

- 기능(역할)을 제공할 객체 후보 선별
  - 내부에서 필요한 데이터 선별
- 객체 간 메시지 흐름 연결
- 위 과정 되풀이

캡슐화

캡슐화(Encapsulation)

- 객체가 내부적으로 어떻게 기능을 구현했는지는 감춤

- 캡슐화를 통해 객체의 내부 구현이 변경되더라도 객체를 사용하는 코드는 변경되지 않도록 함
  - 코드 변경에 따른 비용 최소화

- 객체 지향의 가장 기본

데이터 중심 구현: 절차 지향 스톱워치 - 최초 코드

```
public class ProceduralStopWatch {
    public long startTime; // 밀리초(1/1000초) 단위
    public long stopTime; // 1/1000초 단위
    public long getElapsedTime() {
        return stopTime - startTime;
    }
}
```

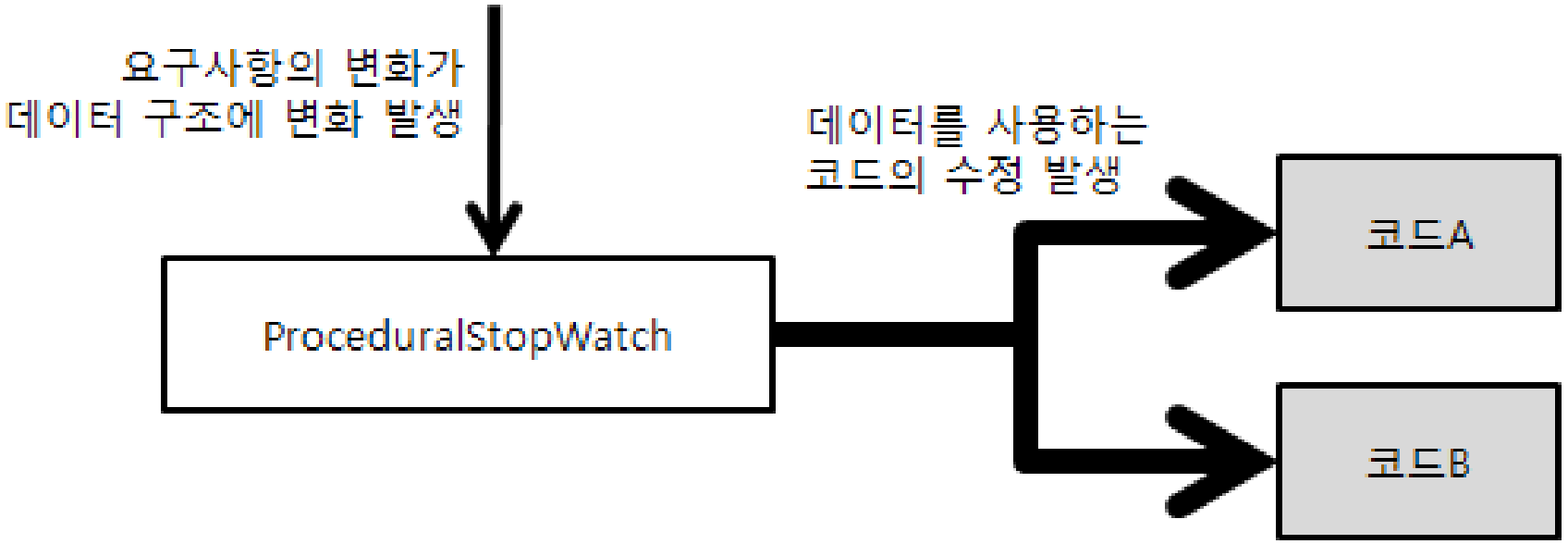
```
ProceduralStopWatch stopWatch = new ProceduralStopWatch();
stopWatch.startTime = System.currentTimeMillis(); // 시작 시간
// 측정 대상 기능 실행
stopWatch.stopTime = System.currentTimeMillis(); // 종료 시간
long elapsedTime = stopWatch.getElapsedTime(); // 시간 차이
```

데이터 중심의 절차 지향 구현 방식 - 기능 추가 여파

```
public class ProceduralStopWatch {
    public long startTime;
    public long stopTime;
    public long startNanoTime;
    public long stopNanoTime;
    …
    public long getElapsedNanoTime() {
        return stopNanoTime - startNanoTime;
    }
}
```

```
ProceduralStopWatch stopWatch = new ProceduralStopWatch();
stopWatch.startNanoTime = System.nanoTime(); // 시작 시간
// 측정 대상 기능 실행
stopWatch.stopNanoTime = System.nanoTime(); // 종료 시간
long elapsedTime = stopWatch.getElapsedNanoTime(); // 차이
```

절차 지향의 한계 - 요구사항 변경에 따른 수정 전파



객체 지향 방식 - 데이터가 아닌 기능/모델을 제공

```
public class StopWatch {
    private long startTime;
    private long stopTime;

    public void start() {
        startTime = System.currentTimeMillis();
    }

    public void stop() {
        stopTime = System.currentTimeMillis();
    }

    public Time getElapsedTime() {
        return new Time(stopTime - startTime);
    }
}
```

객체가 제공하는 기능을 사용/모델을 사용

```
StopWatch stopWatch = new StopWatch();
stopWatch.start(); // startTime 필드에 값을 할당 아닌, 기능 실행
// 코드
stopWatch.stop(); // stopTime 필드에 값을 할당 아닌, 기능 실행
Time time = stopWatch.getElapsedTime(); // long 타입이 아님

time.getMilliTime();
```

객체 지향 - 요구사항 변경에 따른 변화가 객체로 수렴

```
public class StopWatch {
    private long startTime;
    private long stopTime;
    public void start() {
        startTime = System.nanoTime();
    }
    public void stop() {
        stopTime = System.nanoTime();
    }
    public Time getElapsedTime() {
        return new Time(stopTime - startTime);
    }
}
```

```
public class Time {
    private long t;

    public Time(long t) {
        this.t = t;
    }
    public long getMilliTime() {
        return t / 1000000L;
    }
    public long getNanoTime() {
        return t;
    }
}
```

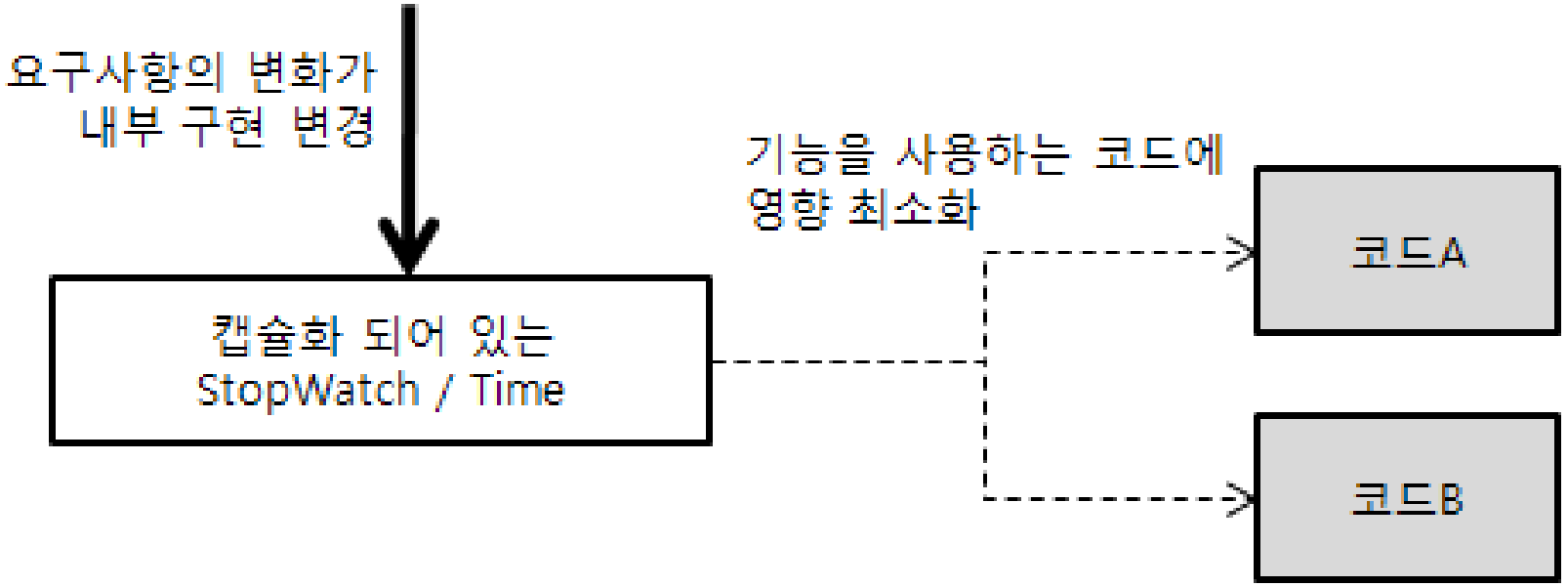
절차 지향에 비해 확실히 작은 변경의 여파

```
StopWatch stopWatch = new StopWatch();
stopWatch.start();
// 코드
stopWatch.stop();
Time time = stopWatch.getElapsedTime();

time.getNanoTime();
```

수정 비용의 감소

```
ProceduralStopWatch stopWatch = new ProceduralStopWatch();
stopWatch.startNanoTime = System.nanoTime(); // 시작 시간
// 측정 대상 기능 실행
stopWatch.stopNanoTime = System.nanoTime(); // 종료 시간
long elapsedTime = stopWatch.getElapsedNanoTime(); // 차이
```

캡슐화 - 구현 변경의 유연함!



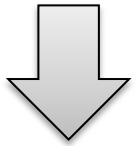
기본 규칙: Tell, Don't Ask

Tell, Don't Ask

- 데이터 달라 하지 말고, 기능 실행하라고 하기
  - 다른 말로 하면
    - 데이터가 흘러 다니지 않도록
    - 즉, 절차 지향이 되지 않도록

- 간단한 예

```
if (member.getExpireDate().getTime() < System. System.currentTimeMillis) {
```

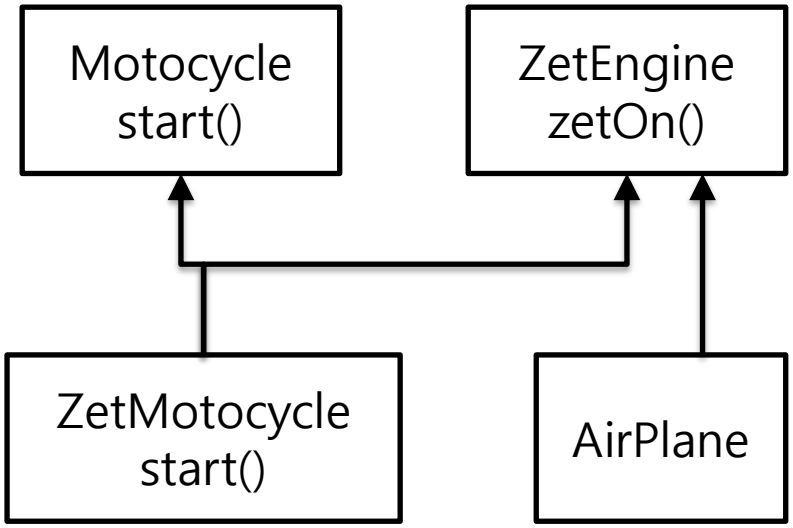


```
if (member.isExpired()) {
```

추상화와 다형성

다형성Polymorphism이란?

- 객체가 여러 역할을 수행할 수 있음
  - 객체 지향에서는 '상속Inheritance'으로 구현
    - 타언어의 Duck Typing 등도 존재




```
ZetMotocycle zm = new ZetMotocycle();
zm.start();
zm.zetOn();

Motocycle mc = zm;
mc.start();

ZetEngine ze = zm;
ze.zetOn();

ZetEngine ap = new AirPlane();
ap.zetOn();
```

상속의 두 가지

- 구현 상속
  - 상위 타입의 구현을 재사용하는 방법

```
public class ZetEngine {
    public void zetOn() { … }
}
public class AirPlane extends ZetEngine {
    public void turnManualMode() { … }
}


AirPlane ap = new AirPlane();
ap.zetOn();
```

- 인터페이스 상속
  - 다형을 갖는 방법

```
public interface ZetEngine {
    public void zetOn();
}
public interface Motocycle {
    public void start();
}
public class ZetMotocycle
    implements ZetEngine, Motocycle {
    public void turnManualMode() { … }
    public void start(0 { … }
}


ZetMotocycle zm = new ZetMotocycle();
ZetEngine ze = zm;
Motocycle mc = zm;
```

추상화Abstraction

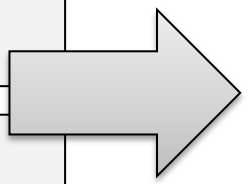
- 데이터/프로세스 등을 의미가 비슷한 개념/표현 정의하는 과정

타입 추상화

- 공통된 데이터나 프로세스를 제공하는 객체들을 하나의 (인터페이스) 타입으로 추상화하는 것

```
class FtpLogCollector {
    private String ftpServer;
    public FtpLogSet collect() { … }
}
class FtpLogSet {
    …
    Iterator iterator() { … }
}
```

```
class DBLogCollector {
    private String jdbcUrl;
    public DBRowLogSet collect() { … }
}
class DBrowLogSet {
    …
    Iterator iterator() { … }
}
```



```
interface LogCollector {
    LogSet collect();
}

interface LogSet {
    Iterator iterator();
}
```

구현 없이 기능만 정의

추상 타입과 구현과의 연결: 상속

```
class FtpLogCollector implements LogCollector {
    private String ftpServer;
    public LogSet collect() {
        …
        return new FtpLogSet(…);
    }
}
class FtpLogSet  implements LogSet {
    …
    Iterator iterat
}
```

```
class DBLogCollector implements LogCollector {
    private String jdbcUrl;
    public LogSet collect() {
        …
        return new DBRowLogSet(…);
    }
}
class DBrowLogSet implements LogSet {
    …
    Iterator iterator() { … }
}
```

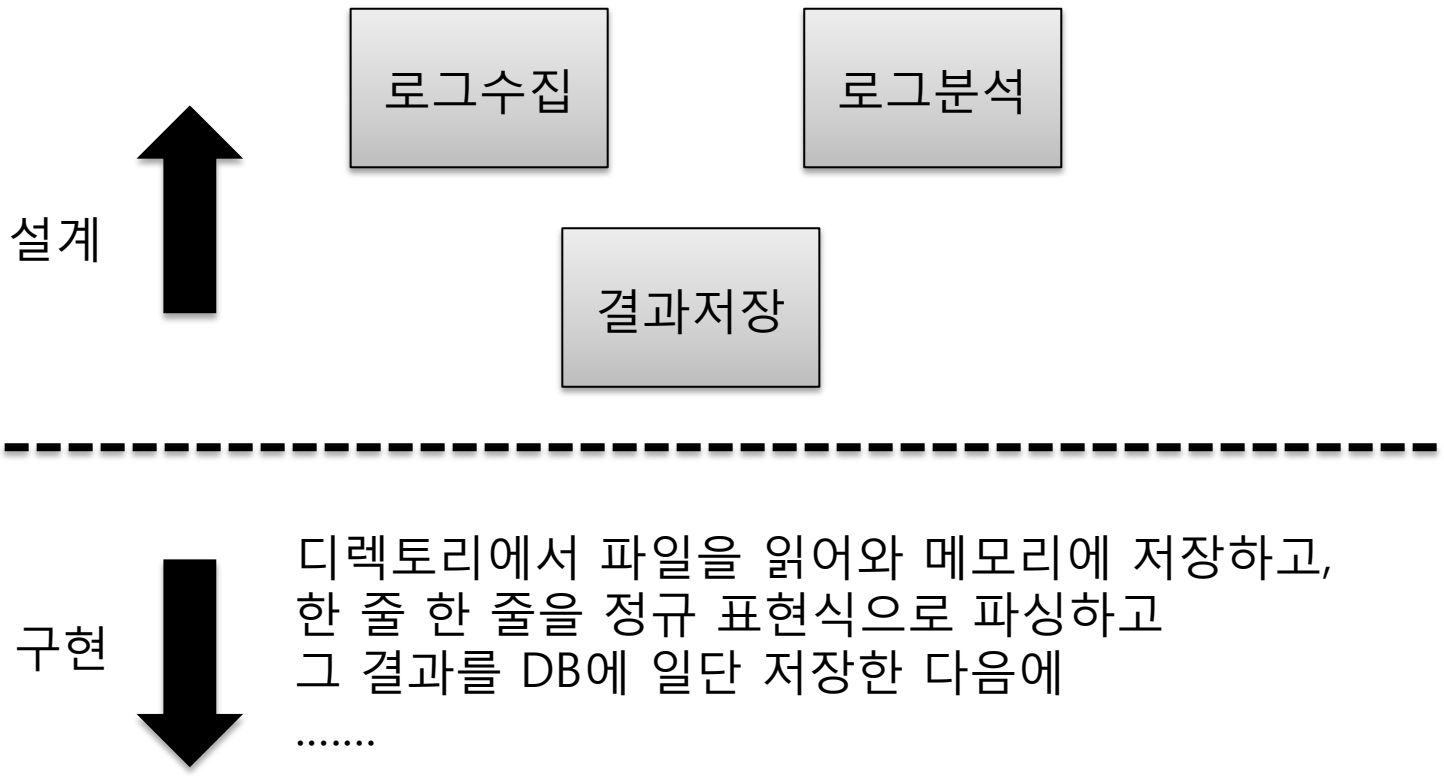
추상화와 다형성

```
LogCollector collector = new FtpLogCollector(ftpServer);

LogSet logSet = collector.collect();
Iterator iter = logSet.iterator();
```

구현 클래스가 아닌 추상화 타입을 이용한 코드 작성 가능

```
LogCollector collector = LogCollectorFactory.create();

LogSet logSet = collector.collect();
Iterator iter = logSet.iterator();
```

추상화와 뇌


로그수집
로그분석
결과저장
설계
상세 구현
디렉토리에서 파일을 읽어와 메모리에 저장하고,
한 줄 한 줄을 정규 표현식으로 파싱하고
그 결과를 DB에 일단 저장한 다음에
.....
.....

추상화/다형성과 유연함

구현 클래스를 직접 사용한다면,

```
FtpLogCollector collector = new FtpLogcollector();
FtpLogSet logSet = collector.collect();
Iterator iter = logSet.iterator();
```

파일로 바꿔주세요

```
FileLogCollector collector = new FileLogcollector();
FileLogSet logSet = collector.collect();
Iterator iter = logSet.iterator();
```

바꿔주세요

```
XXXLogCollector collector = new XXXLogcollector();
XXXLogSet logSet = collector.collect();
Iterator iter = logSet.iterator();
```

추상화 타입에 대고 프로그래밍하기

- (가능한) 구현 클래스가 아닌 인터페이스 사용

```
LogCollector collector = ...;
LogSet logSet = collector.collect();
Iterator iter = logSet.iterator();
```

바꿔주세요

```
LogCollector collector = ...요기;
LogSet logSet = collector.collect();
Iterator iter = logSet.iterator();
```

바꿔주세요

```
LogCollector collector = ...요기;
LogSet logSet = collector.collect();
Iterator iter = logSet.iterator();
```

추상화/다형성의 혜택: 구현 교체의 유연함

```
LogCollector collector = LogCollectorFactory.create();
LogSet logSet = collector.collect();
Iterator iter = logSet.iterator();
```


```
LogCollector collector = LogCollectorFactory.create();
LogSet logSet = collector.collect();
Iterator iter = logSet.iterator();
```


```
LogCollector collector = LogCollectorFactory.create();
LogSet logSet = collector.collect();
Iterator iter = logSet.iterator();
```

중요한/필요한 역량: 인터페이스(추상 타입) 뽑아내기!

- 인터페이스 후보
  - public 메서드
    - public 메서드는 기능을 제공할 가능성이 높고,
    - 기능은 공통된 프로세스일 가능성이 높음
  - 여러 클래스에서 중복되는 구현
    - 공통의 프로세스일 가능성 높음

- 인터페이스는 항상?
  - 미래를 너무 대비할 필요는 없다.
    - 모든 클래스에 대해 상위 인터페이스를 만들 필요 없음
  - 단, 객체 간 호출이 많다면 인터페이스 분리 좋음

기본규칙:Composition over inheritance

(잘못된) 구현 상속의 문제

```
public class LuggageCompartment
        extends ArrayList<Luggage> {
    private int restSpce;
    public void add(Luggage piece) {
        this.restSpace -= piece.getSize();
        super.add(piece);
    }

    public void canContain(Luggage piece) {
        return this.restSpace > piece.size();
    }

    public void extract(Luggage piece) {
        this.restSpace += piece.getSize();
        super.remove(piece);
    }

}
```

```
LuggageCompoartment lc =
        new LuggageCompartment();
lc.add(new Luggage(10));

// 앗!! restSpace가 계산되지 않는다!
lc.remove(someLuggage);

lc.extract(anyLuggage);
lc.canContain(aLuggage); // 잘못된 결과
```

상속은 IS-A 에 대한 것

- 수화물 창고 != 배열기반 목록
- 동일 역할인 경우에만 구현 상속 통한 재사용
  - 예, ArrayList is a AbstractList
- 구현 상속의 고민거리
  - 구현 상속은 상위 클래스의 변경이 모든 하위 클래스에 영향을 줌

조립을 통한 구현 재사용

- 구현 재사용은 상속 보단 조립(composition)

```
public class LuggageCompartment {

    private List<Luggage> luggages = new ArrayList<Luggage>();
    private int restSpce;

    public void add(Luggage piece) {
        restSpace -= piece.getSize();
        luggages.add(piece);
    }
    public void canContain(Luggage piece) {
        return this.restSpace > piece.size();
    }
    public void extract(Luggage piece) {
        restSpace += piece.getSize();
        luggage.remove(piece);
    }
}
```

조립과 유연함

- 조립과 다형성이 만나 유연함을 증가시킴

```
public class Calculator {
    private PriceStrategy strategy;
    public Calculator(PriceStrategy strategy) {
        this.strategy = strategy;
    }
    public void calculate(...) {
        this.strategy.apply(price);
    }
}
```

```
public interface PriceStrategy {
    void apply(Money price);
}

public class RegularCustomerStrategy
     implements PriceStrategy { ... }

public class FirstCustomerStrategy
     implements PriceStrategy { ... }
```

```
PriceStrategy strategy = new RegularCustomerStrategy();
Calculator cal = new Calculator(strategy);
cal.calculate();

PriceStrategy strategy = new FirstCustomerStrategy();
Calculator cal = new Calculator(strategy);
cal.calculate();
```

Calculator 변경없이 새로운 가격정책 적용이 가능

정리

모두 아우르는 大 원칙!

- High Cohesion (높은 응집)
  - 관련된 것들은 최대한 한 곳에 모은다
  - 변화가 한 곳에 집중되기 때문에, 변화에 따른 영향 최소화
- Low Coupling (낮은 결합도)
  - 상호 간 얽혀 있는 부분을 최소화
  - 한 코드의 변화로 인해 다른 코드의 변화가 발생하는 것을 최소화

정리

- 비용: 최초 개발 <<<< 유지보수
  - 유지보수 비용을 줄이기 위한 방법
    - 코드 가독성 향상, 변경에 유연한 구조,

- 객체 지향
  - 서로 다른 기능을 제공하는 객체 간의 메시징
    - 자바나 C#과 같은 언어는 클래스와 메서드로 객체/메시징을 구현
  - 캡슐화, 추상 타입을 통한 변화에 따른 영향 최소화
    - 이는 개발 비용의 감소로 이어짐
  - 기본 규칙
    - Tell, Don't Ask
    - Program To Interface
    - Composition over Inheritance

개발자로서 성장하기 위해 해야할 것

- 표현 방법 익히기
  - UML
- 객체 지향 익히기
  - 설계 원칙
  - 패턴 (GoF 패턴): 공통된 상황을 추상화하는 방법
- TDD
  - 기능 중심의 설계를 할 수 있도록 유도!
- 좋은 코드 만들기
  - Clean Code
  - Implementation Pattern

질문